No Need to Teach New Tricks to Old Malware: Winning an Evasion Challenge with XOR-based Adversarial Samples

Fabrício Ceschin
fjoceschin@inf.ufpr.br
Federal University of Paraná
Curitiba, Paraná, Brazil

Marcus Botacin
mfbotacin@inf.ufpr.br
Federal University of Paraná
Curitiba, Paraná, Brazil

Gabriel Lüders
gl19@inf.ufpr.br
Federal University of Paraná
Curitiba, Paraná, Brazil

Heitor Murilo Gomes
heitor.gomes@waikato.ac.nz
University of Waikato
Hamilton, Waikato, New Zealand

Luiz S. Oliveira
lesoliveira@inf.ufpr.br
Federal University of Paraná
Curitiba, Paraná, Brazil

André Grégio
gregio@inf.ufpr.br
Federal University of Paraná
Curitiba, Paraná, Brazil

ABSTRACT

Adversarial attacks to Machine Learning (ML) models became such a concern that tech companies (Microsoft and CUJO AI's Vulnerability Research Lab) decided to launch contests to better understand their impact on practice. During the contest's first edition (2019), participating teams were challenged to bypass three ML models in a white box manner. Our team bypassed all the three of them and reported interesting insights about models' weaknesses. In the second edition (2020), the challenge evolved to an attack-and-defense model: the teams should either propose defensive models and attack other teams' models in a black box manner. Despite the difficulty increase, our team was able to bypass all models again. In this paper, we describe our insights for this year's contest regarding on attacking models, as well defending them from adversarial attacks. In particular, we show how frequency-based models (e.g., TF-IDF) are vulnerable to the addition of dead function imports, and how models based on raw bytes are vulnerable to payload-embedding obfuscation (e.g., XOR and base64 encoding).

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

malware detection, adversarial malware, machine learning

ACM Reference Format:
Fabrício Ceschin, Marcus Botacin, Gabriel Lüders, Heitor Murilo Gomes, Luiz S. Oliveira, and André Grégio. 2020. No Need to Teach New Tricks to Old Malware: Winning an Evasion Challenge with XOR-based Adversarial Samples. In *Reversing and Offensive-oriented Trends Symposium (ROOTS'20), November 19–20, 2020, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3433667.3433669

1 INTRODUCTION

Malicious programs have been a major security concern during the last four decades, with a plethora of proposed solutions over time to counter their threat. More recently, Machine Learning (ML) approaches have been widely applied to malware detection and classification. Although ML has significant advantages over other approaches, such as signature-based ones, it also has significant limitations (e.g., ML models are vulnerable to adversarial attacks).

In the malware context, adversarial attacks consist of modifying samples so as to disturb the classifier to the point of a malware sample being classified as a legitimate, non-malicious software. The field of adversarial attacks has been growing, both academically and industrially, since those attacks are increasingly common in practice.

Adversarial attacks aiming at ML models became so popular that tech companies decided to launch a contest to better understand their actual impact: The Machine Learning Security Evasion Competition (MLSEC). In this contest, the organizers provide working malware binaries to participants, in addition to classifiers able to detect all malware samples given. The participants are then challenged to transform the binaries of the provided samples in a way that those new malware bypasses the classifiers, while maintaining their same previous/original behavior when executed in a sandbox.

In the first contest edition (2019), the teams were challenged to bypass three ML models in a white box manner. Our team bypassed all models and reported interesting insights about models' weaknesses. In the second edition (2020), the challenge evolved to an attack-and-defense model, with teams proposing defensive models, as well as attacking the models produced by other teams in a black box manner. Although the use of a black box approach certainly increased the challenge difficulty, our team was still able to bypass all models and, consequently, the first team to achieve maximum scoring in the context.

In this paper, we describe the experience gathered in the 2020's MLSEC contest, and the insights gained on both attacking ML models and defending them from adversarial attacks. On the one hand, our experience in the development of defensive models showed that the function distribution in 32-bit and 64-bit Windows libraries, and between their Debug and Release compilations are different, which affects detection. On the other hand, our experience in attacking models showed that embedding the malware payload into another binary eliminates most detection capabilities presented by the models.

In this year's contest, we were challenged to bypass three models. We discovered that: (i) the first model operates by looking into Portable Executable (PE) header features, thus being evaded by embedding malicious payloads in a dropper executable; (ii) the second model classifies function imports and libraries using a TF-IDF method, being evaded by the addition of fake imports to the dropper; (iii) the last model also checks for strings in the embedded content, being evaded by the encoding of the payload using XORing or base64 techniques.

One of the main contributions of this paper is to show that adversarial attacks are more practical in real life models than previously thought. Whereas there are approaches for adversarial attacks generation based on complex techniques, some even leveraging the same ML techniques used to defend against attacks (e.g., reinforcement learning [19]), we show that it is possible to generate attacks using known, simple techniques, such as XORing strings and encoding binaries in base64. Unlike automated attack generators, our approach is fully explainable and our insights can be used as feedback for the development of more robust models.

We highlight the impact of these techniques in practice by demonstrating that the detection rate of antiviruses (AV) from Virus Total decreased when the evasive samples were submitted to scanning (in comparison to the original malware), even though no specific AV was targeted in the competition. Since adversarial attacks pose a significant problem, we decided to make our attack and defense solutions available to the community, so anyone can train with it and design new security solutions. More specifically, we are releasing the source code of a dropper, and a Web interface in which users are able to generate adversarial malware using this dropper and test the transformed samples against multiple classifiers, including the those present in the previous challenge editions.

In summary, our contributions are as follows:

- We describe the experience in an ML-based malware detection evasion challenge.
- We describe our defensive ML model and discuss considerations to be made when developing a detection model.
- We present the attack techniques we leveraged in the contest to bypass all ML models.
- We discuss the impact of adversarial malware in practice via the detection rate of the evasive samples when inspected by real AVs.
- We release code and a platform for the development of experiments with adversarial malware.

This paper is organized as follows: in Section 2, we present the contest and our accomplished achievements there; in section 3, we discuss the reasons why defending from adversarial attacks may be hard; in Section 4, we discuss our findings and how they provide insights and feedback for future security solutions; in Section 5f, we present the related work; finally, we draw our conclusions in Section 6.

2 THE CHALLENGE

We start this section presenting the contest rules, and the provided samples and models. Then, we detail how we managed to implement a defensive model. Finally, we show how we attack the models.

