HEAVEN: a Hardware-Enhanced AntiVirus ENgine to accelerate real-time, signature-based malware detection

Marcus Botacin[1] Marco Antonio Zanata Alves[1] Daniela Oliveira[2] André Grégio[1]

[1]Federal University of Paraná (UFPR-BR)
{mfbotacin,mazalves,gregio}@inf.ufpr.br
[2]University of Florida (UF-USA)
daniela@ece.ufl.edu

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.

1 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 [69], since this is the fastest way to respond to incidents related to newly-discovered samples (from 0-days to 1-days). In the signature matching paradigm, a signature—typically byte sequences—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% [67] (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 [37], and up to 7% overhead when running benchmarks [7] (some AVs are more affected than others [6]). 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 [8]. 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 to scan the embedded payload. To mitigate performance degradation, some AVs turn off real-time checking options [64], 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 [55].

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 [72] can also serve as malware signatures. While previous approaches have already proposed the mitigation of AV overhead with hardware checks [58, 27], 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 [48] 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 2, we provide the background information required for understanding AV operation and branch prediction; in Section 3, we present the threat model and assumptions for HEAVEN, and provide the details of its design and implementation; in Section 4, we show the evaluation methodology for HEAVEN and the obtained results; in Section 5, we discuss the applicability of HEAVEN in actual scenarios, and its limitations; in Section 6, we present the related work and, in Section 7, our concluding remarks.

2 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.

2.1 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 to 1-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 [5] as malicious. This is accomplished with the use of whitelists composed of benign applications [45, 26, 9]. 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 [36]), as well as the time required to filter out candidate signatures against benign binaries (e.g., more than 30 minutes for Ikarus [5]).

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.

2.2 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 [34], which represents the last 16 branches).

In Figure 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.

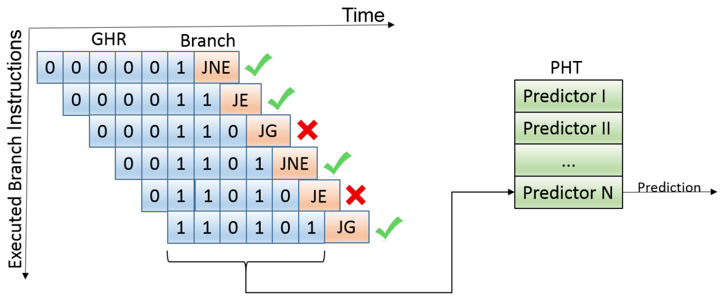


Figure 1: **Two-level branch predictor.** A sequence window of taken (1) and not-taken (0) branches is stored in the Global History Register (GHR).

The 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).

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.

3 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.

3.1 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.

3.2 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 4).

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 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 2a) 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 2b); 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 2c).

3.3 HEAVEN's Design

HEAVEN's architecture is composed by three main components (Figure 3): the Signature Matcher; the HEAVEN Manager; and the Threat Intelligence Manager (TIM).

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) [40]—overflow after reaching their storage thresholds.

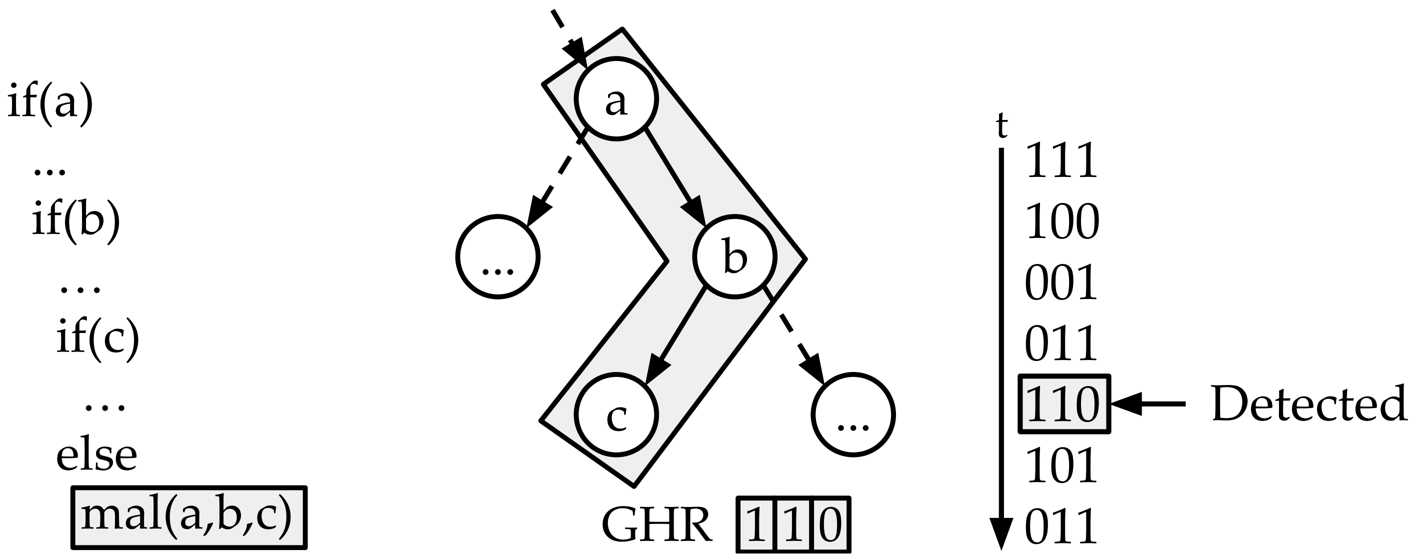
As the signature matching is performed via branch patterns, stored at the architectural level for usage by the Branch Prediction Unit (BPU), HEAVEN does not need to introduce any data collection mechanism. Unlike previous hardware malware detection solutions requiring extensive hardware modifications, 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.

