Assessing static and dynamic features for packing detection

Charles-Henry Bertrand Van Ouytsel, Axel Legay, Serena Lucca, and Dimitri Wauters

INGI, ICTEAM, Universite Catholique de Louvain, Place Sainte Barbe 2, LG05.02,01, 1348 Louvain-La-Neuve, Belgium
{charles-henry.bertrand, axel.legay, serena.lucca,d.wauters}@uclouvain.be

Abstract. Packing is a widely used obfuscation technique for malware to bypass detection tools and hinder reverse engineering. Existing research has already covered methods to detect packing, both with static and dynamic analysis. These methods are based on various features: headers, entropy, API calls, section permissions, etc. While dynamic features are generally more informative, their contribution compared to static features is not always clear. This paper compares the impact of these static and dynamic features on different machine learning classifiers. We propose a study on different datasets to determine whether the information provided by dynamic analysis outweighs its significant extraction time.

Keywords: Packing · Malware · Machine Learning · Static Analysis · Dynamic Analysis · Experimental Comparison

1 Introduction

The growing number of malware discovered each year [25] requires building accurate and efficient malware detection mechanisms to protect the cyberspace. Currently, different malware detection tools available on the market use signature-based detection for their high efficiency. These signatures represent malware by syntactic properties (e.g., length, byte sequences, entropy, etc.), but they could also exploit artifacts observed during malware execution in a monitored environment (e.g., network communication, interaction with the system,...). To avoid the need to manually create such signatures, researchers developed machine learning classifiers based on the same features (i.e., byte sequence, API calls observed during monitored execution,...). However, these signatures and classifiers are vulnerable to obfuscation, which changes the syntactic properties of the malware [11]. Moreover, evasion techniques allow the malware to exhibit its malicious behavior under selected conditions (e.g., absence of sandbox, waiting for a certain day,...) that are not necessarily fulfilled during the analysis. All these techniques make malware analysis challenging and require the development of new approaches to ensure the detection of malicious behavior.

Aware of these challenges, researchers have proposed the use of symbolic execution for malware analysis. Symbolic execution is a formal verification technique that explores the different execution paths of a binary by considering all variables of the execution environment as symbolic. During symbolic execution of the binary, constraints are added to symbolic variables and the different possible execution paths are explored. A Satisfiability Modulo Theory (SMT) solver is responsible for verifying the satisfiability of the collected symbolic constraints and thus the validity of the execution path. Symbolic execution can counter evasion and obfuscation techniques [14,13] and thus ensure the triggering of the malicious behavior [11]. It has led to the development of a new set of machine learning based malware classification methods [43,19,7]. In [43], the authors have proposed to combine symbolic execution with Gspan [52], a graph mining algorithm that detects the largest common subgraphs between two graphs. They first collect API calls via symbolic execution and then connect them in a System Call Dependency Graph (SCDG), i.e., a graph that abstracts the flow of information between these calls. In the training phase, Gspan computes the largest common subgraphs between malware in a given family. These then represent the signature for the family. In the classification phase, they extract the SCDG from the binary and compare it to the signature of each family. This approach has been extended with a focus on using efficient tactics for the SMT solver [44] and path prioritization techniques to improve malware code coverage as well as more efficient classifiers than Gspan [9].

Unfortunately, symbolic execution remains expensive to apply, therefore it is generally not possible to explore all possible execution paths of the binary. In particular, the packing obfuscation technique hinders the efficiency of the symbolic engine and the scalability of this approach. Packing can obfuscate a binary with different strategies: compression, encryption, virtualization,... It directly affects the syntactic properties of the target binary, making it more difficult to analyze. While packing can be used for legitimate purposes (e.g., intellectual property protection), it is often used by malware authors to hide their code [38]. Packing detection could improve the efficiency of symbolic execution for malware analysis. If a packed sample is identified, an unpacking procedure could be used to extract the original binary. It could then be analyzed symbolically to understand its behavior.

While signatures have also been developed to detect specific packers, they are easily bypassed by unknown packers [12]. Thus, different heuristics based on syntactic properties have been proposed. The literature includes works based on the entropy of the binary [34], section names and permissions [41], IAT table [48], metrics related to section sizes [18], N-grams [50], etc. All these characteristics have been extensively used as features for supervised machine learning techniques [20]. For example, Biondi et al. [10] extract 119 features from the literature including entry bytes, entropy, imported functions, metadata, resource and sections to train their classifiers. Given that these attributes have been clearly defined in their paper and originate from a recent review of the relevant literature, they will serve as a baseline for this work.

Although such classifiers are generally efficient, recent works highlight their limitations regarding low entropy packing scheme [35] but also the overlap between classifiers to detect packing and malware [1]. Static features focus on syntactic properties of the sample and could fail to grasp the semantics of the analyzed sample (i.e., this is called the semantic gap [46]). On the other hand, features extracted dynamically could allow to discover semantic properties of the sample [26] (i.e, its behavior at run-time) but lead to high extraction cost. Unfortunately, few works propose to compare the efficiency of these two types of features. Regarding the malware detection problem [21], dynamic features show a better ability to generalize but are also more error-prone to extract.

The main contribution of this paper is the comparison of features extracted statically and dynamically to detect packing. For each static feature commonly used, the literature has been explored and equivalent features extracted dynamically have been implemented. Three sets of features are used, including either static features, dynamic features, or both. These sets are then fed to a machine learning algorithm to compare their classification performance. The individual importance of these different features are then exposed thanks to SHAP value, an explainable machine learning technique. In our experiments, we use two different datasets, one with benign samples that have been manually packed, and one composed of various malware found in the wild. These two datasets are then combined to create a third, more diverse dataset. We train Decision Trees, Random Forests, and Gradient Boosted Decision Trees with the three feature sets defined above and extracted from our datasets. These experiments will allow us to show which features and models perform best at packing detection.

This paper is divided into five sections. Section 2 introduces general background about packing, machine learning and our malware analysis workflow. Section 3 discusses the dynamic features used in this work and how they are extracted. Section 4 explains our methodology for the experiments by discussing our datasets, the feature extraction and the model training. Section 5 evaluates tree-based classifiers considering different feature sets and the individual importance of the features in the classification process. Section 6 discusses future works and concludes.

Message for Tiziana Margaria Dear Tiziana, There are so many things to say about the relationship that binds us, but I think the best way to summarize it is in these few words: kindness, trust, dedication, and compassion. Thank you for all these years of support and for your presence and loyalty (as well as Bernhard's) during difficult times when others were no longer there. Our shared passion for formal verification research has connected us for many years. The article you are about to read represents a practical culmination of these results in the very applied field of cybersecurity. Your constant drive to engage with industry has likely influenced me. I am confident that you will appreciate the proposed results and that they will open new avenues of research for you, especially within the RAISE project.

