When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features

Hojjat Aghakhani*, Fabio Gritti*, Francesco Mecca†, Martina Lindorfer‡,
Stefano Ortolani§, Davide Balzarotti¶, Giovanni Vigna* and Christopher Kruegel*
*University of California, Santa Barbara †Università degli Studi di Torino ‡TU Wien §Lastline Inc. ¶Eurecom
*{hojjat, degrigis, vigna, chris}@cs.ucsb.edu †francesco.mecca402@edu.unito.it ‡mlindorfer@iseclab.org
§ortolani@lastline.com and ¶davide.balzarotti@eurecom.fr

***Abstract*—Machine learning techniques are widely used in addition to signatures and heuristics to increase the detection rate of anti-malware software, as they automate the creation of detection models, making it possible to handle an ever-increasing number of new malware samples. In order to foil the analysis of anti-malware systems and evade detection, malware uses packing and other forms of obfuscation. However, few realize that benign applications use packing and obfuscation as well, to protect intellectual property and prevent license abuse.**

In this paper, we study how machine learning based on static analysis features operates on packed samples. Malware researchers have often assumed that packing would prevent machine learning techniques from building effective classifiers. However, both industry and academia have published results that show that machine-learning-based classifiers can achieve good detection rates, leading many experts to think that classifiers are simply detecting the fact that a sample is packed, as packing is more prevalent in malicious samples. We show that, different from what is commonly assumed, packers do preserve some information when packing programs that is "useful" for malware classification. However, this information does not necessarily capture the sample's behavior. We demonstrate that the signals extracted from packed executables are *not rich enough* for machine-learning-based models to (1) generalize their knowledge to operate on unseen packers, and (2) be robust against adversarial examples. We also show that a naïve application of machine learning techniques results in a substantial number of false positives, which, in turn, might have resulted in incorrect labeling of ground-truth data used in past work.

I. Introduction

Anti-malware software provides end-users with a means to detect and remediate the presence of malware on their machines. Most anti-malware software traditionally consists of two parts: a signature-based detector and a heuristics-based classifier. While signature-based methods detect similar versions of known malware families with a small error rate, they become insufficient as an ever-increasing number of new malware samples are being identified [56]. VirusTotal reports that, on average, over 680,000 new samples are analyzed per day [114], of which some are merely re-packed versions of previously seen samples with identical behavior. Over the last few years, the need for techniques that generalize to new, unknown malware samples while removing expensive human experts from the loop has led to approaches that leverage both static and dynamic analyses combined with data mining and machine learning techniques [26, 59, 80, 81, 93, 97, 98, 103].

Although dynamic analysis provides a clear picture of an executable's behavior, it has some issues in practice: for example, dynamic analysis of untrusted code requires either kernel-level privileges [27], thus expanding the attack surface, or a virtual machine [27], which requires a substantial amount of computing resources. In addition, malware usually employs environmental checks to avoid detection [32, 58, 83], and the virtualized environment may not reflect the environment targeted by the malware [90]. To avoid such limitations, some approaches [51, 56, 70, 82, 93] heavily rely on features extracted through static analysis. These approaches are appealing to anti-malware companies that want to replace anti-malware systems based on dynamic analysis. These static-analysis-based anti-malware vendors, which have quickly grown into billion-dollar companies, boast that their tools leverage "AI techniques" to determine the maliciousness of programs solely based on their static features (i.e., without having to execute them). However, static analysis has known issues when applied to obfuscated and packed samples [67, 77].[1]

It is commonly assumed that packing greatly hinders machine learning techniques that leverage features extracted from static (file) analysis. However, both industry and academia have published results showing that machine-learning-based classifiers can achieve good detection rates. Many experts assume that these results are due to the fact that classifiers just learn to distinguish between packed and unpacked programs. In fact, we would expect that machine-learning-based classifiers will deliver poor performance in real-world settings, where packing is increasingly seen in both malicious and benign software [10, 59, 84]. Unfortunately, most related work did not consider or only briefly discussed the effects of packing when proposing machine-learning-based classifiers [43, 56, 80, 81, 82, 96, 97]. Surprisingly, our initial experiments showed that machine-learning-based classifiers can distinguish between packed benign and packed malicious samples in our dataset. This led us to the following research question: does static analysis on *packed* binaries provide a *rich enough* set of features to build a malware classifier using machine learning?

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24310
www.ndss-symposium.org

[1] While packing can be applied to any program, hereinafter we focus on the packing of Windows x86 binary programs.

Our experiments require a ground-truth dataset for which we can determine if each sample is (1) packed or unpacked and (2) malicious or benign. We created our first dataset, the *wild dataset*, with executables provided by a commercial anti-malware vendor, which uses dynamic features, combined with the labeled benchmark dataset EMBER [3]. We leveraged the vendor's sandbox, along with VirusTotal, to remove samples with inconsistent benign/malicious labels from the dataset. For identifying packed executables, we used the vendor's sandbox combined with the Deep Packer Inspector [110] tool and a number of static tools. The fact that we built the dataset mainly based on the runtime behavior of samples gives us high confidence in our ground truth labels. We created a second dataset, the *lab dataset*, by packing all the executables in the *wild dataset* with widely used commercial and free packers. Following a detailed literature study, we extracted nine families of features from the executables in the two datasets. Even though in our experiments we used SVM, deep neural networks (i.e., MalConv [80]), and different variants of decision-tree learners, like random forest, we only discuss the results of the random forest approach as (1) we observed similar findings for these approaches, with random forest being the best classifier in most experiments, and (2) random forest allows for better interpretation of the results compared to neural networks [35].

As a naïve experiment, we first trained the classifier on *packed malicious* and *unpacked benign* samples. The resulting classifier produced a high false positive rate on *packed benign* samples, which shows that the classifier is biased towards detecting *packing*. Using n-grams, Perdisci et al. [77] also observed that *packing detection* is an easier task to learn compared to detecting maliciousness. In addition, we demonstrated that "packer classification" is a trivial task by training a packer classifier using samples from each packer (class) in the *lab dataset*. The classifier achieved precision and recall greater than 99.99% for each class. This indicates that a bias in the training set regarding packers may cause the classifier to learn specific packing routines as a sign of maliciousness. We verified this by training the classifier on benign and malicious executables packed by two non-overlapping subsets of packers, which we refer to as *good* and *bad* packers, respectively. The resulting classifier learned to label anything packed by *good* packers as benign, and anything packed by *bad* packers as malicious, regardless of whether or not the sample is malicious.

We extended the naïve experiment by training the classifier on different training sets with increasing ratios of *packed benign* samples. To avoid the bias introduced by the use of *good* and *bad* packers, we selected packed samples from the *lab dataset* uniformly distributed over packers. Surprisingly, despite the popular assumption that packing hinders machine-learning-based classifiers, we found that increasing the packed benign ratio in the training set helped the classifier to maintain relatively low false positive and false negative rates. This shows that packers preserve some information about the original binary that can be leveraged for malware detection. For example, most packers keep `.CAB` file headers in the resource sections of the executables. Jacob et al. [44] found a similar trend for packers that employ weak encryption or compression. By training on one packer at a time, we observed that the information preserved about the original binaries is not necessarily associated with malicious behavior, but is "useful" for malware detection. Nevertheless, we argue that such a classifier still suffers from three issues: (1) inability to generalize, (2) failure in the presence of strong encryption, and (3) vulnerability to adversarial samples.

Generalization. Training the classifier on packed samples is not guaranteed to generalize to packers that are not included in the training set. We excluded one packer at a time from the training dataset and evaluated the classifier against samples packed with the excluded packer. We observed false positive rates of 43.65%, 47.49%, and 83.06% when excluding tElock, PECompact, and kkrunchy, respectively. Moreover, the classifier trained on *all* packers from the *lab dataset* produced a *false negative* rate of 41.98% on packed executables from the *wild dataset*. This means that although packers preserve some information, the trained classifier fails to generalize to previously unseen packing routines. This is a severe problem as malware authors often prefer customized packing routines to off-the-shelf packers [34, 66, 110].

Strong & complete encryption. We argue that an executable might be packed in a way that reveals no information related to its behavior until it is executed. As a preliminary step, we packed all executables in the *wild dataset* with our own packer, called *AES-Encrypter*, which encrypts the executable with AES and injects it as the overlay of the packed binary. When the packed program is executed, *AES-Encrypter* decrypts the overlay and executes the original program within a new process. All static features are always the same, except for features extracted from the encrypted overlay. We trained and tested the classifier on executables packed by the *AES-Encrypter*, and, as expected, the classifier could not distinguish between benign and malicious executables packed by *AES-Encrypter*. This shows that packing can be performed without transferring any (static) initial pattern to the packed program, if properly optimized for this purpose.

Adversarial samples. Machine-learning-based malware classifiers have been shown to be vulnerable against adversarial samples, especially those that use only static analysis features [33, 41, 89]. We expect that generating such adversarial samples would be easier in our case, as static analysis of packed binaries does not provide features that capture a sample's behavior. We first trained the classifier on a dataset whose benign and malicious samples are packed with the same packers so that the classifier is not biased to detect specific packing routines as a sign of maliciousness. The classifier maintained a low error rate. From all malicious samples that the classifier detected successfully, we managed to generate new samples that the classifier no longer detects as malicious. Specifically, we identified "benign" sequences of bytes that occurred more frequently in benign samples and injected them into the target binary without affecting the sample's behavior. Very recently, a group of researchers used a very similar technique to trick Cylance's AI-based anti-malware engine into thinking that malware like WannaCry and tools such as Mimikatz were benign [105]. They did this by taking strings from an online gaming program and injecting them into malicious files. Since games are highly obfuscated and packed, they confront such an engine with a dilemma; either inherit a bias towards games or produce high rates of false positives for them [1].

To investigate how real-world malware detectors operate on packed executables, we submitted benign and malicious executables packed by each packer to VirusTotal. We only focused

on six machine-learning-based engines that use only static analysis features according to their description on VirusTotal or the company's website. Unfortunately, we observed that all these six engines learned that packing implies maliciousness. It must be noted that, we used commercial packers, like Themida, PECompact, PELock, and Obsidium, that legitimate software companies use to protect their software. Nevertheless, benign programs packed by these packers were detected as malware.

As packing is being increasingly adopted by legitimate software [84], the anti-malware industry needs to do better than detecting packers, otherwise good and bad programs are misclassified, causing pain to users and eventually resulting in alert fatigue and missed detections. This is especially a concern for previous studies that rely on anti-malware products for establishing ground truth, as misclassification of packed benign programs might have biased those studies [22, 43, 86, 88, 97].

In summary, we make the following contributions:

- We study the limits of machine-learning-based malware classifiers that use only static features. We show that the lack of overlap between packers used in benign and malicious samples causes the classifier to associate specific packers with maliciousness. We show that, if trained correctly, the classifier is able to distinguish between benign and malicious samples packed by real-world packers, though it remains susceptible to unseen packing routines or, even worse, to the application of strong encryption to the entire program. Furthermore, we show that it is possible to craft evasive samples that bypass detection via a naïve adversarial attack.
- Our evaluation of six products on VirusTotal shows that current *static* machine-learning-based anti-malware engines detect packing instead of maliciousness.
- We release a dataset of 392,168 executables for which we know whether each sample is (1) benign or malicious, and (2) packed or unpacked. We also know the specific packer for the *lab dataset*, which includes 341,444 executables.

We release the source code of all experiments in a Docker image at `https://github.com/ucsb-seclab/packware` to support the reproducibility of our results.

II. Motivation

Packing has long been an effective method for malware authors to evade the signature-based detection of anti-malware engines [71], but little is known about its legitimate usage in benign applications. As the first step in this direction, in 2013, Lakshman Nataraj [69] explored how anti-malware scanners available on VirusTotal handle packing. He packed 16,663 benign system executables from various Windows OS versions with four different packers (UPX, Upack, NSPack, and BEP), and submitted them to VirusTotal. He showed that 96.7% of the files packed with Upack, NSPack, and BEP triggered at least ten detections on VirusTotal. Another recent study [116] mined byte pattern-based signatures of anti-malware products to force misclassifications of benign files, and also found that the artifacts of packers are effective as "malicious markers." We argue that these results stem from the fact that packing historically has been associated with malware only. Consequently, a naïve detection approach only based on static features from packed samples will be heavily biased

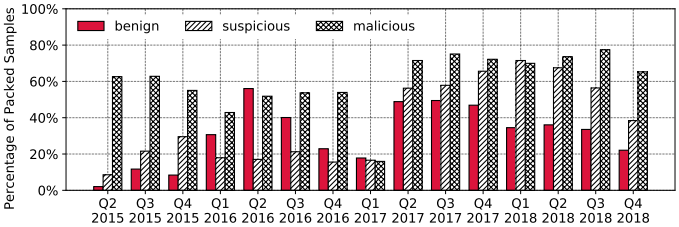


Fig. 1: Prevalence of packed samples in the wild.

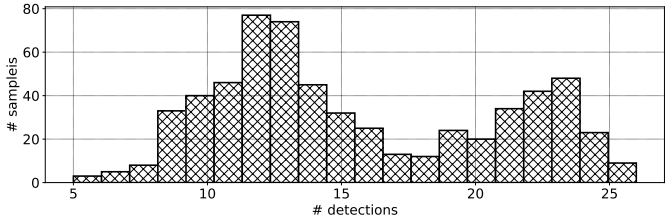


Fig. 2: The histogram of the number of detections on VirusTotal for Windows 10 binaries packed with Themida.

towards associating packing with malicious behavior. In fact, static analysis features that are shown to be useful for packing detection [5, 15, 37, 44, 59, 76, 77, 102, 109] are also being used by machine-learning-based malware detectors [28, 68, 82, 101, 102, 103].

We collected a large-scale, real-world dataset of malicious, suspicious, and benign files from a commercial vendor of advanced malware protection products. This dataset includes samples that the vendor analyzed from customers around the globe over the past three years. As Figure 1 shows, packing is not only widespread in malware samples (75%), but also common in benign samples (50% in the worst case). Note that Figure 1 presents a lower bound for the ratio of packed executables. Our findings overlap with the findings of Rahbarinia et al. [84], who studied 3 million web-based software downloads over 7 months in 2014, finding that *both* malicious and benign files use known packers (58% and 54%, respectively). Making matters even worse, more than half of the 69 unique packers they observed (e.g., INNO, UPX) are being used by both malicious and benign software. While some packers (e.g., NSPack, Molebox) were exclusively used to pack malware in their dataset, they conclude that packing information alone is not a good indicator of malicious behavior. We further packed 613 executables from a fresh installation of Windows 10 (located in *C:\Windows\System32*) with Themida and submitted them to VirusTotal. Figure 2 shows the histogram of the number of detections. Unsurprisingly, out of 613 binaries, 564 binaries were detected as malicious by more than 10 anti-malware tools. If we consider only the six machine-learning-based anti-malware engines on VirusTotal, out of 613 binaries, 553 binaries were detected as malicious by more than four tools.