2.1 Definitions

In 2019, tech companies (Elastic, Endgame, MRG-Effitas, and VM-Ray) launched a contest challenging participants to bypass ML-based malware detectors with adversarial samples [2]. In this contest, the organizers provided participants with working malware binaries and malware classifiers (white box model) that initially detected all these malware samples. The participants were challenged to provide new binaries for the same malware samples, and those new malware should be able to bypass the classifiers, whereas still presenting the same original behavior when executed in a sandbox. Our team joined this challenge and we were able to generate adversarial samples that bypassed all the three models. Our findings were reported in a previous paper [14].

In 2020, other tech companies (Microsoft and CUJO AI's Vulnerability Research Lab) joined the mission of exercising adversarial attacks in practice. That second edition of the Machine Learning Security Evasion Competition (MLSEC) [5] was an incremental version of its previous edition. In this year's competition, the organizers added one extra step: the generation of defensive solutions to be further attacked by the participants.

In the "defender's challenge", the participants were free to develop their own machine learning defensive solutions, with models of their own choice and trained using any dataset. The entire defensive solution should be saved by the participants in a docker image, which was then tested using an unknown dataset with three requirements: (i) the model should accomplish less than 1% of False Positive Rate (FPR), (ii) less than 10% of False Negative Rate (FNR), and (iii) it must return a response within 5 seconds for any presented sample. In total, three models met the proposed requirements: `ember` (provided by the organizers), `needforspeed` (our model), and `domumpqb` (provided by another team that published their solution after the final results [31]). We do not know how many teams participated in this step of the challenge, apart from the fact that only those aforementioned models have met the requirements. Furthermore, the results of this part of the challenge would depend on how the attackers perform against each model, i.e., the one that performs better against adversarial attacks wins.

In the "attacker's challenge", all models that achieved the previous requirements were made available to be attacked by further black box attacks. Thus, the attackers would have access only to the output produced by those models, without directly accessing them [22]. Black box attacks were conducted with 50 unique Windows malware samples provided by the contest organizers, which should be modified in order to bypass their detection in the defense solutions. Each new sample produced by the participants should have its behavior identical to the original malware sample from which it was based on. Behavioral validation is accomplished based on running the modified sample in a sandbox made available in the Web site of the contest, which should result in the same Indicators of Compromise (IoCs) of the original ones. Thus, each bypassed classifier for each binary accounts for 1 point, summing up to 150 points.

Moreover, as a tiebreaker rule in case of similar bypass scores, the competition also stores the number of ML queries (number of times that samples are tested) used by each participant. Therefore, the team that achieves the lowest number of queries wins.

We submitted the 50 samples provided by the organizers to the Virus Total API and then used AVClass [36] to normalize the resulting labels. In Figure 1a, we show the number of samples distributed in malware families, with two families with higher prevalence (`gamarue` and `remcos`) and 30 families in total. Notice that all malware available for this challenge consist of real samples and have already been seen in the wild, such as the family that steal crypto-currency from its victims [8]. The `gamarue` family can give a malicious hacker control of the PC, stealing sensitive information and changing security settings [27]. In addition, the `remcos` family embed a XML code that allows for any binary with parameters to be executed, in this case a REMCOS RAT, which gives the attackers full control over the infected PC, allowing them to run keyloggers and surveillance tools [26]. Thus, despite of being just a competition, all the malicious samples presented in this work may present real risks.

2.2 Defenders challenge

To develop our defense solution, we selected a model developed by our research team that achieved good metrics using textual features (TF-IDF) on top of static analysis and Random Forest classifier [15] as **baseline**. We considered this challenge as an opportunity to test our research model against adversarial attacks in practice.

Initially, we considered using our baseline model as it is originally presented in our paper [15], with malware samples collected in the Brazilian cyberspace as the training set. Our choice is usually enough to reach good result metrics (almost 98% of f1-score with a low false-negative rate). When we tested our baseline model against the samples provided by the organization in the 2019 edition of the competition, the results were surprisingly bad and totally different from what we expected them to be. We hypothesized that those results might be biased due to the distinct characteristics of the samples considered by the organizers in contrast to ours. More specifically, since the threat landscape in Brazil is very different from the rest of the world [11], we hypothesized that the classifiers we used in Brazilian cyberspace were not the most suitable ones for classifying the likely-global samples provided by the organization.

To validate this hypothesis, we decided to retrain the classifier using the `ember` dataset as input, similar to the organizers' approach to train their LightGBM model [3]. When we compared the results of training our model by testing it with the Brazilian and the Ember samples, we noticed that the latter indeed generalize better. In Figure 2, we show the False-Negative Rate (FNR) of the two versions of our model (trained with Brazilian and Ember samples, respectively) after evaluation against a subset of these same datasets. We noticed that each classifier works marginally well in their own region, in a way that they could detect malware without too many false-positives, i.e., the model trained with global samples presents high detection rate, but performs extremely bad with Brazilian samples, and vice-versa.

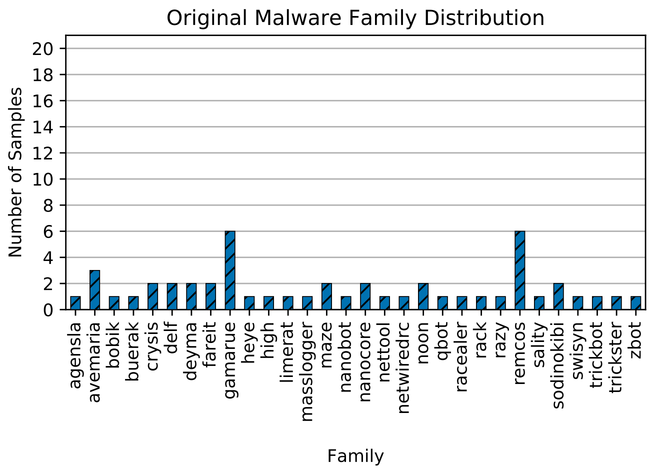
Consequently, if these classifiers were applied in actual scenarios as a generalization of global threats, they would let many threats originated from other regions than the ones they were trained to be executed without raising warnings. Thus, we conclude that ML models must be specially crafted for each region in which they are going to operate, given that they may detect more samples and be more effective. For the remainder of this paper, we refer to our model as the one trained with EMBER, since that was the more suitable dataset for the task at hand. It is worth to mention that the competition organizers also made all the adversarial samples from last year challenge available, but we decided to not use them in the training step, so we could use them to verify our model's robustness in the testing step.

To create our definitive model for the competition, we selected the attributes from the EMBER datasets [3] (both 2017 and 2018 version) we believed to be less prone to be affected by adversaries, according to our previous experience. We categorized the attributes in three types: numerical, which are integer or float numbers; categorical, which represents categories; and textual, which are a set of strings. To train our model, we used the EMBER's 1.6 million labeled samples as input to the scikit-learn Random Forest [34] with 100 estimators. Below, we list the attributes we selected for our model. The detailed description of all of them is available in the EMBER's dataset paper and source code [3].