2 Background

In this section, we present background related to packers and their influence on the characteristics of an executable. Following that, we will proceed to introduce machine learning models and the key metrics employed in their evaluation. Finally, we present our malware analysis workflow and the importance of packing detection to make it efficient.

2.1 Packing

A packer is a program designed to transform a binary input into an obfuscated binary, typically involving compression or encryption processes but also anti-analysis tricks (e.g.: erasing import, junk code,...). The resulting obfuscated binary encompasses both the concealed code and the unpacking procedure, as illustrated in Figure 1. During runtime, the unpacking routine is executed as a preliminary step to reconstruct the original code into the process memory. Afterward, the original code is executed by means of a jump to the original entry point (OEP). Malicious actors frequently employ packers with the intent of concealing their malware from antivirus detection mechanisms.

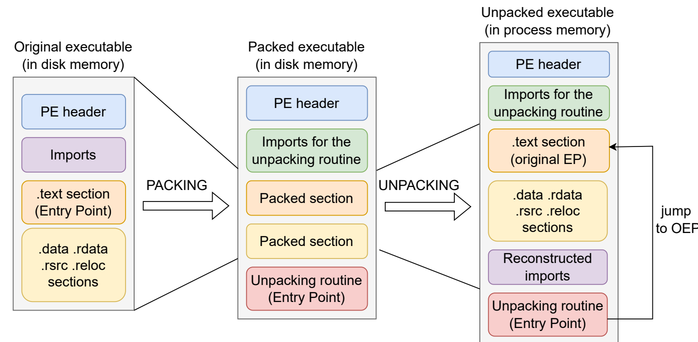


Fig. 1: Packing and unpacking process

Packers often engage in the alteration of the attributes inherent to the original executable. To illustrate, one notable change is the augmentation of entropy within a packed executable, which typically surpasses the entropy level found in the original binary. Furthermore, packed executables may exhibit non-standard section names, as multiple sections are generated to accommodate the original code segments. Another prevalent modification technique employed by packers involves the reduction of the imported functions list. In practice, the list of imported functions is often compressed alongside the original executable and subsequently replaced with a more concise set of functions, specifically tailored to the requirements of the unpacking routine.

2.2 Machine Learning

Machine Learning (ML) allows a system to learn from a dataset in order to make predictions on new data. We will use supervised machine learning which learns from a labeled dataset.

Tree based models are among the most popular models in machine learning. Decision Trees (DT) consist of branches and leaves that represent choices based on features and predicted outputs, respectively. Random Forests (RF) consist in ensembles of different DTs and are less prone to over-fitting than DTs. Finally, Gradient-Boosted Decision Trees (GBDT) are based on a set of DTs that are built sequentially to correct mistakes in previous DTs. They usually result in robust models with good accuracy. Recent research[8,10] shows that tree-based models provide good prediction performance for packing detection with low training time and memory consumption. Therefore, we choose to focus our study on these models.

Performance metrics are needed to evaluate and compare models. We will use well-known ML metrics to assess the strengths and weaknesses of our models. These metrics are computed from 4 measures: TP (True Positives, i.e: packed classified as packed), FP (False Positives, i.e: not packed classified as packed), TN (True Negatives, i.e: not packed classified as not packed) and FN (False Negatives, i.e: packed classified as not packed).

- The *balanced accuracy* is the mean between the proportion of well classified packed samples and the proportion of well classified not-packed samples. $\frac{1}{2}(\frac{TP}{TP+FN} + \frac{TN}{TN+FP})$
- The *precision* is the proportion of packed sample identifications that are actually correct. A low precision indicates that the model classifies many not packed samples as packed. $\frac{TP}{TP+FP}$
- The *recall* is the proportion of actual packed samples that were correctly identified. If this metric is low, it puts in light that many packed samples are missed by the model. $\frac{TP}{TP+FN}$
- The *F1-score* is the harmonic mean of the recall and the precision. This metric works well on imbalanced data. $\frac{2*precision*recall}{precision+recall}$

Finally, we use ROC curves to compare the tradeoffs between FP rate and TP rate with different classification thresholds on our models.

2.3 Malware analysis workflow

Detection of packed binaries allows malware analysis techniques to be adapted to ensure efficient detection of malicious behavior. This section describes SEMA - Symbolic Execution for Malware Analysis - and the importance of packing detection to improve its efficiency.

SEMA [6] is an open source tool for analyzing, detecting and classifying malware using symbolic execution and machine learning. It is based on the angr

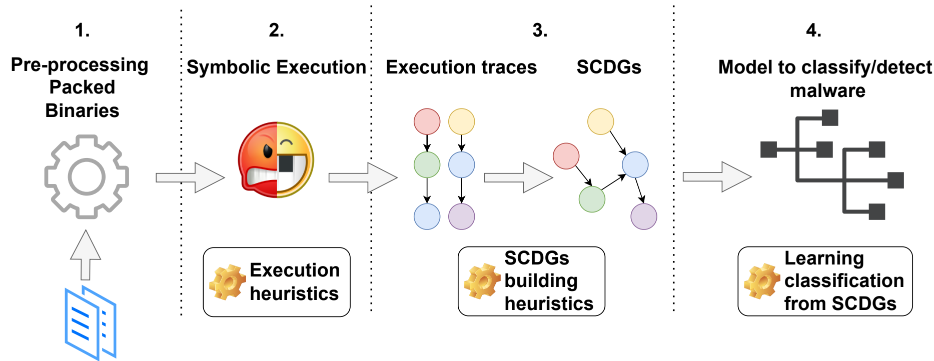


Fig. 2: Malware analysis workflow with SEMA

symbolic execution framework and implements different strategies to improve binary code coverage: path prioritization strategies, loop handling, optimized SMT solver strategies,... The general SEMA workflow is illustrated in Figure 2 and relies on the following key building blocks. The first step, which is the focus of this paper, is to identify packed binaries using packing detection models. The packed binaries are further analyzed to identify the packer and a possible unpacking procedure. We then use symbolic execution on non-packed (or unpacked) binaries from different malware families. During each symbolic execution, we record the API calls and their arguments according to their respective execution paths to create execution traces. These execution traces are used to build the binary's SCDG - System Call Dependency Graph of the binary. Multiple SCDGs built from different binaries are then used to train machine learning algorithms to either distinguish a malicious file from a benign one, or to identify the malware family. When a new binary needs to be classified, we build its SCDG and use the previously trained classifier to output a decision.