As these numbers show, any approach that fails to consider packed benign samples when designing and evaluating a malware detection approach ultimately results in a substantial number of false positives on real-world data. This is especially a concern for machine-learning-based approaches, which, in the absence of reliable and fresh ground truth, frequently rely on labels from anti-malware products available on VirusTotal [22, 43, 86, 88, 97]. Given the disagreement of anti-malware products in labeling samples [42, 48, 65, 99], a common

practice is to sanitize a dataset, for example, by considering decisions from a selected set of anti-malware products, or, as another example, by using a voting-based consensus. While this approach is problematic for various reasons [42, 65], we believe that one main aspect is particularly troublesome: *Dataset pollution*. Packed benign samples that are detected by anti-malware products as malicious are incorrectly used as malware samples. For example, a recent related work [22] used a similar procedure for labeling, as stated by the authors: "We train a classifier using supervised learning and therefore require a target label for each sample (0 for benign and 1 for malware). We use malware indicators from VirusTotal. For each sample, we count the number of malicious detections from the various engines aggregated by VirusTotal, weighted according to a reputation we give to each engine, such that several well-known engines are given weight >1, and all others are weighted 1. We use the result to label a sample benign or malicious." While we do not know which weights are used by the authors, there is a good chance that their dataset is skewed, since, as we showed above, a number of anti-malware engines on VirusTotal detect packed benign samples as malware.

As studied by the anti-malware community, evaluating existing malware detection methodologies poses substantial challenges [57, 65, 90]. For example, Rossow et al. [90] presented guidelines for collecting and using malware datasets. Our work aims to find whether packing even retains *rich enough* static features from the original binary to detect anything meaningful besides the packing itself. To the best of our knowledge, no prior work has considered the effects of packed executables on machine-learning-based malware detectors that leverage only static analysis features.

III. Background

A. Executable Packers

A packer is a software component that applies a set of routines to compress or encrypt a target program. The simplest form of packing consists of the decryption or decompression (at runtime) of the original payload followed by a jump to the memory address that contains the target payload (this technique is called "tail jump"). Ugarte et al. [110] classify packers into six types, with an increasing level of complexity in the reconstruction of the target payload:

Type I: A single unpacking routine is executed to transfer the control to the original program. UPX is the most popular packer in this class. **Type II:** The packer employs a chain of unpacking routines executed sequentially, with the original code recomposed at the end of the chain. **Type III:** Unpacking routines include loops and backward edges. Though the original code is not necessarily reconstructed in the last layer, a tail transition still exists to separate the packer and the application code. **Type IV:** In each layer of packing, the corresponding part of the unpacking routine is interleaved with the corresponding part of the original code. However, the entire original code will be completely unpacked in memory at some point during the execution. **Type V:** The packer is composed of different layers in which the unpacking code is mangled with the original code. There are multiple tail jumps that reveal only a single frame of the original code at a time. **Type VI:** Packers reveal (unpack) only a single fragment of the original code (as little as a single instruction) at any given time.

We discuss approaches that are proposed for packing detection, packer identification, and automated unpacking in Appendix A. Here, we discuss the limitations of these methods.

Limitations of packing detection. Signature-based approaches to packing detection have a high false negative rate, as they require a priori knowledge of packed executables generated by each packer. As an example, PEiD is shown to have approximately a 30% false negative rate [76]. Other approaches apply static analysis to extract a set of features or use hand-crafted heuristics to detect packed executables. However, they are vulnerable to adversaries. As an example, the Zeus malware family applies different techniques, such as inserting a selected set of bytes into executables, in order to keep the entropy of the file and its sections low [112]. Such malware evades entropy-based heuristics, as they are often used to determine if an executable is packed [59]. Dynamic approaches seem to perform better, since they often look for a write-execute sequence in a memory location, which is the definition of packing. However, packed executables usually employ different techniques to evade analysis, like conditional execution of unpacking routines [21].

Limitations of generic unpackers. Packers usually employ different techniques to evade analysis approaches utilized by generic unpackers. For example, tELock and Armadillo leverage several anti-debugging routines to terminate the execution in a debugging setting [9, 13]. Although some unpackers exploit hardware virtualization to achieve transparency [24], the introduced performance overhead could be unacceptable [117]. Themida applies virtualization obfuscation to its unpacking routine, which can cause slice size explosion [64]. In general, generic unpackers rely on a number of assumptions that do not necessarily hold in practice [110]: (1) the entire original code is in memory at a certain point, (2) the original code is unpacked in the last layer, (3) the execution of the unpacking routine and the original code are completely separated, and (4) the unpacking code and the original code run in the same process without any inter-process communication. These simplifications make these unpackers inadequate for handling the challenges introduced by complex, real-world packers. Moreover, generic unpackers often rely on heuristics that are designed for specific packers [110].

B. Packing vs. Static Malware Analysis

In Appendix B, we discuss how machine learning is being adopted by the anti-malware community to statically analyze malicious programs. In particular, we reviewed a wide range of static malware analysis approaches based on machine learning [2, 7, 28, 39, 43, 44, 49, 51, 52, 53, 54, 56, 62, 63, 68, 70, 80, 81, 82, 86, 93, 94, 96, 97, 98, 102, 103, 104, 107, 108, 118]. Although static malware detectors have been shown to be biased towards detecting packing [69, 77, 116], we observed a number of limitations in related work when it comes to the handling of packed executables. In particular, out of the 30 papers mentioned above: **(1)** Ten papers [2, 28, 39, 63, 68, 81, 82, 86, 98, 118] do not mention packing or obfuscation techniques. **(2)** Ten approaches [7, 54, 56, 62, 93, 94, 96, 104, 107, 108] work only on unpacked executables, as mentioned by the authors. They used either unpacked executables or executables that they managed to unpack. **(3)** Seven papers [43, 49, 52, 53, 70, 80, 97] claim to perform well in malware

classification regardless of whether or not the executables are packed. However, the authors did not discuss whether any bias in terms of packing was present in their dataset or not. More precisely, they did not mention using packed benign executables in their dataset, or brief examinations have been done on the effects of packed executables [49, 70], though the evaluation has been thoroughly carried out only on unpacked executables. **(4)** Only three papers [51, 102, 103] focused on packed executables. However, they have two major limitations: (a) they use signature-based packer detectors, such as PEiD, to detect packing, while PEiD has approximately a 30% false negative rate [76], and (b) they augmented their datasets by packing benign executables using only a small number of packers. However, malicious executables might be packed with a different set of packers, which can result in a bias towards detecting specific packing techniques. Jacob et al. [44] detect similar malware samples even if they are packed, yet, their method is resilient only against packers that employ compression or weak encryption, as they acknowledge.

Finally, most related work did not publish their datasets, hence these approaches cannot be fairly compared to each other.

IV. Dataset

Our experiments require a dataset composed of executable programs for which we know if they are: (1) benign or malicious and (2) packed or unpacked. We combined a labeled dataset from a commercial vendor with the EMBER [3] dataset (labeled) to build our *wild dataset*. We leveraged a hybrid approach to label an executable as packed or unpacked. We built another ground-truth dataset, the *lab dataset*, by packing all executables in the *wild dataset* with widely used commercial and free packers and our own packer, *AES-Encrypter*. Following a detailed study of the literature, we extracted nine families of features for all samples.

A. Wild Dataset

We used two different sources to create our *wild dataset* of Windows x86 executables. **(1)** A commercial anti-malware vendor provided 29,573 executables. These samples, observed "in the wild," were randomly selected from an original pool that was analyzed by the anti-malware vendor's sandbox in the US during the period from 2017-05-15 to 2017-09-19. Along with the benign/malicious label and the malicious behaviors observed during the execution, the vendor identified which executable was packed or not. **(2)** A labeled benchmark dataset, called EMBER, was introduced by Anderson et al. [3] for training machine learning models to statically detect Windows malware. This dataset consists of 800,000 Windows executables that are labeled. However, no information is provided regarding packing. We randomly selected 56,411 x86 executables from this dataset and submitted each sample to the commercial anti-malware vendor's sandbox, in order to identify if the sample is packed. This also provides us confirmation whether an executable is malware or benign software, as the sandbox detects malicious behavior. Note that samples from these two sources were observed "in the wild" sometime in 2017, allowing more than enough time for current anti-malware engines to have incorporated means to detect them. As these two sources might have samples that are incorrectly labeled, we performed a careful and extensive post-processing step, which we describe in the following paragraphs.

TABLE I: A: all, N/A: not available, B: benign, M: malicious. Note that this is **not** the final version of the *wild dataset*.

Samples' Origin	Malicious/Benign Label's Source			# of Samples
	VirusTotal	Comm. Anti-mal.	EMBER	
(1) Comm. Anti-malw.	A	A	N/A	29,573
	A	B	N/A	15,736
	B	B	N/A	**13,046**
	A	M	N/A	13,837
	M	M	N/A	**13,536**
(2) EMBER	A	A	A	56,411
	A	A	B	24,348
	A	B	B	24,225
	B	B	B	**24,223**
	A	A	M	32,063
	A	M	M	31,087
	M	M	M	**31,066**
(1) ∪ (2)	A	A	A	**85,984**
	B	B	B	**37,269**
	M	M	M	**44,602**

Malicious vs. benign. We used three different sources to detect whether an executable is malicious or benign. **(1) VirusTotal:** We obtained reports for our entire dataset by querying VirusTotal. All 85,984 executables in our dataset have been available on VirusTotal for more than one year. From all engines used by VirusTotal, we considered only seven tools that are well-known as strong products in the anti-malware industry and labeled each executable based on the majority vote. **(2) The anti-malware vendor:** Since we sent samples extracted from the EMBER dataset to the vendor's sandbox, we have the benign/malicious label for all samples. **(3) EMBER dataset:** All samples that we selected from the EMBER dataset are labeled by Endgame [29].

We discarded 4,113 samples for which there was a disagreement about their benign/malicious nature between the three sources. As Table I shows, at the end of this step, we have 37,269 benign and 44,602 malicious samples left (a total of 81,871 executables).

Packed vs. unpacked. Due to the limitations discussed in Section III-A, we leveraged a hybrid approach to determine if an executable is packed. In particular, for each sample, we took the following steps: **(1) The anti-malware vendor:** We submitted the sample to the vendor's sandbox, and given the downloaded report, we detected whether unpacking behavior had occurred or not. The anti-malware tool detects the presence of packed code by running the executable in a custom sandbox that interrupts the execution every time there is a write to a memory location followed by a jump to that address. At that point in time, a snapshot of the loaded instructions is compared to the original binary, and if they differ, the executable is marked as packed. **(2) Deep Packer Inspector (*dpi*):** We used *dpi* [110] to further analyze each sample. This framework measures the runtime complexity of packers. Adding an extra dynamic engine helps us to identify packed executables that are not detected as packed by the first dynamic engine. For example, the host configuration might make the sample terminate before the unpacking process starts. In addition, this framework gives us insights about the runtime complexity of packers in our dataset. As *dpi* is not operating on .NET executables, we removed all 13,489 .NET executables, 10,681 benign and 2,808

malicious, from our dataset, resulting in 68,382 executables, 26,588 benign and 41,794 malicious. **(3) Signatures and heuristics:** We used *Manalyze* [60], *Exeinfo PE*, *yara* rules, *PEiD*, and *F-Prot* (from VirusTotal) to identify packers that leave noticeable artifacts in packed executables.

In particular, we labeled an executable as packed in our dataset if one among the vendor's sandbox, *dpi*, and signature-based tools detects the executable as packed. In total, we labeled 46,328 samples as packed divided into 12,647 benign and 33,681 malicious samples. We further used heuristics proposed by *Manalyze* for packing detection to determine samples that **might** be packed. *Manalyze* labeled 24,911 samples as "possibly packed," of which 6,898 samples are not detected as packed by other tools. We argue that this discrepancy might be due to limitations with packing detection, which we discuss in Section III-A. Nevertheless, we discarded all these samples as we were not completely sure if they are packed or not.

Table X in the Appendix shows statistics about packed executables that are detected by each approach. Of 17,043 benign executables, 12,647 executables are packed, and 4,396 executables are unpacked, and of 40,031 malicious executables, 33,681 executables are packed, and 5,752 executables are unpacked. While unpacked malware is shown to be rare [10, 59, 61], we did not detect packing for 5,752 (13.61%) malicious samples. Since this percentage could be considered somewhat higher than expected, we attempted to verify our packer analysis by randomly selecting 20 samples, and manually looking for the presence or absence of unpacking routines. We observed the unpacking routine code for 18 samples, but our packer detection scheme did not detect them due to the anti-detection techniques that these samples use. Since we do not need any *unpacked malicious* executables for our experiments, we discarded all 5,752 malicious samples that our system labeled as unpacked. To confirm that all 4,396 benign samples that we identified as unpacked are not packed, we manually looked into 100 unpacked benign executables and did not find any sign of packing. Simple statistics guarantee that more than 97.11% (95.59%) of these samples are labeled correctly with the confidence of 95% (99%).

We further noticed that our dataset was skewed in terms of DLL files, containing 4,005 benign DLLs but only 598 malicious ones. We removed all these samples from our dataset. In the end, the *wild dataset* consists of 50,724 executables divided into 4,396 unpacked benign, 12,647 packed benign, and 33,681 packed malicious executables.

Packer complexity. As Table X in the Appendix shows, *dpi* detects the unpacking behavior for 34,044 executables in the *wild dataset*. Table XI presents the packer complexity classes, as defined by Ugarte et al. [110], for these executables.

Packers in the wild. Using *PEiD*, *F-Prot*, *Manalyze*, *Exeinfo PE*, and *yara* rules, we matched signatures of packers for 9,448 executables, 1,866 benign and 7,582 malicious. We found the artifacts of 48 packers in the *wild dataset*. As Table XII in the Appendix shows, some packers like dxpack, MPRESS, and PECompact have been used mostly in malicious samples.