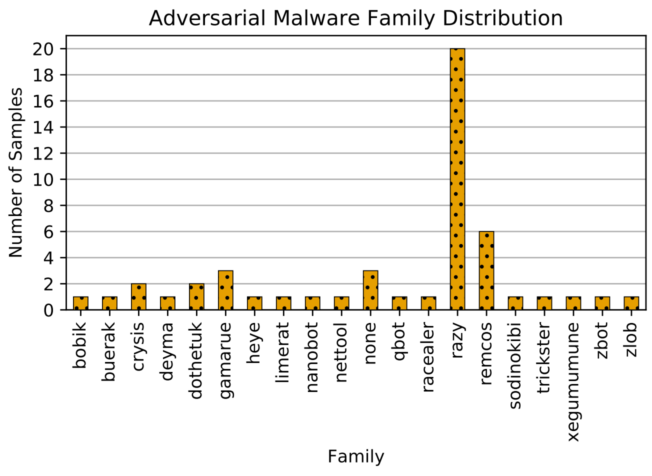
(1) Numerical
- string_paths
- string_urls
- string_registry
- string_MZ
- virtual_size
- has_debug
- imports
- exports
- has_relocations
- has_resources
- has_signature
- has_tls
- symbols
- timestamp
- numberof_sections
- major_image_version
- minor_image_version
- major_linker_version
- minor_linker_version
- major_operating_system_version
- minor_operating_system_version
- major_subsystem_version
- minor_subsystem_version
- sizeof_code
- sizeof_headers
- sizeof_heap_commit

(2) Categorical
- machine
- magic

(3) Textual
- libraries
- functions
- exports_list



(a) Original malware family distribution. In total, 30 different families were present in the samples provided this year, 9 more than last year (21 families).



(b) Adversarial malware family distribution. From the 30 original families, our samples were re-classified into only 19 ones, and 3 of them could not be classified at all.

Figure 1: Malware families distribution. Differences between original malware samples and adversarial ones are notable.

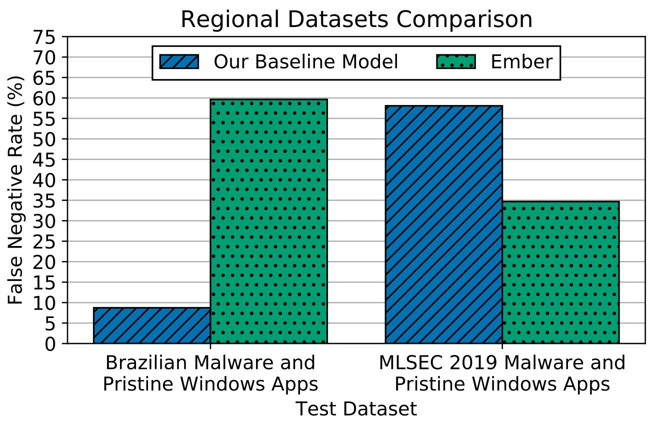


Figure 2: Regional datasets and models. Each model performs better in their own region, indicating that detectors must be specially crafted for a given region.

- dll_characteristics_list
- characteristics_list

The categorical attributes `machine` and `magic` were obtained from the PE header and transformed into one-hot encoding array [33] where each binary column represent a value (1 for present, 0 otherwise). The textual features, such as libraries and functions used by a software, were also obtained from the PE header and transformed into texts, separated by spaces. These texts were used as input to TF-IDF [35], transforming every text into a sparse array containing the TF-IDF values for the 300 top words in all the texts created (the ones with most frequency). The numerical features were all extracted from the PE header, except `string_path`, `string_urls`, `string_registry`, and `string_MZ`, which extracts the number of strings that contains a system paths, URLs, registries, and MZ headers, respectively. Finally, after we transformed all attributes into numerical features, we normalized them using MinMaxScaler [32] to use as input of our model. Given that EMBER dataset is available in CSV format (with all the attributes already extracted), we also created a module that extracts the very same attributes from raw PE binaries. This module is used to test attackers' samples, and it also could be used by any real-world testing solution.

"*The sad state of PE parsing*" (Part 1): The lack of a standard and complete library for PE parsing and manipulation is a long-term complaint [24] and this affected the development of both the defensive and the attacking solutions. In the first case, here reported, our original solution was deployed on top of `PEfile` [13], but it did not achieve the required performance by the challenge, resulting in classification timeout (some samples were taking more than 5 seconds to be parsed). We then ported our implementation to `lief` [30], since this was the tool used for binary parsing in the contest demonstration script. `Lief` is in fact much faster than `PEfile` (the same aforementioned samples were taking about 2 seconds when using `lief` as parser), but its parsing results are a bit different. Due to this fact, we had to slightly change our model to consider some features as categorical instead of numerical, since fields such as `machine` and `magic`, which are parsed as integer numbers by `PEfile`, are represented by strings (flags) in `lief` (it ended up not affecting the results of our classifier but it could).

To fine tune our model, we created a new prediction function that uses the model class probabilities as input. To do so, we defined a threshold T and used it to define the output class: if the probability of being a "goodware" is greater than T, the current sample will be classified as a goodware. Otherwise, it will be classified as a malware. It was required to make our classifier perform as required by the competition, achieving less than 0.1% of FPR with a threshold $T = 80\%$. This technique has proven to be much better than using the default Random Forest prediction function, which achieved a FPR of 8.5%.

Our model vs. last year adversaries: The initial test of our model consisted of submitting the adversarial samples provided by the organizers from last year's challenge to it, and then analyzing the resulting detection rate. In total, there a were 594 samples, all of them variations of the 50 original samples from last year's challenge. Our model was able to detect 88.91% of the samples. Considering that all 2019 models were bypassed by those samples, we have agreed that this was a significant good result. It also confirmed our findings from the previous challenge, i.e., that models based on parsing PE files are better than the ones that make use of raw data [14].

2.3 Attackers challenge

We started our attacks by trying to replicate the strategy leveraged in the previous year (2019) with this year's (2020) classifiers and samples. Thus, we first appended goodware strings and random bytes to the original samples. This strategy resulted in 44 points, with 36 samples bypassing `ember`, 8 bypassing `needforspeed`, and none of them bypassing `domumpqb`. These results show that this year's models were really stronger than the previous ones.