To illustrate the importance of packing detection, consider the simple program introduced in [6] and shown in Listing 1.1. This program prints the message "I'm evil!!" as its malicious behavior. While static analysis is bypassed by splitting the string (Lines 13 and 14), dynamic analysis is bypassed when using a debugger (Line 4) or not updating the tick count (Line 4) to simulate a sleep of 500000 seconds (Line 2). SEMA successfully builds an SCDG representing the entire binary behavior in a few seconds. We use UPX [40] to pack this example and observe the effect on symbolic execution. It appears that SEMA is unable to record any API call, even with a 15 minute timeout, because the symbolic execution is slowed down by the unpacking routine. This observation may generalize to other packers. Therefore, it is essential to detect packing to pre-process the binary before applying symbolic execution to improve its scalability.

```
1 ULONGLONG uptime = GetTickCount();
2 Sleep(500000);
3 ULONGLONG uptimeBis = GetTickCount();
```

```
4  if ((uptimeBis - uptime)<500000 || IsDebuggerPresent())
5      {MessageBox(NULL,"Hello world!","", MB_OK);}
6  else
7      {char message[20] = "";
8      HINSTANCE hlib = LoadLibrary("msvcrt.dll");
9      MYPROC func = (MYPROC) GetProcAddress(hlib,"strcat");
10     (func) (message, "I'm ");
11     (func) (message, "evil!!");
12     MessageBox(NULL, message, "", MB_OK);}
```

Listing 1.1: Malware code toy example from [6]

Packing detection based on static features has been widely studied in the literature [14,39,48]. While these features are fast to extract, they are easily modified [8,47] and thus their efficiency is limited for packing detection to support symbolic execution. Recent studies suggest the use of features extracted from dynamic analysis [26,5,35,8], as they should improve the efficiency and robustness of machine learning models for packing detection. However, a comprehensive evaluation of the benefits associated with these dynamic features is lacking in the literature. Therefore, we propose to explore and evaluate these features to observe if they can really improve the efficiency of packing detection despite the higher extraction time.

3 Dynamic Features

This section describes in details the dynamic features which have been investigated for packing detection.

Entropy This method measures the statistical variation of a byte sequence in an executable. Lyda and al. [34] were the first to use entropy to detect packing since packed binaries have generally a higher entropy than a normal executable due to the encryption/compression. Static analysis could only gather entropy of the sample before the execution of the sample and the deployment of the payload. It should be noted that some packers will enforce a low entropy for their sections [35], hindering entropy measures based on static analysis. Dynamic analysis could measure the entropy multiple times through the execution of the sample and expose a more detailed representation.

Bat-Erdene M. et al. [5] identify multiple classes of patterns that can be extracted from the entropy of executable section:

- *Increasing class*
  The section that will contain the unpacked code initializes its memory to 0, setting the entropy close to 0 as well. When the section will be populated with the unpacked executable, the entropy will increase as illustrated in Figure 3.

- *Decreasing class*
  Contrary to the increasing class, the section of this class will not be initialized to 0 and left with the random bytes present when the memory was allocated. Therefore, the initial entropy of the section will be high as the distribution of the bytes will be close to random. Once the section is populated, the entropy will decrease a little as the bytes will represent assembly instructions as illustrated in Figure 4.
- *Combination class*
  This class combines the entropy increase/decrease with a constant class. The initial entropy of this class will either be low, increasing to a maximum, or high, decreasing to a minimum. After those maximum and minimum, the entropy will almost stay constant as illustrated in Figure 5.
- *Constant class*
  The last observed class of entropy is the constant class. In this class, the entropy does not variate and stays constant for the whole execution. Therefore, we can assume that the sample is not packed.

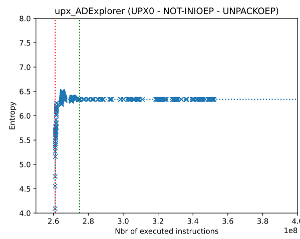


Fig. 3: Increasing class (Packer: UPX)

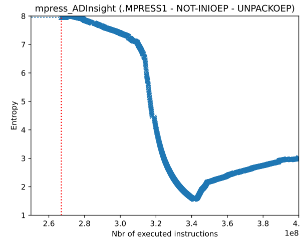


Fig. 4: Decreasing class (Packer: MPRESS)

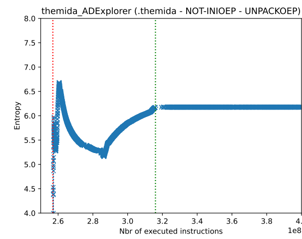


Fig. 5: Combination class (Packer: THEMIDA)

These classes are useful for packing detection since non-packed executable have generally an almost constant entropy. In this implementation, the value of the entropy is computed with a fixed granularity. Meaning that a fixed number of basic blocks will be skipped between every computation to reduce the overhead of this technique. The obtained basic block count and entropy values allow to extract features used by machine learning to deduce if the sample is packed. Those features include statistics such as `maximum`, `minimum`, `delta`, `mean`, `median`, `variance` and `standard deviation` of the recovered entropy points.

IAT functions The Import Address Table (IAT) of a PE executable contains multiple imported Windows functions. These functions may originate either from internal Windows Dynamic Link Libraries (DLLs), such as `kernel32.dll`, or external DLL sources.

From the IAT, static analysis could extract features for packing detection by recovering the number and names of imported functions [54]. However, it

is not able to consider dynamically loaded function since they do not appear in the IAT. A dynamically loaded function is a function that is loaded during the execution of the binary thanks to specific functions present in the IAT (e.g: `GetProcAddress`). Even though the IAT is empty, the executable can recover the functions needed to reconstruct it. For example, it could manually compute the address of the functions [28].

This obfuscation can be counteracted with dynamic analysis: before the execution the addresses of each memory-loaded functions is recovered and saved by looking into the memory mappings of the sample. With this pre-processing, the addresses of these functions are easily recognizable during the execution. Moreover, every call to `GetProcAddress` is monitored and saved (i.e.: the function name and its address). As illustrated in Figure 6, this monitoring offers more information than simple static analysis.