B. Lab Dataset

Some of our experiments require us to know with certainty which packer is used to pack a program. Therefore, we obtained nine packers that are either commercially available or freeware (namely Obsidium, PELock, Themida, PECompact, Petite, UPX, kkrunchy, MPRESS, and tElock) and packed all 50,724 executables in our *wild dataset* to create the *lab dataset*. None of the packers were able to pack all samples. For example, Petite failed on most executables with a GUI, while Obsidium in some cases produced empty executables. We looked at logs generated by these packers and removed those executables that were not properly packed. We also verified that all packed executables have valid entry points. Finally, we developed our own simple packer, called *AES-Encrypter*, which, given the executable P, encrypts P using AES with a random key (which is included in the final binary), and injects the encrypted binary as the overlay of the packed binary P'. When P' is executed, it first decrypts the overlay and then executes the decrypted (original) binary. Table II lists the number of samples we packed successfully with each packer. In total, we generated 341,444 packed executables. To ascertain if packing does, in fact, preserve the original behavior, we compared the behavior of these samples with the original samples. Our results confirm that 94.56% of samples exhibit the original behavior. We explain in Appendix C how we conducted this comparison.

TABLE II: Overview of the *lab dataset*.

Packer	# Benign Samples	# Malicious Samples	Keeps Rich Header?	# Invalid Opcodes
Obsidium	16,940	31,492	29.82%	0
Themida	15,895	26,908	✓	0
PECompact	5,610	28,346	✓	723
Petite	13,638	25,857	✓	318
UPX	9,938	20,620	✓	0
kkrunchy	6,811	15,494	19.68%	61
MPRESS	11,041	11,494	19.84%	629
tElock	5,235	30,049	✓	8
PELock	6,879	8,474	20.60%	461
AES-Encrypter	17,042	33,681	✗	0

TABLE III: Summary of extracted features.

PE headers	28	**Byte n-grams**	13,000
PE sections	570	**Opcode n-grams**	2,500
DLL imports	4,305	**Strings**	16,900
API imports	19,168	**File generic**	2
Rich Header	66		

C. Features

Following a detailed analysis of the literature (see Section B), we extracted nine families of static analysis features that were shown to be useful in related work. We used *pefile* [73] to extract features from three different sources: the PE structure, the program's assembly, and the raw bytes of the binary. As Table III shows, we extracted a total of 56,543 individual features from the samples in our dataset.

(1) PE headers. Features related to PE headers have been widely used in related work. In our case, we use all fields in the PE headers that exhibit some variability across different executables (some header fields never change [56]). We extracted 12 individual features from the Optional and COFF headers, which are described in Table XX in the Appendix. Moreover, from the characteristics field in the COFF header, we extracted 16 binary features, each representing whether the corresponding flag is set for the executable or not. Thus, we extracted 12 integer and 16 binary features from the PE headers, resulting in a total of 28 features.

(2) PE sections. Every executable has different sections, such as the `.data` and `.text` sections. For each section, we extracted 8 individual features as described in Table XXI in the Appendix. Moreover, from the characteristics field in the section header, we created up to 32 binary features for each bit (flag). For example, the feature corresponding to the 30th bit is true when the section is executable. We ignored the bits (flags) that do not vary in our dataset. For each section of the PE file, we computed 32 (at most) binary, 7 integer, and one string feature, named `pesection_sectionId_field`. The maximum number of sections that an executable has in our dataset is 19. For each executable, we built a vector of 516 different features obtained from its sections followed by the `default` values for sections that the sample does not include. Based on the related work, we augmented this set of features with the following processing steps: (1) We extracted the above-mentioned features for the section where the executable's entry point resides and added them to the dataset separately; (2) We calculated the mean, minimum, and maximum entropy of the sections for each executable. We did the same for both the size and the virtual size attributes. As a result, we extracted a total of 570 features from the PE sections.

(3) DLL imports. Most executables are linked to dynamically-linked libraries (DLLs). For each library, we use a binary feature that is true when an executable uses that library. In total, we have 4,305 binary features in this set.

(4) API imports. Every executable has an Import Directory Table that includes the APIs that the executable imports from external DLLs. We introduce a binary feature for each API function that is true if the executable imports that function. In total, we have 19,168 binary features in this set.

(5) Rich Header. The Rich Header field in the PE file includes information regarding the identity or type of the object files and the compiler used to build the executable. Webster et al. [115] have shown that the Rich Header is useful for detecting different versions of malware, as malware authors often do not deliberately strip this header. In particular, they observed that "most packers, while sometimes introducing anomalies, did not often strip the Rich Header from samples." Based on our observation, as Table II shows, while Obsidium, kkrunchy, MPRESS, and PELock stripped the Rich Header for 70–80% of binaries in the *wild dataset*, other packers always kept this header, except for *AES-Encrypter*, which always produces the same header. We followed the procedure by Webster et al. [115] to encode this header into 66 integer features.

(6) Byte n-grams. Given that an executable file is a sequence of bytes, we extracted byte n-grams by considering every n consecutive bytes as an individual feature. Given the practical impossibility of storing the representation of n-grams for $n \geq 4$ in main memory, a feature selection process is needed [82]. Raff et al. [82] observed that 6-grams perform best over their dataset. We used the same strategy to select the most important 6-gram features, where each feature represents if the executable contains the corresponding 6-gram. We first randomly selected a set of 1,000 samples and computed the number of files containing each individual 6-gram. We observed 1,060,957,223 unique 6-grams in these samples. As Figure 10a in the Appendix shows, and as Raff et al. [82] observed, byte 6-grams follow a power-law type distribution, with 99.99% 6-grams occurring ten or fewer times. We reduced our set of candidate 6-grams by selecting 6-grams that occurred in more than 1% of the samples in the set, which results in 204,502 individual 6-gram features. Then, we selected the top 13,000 n-gram features based on the Information Gain (IG) measure [79], since our dataset roughly converges at this value, as depicted in Figure 10b.

(7) Opcode n-grams. We used the Capstone [12] disassembler to tokenize executables into sequences of opcodes and then built the opcode n-grams. While a small value may fail to detect complex malicious blocks of code, long sequences of opcodes can easily be avoided with simple obfuscation techniques [93]. Moreover, large values of n introduce a high performance overhead [49, 93]. For these reasons, similarly to most related work, we use sequences up to a length of four. We represent opcode n-grams by computing the TF-IDF [92] value for each sequence. While we could extract the assembly for all samples in the *wild dataset*, out of the 341,444 samples in the *lab dataset*, we could not disassemble 2,200 samples (see Table II). For these programs, we put `-1` as the value of opcode n-grams features. In total, we extracted 5,373,170 unique opcode n-grams, from which, only 51,942 n-grams occurred in more than 0.1% of executables in the *lab dataset* (Figure 10c). We only consider these opcode n-grams (reduction of 98.47%). Figure 10d presents the Information Gain (IG) measure of these opcode n-grams. We selected the top 2,500 opcode n-grams (based on IG value) with their TF-IDF weights as feature values, resulting into 2,500 float features.

(8) Strings. The (printable) strings contained in an executable may give valuable insights into the executable, such as file names, system resource information, malware signatures, etc. We leveraged the GNU *strings* program to extract the printable character sequences that are at least 4 characters long. We represent each printable string with a binary feature indicating if the executable contains the string. We observed 1,856,455,113 unique strings, from which more than 99.99% were seen in less than 0.4% of samples. After removing these rare strings, we obtained 16,900 binary features.

(9) File generic. We also computed the size of each sample (in bytes), and the entropy of the whole file. We further reference to this small family of features as "generic."

V. Experiments and Results

In this work, we aim to answer the following question: does static analysis on packed binaries provide *rich enough* features to a malware classifier? We analyze multiple facets of this question by performing a number of experiments. As explained in the introduction, even though we used several machine learning approaches (i.e., SVM, neural networks and decision tress), we only discuss the results of the random forest approach as (1) we observed similar findings for these approaches, with random forest being the best classifier in most experiments, and (2) random forest allows for better interpretation of the results compared to neural networks [35]. Following a linear search over different configurations of random forest, we found a suitable trade-off between learning time and test accuracy. Table XIX in the Appendix shows the parameters of the model.

Note that all malicious executables in our datasets are *packed*. Unless stated otherwise: **(1)** we always partition the

dataset into training and test sets with a 70%-30% split, and both the training and test sets are balanced over benign and malicious executables; **(2)** We repeat each experiment five times by randomly splitting the dataset into training and test sets each time, and average the results of all five rounds; **(3)** We use all 56,543 features to train the classifier; **(4)** We focus only on real-world packers (we do not include *AES-Encrypter* except for Experiment X).

We introduce and motivate research questions that help us answer our main hypothesis. For each, we describe one or more experiments followed by the corresponding results. Our results fit into four major findings, which we divide as follows. **(I)** Finding 1 and 3 may be intuitively known in the community, though mostly based on anecdotal experience. We confirm these findings with solid experiments. **(II)** Previous works have shown preliminary evidence of Finding 2, but with major limitations. We provide extensive evidence for this finding. **(III)** We present additional evidence for Finding 4, which is a fairly established fact confirmed by related work.

A. Effects of Packing Distribution During Training

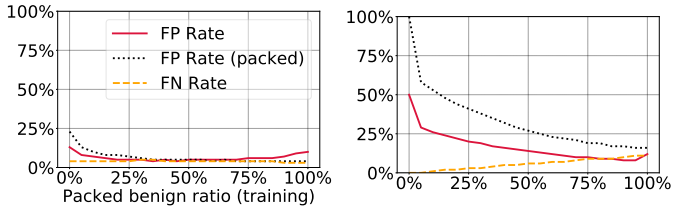
RQ1. Does a bias in the distribution of packers used in benign and malicious samples cause the classifier to learn specific packing routines as a sign of maliciousness?

RQ1 is important for two reasons: **(1)** Machine learning is increasingly being used for malware detection, while, as discussed in Section III-B, most related work does not specify considering *packed benign* executables, and the remaining few neglect the bias that may be introduced by the overlap between packers used in benign and malicious samples; **(2)** Nowadays, packing is also widespread in benign samples [84]. To answer RQ1, we conducted three experiments.

Experiment I: "no packed benign". We trained the classifier on 3,956 *unpacked benign* and 3,956 *packed malicious* executables from the *wild dataset*. The resulting classifier produced a false positive rate of 23.40% on 12,647 *packed benign* samples. It should be noted that the classifier is fairly well calibrated, with false negative and false positive rates of 3.82% and 2.64% for 440 (unseen) packed malicious and 440 (unseen) unpacked benign samples. While this is a naïve experiment, it delivers an important message: excluding *packed benign* samples from the training set makes the classifier biased towards interpreting packing as an indication of maliciousness, and such a classifier will produce a substantial number of false positives in real-world settings, where packing is also widespread in benign samples. This experiment shows that *packed benign* executables must be considered when training the classifier.

The overlap between packers that are used in benign and malicious samples may cause the classifier to distinguish between packing routines, i.e., packers. To further investigate this issue, we performed the following two experiments.

Experiment II: "packer classifier". We used the *lab dataset* to create a *packer* classifier. We defined nine classes for the classifier, one per packer. We trained and tested the classifier on datasets with samples uniformly distributed over all classes. In particular, we trained the classifier on 107,471 samples and evaluated it against 46,059 samples. Note that we discarded



(a) *wild dataset* **(b)** *lab dataset*

Fig. 3: Experiment "different packed ratios"

the benign and malicious labels of samples. The classifier maintained the precision and recall of 99.99% per class. This result shows that "packer classification" is a simple task for the classifier, which indicates that the lack of overlap between packers that are used in benign and malicious samples of the dataset might bias the classifier to associate specific packing routines with maliciousness.

Experiment III: "good-bad packers". We trained the classifier on a dataset in which benign samples are packed by four specific packers, and malicious samples are packed by the remaining five packers. We refer to these two non-overlapping subsets of packers as *good* and *bad* packers, respectively. Then, we tested the classifier on benign and malicious samples that are packed by *bad* and *good* packers, respectively. We repeated this experiment for each split of packers. The accuracy of the classifier varied from 0.01% to 12.57% across all splits, showing that the classifier was heavily biased to distinguish between *good* and *bad* packers.

Finding 1. The lack of overlap between packers used in benign and malicious samples will bias the classifier towards distinguishing between packing routines.

B. Packers vs. Malware Classification

RQ2. Do packers prevent machine-learning-based malware classifiers that leverage only static analysis features?

It is commonly assumed that machine learning combined with only static analysis is not able to distinguish between benign and malicious samples that are packed. We performed the following three experiments to validate this assumption.

Experiment IV: "different packed ratios (wild)". We trained the classifier on different subsets of the *wild dataset* by increasing the ratio of packed benign executables in the training set, with steps of 0.05. The "packed benign ratio" is defined as the proportion of benign samples that are packed. We always used datasets of the same size to fairly compare the trained models with each other, and tested models against the test set with a "wild ratio" of packed benign samples, i.e., the maximum ratio of packed benign executables that the vendor has seen in the wild (i.e., 50% packed benign, see Figure 1). As Figure 3a shows, increasing the packed benign ratio helps the classifier to maintain a lower false positive rate on *packed* samples, while the false negative rate slightly increases. However, the false positive rate on *unpacked* samples considerably increases from 3.18% to 16.24% as the classifier sees fewer unpacked samples, which indicates that a classifier that is trained only on packed samples cannot achieve high accuracy on unpacked samples. As illustrated by Table IV, we always used training sets of the same size, uniformly distributed over benign and malicious executables. Table IV also demonstrates that as we

TABLE IV: Experiment "different packed ratios (wild)." Each row represents features that are important to the classifier.