Then, we moved to the next strategy that was successful last year: embedding the original sample in a "Dropper", a new binary that embeds the original malware sample as a resource, writes it to a file in runtime, and launches it from there. In most cases, the samples were executed directly, as they were typical PE files. In one of the cases, the original malware sample was a DLL. Thus, we modified our Dropper from last year to inject this DLL into a host process. As this DLL did not export any function, we launched its main function, invoking it from its ordinal number (`rundll32 dll_name,#1`). This approach succeeded on fully bypassing the first model (`ember`), which was based on PE headers. However, while this step was the final one in the last year's challenge, now we just accomplished the first third of the challenge.

Despite bypassing the first detector, the dropper malware was not able to fully bypass the other detectors. We then focused our attention in bypassing our own model, since we could leverage previous knowledge on the attack. As our model is based on the library imports and their respective functions, we guessed that our model was detecting the dropper as malicious. This might be happening, for instance, due to the presence of a function such as `FindResource`, which is largely used by malware droppers (and also by a few benign applications). Our first thought was to hide the `FindResource` API calls from the classifier. To do so, we tried to compress our samples with `Telock` [45], `PELock` [29], and `Themida` [44] packers. Interestingly, reducing the number of imports only increased the confidence on the malware label, which reinforces our last year's claim that most classifiers learn packers as malicious feature regardless of the binary content. This phenomenon was also reported to happen for real AVs [1].

The remaining alternative to bypass this model was to search for some benign sample likely used to test the model that present the same imports. Interestingly, the Calculator (`calc.exe`) presents these characteristics, importing a series of functions, including `FindResource`, and was report as benign with 100% of confidence level by our classifier. Thus, our goal turned into building a new dropper binary mimicking the calculator.

```python
import lief
# Parse
gw = lief.parse(GOODWARE)
mw = lief.parse(MALWARE)
# Get Sections
gw_sections = [s for s in gw.sections]
mw_sections = [s for s in mw.sections]
# Add Missing Sections
sec_diff = len(gw_sections) - len(mw_sections)
for i in range(1,sec_diff):
  mw.add_section(gw_sections[i])
for lib in gw.imports:
  lib_name = lib.name
  # Add Missing Libs
  if lib_name not in mw.libraries:
    mw.add_library(lib_name)
    # Add Missing Functions for the Lib
  for func in lib.entries:
    func_name = func.name
    if func_name != '':
      if func_name not in [f.name for f in mw.
           imported_functions]:
        mw.add_import_function(lib_name,func.name)
# Build New Binary
builder = lief.PE.Builder(mw)
builder.build_imports(True)
builder.patch_imports(True)
builder.build()
builder.write(NEW_MALWARE)
```

Code 1: Lief script example. Automatic Section and Function Inclusion.

```c
void dead()
{
  ShellMessageBox(NULL,NULL,NULL,NULL,NULL);
  RegEnumKeyExW(NULL,NULL,NULL,NULL,NULL,NULL,NULL);
  BSTR_UserFree(NULL,NULL);
  CoInitialize(NULL);
  IsThemeActive();
  ...
}
```

Code 2: Dead code insertion. These functions play no role in the Dropper execution.

"*The sad state of PE parsing*" (Part 2): The lack of a standard library for PE manipulation also affected the development of the adversarial samples. `PEfile` has no native support for section inclusions and whereas it can be extended for this task [16], the whole process is manual and laborious. In turn, the `lief` solution has native methods for adding sections and even function imports. We implemented a code on top of it to perform this task, as shown in Code 1. Unfortunately, `lief` has some known issues [41] regarding these functions and it ended up breaking all malware binaries (although it worked with the simplest test cases we developed).

Since the existing solutions did not allow us to patch the compiled binaries, we opted to compile the Dropper with the extra function. However, as these functions do not play any role regarding the Dropper's operation, we added them as dead code into a never called function, as shown in Code 2. We compiled the code without optimization, since these functions are not eliminated. Therefore, their imports were available to the ML models, but they did not affect the Dropper execution. This strategy resulted in the complete bypass of our model.

It is important to notice that the correct working of this approach depends on the complete mimicry of all libraries and functions, which includes compiling the code for the same architecture and

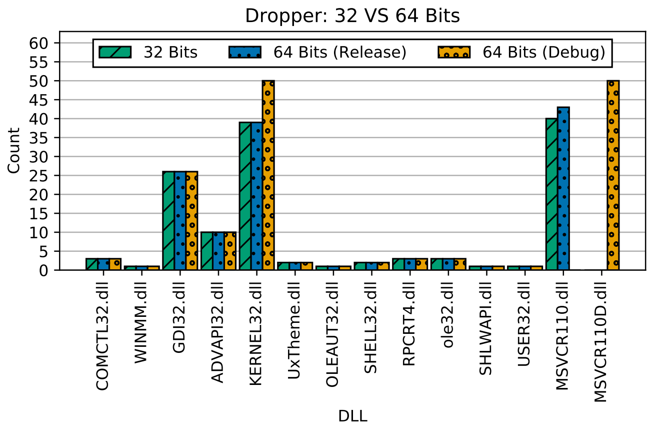


Figure 3: Number of functions in each library. Compiling libraries for 32-bit and 64-bit systems and in the Debug or in the Release mode affect their distribution.

linking it with the same libraries as the goodware binary. Otherwise, the sample could still be detected due to the combination of the extracted functions and libraries.

In Figure 3, we show the impact of multiple compilation settings on the import of the same functions. We observed that changing the compilation from 32 to 64 bits results in the addition of three additional imports to the `MSVCR110.dll`. Even worse, compiling the code in the Debug mode results in the linkage of a distinct library `MSVCR110D.dll`—the debug version—with a distinct number of function imports. Since we considered the Calculator binary as goodware, and it was compiled with Release configurations and for the 64-bit architecture, all dropper binaries were configured the same way. The evasion was only possible using this setting.

Black box is harder, but not impossible. So far, we have bypassed two of the three models. Although we performed the black box tests, we have some degree of previous knowledge about the models because they were either deployed in the last year's contest, or developed by us. To bypass the last model, we had to deploy a full black box attack.

At this point, a few samples had already bypassed the third model (21 samples). Since all droppers were similar by construction, we hypothesized that the third model was detecting some part of the embedded payload, as it was the only part allowed to change from one compilation to another. Therefore, we should hide the embedded payload to bypass this detector. We tested two approaches for this task: (i) encoding the malware binary as a base64 string; and (ii) XORing the malware binary with a key. In the former case, the resource must be decoded before being written in disk, whereas in the latter, the same XOR key can be used to decode the buffer before it is dumped to the file system.