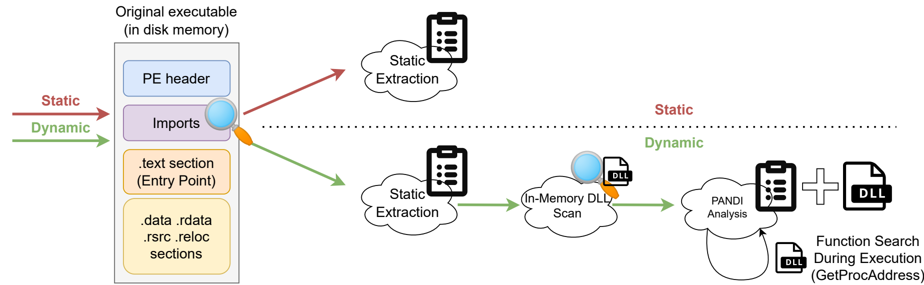


Fig. 6: Representation of IAT static and dynamic analysis workflow

First bytes In signature based approach (e.g.: YARA rules), it is common to examine the first bytes at the entry point of an executable (i.e.: corresponding to the first instructions executed). These features have also been used in machine learning approaches [27]. However, is not always the real entry point of the executable [53]. For example, Thread Local Storage (TLS) callback is a piece of code usually used to initiate threads and called before instructions at the entry point. Code executed by TLS callbacks are not caught by debuggers and a malicious actor could use this anti-debugging technique to bypass signatures [48]. Other techniques [51] such as trampoline (i.e.: creating a new entry point which immediately jumps to the original entry point) could hinder this feature extraction. Dynamic analysis allows to extract first bytes which are effectively executed [39,5,31] by the sample.

Executable sections permission After the execution of the unpacking routine, the original code is written in memory and executed. Since proper section permissions are required to write and execute the code, those have been used as features to detection packing [2,50]. There are three possible values for the

permissions indicated in the PE header: `read`, `write` and `execute`. The most common permission is `read` and the most suspicious pair of permissions regarding packing detection is `write-execute`. While static analysis could extract permissions from the PE header, it has no means to detect a permission change at runtime.

In our dynamic analysis approach, we extract permissions from the PE header and follow permissions modifications during the execution. New memory regions could be allocated with API calls like `LocalAlloc` and their permissions modified with calls like `VirtualProtect`. We monitor these memory regions by following such API calls and monitoring the memory mapping of the sample to verify its permissions. If a memory write occurs in a region initially read-only, it is logged. OmniUnpack [36] followed a similar approach by monitoring suspicious calls and memory access. Note that in our approach, monitoring memory read/write implies a high overhead regarding feature extraction time.

Memory write-then-executed A technique widely used in literature [35,36,49] to detect packing is to monitor *write-then-executed* instructions. This provides a simple and more fine-grained solution to detect packing than memory permissions. Each memory write address is registered in a list. Then, if the currently executed instruction is in the list, it is moved to a *write-then-executed* list. When this second list goes beyond a defined threshold (i.e.: to reduce the false-positives), the sample is considered packed. By design, this technique requires dynamic analysis and is resource intensive (i.e.: regarding execution time and memory). Moreover, a non-packed executable may still be detected in the case of a manual DLL loading and execution, making this technique not totally reliable.

This method is popular to detect packing with dynamic analysis as it does not need a second phase of machine learning, the result is usable directly. This is not the case of the other options proposed for this contribution, they will need another layer to be able to answer to the ”packed or not” question. In return, the other options are quicker to execute and need much less resources as they (almost) never observe the memory.

4 Methodology

This section details our experimental setup: the datasets, the feature extraction process and the complete machine learning workflow.

Datasets We use three datasets. The first dataset CleanPack [15] consists of 1863 packed and 335 not packed cleanware. The packed samples have been manually packed with known packers: Alienyze, Amber, ASPack, BeRoEXEPacker, EnigmaVirtualBox, EronanaPacker, Exe32pack, EXpressor, FSG, JDPack, MEW, Molebox, MPRESS, Neolite, NSPack, Packman, PECompact, PEtite, RLPack, TELock, Themida, UPX, WinUpack, Yoda-Crypter, Yoda-Protector. The second dataset WildPack comes from Aghakhani et al. [1]. The files have been

gathered in the wild around 2017. They have already been labeled as packed or not packed. For our use case, we selected 5728 samples (4953 packed and 775 not packed) from their wild dataset. It includes benign and malicious packed samples alongside benign not packed samples. Finally, we combined the CleanPack and the WildPack datasets to create the MixPack dataset. Both datasets have a similar proportion of packed samples. By merging these datasets, we hope to create more diversity since samples come from different sources (manual packing and wild collection).

Feature Extraction The dynamic features used by our models are extracted with our tool PANDI [33]. This tool is built on top of PANDA [42], a platform for architecture-neutral dynamic analysis, that is used to perform the execution in a virtual environment. The purpose of PANDI is to handle the executable file that will be analyzed by feeding it to PANDA. Once the execution is finished, PANDA will save the execution workflow to be able to replay it later [24]. PANDI replays this execution workflow to analyze the sample thanks to the API made available by PANDA. This API can trigger callbacks that will be caught by PANDI. The callbacks can be, for example, when the sample writes something to memory or when the sample executes a system call. While malware could detect the sandbox used, we limit this possibility by not implementing any detection mechanism directly in the sandbox. Nevertheless, a malware can still implement some more complex techniques to detect the sandbox, like waiting for user input or analyzing the uptime of the machine before extracting the packed executable.

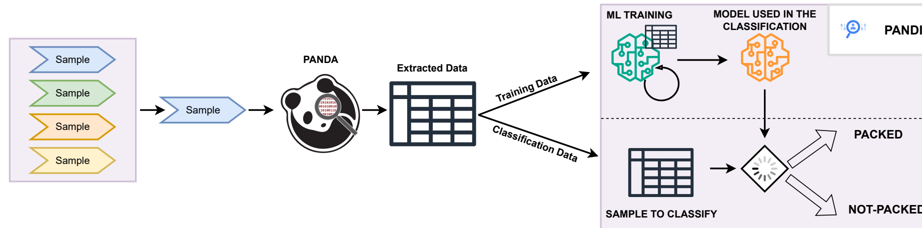


Fig. 7: Representation of the interactions between PANDI and PANDA

Initially, 110 dynamic features were gathered. They represent entropy values (16 features), first few executed bytes (64 features), permissions changes (2 features), write-then-execute sequence (1 feature), imported functions (3 features) and functions calls (24 features). We decided to not include the features linked to permission changes and write-then-execute due to their high extraction cost. We end up with 107 dynamic features. In addition to the dynamic features, a static feature set is extracted thanks to PeLib library [4]. This set is made of 119 features [10] grouped into different categories: metadata, sections, entropy, entry bytes, imported functions and resources.