PB Ratio	Training Set #B (packed)	#B (unpacked)	#M	# Features used by the classifier (Top 50) import	dll	rich	sections	header	strings	byte n-grams	opcode n-grams	generic	all
.0	0	3,077	3,077	1,446 (0)	53 (0)	37 (**3**)	148 (0)	20 (0)	2,278 (**1**)	2,674 (**44**)	2,067 (**2**)	2 (0)	8,725 (50)
.2	615	2,462	3,077	1,560 (**1**)	50 (0)	49 (0)	173 (0)	18 (0)	2,661 (**1**)	2,980 (**48**)	2,088 (0)	2 (0)	9,581 (50)
.4	1,231	1,846	3,077	1601 (**1**)	62 (0)	51 (0)	183 (0)	20 (0)	2,742 (0)	3,012 (**49**)	2,084 (0)	2 (0)	9,757 (50)
.6	1,846	1,231	3,077	1,571 (**1**)	55 (0)	45 (0)	200 (0)	19 (0)	2,754 (0)	2,976 (**49**)	2,081 (0)	2 (0)	9,703 (50)
.8	2,462	615	3,077	1,608 (**1**)	59 (0)	49 (0)	191 (0)	18 (0)	2,797 (0)	3,022 (**49**)	2,117 (0)	2 (0)	9,863 (50)
1.	3,077	0	3,077	1,404 (0)	50 (0)	42 (0)	200 (0)	20 (0)	2,662 (**1**)	2,911 (**49**)	2,081 (0)	2 (0)	9,372 (50)

TABLE V: Experiment "different packed ratios (lab)"

PB Ratio	Training Set #B (packed)	#B (unpacked)	#M	# Features used by the classifier (Top 50) import	dll	rich	sections	header	strings	byte n-grams	opcode n-grams	generic	all
.0	0	3,077	3,077	381 (**8**)	19 (0)	29 (**1**)	86 (**5**)	14 (0)	730 (**12**)	897 (**24**)	861 (0)	2 (0)	3,019 (50)
.2	615	2,462	3,077	508 (6)	48 (0)	49 (**1**)	158 (**3**)	24 (0)	2,463 (**2**)	2,729 (**33**)	2,034 (**3**)	2 (**2**)	8015 (50)
.4	1,231	1,846	3,077	504 (**1**)	56 (0)	46 (0)	161 (**2**)	25 (0)	2,871 (0)	2,939 (**44**)	2,195 (**1**)	2 (2)	8,799 (50)
.6	1,846	1,231	3,077	517 (0)	62 (0)	48 (**1**)	169 (**3**)	23 (**1**)	3,148 (0)	2,999 (**43**)	2,267 (0)	2 (**2**)	9,235 (50)
.8	2,462	615	3,077	496 (0)	77 (0)	47 (0)	183 (**10**)	25 (**3**)	3,372 (0)	3,151 (**35**)	2,273 (0)	2 (**2**)	9,626 (50)
1.	3,077	0	3,077	388 (0)	80 (0)	51 (**1**)	174 (**14**)	26 (**4**)	3,412 (0)	3,094 (**29**)	2,183 (0)	2 (**2**)	9,410 (50)

TABLE VI: Experiment "single packer"

Packer	FPR (%)	FNR (%)	ROC AUC	F-1 Score	# Features used by the classifier (Top 50) import	dll	rich	sections	header	strings	byte n-grams	opcode n-grams	generic	all
PELock	**7.21%**	2.70%	0.95	0.95	752 (0)	118 (0)	33 (0)	101 (**1**)	20 (0)	1,409 (**1**)	2,188 (**48**)	1709 (0)	2 (0)	6,332 (50)
PECompact	**9.93%**	**6.02%**	0.93	0.93	565 (0)	81 (0)	56 (**3**)	110 (**24**)	22 (**5**)	2,856 (0)	2,974 (**16**)	1,868 (0)	2 (**2**)	8,534 (50)
Obsidium	5.53%	4.39%	0.95	0.95	507 (0)	4 (0)	0 (0)	54 (**14**)	10 (**5**)	2,546 (0)	2,274 (**30**)	1,110 (0)	2 (**1**)	6,507 (50)
Petite	3.54%	3.17%	0.97	0.97	769 (0)	173 (1)	54 (**1**)	123 (**9**)	22 (**1**)	1,708 (0)	2,403 (**38**)	1,866 (0)	2 (0)	7,120 (50)
tElock	**6.06%**	**8.85%**	0.93	0.93	4 (0)	3 (0)	59 (**2**)	200 (**40**)	22 (**5**)	2,419 (0)	2,628 (**2**)	1,027 (0)	2 (**1**)	6,364 (50)
Themida	**6.45%**	3.23%	0.95	0.95	2 (0)	2 (0)	52 (0)	127 (0)	21 (0)	4,091 (0)	3,678 (**50**)	1,190 (0)	2 (0)	9,165 (50)
MPRESS	**8.10%**	4.18%	0.94	0.93	633 (0)	145 (0)	0 (0)	45 (**3**)	20 (0)	1,427 (0)	2,861 (**47**)	2,130 (0)	2 (0)	7,263 (50)
kkrunchy	**9.38%**	**6.93%**	0.92	0.92	0 (0)	0 (0)	0 (0)	29 (**23**)	22 (**5**)	997 (0)	1,371 (**20**)	1,633 (0)	2 (**2**)	4,054 (50)
UPX	3.95%	4.98%	0.96	0.96	750 (1)	175 (0)	52 (**1**)	37 (**23**)	19 (**6**)	3,913 (0)	5,058 (**17**)	1,217 (0)	2 (**2**)	11,223 (50)

increase the ratio of packed benign executables in the training dataset, byte n-gram features play a much more significant role compared to other feature families.

Note that the performance of the classifier might be due to features that do not necessarily capture the real behavior of samples. For example, packed benign executables might be packed by a different set of packers compared to malicious executables. Table XII in the Appendix shows that the distribution of packers being used by benign samples is very different from packers used by malicious samples. For example, there are 13 packers for which we found signatures only in malicious executables in our dataset (e.g., FSG, VMProtect, dxpack, and PE-Armor). Although this discrepancy might not hold for the entire *wild dataset*, it indicates that such a difference may make the classifier biased to distinguish between *good* and *bad* packers, and thus, results can be misleading.

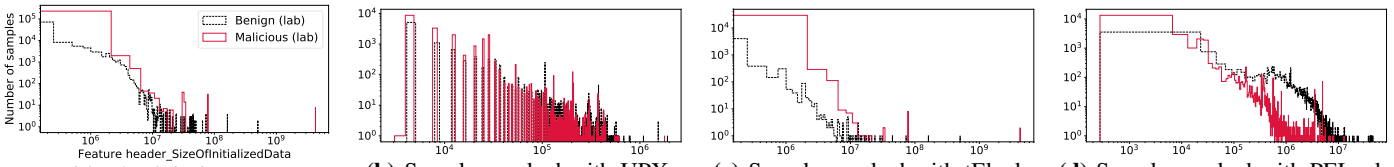
Experiment V: "different packed ratios (lab)". To mitigate the uncertainty about the distribution of packers in the dataset, we repeated the previous experiment on the *lab dataset* combined with *unpacked benign* executables from the *wild dataset*. We selected packed samples uniformly distributed over the packers for training and test sets. Surprisingly, unlike the popular assumption that packing greatly hinders machine learning models based on static features, the classifier performed better than our expectations, even when there was no unpacked sample in the training set, with false positive and false negative rates of 12.24% and 11.16%, respectively. As Figure 3b presents, the false positive rate for *packed* executables decreases from 99.76% to 16.03% as we increase the ratio of packed benign samples in the training dataset. Unsurprisingly, when there is no *packed benign* executable in the training set, the classifier detects everything packed by the packers in the *lab dataset* as malicious. Table V presents the important features for the classifier based on the ratio of packed benign executables in the dataset. Byte n-grams and PE sections are the most useful families of features. We focused on one packer at a time in the next experiment to identify useful features for each packer.

Experiment VI: "single packer". For each packer, we trained and tested the classifier on only benign and malicious executables that we packed with that packer. Table VI presents the performance of the classifier corresponding to each individual packer. In all cases, the classifier performed relatively well, with byte n-gram and PE section features as the most useful.

We are also curious to see how packers preserve information when packing programs. To this end, for each packer, we built different models by training the classifier on *one family* of features at a time. In particular, we observed the following:

Rich Header. The Rich Header family alone helps the classifier to achieve relatively high accuracy, except for those packers that often strip this header (see Table II). As an example, using only Rich Header features, the classifier that is trained on executables packed with Themida maintains an accuracy of 89.03%. Webster et al. [115] also showed that the Rich Header is useful for detecting similar malware.

API imports. If we use tElock, Themida, and kkrunchy, API import features are no longer useful for malware detection. However, other packers preserve some information in these features. For example, we trained the classifier on executables that are packed with UPX and observed an accuracy of 89.11%. We noticed a similar trend for the DLL imports family. Among the packers affected by these features, the number of API imports was one of the most important features for the classifier. Figure 5 presents the distribution of this feature for UPX, Petite, and PECompact. We also observed specific API imports to be very distinguishing, like `ShellExecuteA`. Table XXII in the Appendix shows the number of benign



(a) The *lab dataset*. **(b)** Samples packed with UPX. **(c)** Samples packed with tElock. **(d)** Samples packed with PELock.

Fig. 4: The histogram of the feature `header_SizeOfInitializedData`.

and malicious samples that import each of these APIs. For example, Obsidium keeps importing the API `FreeSid` when packing a binary, or it is well-known that UPX keeps one API import from each DLL that the original binary imports to avoid the complexity of loading DLLs during execution. This indicates that packers preserve some information in the Import Directory Table when packing programs.

Opcode n-grams. For each of Obsidium, tElock, and Themida, we trained the classifier using opcode n-grams, and the accuracy dropped to ∼50%. However, we observed the accuracy of 89.01%, 88.72%, 88.27%, 77.25%, 77.04%, and 65.75% while training on samples packed with Petite, PELock, Mpress, kkrunchy, UPX, and PECompact, respectively.

PE headers. For all packers, the classifier had an accuracy above 90%. In particular, the "size of the initialized data" was the most important feature in all cases but UPX. However, the distribution of this feature differs across packers (see Figure 4). While malicious samples packed with kkrunchy, Obsidium, PECompact, tElock, and Themida have bigger initialized data compared to benign executables, the same malicious samples, packed with MPRESS, PELock, and Petite have smaller initialized data. Interestingly, malicious samples packed with UPX follow a distribution very similar to the distribution observed for benign samples.

PE sections. The accuracy of the classifier was above 90% for all packers, varying from 91.23% to 96.72%. As Figure 7 shows, the importance weights of features significantly differ across different models. For example, the entropy of the entry point section is a very important feature for the classifier that is trained on MPRESS. However, this feature is not helpful when we train the classifier on samples packed with Obsidium, Themida, or PELock. The entry point of binaries packed with MPRESS resides in the second section, `.MPRESS2`, for which benign and malicious executables have a mean entropy of 6.16 and 5.77. However, for Obsidium, the entry point section always has a high entropy, close to 8.

Finding 2. Packers preserve some information when packing programs that may be "useful" for malware classification, however, such information does not necessarily represent the real nature of samples.

We should emphasize that related work has provided preliminary evidence of Finding 2. Jacob et al. [44] showed that some packers employ weak encryption, which can be used to detect similar malware samples packed with these packers. Webster et al. [115] also showed that some packers do not touch the rich header, leaving it viable for malware detection.

C. Malware Classification in Real-world Scenarios

RQ3. Can a classifier that is carefully trained and not biased towards specific packing routines perform well in real-world scenarios?

RQ3 is a key question in the development of machine-learning-based malware classifiers. In this work, we focus on three specific issues:

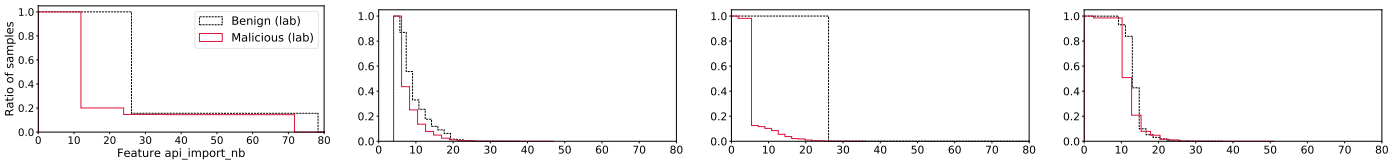
- *Generalization.* Nowadays, runtime packers are evolving, and malware authors often tend to use their own custom packers [34, 66, 110]. This raises serious doubt about how a classifier performs against previously unseen packers.
- *Strong & Complete Encryption.* Malware authors might customize the packing process to remove the static features that machine-learning-based classifiers can reasonably be expected to leverage. Can malware classifiers be effective in the presence of strong and complete encryption?
- *Adversarial Examples.* Despite their limited scope, recent work [33, 41, 89] has shown that machine-learning-based malware detectors are vulnerable to adversarial examples. Is it possible to use the learned model to drive evasion?

To investigate the *generalization* question, we carried out the next three experiments.

Experiment VII: "wild vs. packers". First, we trained the classifier on a dataset with a "wild ratio" of packed benign samples extracted from the *wild dataset*, and tested it on the *lab dataset*. As Table VIII shows, the classifier performed poorly against all packers, with the highest accuracy being 78.19% against Themida. This is interesting, as we knew that at least 50% of the packers in our dataset keep the Rich Header, and, therefore, the classifier still should have maintained high accuracy based on the earlier results. We argue that this happened because the classifier chose features with more information gain, and, while testing on the *lab dataset*, those features are not helpful anymore. In fact, we trained the classifier using only the Rich Header, and the classifier's accuracy against packers that keep the Rich Header increased considerably, up to over 90%.

Experiment VIII: "withheld packer". Second, we performed several rounds of experiments on the *lab dataset*, in which we withheld one packer from the training set and then evaluated the resulting classifier on packed executables generated by this packer (one round for each of the nine packers). To have a fair comparison between rounds, we fixed the size of the training set to 83,760, by selecting 5,235 benign and 5,235 malicious executables for each of the packers. We then tested the classifier on 5,235 benign and 5,235 malicious executables packed with the withheld packer. As Table VII shows, except for the three noticeable cases of PECompact, tElock, and kkrunchy, the classifier performed relatively well, with an F-1 score ranging from 0.90 to 0.95.

In all cases, we identified byte n-gram features extracted from `.CAB` file headers (reside in the resource sections) as the most important features. There are 6,717 benign and 1,269 malicious executables having these features enabled in the *wild dataset*. In the previous experiment, the classifier did not learn these features as there were more distinguishing features. However, as packers mostly keep headers of resources despite



(a) The *lab dataset*. **(b)** Packed with UPX. **(c)** Packed with PECompact. **(d)** Packed with Petite.

Fig. 5: The CCDF of the feature `api_import_nb`, i.e., the number of API imports.