These strategies are enough to bypass most of the verification performed by the model, as it hides the original strings and magic numbers. For instance, the `MZ` flag of the embedded payload is not present anymore in the resource section, thus not being identified by mechanisms that look for embedded files (as is the case of our own model). In addition, XORing strings often result in non-printable characters that are not handled by the `strings` utility. By combining these strategies, we were able to fully bypass the third model. Thus, despite requiring additional reasoning and implementation efforts, the black box bypass of ML detectors is completely viable.

Table 1: Average number of queries. We bypassed all models with an average rate lower than 5 queries per sample.

Team	Bypasses	Queries	Average
Ours	150	741	4.94
2nd	47	162	3.44
3rd	44	150	3,40
4th	1	78	78

It is worth to emphasize the robustness of this approach, despite its relative simplicity. We first hypothesized that the dropping approach could be detected by a model that considers section entropy values as a feature (e.g., histogram of section's entropy), because packing, compression, and embedding often result in entropy increase [47]. In practice, this effect was not observed. In Figure 4, we show the section maximum entropy values for the original samples and for the multiple dropper variations. We observe that most of the dropper's section maximum entropy values are equal to the ones of the original samples (in most cases, these values were already high). In some cases, noticeably when using base64, the values are even lower than the original samples. Therefore, the embedding of content in the droppers would not be detectable in an indistinguishable manner when compared to the high entropy of the original binaries.

In Table 1, we show the total and average number of queries performed until breaking all the models. On average, it took us less than 5 queries per sample to bypass the three models. We consider this number very low, even when considering that we had previous knowledge about some models, which can also be expected from a skilled, motivated attacker performing targeted attacks against real systems. Worse than that, if these results hold true for an actual security solution, it is plausible to hypothesize that five attempts is even below the threshold of a typical Intrusion Detection System (IDS), thus an intrusion could occur unnoticeable.

To better demonstrate the impact of adversarial attacks over real systems, we submitted samples to the Virus Total service [46] and compared the detection rate of the original and the evasive samples. Figure 5 shows that the AVs were also affected by the samples' modifications, even though the challenge originally did not target any AV. This happens because (i) hiding the payload from the ML models also hides them from the AV scanners; and (ii) the ML models used by the AVs are also affected by the changes in the binaries features introduced by the malware dropper. This latter phenomenon can be clearly observed in practice if we focus on the detection labels provided by the AVs that claim to use ML [6, 9, 17, 18, 28], as shown in . Table 2 for the samples that less (22) and most (27) affected AV detection. We highlight that even the samples which less affected AVs bypassed at least one ML-based malware detection solution.

The impact of hiding the payload from the AV can also be observed in the assigned labels, as shown in Figure 1b. We noticed that

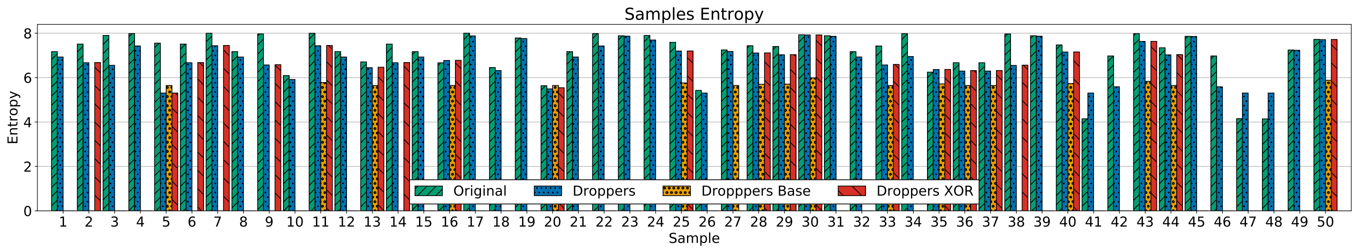


Figure 4: Sample's maximum section entropy. Embedding the original malware samples into binary droppers did not generate sections with greater entropy values.

Table 2: ML and AntiVirus. AVs that claim to use ML are also affected by our adversarial malware samples.

Sample	Version	AntiVirus				
		CrowdStrike [17]	Cylance [9]	Cynet [18]	Elastic [6]	Paloalto [28]
22	Original	True (100%)	True	True (100%)	True (high confidence)	True
	Adversarial	True (60%)	True	False	False	False
27	Original	True (100%)	True	True (100%)	True (high confidence)	True
	Adversarial	False	False	False	True	False

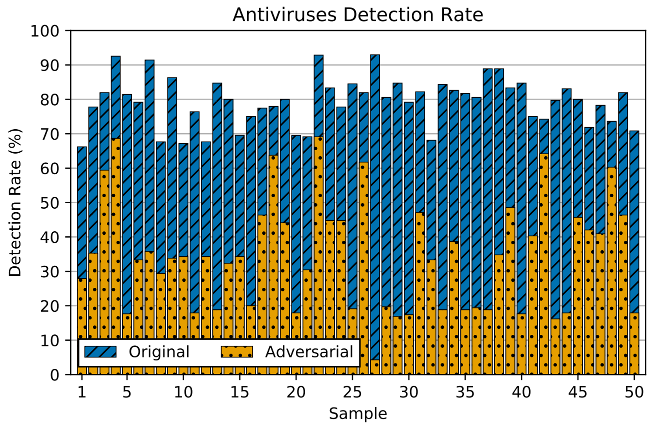


Figure 5: Detection rate of AVs. Real AVs were also affected by our deployed evasion techniques.

the labels assigned to the adversarial samples were significantly different from those assigned to the original samples, which suggests that they were detected using distinct rules, heuristics, patterns etc. In our case, the majority of the samples (20) were turned into `razy` family, which consists in malware that attack browser extensions to steal crypto-currency [8]. We believe that this phenomenon might be related to the fact that many `razy` samples are distributed in the form of a dropper.

A side-effect of embedding the payloads into a dropper is that the dropper binaries become similar, as they share the same headers, instructions, libraries, and so on. In Figure 6, we present the samples similarity according to `ssdeep` [40]'s scores. It shows that embedding the original malware samples into dropper binaries increased the number of samples reported as similar, even though reducing the relative frequency of very similar sample's scores. The first effect occurs because the dropper's similarities are identified by the similarity matching solution. In turn, the second effect occurs because the similar bytes between two binaries are "diluted" among the dropper's bytes, thus reducing the similarity score.