Figure 8 illustrates the time taken to collect data from each group of features across 1000 samples. Measurements were performed on an 8-cores KVM virtual

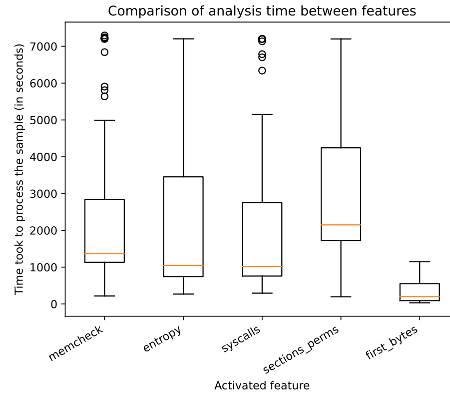


Fig. 8: Time comparison of the extraction for each type of feature

machine (1 sample per core concurrently) and based on Ubuntu 22.04 with 32 GB of RAM. Regarding static features, Biondi et al. [10] have already discussed their extraction time. Showing extraction costs of metadata, section and entry bytes were negligible; extraction costs of entropy and resources were dependent of file size and properties; and extraction cost of imported functions was the most important. Nonetheless, all extraction costs related to static features are negligible compared to extraction costs of dynamic features.

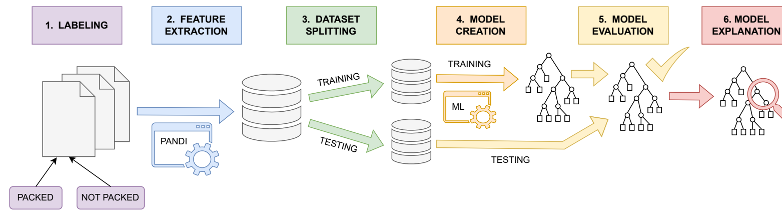


Fig. 9: Machine Learning Workflow

Machine Learning Pipeline We use the scikit-learn 1.3.0 library during our experiments and follow the workflow depicted in Figure 9.

1. Labeling The datasets that we use are already labeled. The CleanPack dataset is composed of cleanware that are manually packed. Hence, the labeling is straightforward. The binaries of WildPack dataset has been labeled as packed or not packed using multiple dynamic and static techniques such as deep packer

inspection, sandbox execution, signatures and heuristics (Manalyze [30], Exeinfo PE, Yara rules, PEiD [22], and F-Prot) as explained in [1]. A sample from this dataset is labeled as packed if at least one of the detection techniques considers it to be packed.

2. Feature Extraction We mentioned previously the methods used to extract the static and dynamic features. Each experiment will be performed using the set of dynamic features, then the static features and finally using the combination of both feature sets. This will allow to compare the dynamic approach to the more classical static approach and their combination.

3. Dataset Splitting We split the datasets into a training and a test set. The training set is only used for the model creation (step 4) while the test set will only intervene for the evaluation of the model (step 5). We choose to keep 70% of the data for training and the remaining 30% for testing. Moreover, we shuffle the datasets and split them with a stratified sampling method.

4. Model Creation The performances of the different ML models depend greatly on their hyper-parameters. We use a grid search to evaluate different hyper-parameters combinations and build the best models for packing detection. The hyper-parameters grid tested for each algorithm is depicted in Table 1. In our experiment, we use the stratified 10-fold cross-validation on the training set and the balanced accuracy scoring method to evaluate each combination of hyper-parameters.

<table>
<tr><td colspan="3">DT</td><td></td></tr>
<tr><td colspan="4">RF</td></tr>
<tr><td></td><td colspan="2">GBDT</td><td></td></tr>
<tr><td>criterion</td><td>min_sample_leaf</td><td>max_depth</td><td>n_estimators</td></tr>
<tr><td>gini/entropy</td><td>2-12</td><td>2-12</td><td>10-39</td></tr>
</table>

Table 1: Hyper-parameters for each type of model

5. Model Evaluation Using the best hyper-parameters found with the training set, we evaluate the models on the test set. We use different metrics: the balanced accuracy, the precision, the recall, the F1-score and the ROC curve.

6. Model Explanation Finally, we use explainable artificial intelligence methods (XAI) to analyse our classifiers and their decision with SHAP values (SHapley Additive exPlanations). This technique, which comes from the game theory field, determines the individual contribution of each feature on the predictions made by the model. Analyzing models with these techniques is good practice to avoid spurious correlation [3] and better understand our models.

5 Experiments and results

This section presents the different experiments with dynamic features extracted thanks to PANDI [33] and static features already explored in literature [10].

Training on the* CleanPack *dataset The results of the models trained on the CleanPack dataset are presented in Table 2. This dataset is classified with high accuracy by all models. The ROC curves of the 9 models are depicted in Figure 10. The DT models are slightly less efficient than the RF and GBDT models as we can see from their AUC values.

These excellent results are likely linked to the simplicity of the dataset with well-known packers. The samples have been manually packed, likely resulting in a substantial degree of commonality among them, thereby rendering their classification more straightforward.

We evaluate these models on the WildPack dataset to see if they generalize to more complex datasets. Unfortunately, the results were not satisfactory. The recall scores were very low which indicates that many packed samples were misclassified. This is likely due to the variety and complexity of packers that appear in the WildPack dataset. Moreover, the CleanPack dataset is solely composed of cleanware while the WildPack dataset mostly comprises malware.

		BA	P	R	F1
dynamic	DT	0.97	0.99	1.0	0.99
	RF	0.99	0.99	1.0	1.0
	GBDT	0.98	0.99	1.0	1.0
static	DT	0.99	1.0	1.0	1.0
	RF	0.99	0.99	1.0	1.0
	GBDT	1.0	1.0	1.0	1.0
both	DT	0.99	0.99	1.0	1.0
	RF	0.99	1.0	1.0	1.0
	GBDT	0.99	1.0	1.0	1.0

Table 2: Metrics for the models trained on the CleanPack dataset

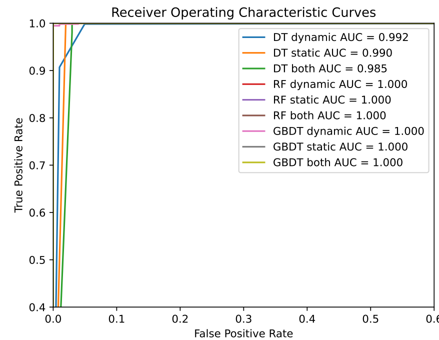


Fig. 10: ROC curves

Training on the* WildPack *dataset The metrics of the models on the second dataset are listed in Table 3. We notice that the precision, recall and F1-scores are high for most models. The best performing models are the RF and the GBDT trained on the combination of both feature sets (as we can see in Table 3 and Figure 11). The high recall but lower balanced accuracy put into light that the models detect the packed samples better than the not packed samples. In the context of a malware analysis toolchain, it is important to have a high recall to

detect packing and perform a correct analysis of the sample (even at the cost of a few false positives).