3.4 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 [48], 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 [3]. 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



(a) **Code.** (b) **Flow.** (c) **Signature**

Figure 2: **Signature Generation Policy.** Associating high-level code constructs with their occurrence in the execution flow.

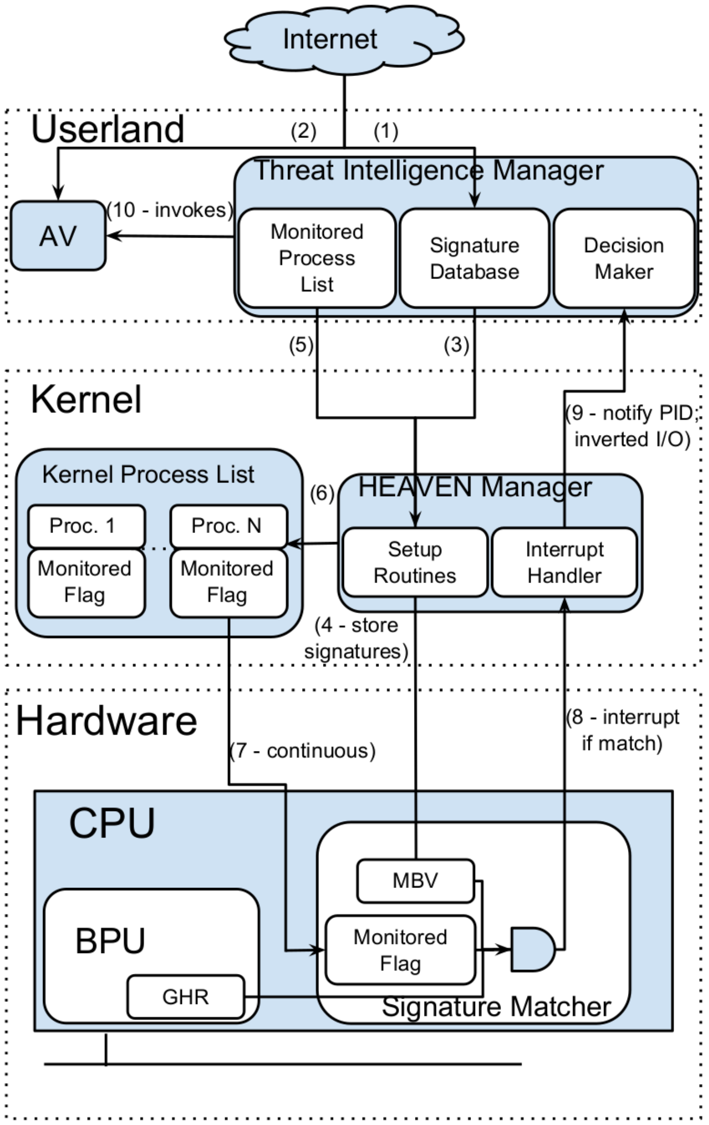


Figure 3: HEAVEN Architecture's design: modules in userland, kernel and hardware levels.

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 [52].

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 [4]. 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 [14]: 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 [51].

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 [21, 46, 44, 70, 61]. 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 [61]). 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 [65], 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 A). 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 2). In this case, even though a malware and a goodware application share a common branch pattern (blue one in Figure 4), the malware sample can still be fingerprinted by another pattern unique to it (red one in Figure 4).

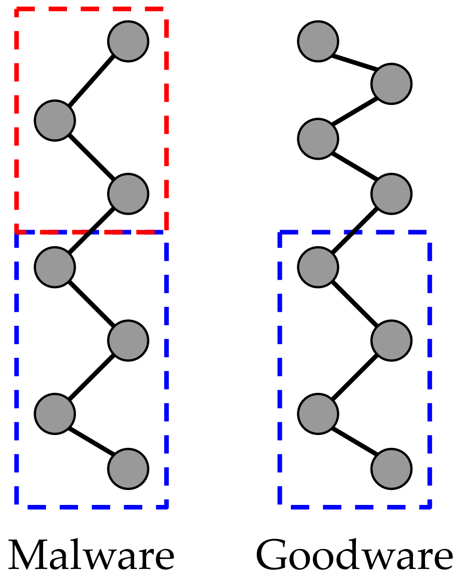


Figure 4: **Malware-Goodware Disambiguation.** Shared patterns are ignored and unique patterns are selected to fingerprint samples.