TABLE VII: Experiment "withheld packer"

Withheld Packer	All features			NO byte n-grams		
	FPR (%)	FNR (%)	F-1	FPR (%)	FNR (%)	F-1
PELock	7.30%	3.74%	0.95	**26.06%**	1.51%	0.88
PECompact	**47.49%**	2.14%	**0.80**	**42.75%**	2.83%	**0.81**
Obsidium	**17.42%**	3.32%	0.90	**70.09%**	0.73%	**0.74**
Petite	5.16%	4.47%	0.95	**12.45%**	4.22%	0.92
tElock	**43.65%**	2.02%	**0.77**	**73.98%**	1.07%	**0.72**
Themida	6.21%	3.29%	0.95	**21.28%**	10.37%	0.85
MPRESS	5.43%	4.53%	0.95	**28.65%**	1.87%	0.87
kkrunchy	**83.06%**	2.50%	**0.70**	**55.97%**	0.38%	**0.78**
UPX	11.21%	4.34%	0.92	**17.52%**	9.02%	0.87

TABLE VIII: Experiment "wild vs. packers"

Packer	All features			Rich Header (only)		
	FPR (%)	FNR (%)	F-1	FPR (%)	FNR (%)	F-1
PELock	60.79%	0.0%	0.80	99.72%	0.0%	**0.67**
PECompact	66.48%	0.23%	0.76	**22.56%**	1.44%	0.89
Obsidium	82.10%	0.0%	0.73	100.0%	0.0%%	**0.67**
Petite	74.85%	0.02%	0.78	**8.44%**	1.54%	0.95
tElock	99.28%	0.03%	0.67	**32.38%**	1.35%	0.86
Themida	43.41%	0.21%	0.80	**12.23%**	1.44%	0.94
MPRESS	89.93%	1.23%	0.69	100.0%	0.0%	**0.67**
kkrunchy	100.0%	0.0%	0.67	100.0%	0.0%	**0.67**
UPX	50.46%	1.72%	0.79	**18.32%**	1.86%	0.91

the encryption of the body, this initial bias is intensified as we packed each sample with multiple packers. In particular, there are 28,765 benign and 2,428 malicious executables in the *lab dataset* that include these sequences of bytes. However, for PECompact the situation is a bit different, as we could pack only 1,095 benign and 451 malicious samples that have `.CAB` headers. For tElock, we could pack only 181 benign and 444 malicious samples with `.CAB` headers. This explains why the accuracy of the classifier is low against PECompact and tElock. We looked at the most important features when we withheld kkrunchy in the learning phase, and we found that byte n-grams extracted from the *version info* field of resources are very helpful for the classifier. Other packers usually keep this information, hence the classifier learns it, but fails to utilize that against samples packed with kkrunchy, as the packer strips this information. We repeated the experiment by excluding byte n-grams features, and the accuracy of the classifier dropped significantly in all cases, except when we withheld PECompact or kkrunchy (see Table VII).

Experiment IX: "lab against wild". In this third experiment, we trained the classifier on the *lab dataset* and evaluated it on *packed executables in the wild dataset*. This experiment is important as malware authors often prefer customized packing routines to off-the-shelf packers [34, 66, 110]. To avoid any bias in our dataset toward any particular packer, benign and malicious executables were uniformly selected from the various packers. We observed the false negative rate of 41.84%, and false positive rate of 7.27%.

Experiments VII, VIII, and IX demonstrate that when using static analysis features, the classifier is not guaranteed to generalize well to previously unseen packers. As a preliminary step towards the *Strong & Complete Encryption* issue, we performed the following experiment.

Experiment X: "Strong & Complete Encryption". In this experiment, we trained the classifier on 11,929 benign and 11,929 malicious executables packed with *AES-Encrypter* and evaluated it against 5,113 benign and 5,113 malicious executables packed with *AES-Encrypter*. As *AES-Encrypter* encrypts the whole executable with AES, we would expect that static analysis features are no longer helpful for a static malware classifier. Surprisingly, the classifier performed better than a random guess just because of two features, "`file size`" and "`file entropy`," with accuracy of 72.67%. As benign samples are bigger in the *wild dataset*, obviously, packed benign executables are still larger than packed malicious executables as *AES-Encrypter* just encrypts the original binary. Also, the entropy of the packed executable is affected as a bigger overlay increases the entropy of the packed program more. All other static analysis features are the same across executables packed with *AES-Encrypter*, except for byte n-grams and strings features, as executables have different (encrypted) overlays. Since we have more malicious samples in the *wild dataset*, our feature selection procedures for extracting byte n-grams and strings (see Section IV-C) tend to select those features that appear in malicious samples with a higher probability, thus, we expect that the accuracy of the classifier is still slightly better than 50%. In particular, removing the features "`file size`" and "`file entropy`" from the dataset resulted in a classifier with an accuracy of 56.85%. In fact, we repeated the feature selection procedure for a balanced dataset of only executables packed with *AES-Encrypter*, and we got an accuracy of 50% for the classifier when removing these two features.

Experiment X raises serious doubts about machine learning classifiers. When packing hides all information about the original binary until execution, the classifier has no choice but to classify any sample packed by such a packer as malicious. This is an issue, as packing is increasingly being adopted by legitimate software [84].

Experiment XI: "adversarial samples". Recent work [33, 41, 89] has shown that machine-learning-based malware detectors, especially those that are based on only static analysis features, are vulnerable to adversarial samples. In our case, this issue becomes magnified as packing causes machine learning classifiers to make decisions based on features that are not directly derived from the actual (unpacked) program. Therefore, generating such adversarial samples would be easier for an adversary.

TABLE IX: The false positive and false negative rates (%) of six machine-learning-based engines integrated with VirusTotal.

Packer	AV1		AV2		AV3		AV4		AV5		AV6	
	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR	FPR	FNR
PELock	81.48	18.32	72.35	28.26	81.49	19.09	88.34	12.84	89.96	10.29	91.14	8.84
PECompact	81.81	18.36	72.18	28.31	80.53	19.55	88.39	12.06	89.74	10.06	90.88	8.89
Obsidium	79.01	22.24	70.18	30.54	77.03	24.42	83.69	17.42	67.53	32.71	86.55	13.82
Petite	78.73	20.99	68.48	31.07	74.52	25.5	84.79	16.11	72.13	26.92	85.45	14.68
tElock	78.92	20.49	67.49	30.77	74.51	25.06	84.37	14.58	72.53	26.53	84.79	14.17
Themida	79.18	21.58	70.18	30.45	76.95	23.88	84.13	15.95	67.5	33.88	86.23	14.28
MPRESS	82.3	18.58	72.75	28.44	82.04	19.68	88.37	12.73	89.94	9.43	91.58	8.94
kkrunchy	78.79	21.15	69.71	30.32	77.27	23.28	83.11	16.89	67.29	32.07	86.72	13.98
UPX	78.37	22.36	70.04	30.62	76.04	23.72	82.8	17.01	67.34	33.07	85.54	13.98
AES-Encrypter	79.77	21.27	69.14	31.27	75.47	24.4	85.29	15.2	73.25	26.71	85.69	14.19

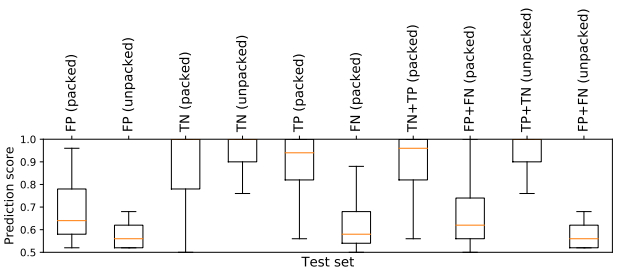


Fig. 6: Confidence of the "best possible" classifier on false positives, false negatives, true positives, and true negatives.

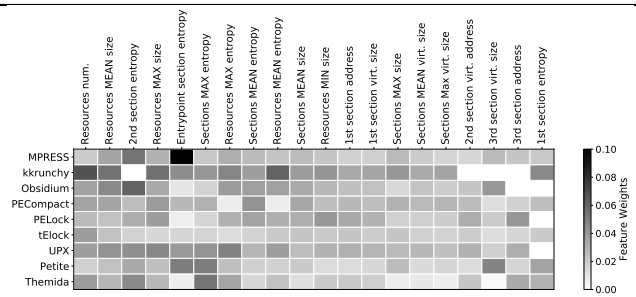


Fig. 7: Experiment "single packer." The weights of the top 20 features while training on only PE sections features.

In this experiment, first we carefully trained the classifier on 3,956 unpacked benign, 3,956 packed benign, and 7,912 malicious executables whose packed benign and malicious samples are uniformly distributed over the same packers from the *lab dataset* and *packed executables in the wild*. We showed that such a classifier is not biased towards detecting specific packing routines as a sign of maliciousness. As expected, the classifier performed relatively well in the evaluation, with false positive and false negative rates of 9.70% and 5.33%, respectively. Figure 6 shows the box and whisker plot of the classifier's confidence score on the test set. The mean confidence of the classifier for packed and unpacked executables that are classified correctly is 0.89 and 0.93, respectively. For benign samples that the classifier misclassified (false positives), the mean confidence is 0.68 and 0.58 for packed and unpacked samples, respectively.

Then, we generated adversarial samples from all 2,494 malicious samples that the classifier detected as malicious (i.e., true positives). To achieve this, we identified byte n-gram and string features that occurred more in benign samples and injected the corresponding bytes into the target program without affecting its behavior. We verified this by analyzing the sample with the *ANY.RUN* [4] sandbox. By injecting 34.24 (69.92) benign features on average, we managed to generate 2,483 (1,966) adversarial samples that cause the classifier to make false predictions with a confidence greater than 0.5 (0.9). We expect that a more complex attack is needed when the classifier is trained using features extracted from dynamic analysis, which represent the sample's behavior.

Finding 3. Although we observed that static analysis features combined with machine learning can distinguish between packed benign and packed malicious samples, such a classifier will produce intolerable errors in real-world settings.

Recently, a group of researchers found a very similar way to subvert Cylance's AI-based anti-malware engine [1, 105]. They developed a "global bypass" method that works with almost any malware to fool the Cylance engine. The evasion technique involves simply taking strings from an online gaming program and appending them to known malware, like WannaCry. The major problem that plagued Cylance was that behaviors that are common in malware are also common in games. Games use these techniques for various reasons, e.g., to prevent cheating or reverse engineering. Tuning the system to flag the malware but not such benign programs is quite difficult and prone to more errors, which in this case, confront Cylance's engine with a dilemma, either produce high false positives for games or inherit a bias towards them.

D. Anti-malware Industry vs. Packers

RQ4. How is the accuracy of real-world anti-malware engines that leverage machine learning combined with static analysis features affected by packers?

In today's world, legitimate software authors pack their products. Therefore, it is no longer acceptable for anti-malware products to detect anything packed as malicious. RQ4 is important because most machine-learning-based approaches rely on labels from VirusTotal in the absence of a reliable and fresh ground-truth dataset [22, 86, 88, 97]. To this end, we identified six products on VirusTotal that, either on the corresponding company's website or on a VirusTotal blog post, are described as machine-learning-based malware detectors that use only static analysis features. It should be noted that, while VirusTotal clearly discourages using their service to perform anti-malware comparative analyses [113], in the next experiment, we aim only to see how these engines assign labels to packed samples in general. We do not intend to compare these tools with each other or against another tool.

Experiment XII: "anti-malware industry". In February 2019, we submitted 6,000 benign and 6,000 malicious executables packed with each packer from the *lab dataset* to VirusTotal to evaluate these six anti-malware products. As Table IX shows clearly, all six engines have learned to associate packing with maliciousness. Other engines on VirusTotal also produced a similarly high error rate as these six engines. As we discussed in Section II, related work have published results showing similar trend [69, 116]. This experiment indicates that as packing is being used more often in legitimate software [84], unless the anti-malware industry does better than detecting packing, benign and malicious software are going to be increasingly misclassified.

Finding 4. Machine-learning-based anti-malware engines on VirusTotal detect packing instead of maliciousness.

VI. Discussion

We showed that machine-learning-based anti-malware engines on VirusTotal produce a substantial number of false positives on packed binaries, which can be due to the limitations discussed in this work. This is especially a serious issue for machine-learning-based approaches that frequently rely on labels from VirusTotal [22, 86, 88, 97], causing an endless loop in which new approaches rely on *polluted datasets*, and, in turn, generate *polluted datasets* for future work.

One might say that this general issue with packing can be avoided by whitelisting samples based on code-signing certificates. However, we have seen that valid digital signatures allowed malware like LockerGoga, Stuxnet, and Flame to bypass anti-malware protections [50]. It should be noted that although we showed that packer classification is an easy task for the classifier to learn over our dataset, *packing detection*, in general, is a challenging task [5, 5, 59, 102], especially when malware authors use customized packers that evolve rapidly [34, 66, 110]. While using dynamic analysis features seems necessary to mitigate the limitations of static malware detectors, malware could still force malware detectors to fall back on static features by using sandbox evasion [116]. For example, Jana et al. [45] discovered 45 evasion exploits against 36 popular anti-malware scanners by targeting file processing in malware detectors. All these issues suggest that malware detection should be done using a hybrid approach leveraging both static and dynamic analysis.

Limitations. As encouraged by Pendlebury et al. [75] and Jordaney et al. [46], malware detectors should be evaluated on how they deal with concept drift. We have observed that machine learning combined with static analysis generalizes poorly to unseen packers, however, we did not consider time constraints in our experiments, which we leave as future work. Also, we focused only on Windows x86 executables in this paper, but our hypothesis might also be applicable to Android apps, for which packing is also getting more common [25].

VII. Related Work

The theoretical limitations of malware detection have been studied widely. Early work on computer viruses [18, 19] showed that the existence of a precise virus detector that detects all computer viruses implies a decision procedure for the halting problem. Later, Chess et al. [14] presented a polymorphic virus that cannot be precisely detected by any program. Similarly, several critical techniques of static and dynamic analysis are undecidable [55, 100], including detection of unpacking execution.

Moser et al. [67] proposed a binary obfuscation scheme based on *opaque constants* that scrambles a program's control flow and hides data locations and usage. They showed that static analysis for the detection of malicious code can be evaded by their approach in a general way. Christodorescu et al. [16] showed that three anti-malware tools can be easily evaded by very simple obfuscation transformations. Later, they developed a system for evaluating anti-malware tools against obfuscation transformations commonly used to disguise malware [17]. ADAM [119] and DroidChameleon [85] used similar transformation techniques to evaluate commercial Android anti-malware tools. In particular, DroidChameleon's results on ten anti-malware products show that none of these is resistant to common and simple malware transformation methods. Bacci et al. [6] showed that while dynamic-analysis-based detection demonstrates equal performance on both obfuscated and non-obfuscated Android malware, static-analysis-based detection has a poor performance on obfuscated samples. Although they showed that this effect can be mitigated by using obfuscated malicious samples in the learning phase, no obfuscated benign sample is used, which raises the doubt that the classifier might have learned to detect obfuscation. Hammad et al. [36] recently studied the effects of code obfuscation on Android apps and anti-malware products and found that most anti-malware products are severely impacted by simple obfuscations.