		BA	P	R	F1
dynamic	DT	0.87	0.97	0.93	0.95
	RF	0.86	0.96	0.98	0.97
	GBDT	0.88	0.97	0.98	0.97
static	DT	0.83	0.96	0.93	0.94
	RF	0.77	0.93	0.98	0.95
	GBDT	0.83	0.95	0.98	0.96
both	DT	0.89	0.97	0.95	0.96
	RF	0.91	0.97	0.98	0.98
	GBDT	0.91	0.98	0.98	0.98

Table 3: Metrics for the models trained on the WildPack dataset

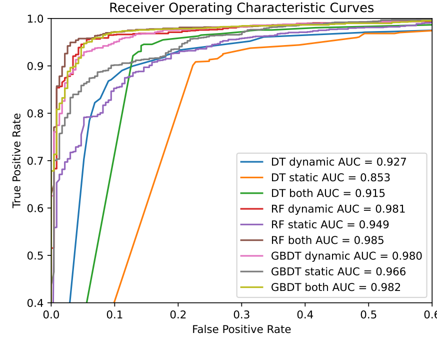


Fig. 11: ROC curves

Training on the* MixPack *dataset We notice that metrics in Table 4 slightly improve compared to Table 3. The AUC values improve as well, as we can see by comparing the ROC curves from Figures 11 and 12.

However, when we analyze in detail the results of the classification on the test set, we notice that the CleanPack part of the test set is much better classified than the WildPack part. If we test the models exclusively on the WildPack part of the dataset, we find that the scores are nearly identical to those from the previous experiment. Therefore, including manually packed samples in the training set does not seem to improve the performance of our classifiers.

		BA	P	R	F1
dynamic	DT	0.90	0.97	0.96	0.97
	RF	0.90	0.97	0.98	0.98
	GBDT	0.91	0.97	0.98	0.98
static	DT	0.87	0.96	0.95	0.96
	RF	0.85	0.96	0.98	0.97
	GBDT	0.86	0.96	0.98	0.97
both	DT	0.91	0.98	0.96	0.97
	RF	0.92	0.98	0.98	0.98
	GBDT	0.93	0.98	0.98	0.98

Table 4: Metrics for the models trained on the MixPack dataset

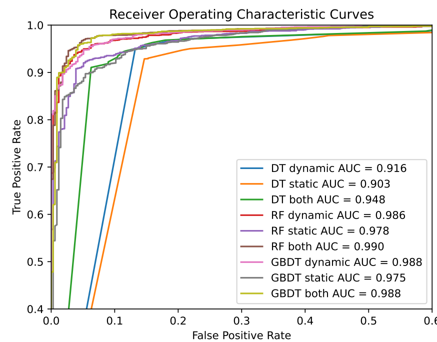


Fig. 12: ROC curves

Interpretable machine learning Analyzing the SHAP values of our models allows us to understand the features that influence their decisions [37]. We investigate our Random Forest models on the different datasets because they perform best in our experiments. We chose to focus on models trained on dynamic features and on the combination of static and dynamic features since models based only on static features have already been studied in the literature [8]. The SHAP values of the top 10 features for each of our 6 models are shown in Figure 13. Figures 13 (a), (c) and (e) represent models trained on the dynamic feature set while Figures 13 (b), (d) and (f) represent models trained on the static and dynamic feature sets. We highlight the dynamic features in pink.

The classification of each sample is visualized as a dot on the graph. Dots located on the left side of the graph indicate a decision towards the *not packed* label. Conversely, dots located on the right side of the graph indicate a decision towards the *packed* label. Red dots indicate high feature values for the given sample, while blue dots indicate low feature values.

In Figure 13(a), the CleanPack dataset is classified using many features related to executed bytes. Furthermore, in Figure 13(b), we see that static features play an important role in the classification of the CleanPack dataset, while dynamic features have more influence on the classification of the WildPack dataset (Figure 13(d)). These observations are consistent because the CleanPack dataset represents a more restricted set of packers than the WildPack dataset. Therefore, it is more suitable for classification by patterns and signatures. The WildPack dataset requires broader and more robust features because it consists of a wider variety of packers.

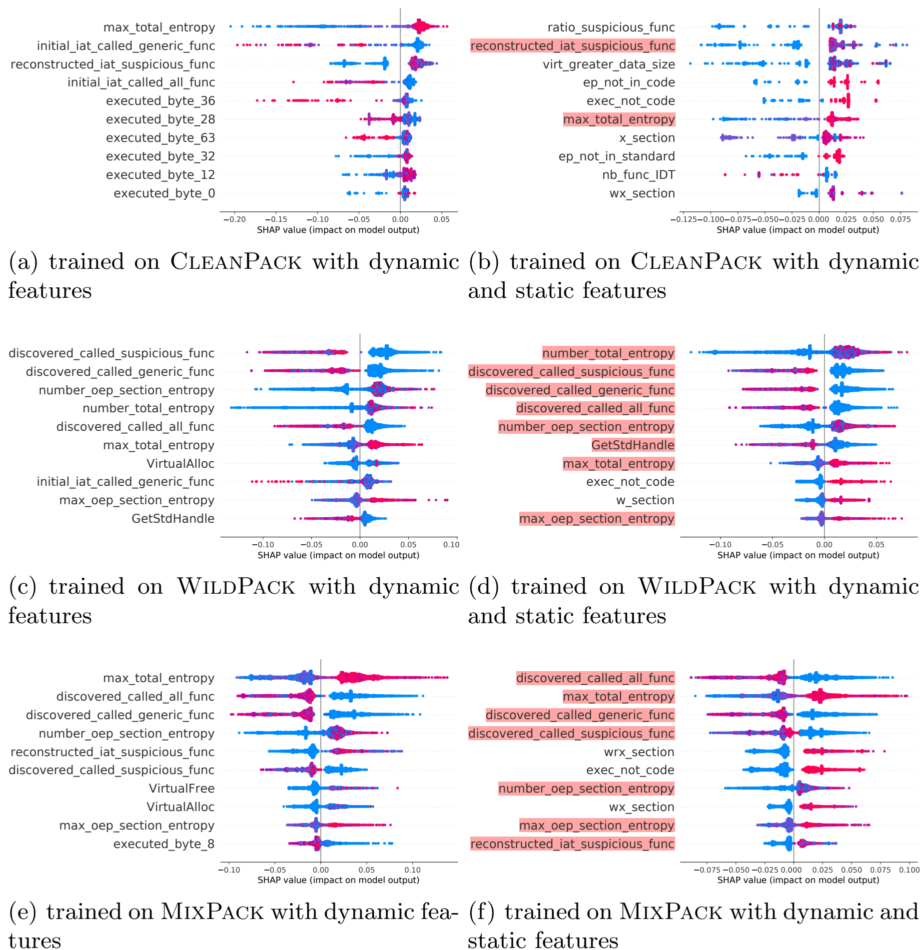
The features max_total_entropy and max_oep_section_entropy represent the maximum entropy that can be obtained from the whole binary and the section containing the OEP, respectively. As we can see, a low entropy value influences the classification towards the *not packed* result and vice versa.
Other important features are related to entropy: number_total_entropy and number_oep_section_entropy. These represent the number of entropy points collected during the execution and are directly related to the number of blocks executed. The classifier tends to consider a sample as packed when these values are high. We can explain this tendency by the fact that the unpacking routine usually implies that many blocks are executed and many entropy points generated.