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 5), we can still generate signatures by looking to its common patterns (red ones in Figure 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.

4 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

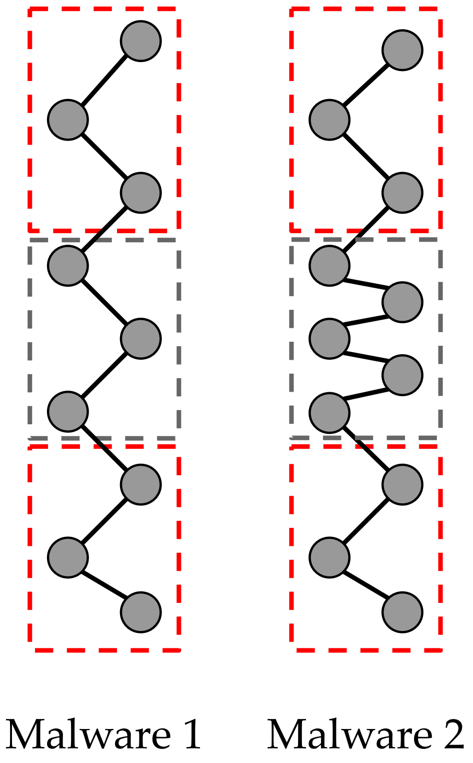


Figure 5: **Probabilistic Malware Execution.** The best signatures are the ones that are common to all sample's executions.

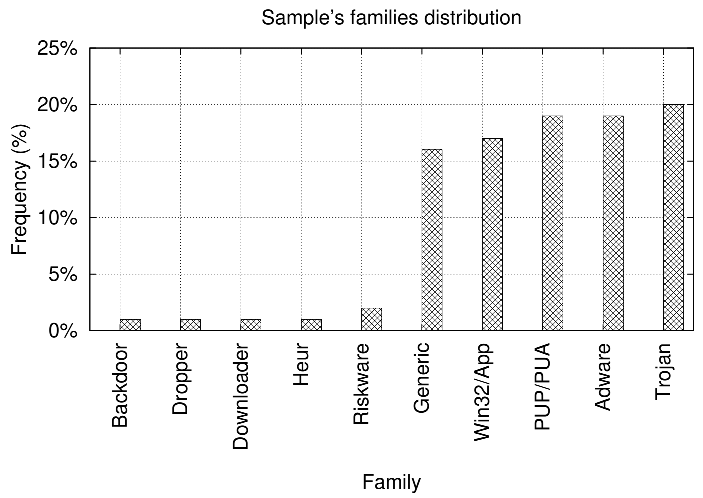


Figure 6: **Sample's families distribution.** Malware dataset balanced according VirusTotal statistics and labeled with AVClass.

AV companies were able to generate perfect signatures and no False Positive (FP) is observed. Further, we discuss HEAVEN operation in scenarios with FPs.

4.1 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 [66]. 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.

Our set of 1,000 benign software was collected from three distinct sources: (i) applications from the SPEC-CPU 2006 benchmark [38]; (ii) Internet browsers (Internet Explorer, Firefox and Chrome); and (iii) popular applications crawled from popular Internet repositories [25, 63]. 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 [12]. We evaluate the browser during its loading and while opening URLs from the Alexa top50 [1] most accessed websites in May/2018 in a loop.

We ran all malware samples for three minutes in a sandboxed environment (which usually suffices [47]). We recorded the branch patterns generated from the execution of all goodware and malware samples considering all runs and all distinct inputs per sample.

4.2 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 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

[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

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.

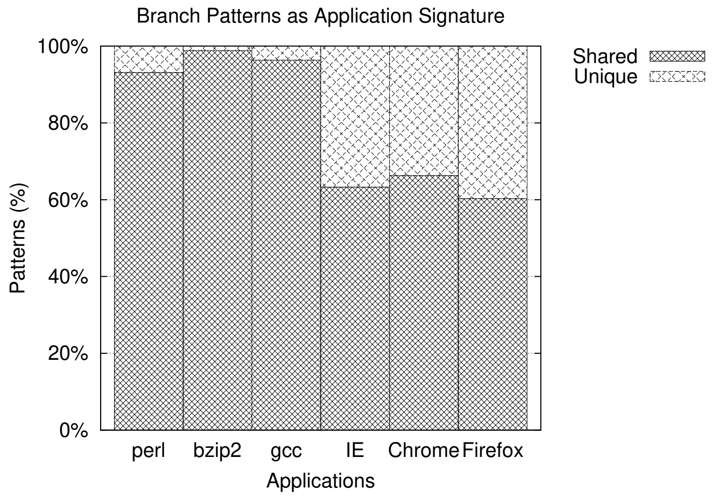


Figure 7: **Branch patterns as signatures.** All applications presented at least one unique branch pattern.