VIII. Conclusions

In this paper, we have investigated the following question: does static analysis on *packed* binaries provide a *rich enough* set of features to a malware classifier? We first observed that the distribution of the packers in the training set must be considered, otherwise the lack of overlap between packers used in benign and malicious samples might cause the classifier to distinguish between packing routines instead of behaviors. Different from what is commonly assumed, packers preserve information when packing programs that is "useful" for malware classification, however, such information does not necessarily capture the sample's behavior. In addition, such information does not help the classifier to (1) generalize its knowledge to operate on previously unseen packers, and (2) be robust against trivial adversarial attacks. We observed that *static* machine-learning-based products on VirusTotal produce a high false positive rate on packed binaries, possibly due to the limitations discussed in this work. This issue becomes magnified as we see a trend in the anti-malware industry toward an increasing deployment of machine-learning-based classifiers that only use static features.

To the best of our knowledge, this work is the first comprehensive study on the effects of packed Windows executables on machine-learning-based malware classifiers that use only static analysis features. The source code and our dataset of 392,168 executables are publicly available at `https://github.com/ucsb-seclab/packware`.

ACKNOWLEDGMENTS

We would like to thank our reviewers for their valuable comments and input to improve our paper. This material is based on research sponsored by a gift from Intel, by the National Science Foundation grant #CNS-1704253, and by DARPA under agreement number #FA8750-19-C-0003. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. This research has also received funding from the European Research Council (ERC) under grant agreement No 771844 – BitCrumbs, from SBA Research (SBA-K1), which is funded within the framework of COMET – Competence Centers for Excellent Technologies by BMVIT, BMDW, and from the federal state of Vienna, managed by the FFG. The financial support by the Christian Doppler Research Association, the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is also gratefully acknowledged.

REFERENCES

[1] "Researchers Easily Trick Cylance's AI-Based Antivirus Into Thinking Malware Is 'Goodware'," https://www.vice.com/en_us/article/9kxp83/researchers-easily-trick-cylances-ai-based-antivirus-into-thinking-malware-is-goodware, July 2019.

[2] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, "N-gram-based detection of new malicious code," in *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, vol. 2. IEEE, 2004, pp. 41–42.

[3] H. S. Anderson and P. Roth, "Ember: An open dataset for training static pe malware machine learning models," *arXiv preprint arXiv:1804.04637*, 2018.

[4] ANY.RUN, "Interactive malware analyzer," https://any.run/, (Accessed: 2019-1-17).

[5] R. Arora, A. Singh, H. Pareek, and U. R. Edara, "A Heuristics-based Static Analysis Approach for Detecting Packed PE Binaries," *International Journal of Security and Its Applications*, 2013.

[6] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Impact of Code Obfuscation on Android Malware Detection based on Static and Dynamic Analysis," in *Proc. of the International Conference on Information Systems Security and Privacy*, 2018.

[7] D. Bilar, "Opcodes As Predictor for Malware," *International Journal of Electronic Security and Digital Forensics*, vol. 1, no. 2, 2007.

[8] C. M. Bishop *et al.*, *Neural networks for pattern recognition*. Oxford university press, 1995.

[9] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions," 2015.

[10] T. Brosch and M. Morgenstern, "Runtime Packers: The Hidden Problem?" *Black Hat USA*, 2006.

[11] D. Bueno, K. J. Compton, K. A. Sakallah, and M. Bailey, "Detecting Traditional Packers, Decisively," in *Proc. of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2013.

[12] Capstone, "Disassembler," https://www.capstone-engine.org/, (Accessed: 2018-11-20).

[13] B. Cheng, J. Ming, J. Fu, G. Peng, T. Chen, X. Zhang, and J.-Y. Marion, "Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018.

[14] D. M. Chess and S. R. White, "An Undetectable Computer Virus," in *Proceedings of the Virus Bulletin Conference (VB)*, vol. 5, 2000.

[15] Y.-s. Choi, I.-k. Kim, J.-t. Oh, and J.-c. Ryou, "Encoded Executable File Detection Technique via Executable File Header Analysis," *International Journal of Hybrid Information Technology*, 2009.

[16] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," in *Proc. of the USENIX Security Symposium*, 2003.

[17] ——, "Testing malware detectors," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 34–44, 2004.

[18] F. Cohen, "Computer Viruses: Theory and Experiments," *Computers & Security*, vol. 6, no. 1, 1987.

[19] ——, "Computational Aspects of Computer Viruses," *Computers & Security*, vol. 8, no. 4, 1989.

[20] W. W. Cohen, "Learning trees and rules with set-valued features," in *AAAI/IAAI, Vol. 1*, 1996, pp. 709–716.

[21] K. Coogan, S. Debray, T. Kaochar, and G. Townsend, "Automatic Static Unpacking of Malware Binaries," in *Proc. of the Working Conference on Reverse Engineering (WCRE)*, 2009.

[22] F. Copty, M. Danos, O. Edelstein, C. Eisner, D. Murik, and B. Zeltser, "Accurate malware detection by extreme abstraction," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 101–111.

[23] S. Debray and J. Patel, "Reverse Engineering Self-Modifying Code: Unpacker Extraction," in *Proc. of the Working Conference on Reverse Engineering (WCRE)*, 2010.

[24] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 51–62.

[25] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, "Things You May Not Know About Android (Un)Packers: A Systematic Study based on Whole-System Emulation," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[26] T. Dube, R. Raines, G. Peterson, K. Bauer, M. Grimaila, and S. Rogers, "Malware Target Recognition via Static Heuristics," *Computers & Security*, vol. 31, no. 1, 2012.

[27] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A Survey on Automated Dynamic Malware Analysis Techniques and Tools," *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, 2012.

[28] Y. Elovici, A. Shabtai, R. Moskovitch, G. Tahan, and C. Glezer, "Applying machine learning techniques for detection of malicious code in network traffic," in *Annual Conference on Artificial Intelligence*. Springer, 2007, pp. 44–50.

[29] ENDGAME, "Endpoint protection," https://www.endgame.com, (Accessed: 2018-12-26).

[30] Exeinfo PE, "Signature-based packer detector," http://exeinfo.atwebpages.com/, (Accessed: 2019-01-07).

[31] C. Feng and D. Michie, "Machine learning of rules and trees," *Machine learning, neural and statistical classification*, pp. 50–83, 1994.

[32] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility Is Not Transparency: VMM Detection Myths and Realities," in *Proc. of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2007.

[33] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 62–79.

[34] F. Guo, P. Ferrie, and T.-c. Chiueh, "A Study of the Packer Problem and Its Solutions," in *Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[35] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "Lemna: Explaining deep learning based security applications," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 364–379.

[36] M. Hammad, J. Garcia, and S. Malek, "A Large-Scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-Malware Products," in *Proc. of the International Conference on Software Engineering (ICSE)*, 2018.

[37] S. Han, K. Lee, and S. Lee, "Packed PE File Detection for Malware Forensics," in *Proc. of the International Conference on Computer Science and its Applications (CSA)*, 2009.

[38] I. U. Haq, S. Chica, J. Caballero, and S. Jha, "Malware Lineage in the Wild," *arXiv preprint 1710.05202*, 2017.

[39] O. Henchiri and N. Japkowicz, "A feature selection and evaluation scheme for computer virus detection," in *Sixth International Conference on Data Mining (ICDM'06)*. IEEE, 2006, pp. 891–895.

[40] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[41] W. Hu and Y. Tan, "Black-box attacks against rnn based malware detection algorithms," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[42] M. Hurier, K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "On the Lack of Consensus in Anti-Virus Decisions: Metrics and Insights on Building Ground Truths of Android Malware," in *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[43] I. Incer, M. Theodorides, S. Afroz, and D. Wagner, "Adversarially robust malware detection using monotonic classification," in *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*. ACM, 2018, pp. 54–63.

[44] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna, "A Static, Packer-agnostic Filter to Detect Similar Malware Samples," in *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2013.

[45] S. Jana and V. Shmatikov, "Abusing File Processing in Malware Detectors for Fun and Profit," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2012.

[46] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting Concept Drift in Malware Classification Models," in *Proc. of the USENIX Security Symposium*, 2017.

[47] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A Hidden Code Extractor for Packed Executables," in *Proc. of the ACM Workshop on Recurring Malcode (WORM)*, 2007.

[48] A. Kantchelian, M. C. Tschantz, S. Afroz, B. Miller, V. Shankar, R. Bachwani, A. D. Joseph, and J. D. Tygar, "Better Malware Ground Truth: Techniques for Weighting Anti-Virus Vendor Labels," in *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*, 2015.

[49] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware phylogeny generation using permutations of code," *Journal in Computer Virology*, vol. 1, no. 1-2, pp. 13–23, 2005.

[50] D. Kim, B. J. Kwon, and T. Dumitraş, "Certified Malware: Measuring Breaches of Trust in the Windows Code-Signing PKI," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[51] D. Kirat, L. Nataraj, G. Vigna, and B. S. Manjunath, "SigMal: A Static Signal Processing Based Malware Triage," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2013.

[52] J. Z. Kolter and M. A. Maloof, "Learning to Detect and Classify Malicious Executables in the Wild," *Journal of Machine Learning Research*, vol. 7, 2006.

[53] ——, "Learning to detect malicious executables in the wild," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2004, pp. 470–478.

[54] D. Kong and G. Yan, "Discriminant malware distance learning on structural information for automated malware classification," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 1357–1365.

[55] W. Landi, "Undecidability of Static Analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 4, 1992.

[56] B. Li, K. Roundy, C. Gates, and Y. Vorobeychik, "Large-Scale Identification of Malicious Singleton Files," in *Proc. of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2017.

[57] P. Li, L. Liu, D. Gao, and M. K. Reiter, "On Challenges in Evaluating Malware Clustering," in *Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2010.

[58] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting Environment-Sensitive Malware," in *Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011.

[59] R. Lyda and J. Hamrock, "Using Entropy Analysis to Find Encrypted and Packed Malware," *IEEE Security and Privacy*, vol. 5, no. 2, 2007.

[60] Manalyzer, "Malware analysis tool," https://manalyzer.org/, (Accessed: 2019-01-07).

[61] L. Martignoni, M. Christodorescu, and S. Jha, "OmniUnpack: Fast, Generic, and Safe Unpacking of Malware," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2007.

[62] M. M. Masud, L. Khan, and B. Thuraisingham, "A scalable multi-level feature extraction technique to detect malicious executables," *Information Systems Frontiers*, vol. 10, no. 1, pp. 33–45, 2008.

[63] E. Menahem, A. Shabtai, L. Rokach, and Y. Elovici, "Improving malware detection by applying multi-inducer ensemble," *Computational Statistics & Data Analysis*, vol. 53, no. 4, pp. 1483–1494, 2009.

[64] J. Ming, D. Xu, Y. Jiang, and D. Wu, "Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *Proceedings of the 26th USENIX Security Symposium*, 2017.

[65] A. Mohaisen and O. Alrawi, "AV-Meter: An Evaluation of Antivirus Scans and Labels," in *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2014.

[66] M. Morgenstern and H. Pilz, "Useful and Useless Statistics about Viruses and Anti-Virus Programs," in *Proc. of the CARO Workshop*, 2010.

[67] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2007.

[68] R. Moskovitch, D. Stopel, C. Feher, N. Nissim, and Y. Elovici, "Unknown malcode detection via text categorization and the imbalance problem," in *Intelligence and Security Informatics, 2008. ISI 2008. IEEE International Conference on*. IEEE, 2008, pp. 156–161.

[69] L. Nataraj, "Nearly 70% of Packed Windows System files are labeled as Malware," http://sarvamblog.blogspot.com/2013/05/nearly-70-of-packed-windows-system.html, 2013.

[70] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath, "Malware Images: Visualization and Automatic Classification," in *Proc. of the International Symposium on Visualization for Cyber Security*, 2011.

[71] J. Oberheide, M. Bailey, and F. Jahanian, "PolyPack: An Automated Online Packing Service for Optimal Antivirus Evasion," in *Proc. of the USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[72] J. Pearl, "Fusion, propagation, and structuring in belief networks," *Artificial intelligence*, vol. 29, no. 3, pp. 241–288, 1986.

[73] PEFILE, "Pe file parser," https://github.com/erocarrera/pefile, (Accessed: 2018-10-28).

[74] PEiD, "Signature-based packer detector," https://www.aldeid.com/wiki/PEiD, (Accessed: 2019-01-07).

[75] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "Tesseract: Eliminating experimental bias in malware classification across space and time," *arXiv preprint arXiv:1807.07838*, 2018.

[76] R. Perdisci, A. Lanzi, and W. Lee, "Classification of Packed Executables for Accurate Computer Virus Detection," *Pattern Recognition Letters*, vol. 29, no. 14, 2008.

[77] ——, "Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables," in *2008 Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2008, pp. 301–310.

[78] M. Polino, A. Continella, S. Mariani, S. D'Alessio, L. Fontata, F. Gritti, and S. Zanero, "Measuring and Defeating Anti-Instrumentation-Equipped Malware," in *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2017.

[79] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[80] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole exe," in *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[81] E. Raff, J. Sylvester, and C. Nicholas, "Learning the PE Header, Malware Detection with Minimal Domain Knowledge," in *Proc. of the ACM Workshop on Artificial Intelligence and Security (AISec)*, 2017.

[82] E. Raff, R. Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. McLean, and C. Nicholas, "An Investigation of Byte N-Gram Features for Malware Classification," *Journal of Computer Virology and Hacking Techniques*, 2016.

[83] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting System Emulators," in *Proc. of the International Conference on Information Security (ISC)*, 2007.

[84] B. Rahbarinia, M. Balduzzi, and R. Perdisci, "Exploring the Long Tail of (Malicious) Software Downloads," in *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, 2017.

[85] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks," in *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2013.

[86] H. Rathore, S. Agarwal, S. K. Sahay, and M. Sewak, "Malware detection using machine learning and deep learning," in *International Conference on Big Data Analytics*. Springer, 2018, pp. 402–411.

[87] RDG Packer Detector, "Signature-based packer detector," http://www.rdgsoft.net/, (Accessed: 2019-01-07).

[88] M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks," *computers & security*, vol. 77, pp. 578–594, 2018.

[89] I. Rosenberg, A. Shabtai, L. Rokach, and Y. Elovici, "Generic black-box end-to-end attack against state of the art api call based malware classifiers," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 490–510.

[90] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. v. Steen, "Prudent Practices for Designing Malware Experiments: Status Quo and Outlook," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2012.

[91] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," in *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2006.

[92] G. Salton and M. J. McGill, "Introduction to modern information retrieval," 1986.

[93] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode Sequences as Representation of Executables for Data-mining-based Unknown Malware Detection," *Information Sciences*, vol. 231, 2013.