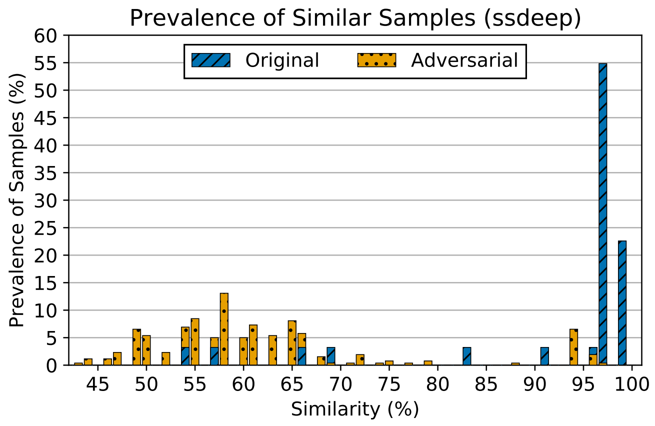


Figure 6: Sample's similarity. Encoding the payload reduces the sample's similarity score.

We believe that the first phenomenon might provide an interesting mechanism for the detection of the adversarial malware. If similarity scores were considered by a security solution in addition to the ML model's verdicts, the solution could be able to use it to correlate a detected dropper with an evasive sample. From the attacker's perspective, it would be now required to bypass all ML models all the time, instead of risking that a single similar, detected dropper raises a detection warning.

Table 3: Original vs updated model. Training with adversarial malware does not help to improve classification performance.

Train Dataset	Test Dataset	False Negative Rate (FNR)	False Positive Rate (FPR)
EMBER only	MLSEC 2020 Original Samples and Pristine Windows Apps	0%	0.1%
EMBER and MLSEC 2019 Adversaries		0%	78.54%
EMBER only	Our MLSEC 2020 Adversaries and Pristine Windows Apps	100%	0.1%
EMBER and MLSEC 2019 Adversaries		0%	78.54%

3 WHY DEFENDING IS HARDER?

A frequent question related to the bypass of ML models is why classifiers did not detect the adversarial samples. In the hereby reported case, the adversarial samples we produced in the attackers challenge were not detected by any model because we made them look like a goodware application (the Calculator), as shown in the previous section. In face of this scenario, it is common to hear proposals for training the models with adversarial samples so as to harden them against evasion. If this were effective, it would increase the security coverage of most security solutions. For instance, AVs would be allowed to update their models to detect adversarial samples as soon as some previously undetected samples were uncovered.

To test this hypothesis, we compared our original model—the one submitted to the competition—with a new one, trained with the same EMBER datasets, but including the 594 MLSEC 2019 adversarial samples provided by the organizers. Then, we tested these models with two datasets: one containing the adversaries we developed, and another containing the original samples. Both of them included the same pristine Windows applications as the "goodware" set.

In Table 3, we present the results for the multiple training and test sets combinations. On the one hand, when we trained our model using only EMBER datasets, it detected all the competition original malware samples ($FNR = 0\%$) and correctly labelled almost all the goodware samples ($FPR = 0.1\%$). However, the model was unable to detect our developed adversarial malware ($FNR = 100\%$). On the other hand, when we trained our model with last year's' adversarial samples in addition to the EMBER datasets, it was able to detect all malware samples from both original and adversarial datasets ($FNR = 0$$). However, the accomplished results incurred in a very high rate of false-positives ($FPR = 78.54\%$), i.e., our model started to recognize the majority of goodware as a malware.

The presented results indicate that the adversarial malware samples created by our attack are very difficult to distinguish apart from goodware, even if we use an updated ML model. This happens because they present the same features of goodware applications. Therefore, our updated model started to consider plenty of goodware samples as malware.

In practice, the problem of detecting adversarial samples might be even more complicated. AV companies will not have all adversarial samples at once, but will collect them over time as soon as they are uncovered. Hence, the problem of adversarial attacks mixes with the concept drift problem [15], when samples context shifts from time T to time $T + 1$. In our particular scenario, the malware samples evolve to prevent being detected. Whereas there are existing approaches to detect concept drift, these might face difficulties to handle adversarial samples, On the one hand, approaches that consider all the models at once [48] might be vulnerable to poisoning [42], which fall backs to the aforementioned scenario. On the other hand, approaches that perform partial retraining (e.g., ADWIN [7], DDM [20], and EDDM [4]) could increase the FP rate, since they discard goodware concepts to reduce the classifier's confusion between the classes. Therefore, there is a long path towards the development of effective and real-world solutions to handle adversarial malware attacks.

4 DISCUSSION

In this section, we revisit our findings after beating the MLSEC 2020, and discuss their implications for ML-based anti-malware.

Increasing ML Robustness. Our results suggest that ML-based malware detectors must be more robust against adversarial attacks to be practical, effective defenses. This type of detector should consider the great variety of malware samples, which may be distributed by attackers with simple modifications on them and, at the same time, exhibit the intended behavior/malicious actions. Despite being more resistant to attackers than previous year's models, all the models submitted in this year's competition were easily evaded. This indicates that the selected features are not robust enough to ensure a good detection model. The problem persisted even when we trained a new classifier with similar adversarial samples. Therefore, the investigation of new, more robust ways to represent malware is still a long path to be followed in future research work.

Explainable Attacks & Defenses. Although there are many existing automated approaches for adversarial attacks, we hereby presented attacks to ML models based on the attacker's knowledge about the models and binary implementations. This approach is more laborious, but it has the significant advantage of providing feedback information for the development of the next generation of security solutions. In an analogy, deep learning models are often criticized for not being explainable, despite being effective. We here extend the criticism to automated attacks, which are not explainable, despite their effectiveness. We believe that knowing when an ML model fails is extremely important to understand how to correct it.

Adversarial Attacks for the Masses. Our results show that adversarial attacks really happen, and they are effective. Thus, we claim that these attacks must be taken into account in threat models, training, and experiments. To encourage this practice, we are publicly releasing the code of our dropper to the community, so anyone may be able to practice with it (and improve it), as well as to consider adversarial attacks in their own research. Moreover, we are also making available a Web-based, automated solution to generate adversarial samples based on files uploaded by the user. Each submitted file is checked against multiple ML classifiers, including the ones from 2019's challenge and our classifier for the

2020's challenge. With that, we hope to allow users in checking the robustness of multiple models and the viability of attacking them.