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 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 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.

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,

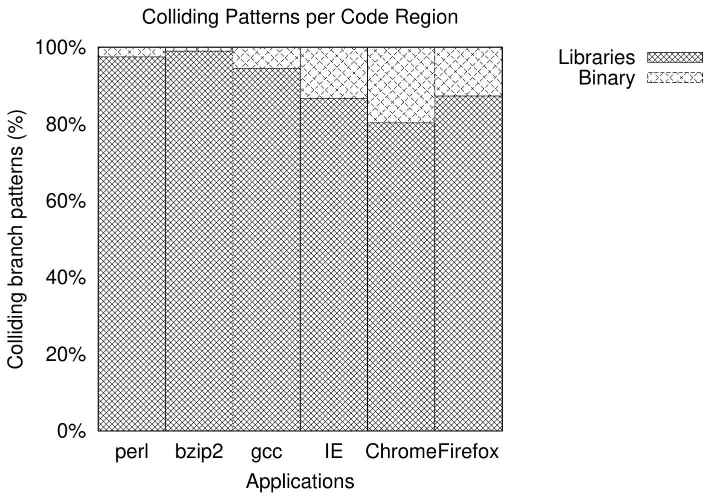


Figure 8: **Colliding branch patterns per code region.** Collisions on branch pattern originated on libraries are more prevalent than collisions on branch patterns originated on the application `.text` section.

the 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 [60, 30, 62]. 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 9).

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

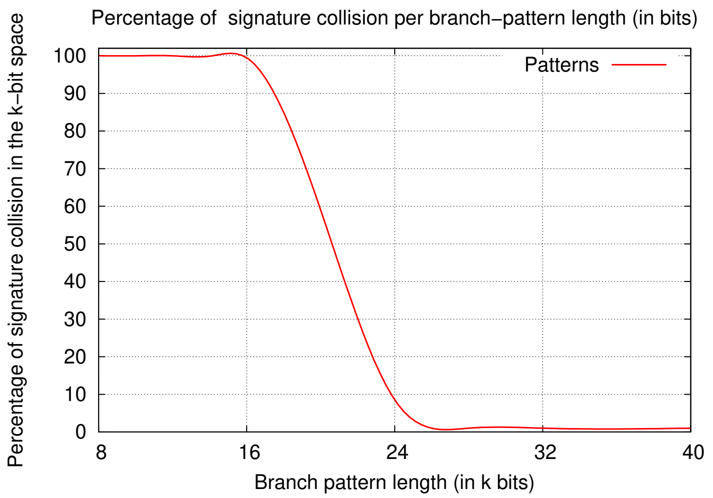


Figure 9: **Branch patterns coverage.** Signatures spanning less than 16 bits are not ideal because of the high collision rate. With 24-bit signatures, less than 10% of the branch patterns collide.

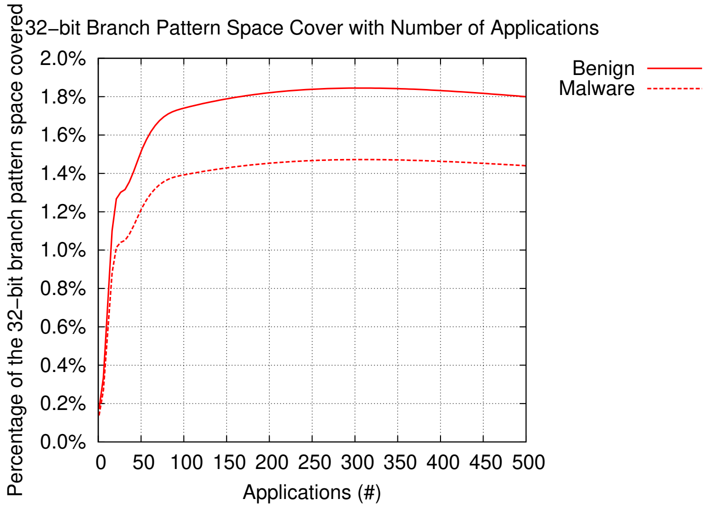


Figure 10: **Pattern Coverage.** Unique patterns are identified for all samples but coverage saturates after approximately 100 samples. Omitting data for the remaining samples due to the lack of variation.

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.

4.3 Malware Detection

We generated signatures for all samples and evaluated the bit space coverage as a function of the number of applications traced (Figure 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.

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 [73]. Although these conclusions might not hold for all types of malware, such as specially-crafted, modern malware samples [19], 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 [39], 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 5). 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 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 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 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 [59], which allowed the association of function calls to instruction pointers and, consequently, to the retrieved branches (Table 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 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 [35], 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%) [35]. 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 [68], thus presenting a more realistic scenario of applications distribution.