The reconstructed_iat_suspicious_func feature represents the number of suspicious functions retrieved by the `GetProcAddress` function. A high number of these indicates that the sample may be packed.

The number of functions called by the original IAT are represented by the features initial_iat_called_generic_func and initial_iat_called_all_func. A low value for these features pushes the classification towards the *packed* result. In fact, a packed sample will only use the original IAT to decompress and reconstruct its original IAT. The majority of calls will be made from the reconstructed IAT.

Finally, the API calls used are represented by discovered_called_suspicious_func, discovered_called_generic_func and discovered_called_all_func. They seem to take higher values for not packed samples than for packed samples. This may be be-

cause malicious samples call native windows functions directly instead of using high level API calls (i.e.: a high-level API is usually based on lower-level API that we all record).



(a) trained on CleanPack with dynamic features

(b) trained on CleanPack with dynamic and static features

(c) trained on WildPack with dynamic features

(d) trained on WildPack with dynamic and static features

(e) trained on MixPack with dynamic features

(f) trained on MixPack with dynamic and static features

Fig. 13: SHAP values

Observing our results, we could conclude that dynamic features have a high impact on machine learning algorithms, improving their accuracy and appearing more robust. In particular, entropy features seem to be selected as an influential feature by all the different models studied. However, extracting dynamic features is costly and it could be interesting to extend PANDI to directly detect the original entry point when a sample is detected as packed. This will further improve our malware analysis workflow while maintaining a similar overhead.

Limitations Our approach faces different limitations related to PANDI and our experimental setup. While PANDI takes some time to analyze, it allows us to extract extensive information about system calls and memory writes. However, we do not have access to the sample's execution state and use a timer to terminate the virtual machine after a certain period of time and then analyze the replay. This method avoids the need for direct feedback from the virtual machine, but could interrupt the execution of the sample before it finishes. As noted by Kuchler et al. [29], the vast majority of malicious activity of a malware is observed within the first two minutes of its execution. Moreover, we focus on the unpacking part of the execution and not the full malicious behavior, which may include more evasion techniques. Therefore, the impact on our results should be limited.

As discussed by Arp et al. [3], the dataset used by machine learning for cybersecurity tasks is difficult to construct without biases. First, the labeling process of the dataset is generally difficult and should rely on other techniques (antivirus, heuristics, manual inspection,...) with their own biases and errors. This problem could be solved by the Packing Box [23] as discussed in the next section. Datasets could also be subject to temporal shift, i.e. the distribution and characteristics of the packed samples could change over time. While we do not consider this type of evaluation, the dataset obtained from [1] already provides a good representation of the packing ecosystem.

6 Conclusion and Future Work

We have studied several methods in the literature that are based on dynamic analysis. Our research shows that the use of dynamic features within a machine learning framework can yield impressive results, matching or even exceeding those achieved with static features. However, their significant extraction cost should also be considered when designing an efficient malware analysis toolchain. There is still room for improvement in several aspects.

We can use feature selection to extract only the more important features (e.g.,*max_total_entropy*), thus reducing the extraction cost. Further exploration of the effectiveness of dynamic features in classifying different types of packers would be an interesting prospect. The machine learning approach could help in identifying the type of packer (protector, compressor, etc.) and/or the complexity of the packer [49]. However, creating ground truth with sufficient information for such experiments is challenging. Contributing to the Packing Box [23] by adding information about packing complexity and packing type for each packer would help our future research as well as other studies. In addition, the ability to pack malware samples could be added to this tool to improve dataset diversity and help our models to generalize to real-world scenarios.

Moreover, it is imperative to address adversarial aspects when applying machine learning to a cybersecurity problem [3]. First, the sandbox used by PANDI should be improved to minimize the risk of malware evasion (e.g., adding automatic mouse movements, launching other processes in the background, etc.). We

could generate adversarial examples based on static features using tools such as MAB-malware [47], but the task becomes significantly more challenging when using dynamic features [45]. Finally, since malware and packers evolve over time, it seems necessary to analyze the performance of our models on an evolving real-world dataset. As a result of this evolution, the need and cost of retraining our models should be studied [8]. We hope that dynamic features would be more robust to temporal evolution, since it is harder to change semantic properties than syntactic properties of a binary.

In addition, our approach could be extended to include unpacking capabilities. Specifically, machine learning algorithms can be used to trigger a memory dump when a sample is identified as unpacked during execution replay. This would require defining features that characterize the end of the unpacking process and the beginning of the execution of the original executable. Existing techniques ([16], [32], [17], [36]) could be adapted to our dynamic analysis and machine learning approach to efficiently detect the original entry point.

Finally, investigating unsupervised machine learning algorithms could help us overcome the lack of information about the samples [39]. This would regroup samples with high similarity. The resulting clusters could then be examined to identify common patterns. Dynamic features would certainly provide new insights into the behavioral similarities of packers. The information identified by this method could be used to develop customized unpacking strategies.

Acknowledgments. This research is supported by the Walloon region's CyberExcellence program (grant #2110186).

References

1. Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., Kruegel, C.: When malware is packin'heat; limits of machine learning classifiers based on static analysis features. In: NDSS 2020 (2020)
2. Ahmadi, M., Ulyanov, D., Semenov, S., Trofimov, M., Giacinto, G.: Novel feature extraction, selection and fusion for effective malware family classification. In: Proceedings of the sixth ACM conference on data and application security and privacy. pp. 183–194 (2016)
3. Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., Rieck, K.: Dos and don'ts of machine learning in computer security. In: USENIX Security 22. pp. 3971–3988 (2022)
4. Avast: Pelib. https://github.com/avast/pelib (2023)
5. Bat-Erdene, M., Park, H., Li, H., Lee, H., Choi, M.S.: Entropy analysis to classify unknown packing algorithms for malware detection. International Journal of Information Security **16**, 227–248 (2017)
6. Bertrand Van Ouytsel, C.H., Crochet, C., Dam, K.H.T., Legay, A.: Tool papersema: Symbolic execution toolchain for malware analysis. In: International Conference on Risks and Security of Internet and Systems. pp. 62–68. Springer (2022)
7. Bertrand Van Ouytsel, C.H., Dam, K.H.T., Legay, A.: Symbolic analysis meets federated learning to enhance malware identifier. In: Proceedings of the 17th International Conference on Availability, Reliability and Security. pp. 1–10 (2022)