[94] I. Santos, J. Nieves, and P. G. Bringas, "Semi-supervised learning for unknown malware detection," in *International Symposium on Distributed Computing and Artificial Intelligence*. Springer, 2011, pp. 415–422.

[95] I. Santos, X. Ugarte-Pedrero, B. Sanz, C. Laorden, and P. G. Bringas, "Collective Classification for Packed Executable Identification," in *Proc. of the Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference (CEAS)*, 2011.

[96] V. S. Sathyanarayan, P. Kohli, and B. Bruhadeshwar, "Signature generation and detection of malware families," in *Australasian Conference on Information Security and Privacy*. Springer, 2008, pp. 336–349.

[97] J. Saxe and K. Berlin, "Deep Neural Network Based Malware Detection Using Two Dimensional Binary Program Features," in *Proc. of the the International Conference on Malicious and Unwanted Software (MALWARE)*, 2015.

[98] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, "Data Mining Methods for Detection of New Malicious Executables," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2001.

[99] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "AVClass: A Tool for Massive Malware Labeling," in *Proc. of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2016.

[100] A. A. Selçuk, F. Orhan, and B. Batur, "Undecidable Problems in Malware Analysis," in *Proc. of the International Conference for Internet Technology and Secured Transactions (ICITST)*, 2017.

[101] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer, "Detection of Malicious Code by Applying Machine Learning Classifiers on Static Features: A State-of-the-art Survey," *Information Security Technical Report*, vol. 14, no. 1, 2009.

[102] M. Z. Shafiq, S. Tabish, and M. Farooq, "PE-Probe: Leveraging Packer Detection and Structural Information to Detect Malicious Portable Executables," in *Proc. of the Virus Bulletin Conference (VB)*, 2009.

[103] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq, "PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime," in *Proceedings of International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2009.

[104] M. Siddiqui, M. C. Wang, and J. Lee, "Detecting internet worms using data mining techniques," *Journal of Systemics, Cybernetics and Informatics*, vol. 6, no. 6, pp. 48–53, 2009.

[105] Skylightcyber, "Cylance, I Kill You!" https://skylightcyber.com/2019/07/18/cylance-i-kill-you/, July 2019.

[106] L. Sun, "Reform: A framework for malware packer analysis using information theory and statistical methods," 2010.

[107] R. Tian, L. Batten, R. Islam, and S. Versteeg, "An automated classification system based on the strings of trojan and virus families," in *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*. IEEE, 2009, pp. 23–30.

[108] R. Tian, L. M. Batten, and S. Versteeg, "Function length as a tool for malware classification," in *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*. IEEE, 2008.

[109] S. Treadwell and M. Zhou, "A Heuristic Approach for Detection of Obfuscated Malware," in *Proc. of the IEEE International Conference on Intelligence and Security Informatics (ISI)*, 2009.

[110] X. Ugarte Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2015.

[111] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "RAMBO: Run-Time Packer Analysis with Multiple Branch Observation," in *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2016.

[112] X. Ugarte-Pedrero, I. Santos, B. Sanz, C. Laorden, and P. G. Bringas, "Countering entropy measure attacks on packed software detection," in *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*. IEEE, 2012, pp. 164–168.

[113] VirusTotal, "Av comparative analyses," https://blog.virustotal.com/2012/08/av-comparative-analyses-marketing-and.html, (Accessed: 2019-3-31).

[114] ——, "File statistics," https://www.virustotal.com/en/statistics/, (Accessed: 2018-11-26).

[115] G. Webster, B. Kolosnjaji, C. von Pentz, Z. Hanif, J. Kirsch, A. Zarras, and C. Eckert, "Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage," in *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2017.

[116] C. Wressnegger, K. Freeman, F. Yamaguchi, and K. Rieck, "Automatically Inferring Malware Signatures for Anti-Virus Assisted Attacks," in *Proc. of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2017.

[117] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, "V2e: combining hardware virtualization and softwareemulation for transparent and extensible malware analysis," *ACM Sigplan Notices*, vol. 47, no. 7, pp. 227–238, 2012.

[118] B. Zhang, J. Yin, J. Hao, D. Zhang, and S. Wang, "Malicious codes detection based on ensemble learning," in *International Conference on Autonomic and Trusted Computing*. Springer, 2007, pp. 468–477.

[119] M. Zheng, P. P. C. Lee, and J. C. S. Lui, "ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems," in *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2013.

APPENDIX A
PACKING DETECTION AND AUTOMATED UNPACKING

Packing detection and packer identification. Detection of packed software is known to be a challenging task [5, 59, 102]. Packer identification tools [30, 60, 74, 87] use signatures to determine if a program is obfuscated with a particular packer.

Lyda et al. [59] and Jacob et al. [44] apply entropy analysis techniques to a binary, assuming that a packed binary has a high entropy. Sun et al. [106] proposed a different method for randomness analysis that generates a randomness profile for a packed executable to identify the packer employed to protect the program. A similar work generates alternative randomness profiles by combining byte histograms with entropy analysis to mitigate common attacks against entropy analysis [112]. Other approaches leverage static features of PE headers and sections [5, 15, 109], also with the use of machine learning classifiers [37, 76, 77, 95, 102]. However, the problem of distinguishing between packed and unpacked executables is undecidable in general [91], although recent work raised hopes that this problem could be tractable under certain space and time constraints [11].

Automated unpacking. There have been many attempts at unpacking executables in order to extract the original payload for analysis [9, 23, 38, 47, 61, 77, 91, 111]. OmniUnpack [61] scans the memory for the presence of malware at every memory write. PolyUnpack [91] first uses static analysis to acquire a static model of the executable code. Then, it executes the binary in an isolated environment and compares the execution context with the static code model. Coogan et al. [21] exploit alias analysis, static slicing, and control-flow analysis to statically construct a customized unpacker for the executable, which can be executed later to obtain the unpacked code. Similarly, Debray et al. [23] use offline analysis of a dynamic instruction trace to automate the creation of custom unpacking routines. Renovo [47] works under the assumption that the entire unpacked binary resides in memory at a certain time. Bonfante et al. [9] take a sequence of memory snapshots to extract instructions of the original program that are executed. Haq et al. [38] augment this approach by taking differential memory snapshots to minimize noise. Polino et al. [78] study common ways used by malware to evade Dynamic Binary Instrumentation (DBI) and present an anti-DBI resistant unpacker. Ugarte et al. [111] proposed domain-specific customized multi-path exploration techniques to trigger the unpacking of all code regions. More recently, Cheng et al. [13] proposed BinUnpack which works under the assumption that the reconstruction of the Import Address Table finishes ahead of the jump to the original entry point.

APPENDIX B
MACHINE LEARNING FOR STATIC MALWARE ANALYSIS

In this section, we discuss how machine learning is being adopted by the anti-malware community to statically analyze malware. We reviewed a wide range of static malware analysis approaches based on machine learning [2, 7, 28, 39, 44, 49, 51, 52, 53, 54, 56, 62, 63, 68, 70, 80, 81, 82, 86, 93, 94, 96, 97, 98, 102, 103, 104, 107, 108, 118].

One of the first papers that proposed to use machine learning for malware detection was presented by Schultz et al. [98]. The authors used three different feature categories, byte n-grams, (printable) strings, and DLL imports to examine three different classifiers, a Naïve Bayes classifier [31], a Multi-Naïve Bayes classifier, and an inductive ruler learner (i.e., RIPPER [20]). Later, Masud et al. [62] used byte n-grams, assembly instructions, and DLL function calls to train different types of classifiers.

Byte n-gram features are one of the most common features used in static malware detection [68, 82, 101]. Abou-Assaleh et al. [2] used the L most frequent n-grams observed in the training set ($20 \leq L \leq 5000$) to create a profile for each sample, and assign each new sample to a particular class using a nearest neighbor classifier. Kolter et al. [52, 53] extracted 500 n-grams features with the highest information gain and trained several classifiers. Zhang et al. [118] also used information gain measures to select the top n-grams, followed by a probabilistic neural network. Henchiry et al. [39] proposed a hierarchical feature selection that considers only those n-grams that appear above a certain threshold in a malware family, as well as in more than a minimum number of malware families. Jacob et al. [44] used bigram distributions to detect similar malware without executing them, to mitigate analyzing duplicate malware. They used a packer detector based on different heuristics, such as code entropy, that automatically configures the distance sensitivity based on the type of packing used. Other related work [70] visualizes executables as gray-scale images by treating bytes as gray-scale pixel values and borrows image processing techniques to build a K-nearest neighbor classifier. Similar to byte n-grams, opcode n-grams have been used for malware detection [68, 93, 101]. Karim et al. [49] tokenized the input programs into sequences of opcodes to track malware evolution. Bilar et al. [7] leveraged statistical differences between the opcode frequency distribution of malware and benign software to detect malicious code.

Related work focused on other types of features extracted from the program disassembly. Menahem et al. [63] augmented byte n-grams and PE header fields using attributes extracted from functions in the disassembled program. Kong et al. [54] constructed a function call graph and applied discriminant distance metric learning to cluster malware. Tian et al. [108] used the function length along with its frequency to classify Trojans. Siddiqui et al. [104] used variable length instruction sequences followed by tree-based classifiers to detect worms. Sathyanarayan et al. [96] used API calls to obtain a signature for each malware family. Although features from the program disassembly are used widely in capturing malware signatures, they are not always obtainable, as some executables cannot be disassembled properly [101].

While many approaches focused on the binary code of the program, some work has considered other parts of the executables, such as PE headers. Shafiq et al. [103] proposed PE-Miner, which uses 189 features from only PE headers followed by a decision tree classifier. To diminish the bias of PE-Miner in detecting packed executables, they introduced PE-Probe [102], in which a multi-layer perceptron classifier uses heuristics studied by Perdisci et al. [76] to detect packed executables. Based on the outcome, the executable is analyzed by two separate specialized structural models. They compared the distribution of each feature for packed and unpacked executables to identify those that are robust to packing (although they did not report these features). Elovici et al. [28] used Bayesian networks [72], decision trees, and artificial neural networks [8] to create five different classifiers based on byte n-grams and PE headers fields. Webster et al. [115] demonstrated how the contents of the Rich Header fields in PE files can help to detect different versions of malware. Saxe et al. [97] applied a deep neural network with two hidden layers using a histogram of byte entropy values, DLL imports, and numerical

TABLE X: Summary of the packing detection tools used to build *wild dataset*.

Tool	Benign		Malicious	
	packed	unpacked	packed	unpacked
(1) vendor's sandbox	10,463	16,162	26,699	15,095
(2) *dpi*	6,049	20,576	27,995	13,799
(3) *Manalyze*	1,239	17,436	5,376	19,457
(4) *PEiD+F-Prot*	1,189	25,436	2,630	39,164
(5) *yara*	1,524	25,101	3,882	37,912
(6) *Exeinfo PE*	1,088	25,537	5,770	36,024
(1)+(2)+(3)+(4)+(5)+(6)	12,647	4,396	33,681	5,752

TABLE XI: Packer complexity in the *wild dataset*.

Type	Benign	Malicious	All
Type I	708 (11.70%)	660 (2.36%)	1,368 (4.02%)
Type II	19 (0.31%)	2,069 (7.39%)	2,088 (6.13%)
Type III	5,321 (87.96%)	25,111 (89.70%)	30,432 (89.39%)
Type IV	0 (0.00%)	151 (0.54%)	151 (0.44%)
Type V	1 (0.01%)	3 (0.01%)	4 (0.01%)
Type VI	0 (0.00%)	1 (0.00%)	1 (0.00%)

PE fields as features. Li et al. [56] applied a combination of a recurrent neural network (RNN) model and an SVM on top of features extracted from PE headers and sections. To avoid explicit feature extraction, Raff et al. [81] proposed using a Long Short-Term Memory (LSTM [40]) network on raw byte sequences obtained from only PE headers. In particular, they considered only MS-DOS, COFF, and Optional headers. MalConv [80] extends this work by training convolutional neural networks on the entire body of executables.

TABLE XIV: Experiment "wild vs. packers" - MalConv

Packer	FPR (%)	FNR (%)	F-1
PELock	65.40%	17.98%	0.68
PECompact	98.81%	0.98%	0.67
Obsidium	91.35%	5.64%	0.67
Petite	93.70%	1.67%	0.67
tElock	96.03%	2.06%	0.67
Themida	92.34%	5.27%	0.66
MPRESS	97.53%	0.59%	0.67
kkrunchy	98.10%	0.66%	0.66
UPX	85.46%	7.59%	0.67

APPENDIX C
LAB DATASET VALIDATION

To ascertain if (re-)packed executables in the *lab dataset* present their original behavior during execution, we analyze each sample in Cuckoo Sandbox and compare its behavior with the original sample. For this comparison, we look at network behavior and interaction with the file system and Windows registry keys. We further look at APIs that are called during the execution. Due to page limit restrictions, we explain the details of our validation scheme in supplementary material, which can be found at `https://github.com/ucsb-seclab/packware`. In a nutshell, packing does preserve the original behavior for more than 94.56% of samples.

APPENDIX D
RESULTS FOR ALTERNATIVE MODELS

Here, we present the results of major experiments for two different types of classifiers, SVM and neural networks (MalConv [80]). As we mentioned earlier, the trend is the same as what was discussed in Section V.

TABLE XII: Packers identified by *PEiD*, *F-Prot*, *Manalyze*, *Exeinfo PE*, and *yara* rules in the *wild dataset*.

	Benign	Malicious		Benign	Malicious
UPX	1,025	2,187	MEW	0	109
Simple Packer (dxpack)	0	2,293	EmbedPE	16	0
Armadillo	678	676	EXEStealth	0	54
MPRESS	3	955	NsPack	1	21
PECompact	49	307	PENinja	0	23
AHTeam EP Protector	0	271	Expressor	0	10
ASPack	54	202	U-Pack	0	18
PE-Armor	0	144	EXECryptor	1	10
ASProtect	14	103	pklite	10	0
VMProtect	0	61	Diminisher	4	33
FSG	0	43	Themida	0	10

TABLE XIII: Experiment "withheld packer" - MalConv

Withheld Packer	FPR (%)	FNR (%)	Accuracy
PELock	41.70%	53.38%	52.46%
PECompact	67.37%	23.56%	54.54%
Obsidium	37.03%	44.16%	59.41%
Petite	7.44%	82.52%	55.02%
tElock	46.78%	37.82%	57.70%
Themida	30.77%	63.49%	52.88%
MPRESS	89.92%	5.88%	52.04%
kkrunchy	51.21%	43.63%	52.58%
UPX	20.05%	58.08%	60.95%

SVM. Figure 8a and Figure 8b show the false positive and false negative rates of the SVM classifier in "different packed ratios (wild)" and "different packed ratios (lab)", as the packed benign ratio increases in the training set. Table XVII and Table XVIII demonstrate the importance of each family of features in these two experiments. Similar to what we have seen for the random forest classifier in "wild vs. packers", but to less extent, training the classifier using only the Rich Header features helps the classifier to achieve better performance (Table XVI) against packers that preserve this header. Table XV also shows that the classifier fails to generalize to previously unseen packing routines.