We evaluated HEAVEN with two AVs: (i) Clamwin [24], a Windows version for the open-source ClamAV with memory scan and real-time [23] support, and (ii) the most downloaded free AV in the Softonic's list [63]. Table 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 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.

4.4 Performance

To evaluate the performance overhead imposed by the multiple checks performed by standard AVs, we leveraged the Novabench benchmark [56] 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 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.

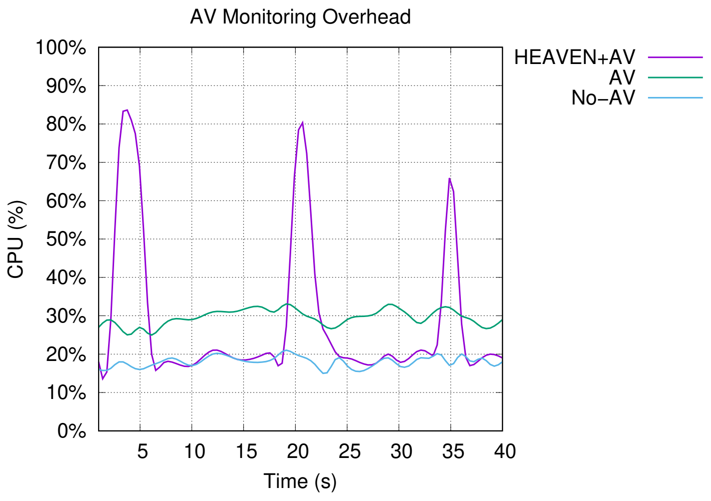


Figure 11: **HEAVEN CPU performance overhead for `monitoring` and `inspection` phases.** The inspection phase causes occasional, and quick bursts of CPU usage. The AV operating alone incurs a continuous 10% performance overhead.

Figure 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 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.

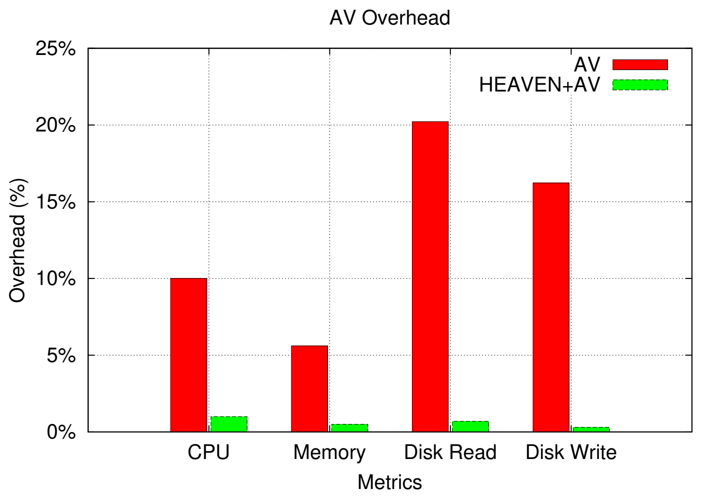


Figure 12: **HEAVEN performance overhead improvements compared to the AV alone.** All numbers are normalized for a system operating with no AV.

RTAV is a kernel driver implementing a file-system filter [50] and registry [49] and process [53] callbacks (These are the data collection mechanisms recommended by Microsoft [54] for AVs development after the adoption of the Kernel Patching Protection mechanism in modern Windows kernels [17]). 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 [71] 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 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 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%.

4.5 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 that uniquely identify a known software execution and that 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.

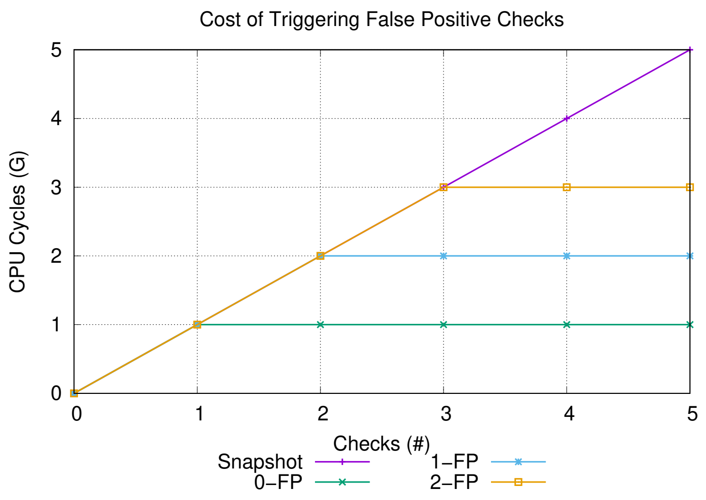


Figure 13: **The impact of FPs on HEAVEN performance.** The more FPs, the more HEAVEN approximates from a snapshot-based solution.

Figure 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.