Feedback for Future Work & Arms-Race. We believe that our findings might provide valuable feedback for the development of the next-generation security solutions. We discuss some insights that might lead to ML models improvement, and how they can be potentially attacked. Our findings show that embedding payloads into a binary is a simple yet effective way to defeat classifiers. Hence, the next-generation ML classifiers cannot be limited to look only into the first binary layer (the Dropper), but they will need to extract embedded payloads (e.g., via file carving) to classify them. When this strategy become mainstream, attackers will probably streamline the encoding of the payloads (e.g., using XOR, as we demonstrated). To handle this case, feature extractors should also try to guess the XOR key [39]. In the challenge, these steps were not performed due to the artificial timing constraints imposed to simulate the actual performance constraints of real systems. Therefore, performance-efficient tools for tasks such as key guessing need to be developed so as to make those approaches become practical. Alternatively, a possibility that might tackle the issue of ML detection using static features would be to change the samples representation, aiming at covering less mutable features. In the Dropper case, a representation based on the instruction disassembly would be more suitable than one based on the PE characteristics, since all Droppers perform the same actions before dumping the payload to the file system. However, in a future in which this detection approach becomes more popular, attackers will likely inject tons of dead code constructions into the binaries (as we did for function imports) to defeat the classifier. Therefore, we can expect the reemergence of the arms-race started in the signature-based malware detection realm and its extension to the ML-based malware detection scenario.

5 RELATED WORK

In this section, we present related work that propose defense or attack solutions for malware detection using machine learning in the literature.

There are many works in literature that propose defense solutions using machine learning. The majority of them consider using dynamic analysis, but they are not recommended for scenarios where a decision must be fast, given that a sandbox environment is required in order to extract dynamic attributes [49]. Thus, static analysis is performed in these cases, looking at the content of the samples without requiring their execution by extracting byte Sequences, opcodes, API and system calls, strings, disassembly code, control-flow and data-flow graphs, or PE file characteristics [3, 15, 21, 49]. Some solutions also consider strategies to defend (or to be more robust) against adversarial attacks, such as Label-based Semi-supervised Defense (LSD), Clustering-based Semi-supervised Defense (CSD) [43], the use of control-flow graphs loop instructions as features [25], or generative adversarial networks (GANs) as classification model [23].

In response to many of these defense solutions, attackers may try to evade them thought a great variation of attacks. Some of them consider the machine learning model being used, such as the perturbations used to attack neural networks and deep learning models [12], Silhouette Clustering-based Label Flipping Attack (SCLFA) [42], or the samples generated by Generative Adversarial Networks (GANs), which may be used to attack other models [23]. Other strategies consists in generating new variants of malware by using different packers [1], creating binary mutations by applying transformations such as code replacement, instruction swapping, variable changes, dead code insertion and control flow obfuscation [10, 37], or by creating automated binary exploitation that automatically discover flaws and exploits [38].

In contrast to these related work, this research presents a practical experience from the 2020 edition of the Machine Learning Security Evasion Competition (MLSEC) regarding on attacking ML models and its effects on them and AVs, producing useful insights for future community's research work.

6 CONCLUSION

In this paper, we reported our experience on a malware detection evasion contest (MLSEC 2020) and presented our insights on how to attack and defend machine learning models focused on classifying programs into malicious or not. We were challenged to bypass the detection of 50 samples, all of them submitted to three distinct ML models in a black box manner. We were able to bypass all models and were the first team to reach 150 points (perfect score) in the contest. During the period of the contest, we discovered that: (i) the first model operate by looking to PE header characteristics, thus being evaded by embedding malicious payloads in a dropper executable; (ii) the second model classified function imports and libraries via a TF-IDF method, being evaded by the addition of fake imports to the dropper; (iii) the last model also checks for strings in the embedded content, being evaded by the encoding of the payload using XORing or base64 techniques. We highlighted the impact of these techniques in practice by demonstrating that the detection rate of the Virus Total AVs decreased when the evasive samples were considered in comparison to the original ones, even though no specific AV was targeted in the competition. By describing the steps we took during the competition, we present how attackers and defenders reason about the problem and the challenges they face. We expect that this work might help those being introduced to the field of adversarial attacks.

Reproducibility. The source code of the developed dropper was released as open-source and it is available on `github`: https://github.com/marcusbotacin/Dropper. The source code of the tool used to append goodware data to our adversaries is open-source and available on `github`: https://github.com/ludersGabriel/BreakingGood. The source code of our detection model is also open-source and available on `github`: https://github.com/fabriciojoc/mlsec2020-needforspeed. A Web-based version of our obfuscation mechanism is available on the `corvus` platform: https://corvus.inf.ufpr.br.

Acknowledgments. Fabrício would like to thank the Coordination for the Improvement of Higher Education Personnel (CAPES, PhD Scholarship, process 88882.382195/2019-01). Marcus would like to thank the the Brazilian National Counsel of Technological and Scientific Development (CNPq, PhD Scholarship, process

164745/2017-3). We also want to thank the 2020 Machine Learning Security Evasion Competition (MLSEC) organizers (especially Hyrum Anderson and Zoltan Balazs) and their sponsors (Microsoft, CUJO AI's Vulnerability Research Lab, MRG Effitas, and VMRay) for the opportunity to participate and to keep developing new competitions that benefit the whole community.

REFERENCES

[1] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. 2020. When Malware is Packin'Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features. In *NDSS Proceedings (NDSS)*. NDSS, US, 1.

[2] Hyrum Anderson. 2019. Machine Learning Static Evasion Competition. https://www.elastic.co/pt/blog/machine-learning-static-evasion-competition/.

[3] Hyrum S. Anderson and Phil Roth. 2018. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *CoRR* abs/1804.04637 (2018), 1. arXiv:1804.04637 http://arxiv.org/abs/1804.04637

[4] Manuel Baena-Garćıa, José del Campo-Ávila, Raúl Fidalgo, Albert Bifet, Ricard Gavaldà, and Rafael Morales-Bueno. 2006. Early Drift Detection Method.

[5] Zoltan Balazs. 2020. CUJO AI Partners with Microsoft for the Machine Learning Security Evasion Competition 2020. https://cujo.com/machine-learning-security-evasion-competition-2020/.

[6] Shay Banon. 2020. Introducing Elastic Endpoint Security. https://www.elastic.co/blog/introducing-elastic-endpoint-security.

[7] Albert Bifet and Ricard Gavaldà. 2007. Learning from Time-Changing Data with Adaptive Windowing, In SIAM Int. Conf. on Data Mining. *SIAM Int. Conf. on Data Mining*.

[8] David Bisson. 2019. Razy Trojan Installs Malicious Browser Extensions to Steal Cryptocurrency. https://securityintelligence.com/news/razy-trojan-installs-malicious-browser-extensions-to-steal-cryptocurrency/.

[9] BlackBerry - Cylance. 2020. Hard on viruses, light on your computer. https://shop.cylance.com/us/.

[10] Jean-Marie Borello and Ludovic Mé. 2008. Code obfuscation techniques for metamorphic viruses. https://doi.org/10.1007/s11416-008-0084-2. *JICVHT*. (2008).