Neural Network. We used the architecture proposed by [80], i.e., MalConv. Following extensive hyperparameter tuning, we achieved the same or better performance on the validation set in most experiments. It should be noted that a dataset of 400,000 samples was used to train MalConv. [2] However, in this work, we used datasets with 20-30 times smaller size across all experiments. As we discussed earlier, the nature of this work requires us to label the samples (i.e., benign/malicious and packed/unpacked) based on their dynamic behavior. Unfortunately, such a requirement makes it extremely hard to build huge datasets. For this reason, we did not achieve the highest performance reported for MalConv for some experiments. Also, as acknowledged by the authors, tuned hyperparameters of MalConv will depend on the distribution of samples. In experiments where we have different datasets, especially Experiment "different packed ratios (lab)", MalConv did not achieve its highest performance. In all experiments, similar to the original work, we trained the neural network for 10 epochs, which was enough for convergence. Figure 9 shows the results of "different packed ratios (wild)" and "different packed ratios (lab)". Table XIII also shows that the classifier performs poorly against previously unseen packers. Table XIV shows a similar trend for MalConv in Experiment "wild vs. packers".

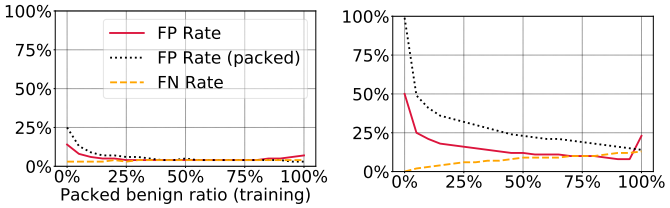
[2] They further used a dataset of 2 million samples to show that MalConv has the capacity to perform better if it is trained on more data.

TABLE XV: Experiment "withheld packer" - SVM

Withheld Packer	All features FPR (%)	All features FNR (%)	All features F-1	NO byte n-grams FPR (%)	NO byte n-grams FNR (%)	NO byte n-grams F-1
PELock	**61.32%**	3.46%	**0.75**	**49.88%**	3.10%	0.79
PECompact	**35.90%**	4.90%	**0.82**	**51.81%**	8.0%	**0.75**
Obsidium	**49.67%**	1.07%	**0.79**	**62.02%**	3.04%	**0.75**
Petite	**21.39%**	0.87%	0.90	18.17%	4.20%	0.90
tElock	**68.07%**	1.34%	**0.74**	**84.65%**	1.62%	**0.69**
Themida	9.89%	9.28%	0.91	10.74%	**50.39%**	0.62
MPRESS	12.17%	6.83%	0.91	**19.44%**	4.09%	0.89
kkrunchy	**59.32%**	0.0%	**0.77**	**56.07%**	4.57%	**0.76**
UPX	7.39%	11.02%	0.91	10.64	14.74%	0.87

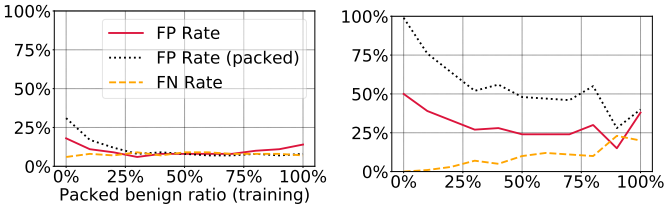
TABLE XVI: Experiment "wild vs. packers" - SVM

Packer	All features FPR (%)	All features FNR (%)	All features F-1	Rich Header (only) FPR (%)	Rich Header (only) FNR (%)	Rich Header (only) F-1
PELock	99.39%	0.77%	0.66	99.72%	0.0%	**0.67**
PECompact	62.56%	4.37%	0.76	**45.14%**	11.96%	0.75
Obsidium	67.99%	9.38%	0.69	100.0%	0.0%%	**0.67**
Petite	76.86%	0.69%	0.71	**26.95%**	13.08%	0.81
tElock	91.08%	0.53%	0.68	**68.87%**	11.96%	0.69
Themida	98.64%	0.29%	0.67	**30.61%**	10.88%	0.81
MPRESS	95.05%	0.25%	0.67	100.0%	0.0%	**0.67**
kkrunchy	99.30%	0.1%	0.67	100.0%	0.0%	**0.67**
UPX	41.55%	3.27%	0.83	43.04%	10.45%	0.77



(a) *wild dataset* **(b)** *lab dataset*

Fig. 8: Experiment "different packed ratios" - SVM



(a) *wild dataset* **(b)** *lab dataset*

Fig. 9: Experiment "different packed ratios" - MalConv

TABLE XVII: Experiment "different packed ratios (wild)."

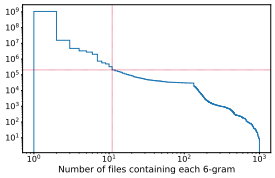
PB Ratio	Training Set #B (packed)	#B (unpacked)	#M	# Features used by the classifier (Top 50) import	dll	rich	sections	header	strings	byte n-grams	opcode n-grams	generic	all
.0	0	3,077	3,077	26 (3)	1 (1)	20 (0)	46 (2)	5 (0)	104 (29)	120 (14)	0 (0)	1 (1)	323 (50)
.2	615	2,462	3,077	24 (4)	3 (0)	23 (0)	60 (3)	6 (1)	130 (28)	192 (14)	0 (0)	1 (0)	439 (50)
.4	1,231	1,846	3,077	33 (3)	5 (1)	22 (0)	57 (4)	4 (0)	140 (30)	209 (12)	0 (0)	1 (0)	471 (50)
.6	1,846	1,231	3,077	29 (3)	3 (1)	21 (0)	64 (5)	3 (0)	132 (30)	187 (11)	0 (0)	1 (0)	440 (50)
.8	2,462	615	3,077	25 (2)	3 (1)	23 (0)	57 (5)	4 (0)	121 (29)	201 (13)	0 (0)	1 (0)	435 (50)
1.	3,077	0	3,077	20 (5)	4 (1)	20 (0)	60 (3)	3 (0)	116 (26)	187 (15)	0 (0)	1 (0)	411 (50)

TABLE XVIII: Experiment "different packed ratios (lab)" - SVM

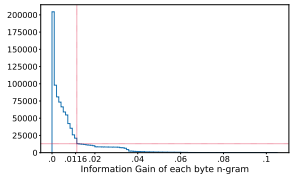
PB Ratio	Training Set #B (packed)	#B (unpacked)	#M	# Features used by the classifier (Top 50) import	dll	rich	sections	header	strings	byte n-grams	opcode n-grams	generic	all
.0	0	3,077	3,077	0 (0)	0 (8)	14 (14)	24 (24)	3 (3)	0 (0)	0 (0)	0 (0)	1 (1)	42 (50)
.2	615	2,462	3,077	3 (0)	4 (1)	20 (0)	49 (2)	9 (0)	113 (15)	179 (32)	1 (0)	0 (0)	378 (50)
.4	1,231	1,846	3,077	6 (1)	7 (1)	18 (0)	56 (2)	11 (0)	199 (28)	247 (17)	2 (1)	0 (0)	546 (50)
.6	1,846	1,231	3,077	7 (2)	7 (0)	22 (0)	58 (1)	9 (0)	236 (39)	386 (7)	3 (1)	0 (0)	728 (50)
.8	2,462	615	3,077	10 (1)	9 (0)	20 (0)	61 (1)	11 (0)	257 (36)	395 (12)	3 (0)	0 (0)	766 (50)
1.	3,077	0	3,077	14 (0)	10 (0)	22 (0)	58 (0)	11 (0)	281 (38)	405 (12)	2 (0)	0 (0)	803 (50)

TABLE XIX: The parameters of the random forest classifier used in the experiments.

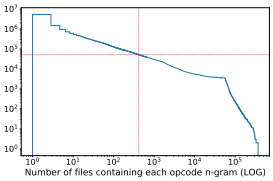
Parameter	Value
# of trees	50
The maximum depth of each tree	`Infinity (Nodes are expanded until leafs)`
The minimum number of samples required to split an internal node	2
The minimum number of samples required to be at a leaf node	1
The number of features to consider when looking for the best split	$\sqrt{\#\ \text{features}}$
Bootstrap: whether bootstrap samples are used when building trees	`True`
The function to measure the quality of a split	`Gini Impurity`



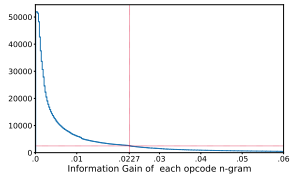
(a) The CCDF of number of 6-grams occurring in x files.



(b) The CCDF of number of 6-grams with IG value of y.



(c) The CCDF of number of opcode n-grams occurring in x files.



(d) The CCDF of number of opcode n-grams with IG value of y.

Fig. 10: Opcode and byte n-grams distributions.

TABLE XX: Features extracted from PE headers.

Name	Source	Description
header_ImageBase	Opt. header	The address of the memory mapped location of the file
header_AddressOfEntryPoint	Opt. header	The address where the loader will begin execution
header_SizeOfImage	Opt. header	The size (in bytes) of the image in memory
header_SizeOfCode	Opt. header	The size of the code section
header_BaseOfCode	Opt. header	The address of the first byte of the entry point section
header_SizeOfInitializedData	Opt. header	The size of the initialized data section/s
header_SizeOfUninitializedData	Opt. header	The size of the uninitialized data section/s
header_BaseOfData	Opt. header	The address of the first byte of the data section
header_SizeOfHeaders	Opt. header	The combined size of the MS-DOS stub, PE headers, and section headers
header_SectionAlignment	Opt. header	The alignment of sections loaded in memory
header_FileAlignment	Opt. header	The alignment of the raw data of sections
header_NumberOfSections	COFF. header	The number of sections
header_SizeOfOptionalHeader	COFF. header	The size of the optional header
header_characteristics_bitX	COFF. header	The corresponding flag to bit X is set for the executable or not

TABLE XXI: Features extracted per each section of the PE file ("features per section"). 'id' is the section number. For example, feature PESECTION_10_NAME represents the name of the 10th section of the executables if present, otherwise none.

Name	Description
pesection_id_name	The section name
pesection_id_size	The section size
pesection_id_rawAddress	The address in the file
pesection_id_virtualSize	The total size when loaded into memory
pesection_id_entropy	The entropy of the section
pesection_id_numberOfRelocations	The number of relocation entries
pesection_id_pointerToRelocations	The address of the first byte of the relocation entries in file
pesection_id_characteristics_bitX	The corresponding flag to bit X is set for the section or not

TABLE XXII: Each row shows the number (percentage) of benign and malicious samples per packer that import the API.

API Import	Obsidium		Petite		UPX		MPRESS		PELock		PECompact	
	#B	#M	#B	#M	#B	#M	#B	#M	#B	#M	#B	#M
RegCloseKey	**2730 (16.12%)**	**16,675 (52.95%)**	**1,870 (13.71%)**	**2,443 (9.45%)**	**2,928 (29.46%)**	**3,276 (15.89%)**	1,770 (16.03%)	1,809 (15.74%)	230 (3.34%)	321 (3.79%)	**190 (3.39%)**	**447 (1.58%)**
InitCommonControls	**637 (3.76%)**	**152 (0.48%)**	13 (0.1%)	128 (0.5%)	**509 (5.12%)**	**61 (0.3%)**	**240 (2.17%)**	**59 (0.51%)**	85 (1.24%)	34 (0.4%)	**270 (4.81%)**	**44 (0.16%)**
RegQueryValueA	**479 (2.83%)**	**31 (0.1%)**	**435 (3.19%)**	**4 (0.02%)**	**428 (4.31%)**	**3 (0.01%)**	**395 (3.58%)**	**0 (0.0%)**	**269 (3.91%)**	**7 (0.08%)**	14 (0.25%)	15 (0.05%)
MessageBoxA	1,075 (6.35%)	2,218 (7.04%)	13,628 (99.93%)	25,473 (98.51%)	**95 (0.96%)**	**398 (1.93%)**	246 (2.23%)	386 (3.36%)	91 (1.32%)	191 (2.25%)	226 (4.03%)	83 (0.29%)
ShellExecuteA	0 (0.0%)	0 (0.0%)	**1,066 (7.82%)**	**2,542 (9.83%)**	1,173 (11.8%)	2,416 (11.72%)	926 (8.39%)	690 (6.0%)	372 (5.41%)	329 (3.88%)	105 (1.87%)	564 (1.99%)
SysFreeString	0 (0.0%)	0 (0.0%)	205 (1.5%)	205 (0.79%)	**541 (5.44%)**	**584 (2.83%)**	**408 (3.7%)**	**687 (5.98%)**	**148 (2.15%)**	**452 (5.33%)**	**484 (8.63%)**	**1,718 (6.06%)**
FreeSid	**107 (0.63%)**	**1,490 (4.73%)**	1,210 (8.87%)	2,172 (8.4%)	1,047 (10.54%)	2,189 (10.62%)	673 (6.1%)	524 (4.56%)	87 (1.26%)	49 (0.58%)	31 (0.55%)	45 (0.16%)
wsprintfA	**174 (1.03%)**	**1,574 (5.0%)**	13,628 (99.93%)	25,473 (98.51%)	**118 (1.19%)**	**622 (3.02%)**	**91 (0.82%)**	**581 (5.05%)**	**56 (0.81%)**	**269 (3.17%)**	9 (0.16%)	13 (0.05%)
InitCommonControlsEx	**803 (4.74%)**	**455 (1.44%)**	216 (1.58%)	383 (1.48%)	316 (3.18%)	349 (1.69%)	192 (1.74%)	159 (1.38%)	101 (1.47%)	86 (1.01%)	**218 (3.89%)**	**62 (0.22%)**
GetDC	**792 (4.68%)**	**635 (2.02%)**	0 (0.0%)	0 (0.0%)	**3,357 (33.78%)**	**3,160 (15.32%)**	1,811 (16.4%)	1,554 (13.52%)	51 (0.74%)	151 (1.78%)	**220 (3.92%)**	**369 (1.3%)**
# Samples	16,940	31,492	13,638	25,857	9,938	20,620	11,041	11,494	6,879	8,474	5,610	28,346