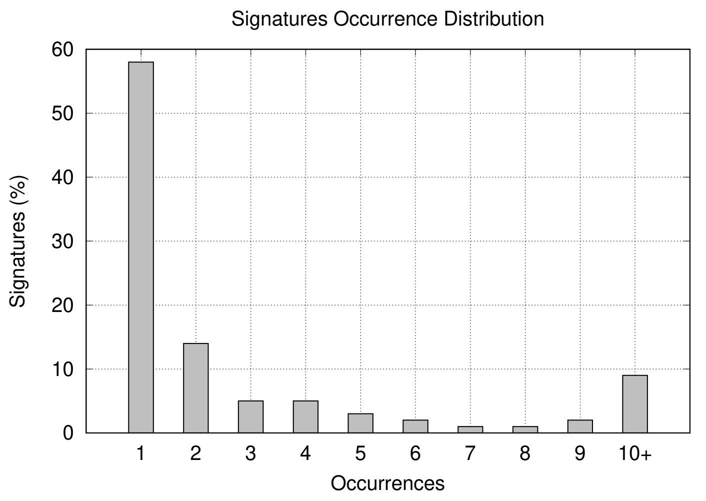


Figure 14: **Average FP impact.** Most branch patterns are unique or repeats few times, limiting the impact of FPs.

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 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 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 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.

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 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

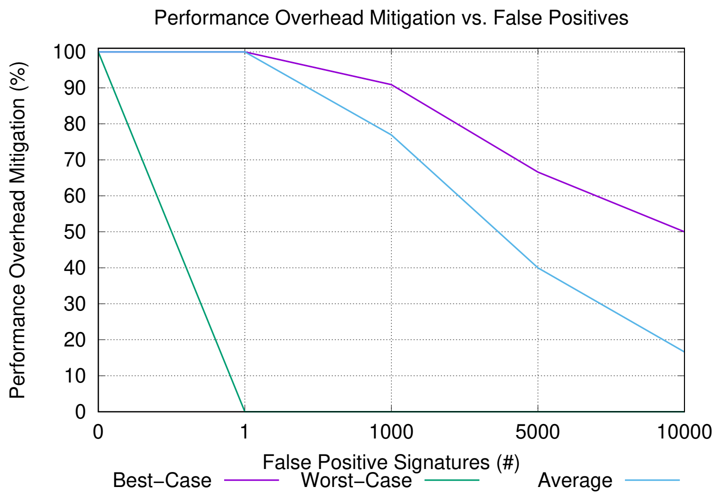


Figure 15: **Multiple FPs scenario.** Most part of the performance degradation comes from the repetition of the colliding patterns rather than from the number of distinct colliding patterns.

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.

5 Discussion

In this section, we revisit our proposal to discuss our contributions and still existing limitations.

HEAVEN Advances. 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) [40], 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 3.2. 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 [77, 28].

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 [42]. 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 [22, 33, 32].

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 [18]. 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 Related Work

John Aycock stated in his book [10] 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. [4] 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. [74] 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 [31]. These approaches present multiple drawbacks [28], 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 [13]; 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 [76]. 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 [57] 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. [11], 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. [29] 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. [58] and the approach proposed by Das et al. [27], 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 [75], 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 [20], 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 [16] that generates interrupts when violations of a given security policy are identified (in HEAVEN's case, malware execution detection).

7 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).

References

[1] Alexa. Alexa top 500 global sites. `https://www.alexa.com/topsites`, 2018.

[2] Andikleen. Simple-pt. `https://github.com/andikleen/simple-pt`, 2018.

[3] I. Arghire. Windows 7 most hit by wannacry ransomware. `http://www.securityweek.com/windows-7-most-hit-wannacry-ransomware`, 2017.

[4] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. Secure embedded processing through hardware-assisted run-time monitoring. In *Design, Automation and Test in Europe*, pages 178–183 Vol. 1, GER, March 2005. ACM.

[5] K. Ask. Automatic malware signature generation. `http://www.gecode.org/~schulte/teaching/theses/ICT-ECS-2006-122.pdf`, 2006.

[6] AV-Comparatives. Business security test. `https://www.av-comparatives.org/tests/business-security-test-2020-august-november/`, 2020.

[7] av comparatives.org. Impact of security software on system performance. `https://www.av-comparatives.org/wp-content/uploads/2017/10/avc_per_201710_en.pdf`, 2017.

[8] AV-Test. Endurance test: Does antivirus software slow down pcs? `https://www.av-test.org/en/news/news-single-view/endurance-test-does-antivirus-software-slow-down-pcs/`, 2015.

[9] Avast. Avast threat lab - file whitelisting. `https://support.avast.com/en-ww/article/Threat-Lab-file-whitelist`, 2018.

[10] J. Aycock. *Computer Viruses and Malware*. Springer, 2006.

[11] M. B. Bahador, M. Abadi, and A. Tajoddin. Hlmd: a signature-based approach to hardware-level behavioral malware detection and classification. *The Journal of Supercomputing*, 75(8):5551–5582, Aug 2019.