[11] Marcus Botacin, Anatoli Kalysch, and André Grégio. 2019. The Internet Banking [in]Security Spiral: Past, Present, and Future of Online Banking Protection Mechanisms Based on a Brazilian Case Study. In *Proceedings of the 14th International Conference on Availability, Reliability and Security* (2019-01-01) *(ARES '19)*. ACM, Canterbury, CA, United Kingdom, 49:1–49:10. https://doi.org/10.1145/3339252.3340103

[12] Nicholas Carlini and David Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *2017 IEEE Symposium on Security and Privacy (SP)*. 39–57. https://doi.org/10.1109/SP.2017.49

[13] Ero Carrera. 2019. PEfile python handler. https://pypi.org/project/pefile/.

[14] Fabrício Ceschin, Marcus Botacin, Heitor Murilo Gomes, Luiz S Oliveira, and André Grégio. 2019. Shallow Security: On the Creation of Adversarial Variants to Evade Machine Learning-Based Malware Detectors. In *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium* (2019-11-28) *(ROOTS'19)*. Association for Computing Machinery, Vienna, Austria. https://doi.org/10.1145/3375894.3375898

[15] Fabrício Ceschin, Felipe Pinage, Marcos Castilho, David Menotti, Luis S Oliveira, and André Gregio. [n.d.]. The Need for Speed: An Analysis of Brazilian Malware Classifers. *IEEE Security Privacy* 16, 6 ([n. d.]), 31–41. https://doi.org/10.1109/MSEC.2018.2875369

[16] Alexandre Cheron. 2017. Code Injection with Python. https://axcheron.github.io/code-injection-with-python/.

[17] CrowdStrike. 2020. Falcon Prevent: Cloud-native Next-Generation Antivirus (NGAV). https://www.crowdstrike.com/endpoint-security-products/falcon-prevent-endpoint-antivirus/.

[18] Cynet. 2020. NEXT-GEN ANTIVIRUS. Proactively Block Zero Day Attacks. https://www.cynet.com/platform/threat-protection/nextgen-anti-virus/.

[19] Bobby Filar. 2020. Malware Bypass Research using Reinforcement Learning. https://github.com/bfilar/malware_rl.

[20] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A Survey on Concept Drift Adaptation. *ACM Comput. Surv.* (2014).

[21] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. 2014. Malware Analysis and Classification: A Survey. *Journal of Information Security* 5, 2 (2014), 56–64.

[22] Chuan Guo, Jacob R. Gardner, Yurong You, Andrew Gordon Wilson, and Kilian Q. Weinberger. 2019. Simple Black-box Adversarial Attacks. *CoRR* abs/1905.07121 (2019), 1. arXiv:1905.07121 http://arxiv.org/abs/1905.07121

[23] Weiwei Hu and Ying Tan. 2017. Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN. *CoRR* abs/1702.05983 (2017). arXiv:1702.05983 http://arxiv.org/abs/1702.05983

[24] lucasg. 2017. The sad state of PE parsing. https://lucasg.github.io/2017/04/28/the-sad-state-of-pe-parsing/.

[25] Aravind Machiry, Nilo Redini, Eric Gustafson, Yanick Fratantonio, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2018. Using Loops For Malware Classification Resilient to Feature-Unaware Perturbations. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 112–123. https://doi.org/10.1145/3274694.3274731

[26] Malwarebytes Labs. 2017. Trojan.Remcos. https://blog.malwarebytes.com/detections/trojan-remcos/.

[27] Microsoft Security Intelligence. 2017. Win32/Gamarue. https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32/Gamarue.

[28] Palo Alto Networks. 2020. Advanced Endpoint Protection Protects You From Dated Antivirus. https://www.paloaltonetworks.com/cyberpedia/advanced-endpoint-protection-protects-you-from-dated-antivirus.

[29] PELock. 2016. PELock. https://www.pelock.com/.

[30] Quarkslab. 2019. Lief. https://lief.quarkslab.com/doc/stable/api/python/pe.html.

[31] Erwin Quiring, Lukas Pirch, Michael Reimsbach, Daniel Arp, and Konrad Rieck. 2020. Against All Odds: Winning the Defense Challenge in an Evasion Competition with Diversification. arXiv:2010.09569 [cs.CR]

[32] scikit learn. 2020. MinMaxScaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html.

[33] scikit learn. 2020. OneHotEncoder. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html#sklearn.preprocessing.OneHotEncoder.

[34] scikit learn. 2020. RandomForestClassifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[35] scikit learn. 2020. TfidfVectorizer. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html.

[36] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. AVclass: A Tool for Massive Malware Labeling. In *Research in Attacks, Intrusions, and Defenses*, Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro (Eds.). Springer International Publishing, Cham, 230–253.

[37] Peng Shao and Randy K. Smith. 2009. Feature Location by IR Modules and Call Graph. In *ACM-SE 47*.

[38] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[39] Alex Smirnov. 2016. Xor-Decrypt. https://github.com/AlexFSmirnov/xor-decrypt.

[40] ssdeep. 2017. ssdeep - Fuzzy hashing program. https://ssdeep-project.github.io/ssdeep/index.html.

[41] stevielavern. 2017. Cannot add section and rebuild PE on Windows 10 #109. https://github.com/lief-project/LIEF/issues/109.

[42] Rahim Taheri, Reza Javidan, Mohammad Shojafar, Zahra Pooranian, Ali Miri, and Mauro Conti. 2019. On Defending Against Label Flipping Attacks on Malware Detection Systems. arXiv:1908.04473 [cs.LG]

[43] Rahim Taheri, Reza Javidan, Mohammad Shojafar, Zahra Pooranian, Ali Miri, and Mauro Conti. 2020. On Defending Against Label Flipping Attacks on Malware Detection Systems. arXiv:1908.04473 [cs.LG]

[44] Oreans Technologies. 2011. Themida. https://www.oreans.com/Themida.php.

[45] Telock. 2008. Telock. http://www.telock.com-about.com/.

[46] Virus Total. 2020. Virus Total. https://www.virustotal.com.

[47] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas. 2015. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *2015 IEEE Symposium on Security and Privacy*. IEEE, US, 659–673.

[48] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu. 2019. DroidEvolver: Self-Evolving Android Malware Detection System. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*.

[49] Yanfang Ye, Tao Li, Donald Adjeroh, and S. Sitharama Iyengar. 2017. A Survey on Malware Detection Using Data Mining Techniques. *ACM Comput. Surv.* 50, 3, Article 41 (June 2017), 40 pages. https://doi.org/10.1145/3073559