[12] M. Botacin, G. Bertão, P. de Geus, A. Grégio, C. Kruegel, and G. Vigna. On the security of application installers and online software repositories. In C. Maurice, L. Bilge, G. Stringhini, and N. Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 192–214, Cham, 2020. Springer International Publishing.

[13] M. Botacin, L. Galante, F. Ceschin, L. C. P. C. Santos, P. L. de Geus, A. Gregio, and M. Zanata. The av says: Your hardware definitions were updated! In *14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2019)*, page 1. IEEE, 2019.

[14] M. Botacin, P. L. D. Geus, and A. Grégio. Enhancing branch monitoring for security purposes: From control flow integrity to malware analysis and debugging. *ACM Transactions on Privacy and Security*, 21(1):4:1–4:30, Jan. 2018.

[15] M. Botacin, P. L. D. Geus, and A. grégio. 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):69:1–69:34, July 2018.

[16] M. Botacin, M. Zanata, and A. Grégio. The self modifying code (smc)-aware processor (sap): a security look on architectural impact and support. *Journal of Computer Virology and Hacking Techniques*, 16(3):185–196, Sep 2020.

[17] M. F. Botacin, P. L. de Geus, and A. R. A. Grégio. The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques*, 14(1):87–98, Feb 2018.

[18] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. *SIGOPS Oper. Syst. Rev.*, 28(5):242–251, Nov. 1994.

[19] A. Calleja, J. Tapiador, and J. Caballero. A look into 30 years of malware development from a software metrics perspective, 09 2016.

[20] N. A. Carreon, S. Lu, and R. Lysecky. Hardware-based probabilistic threat detection and estimation for embedded systems. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 522–529, USA, 2018. IEEE.

[21] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen. 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, 2010. USENIX Association.

[22] Clamav. Clamav. `https://www.clamav.net/downloads\#collapseCVD`, 2018.

[23] ClamSentinel. Clamsentinel. `https://sourceforge.net/projects/clamsentinel/`, 2018.

[24] ClamWin. Free antivirus for windows. `http://www.clamwin.com/`, 2018.

[25] cnet. Cnet: Product reviews, how-tos, deals and the latest tech news. `cnet.com`, 2018.

[26] Comodo. Antivirus whitelist. `https://securebox.comodo.com/antivirus-whitelist/`, 2018.

[27] S. Das, Y. Liu, W. Zhang, and M. Chandramohan. Semantics-based online malware detection: Towards efficient real-time protection against malware. *IEEE Transactions on Information Forensics and Security*, 11(2):289–302, Feb 2016.

[28] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose. 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, 2019.

[29] S. Das, H. Xiao, Y. Liu, and W. Zhang. Online malware defense using attack behavior model. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1322–1325, CA, May 2016. IEEE.

[30] O. E. David and N. S. Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classification. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, Ireland, July 2015. INNS.

[31] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 559–570, New York, NY, USA, 2013. ACM.

[32] EMSISOFT. 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/`, 2015.

[33] ESET. Types of updates. `http://support.eset.com/kb309/?viewlocale=en_US`, 2018.

[34] A. Fog. The microarchitecture of intel, amd and via cpus. `http://www.cs.utexas.edu/~hunt/class/2018-spring/cs340d/documents/Agner-Fog/microarchitecture.pdf`, 2018.

[35] A. Grégio, V. Afonso, D. Simões Fernandes Filho, P. De Geus, and M. Jino. Toward a taxonomy of malware behaviors. *The Computer Journal*, 58:2758–2777, 07 2015.

[36] K. Griffin, S. Schneider, X. Hu, and T.-c. Chiueh. *Automatic Generation of String Signatures for Malware Detection*, pages 101–120. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[37] T. hardware. Do antivirus suites impact your pc's performance? `http://www.tomshardware.co.uk/antivirus-performance-benchmark,review-32294.html`, 2011.

[38] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[39] Intel. *Intel(R) Advanced Vector Extensions Programming Reference*. Intel, 2011.

[40] Intel. Manual. `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-prog` `pdf`, 2016.

[41] Intel. Intelpt. `https://github.com/intelpt/WindowsIntelPT`, 2018.

[42] Intel. Technologies for hardware assisted native malware detection. `https://patentimages.storage.googleapis.com/fb/23/ff/9d11b27884f050/US10540498.pdf`, 2020.

[43] R. Intel. Architecture instruction set extensions programming reference. *Intel, March*, 1(1), 2014.

[44] B. Kang, H. Kim, T. Kim, H. Kwon, and E. Im. Fast malware classification using counting bloom filter. *Information (Japan)*, 15(7):2879–2892, July 2012.

[45] Kaspersky. Whitelist program. `https://usa.kaspersky.com/partners/whitelist-program`, 2018.

[46] J. Koret and E. Bachaalany. *The Antivirus Hacker's Handbook*. Wiley Publishing, US, 1st edition, 2015.

[47] A. Küchler, A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti. Does every second count?time-based evolution of malware behavior in sandboxes. `http://s3.eurecom.fr/docs/ndss21_kuechler.pdf`, 2021.

[48] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 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, 2005. ACM.

[49] Microsoft. Cmregistercallbackex function. `https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdm/nf-wdm-cmregistercallbackex`, 2018.

[50] Microsoft. Fsrtlregisterfilesystemfiltercallbacks function. `https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntifs/nf-ntifs-fsrtlregisterfilesystemfiltercallbacks`, 2018.

[51] Microsoft. Getcurrentprocessid function. `https://msdn.microsoft.com/pt-br/library/windows/desktop/ms683180(v=vs.85).aspx`, 2018.

[52] Microsoft. Peb structure. `https://msdn.microsoft.com/pt-br/library/windows/desktop/aa813706(v=vs.85).aspx`, 2018.

[53] Microsoft. Pssetcreateprocessnotifyroutine function. `https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-pssetcreateprocessnotifyroutine`, 2018.

[54] Microsoft. Avscan file system minifilter driver. `https://docs.microsoft.com/en-us/samples/microsoft/windows-driver-samples/avscan-file-system-minifilter-driver/`, 2019.

[55] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: Toward snoop-based kernel integrity monitor. In *Proc. 2012 ACM Conf. on Comp. and Comm. Sec.*, CCS '12, US, 2012. ACM.

[56] Novabench. Free benchmark. `https://novabench.com/`, 2018.

[57] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev. Malware-aware processors: A framework for efficient online malware detection. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 651–661, US, Feb 2015. IEEE.

[58] M. Rahmatian, H. Kooti, I. G. Harris, and E. Bozorgzadeh. Hardware-assisted detection of malicious software in embedded systems. *IEEE Embedded Systems Letters*, 4(4):94–97, Dec 2012.

[59] C. Sandbox. Cuckoo sandbox: Automated malware analysis. `https://cuckoosandbox.org/`, 2018.

[60] V. S. Sathyanarayan, P. Kohli, and B. Bruhadeshwar. *Signature Generation and Detection of Malware Families*, pages 336–349. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[61] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable hardware memory disambiguation for high ilp processors. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 399–410, US, Dec 2003. ACM.

[62] A. Shabtai, E. Menahem, and Y. Elovici. 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, July 2011.

[63] Softonic. Softonic: App news and reviews, best software downloads and discovery. `softonic.com`, 2018.

[64] Sophos. Default anti-virus scanning options for sophos central. `https://community.sophos.com/kb/en-us/119637`, 2016.

[65] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys Tutorials*, 14(1):131–155, First 2012.

[66] X. Ugarte-Pedrero, M. Graziano, and D. Balzarotti. A close look at a daily dataset of malware samples. *ACM Trans. Priv. Secur.*, 22(1):6:1–6:30, Jan. 2019.

[67] D. Uluski, M. Moffie, and D. Kaeli. Characterizing antivirus workload execution. *SIGARCH Comput. Archit. News*, 33(1):90–98, Mar. 2005.

[68] UPX. Upx: the ultimate packer for executables. `https://upx.github.io/`, 2018.

[69] C. Wressnegger, K. Freeman, F. Yamaguchi, and K. Rieck. 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, 2017. Association for Computing Machinery.

[70] A. Yakunis. Nice bloom filter application. `http://blog.alexyakunin.com/2010/03/nice-bloom-filter-application.html`, 2010.

[71] Yara. Yara - the pattern matching swiss knife for malware researchers. `https://virustotal.github.io/yara/`, 2018.

[72] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *[1992] Proceedings the 19th Annual International Symposium on Computer Architecture*, pages 124–134, AUS, 1992. ACM.

[73] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 297–300, JP, Nov 2010. IEEE.

[74] T. Zhang, X. Zhuang, S. Pande, and W. Lee. Hardware supported anomaly detection: down to the control flow level. `https://smartech.gatech.edu/handle/1853/96`, 2004.

[75] T. Zhang, X. Zhuang, S. Pande, and W. Lee. 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, 2005. ACM.

[76] Z. Zhang, S. Park, and S. Mahlke. 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, 2020. Association for Computing Machinery.

[77] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi. 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, 2018. Association for Computing Machinery.

A 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 [15], 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 [43] as a basis for sandbox development and branch pattern extraction. The PT feature is present on Intel's 6^th 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 [41] and a Intel PT decoder library [2]. In Code Snippet 1, we show that our PoC is able to detect a branch signature after a sequence of `tnt.8` packets:

1. Line 1 shows the first captured `tnt.8` packet;
2. 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;
3. 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;
4. Line 8 shows the detection signal raised when the produced pattern matches a stored signature.

```
1 1st tnt.8: 001110
2 2nd tnt.8: 111111
3 3rd tnt.8: 011111
4 4th tnt.8: 110011
5 5th tnt.8: 111011
6 6th tnt.8: 11----
7 --
8 Obtained signature:
      |11|111011|110011|011111|111111|001110|
```

Code 1: Obtaining signatures using using `tnt.8` packets from Processor Trace.