8. Bertrand Van Ouytsel, C.H., Dam, K.H.T., Legay, A.: Analysis of machine learning approaches to packing detection. Computers & Security p. 103536 (2023)
9. Bertrand Van Ouytsel, C.H., Legay, A.: Malware analysis with symbolic execution and graph kernel. In: Secure IT Systems: 27th Nordic Conference, NordSec 2022, Reykjavic, Iceland, November 30–December 2, 2022, Proceedings. pp. 292–310. Springer (2023)
10. Biondi, F., Enescu, M.A., Given-Wilson, T., Legay, A., Noureddine, L., Verma, V.: Effective, efficient, and robust packing detection and classification. Computers & Security **85**, 436–451 (2019)
11. Biondi, F., Given-Wilson, T., Legay, A., Puodzius, C., Quilbeuf, J.: Tutorial: An overview of malware detection and evasion techniques. In: Leveraging Applications of Formal Methods, Verification and Validation. Modeling: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I 8. pp. 565–586. Springer (2018)
12. Biondi, F., Given-Wilson, T., Legay, A., Puodzius, C., Quilbeuf, J.: Tutorial: An overview of malware detection and evasion techniques. In: ISoLA 2018. pp. 565–586. Springer (2018)
13. Biondi, F., Josse, S., Legay, A.: Bypassing malware obfuscation with dynamic synthesis. ERCIM News (106) (2016)
14. Biondi, F., Josse, S., Legay, A., Sirvent, T.: Effectiveness of synthesis in concolic deobfuscation. Computers & Security **70**, 500–515 (2017)
15. packing box: dataset-packed-pe. https://github.com/packing-box/dataset-packed-pe (2023)
16. Cheng, B., Ming, J., Fu, J., Peng, G., Chen, T., Zhang, X., Marion, J.Y.: 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. pp. 395–411 (2018)
17. Cheng, B., Ming, J., Leal, E.A., Zhang, H., Fu, J., Peng, G., Marion, J.Y.: {Obfuscation-Resilient} executable payload extraction from packed malware. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 3451–3468 (2021)
18. Choi, Y.s., Kim, I.k., Oh, J.t., Ryou, J.c.: Pe file header analysis-based packed pe file detection technique (phad). In: International Symposium on Computer Science and its Applications. pp. 28–31. IEEE (2008)
19. Dam, K.H.T., Given-Wilson, T., Legay, A.: Unsupervised behavioural mining and clustering for malware family identification. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing. pp. 374–383 (2021)
20. Dam, K.H.T., Given-Wilson, T., Legay, A., Veroneze, R.: Packer classification based on association rule mining. Applied Soft Computing **127**, 109373 (2022)
21. Dambra, S., Han, Y., Aonzo, S., Kotzias, P., Vitale, A., Caballero, J., Balzarotti, D., Bilge, L.: Decoding the secrets of machine learning in malware classification: A deep dive into datasets, feature extraction, and model performance. arXiv preprint arXiv:2307.14657 (2023)
22. D'Hondt, A.: peid. https://github.com/packing-box/peid (2023)
23. D'Hondt, A., Van Ouytsel, C.H.B., Legay, A.: Experimental toolkit for manipulating executable packing. arXiv preprint arXiv:2302.09286 (2023)
24. Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., Whelan, R.: Repeatable reverse engineering with panda. In: Proceedings of the 5th Program Protection and Reverse Engineering Workshop. pp. 1–11 (2015)
25. ENISA: Threat landscape report 2022 (2022), https://www.enisa.europa.eu/publications/enisa-threat-landscape-2022

26. Islam, R., Tian, R., Batten, L.M., Versteeg, S.: Classification of malware based on integrated static and dynamic features. Journal of Network and Computer Applications **36**(2), 646–656 (2013)
27. Kancherla, K., Donahue, J., Mukkamala, S.: Packer identification using byte plot and markov plot. Journal of Computer Virology and Hacking Techniques **12**, 101–111 (2016)
28. Kotov, V., Wojnowicz, M.: Towards generic deobfuscation of windows api calls. arXiv preprint arXiv:1802.04466 (2018)
29. Küchler, A., Mantovani, A., Han, Y., Bilge, L., Balzarotti, D.: Does every second count? time-based evolution of malware behavior in sandboxes. In: NDSS (2021)
30. Kwiatkowski, I.: Manalyze. https://github.com/JusticeRage/Manalyze (2023)
31. Li, X., Shan, Z., Liu, F., Chen, Y., Hou, Y.: A consistently-executing graph-based approach for malware packer identification. IEEE Access **7**, 51620–51629 (2019)
32. Lim, C., Ramli, K., Kotualubun, Y.S., et al.: Mal-flux: Rendering hidden code of packed binary executable. Digital Investigation **28**, 83–95 (2019)
33. Lucca, S., Wauters, D.: Pandi. https://github.com/dimitriwauters/PANDI (2023)
34. Lyda, R., Hamrock, J.: Using entropy analysis to find encrypted and packed malware. IEEE Security & Privacy **5**(2), 40–45 (2007)
35. Mantovani, A., Aonzo, S., Ugarte-Pedrero, X., Merlo, A., Balzarotti, D.: Prevalence and impact of low-entropy packing schemes in the malware ecosystem. In: NDSS (2020)
36. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: Fast, generic, and safe unpacking of malware. In: ACSAC. IEEE (2007)
37. Molnar, C.: Interpretable machine learning. Lulu. com (2020)
38. Muralidharan, T., Cohen, A., Gerson, N., Nissim, N.: File packing from the malware perspective: Techniques, analysis approaches, and directions for enhancements. ACM Computing Surveys **55**(5), 1–45 (2022)
39. Noureddine, L., Heuser, A., Puodzius, C., Zendra, O.: Se-pac: A self-evolving packer classifier against rapid packers evolution. In: CODASPY (2021)
40. Oberhumer, M., Molnar, L., Reiser, J.: UPX, the Ultimate Packer for eXecutables, https://upx.github.io/
41. Perdisci, R., Lanzi, A., Lee, W.: Classification of packed executables for accurate computer virus detection **29**(14), 1941–1946
42. panda re: Panda. https://github.com/panda-re/panda (2023)
43. Said, N.B., Biondi, F., Bontchev, V., Decourbe, O., Given-Wilson, T., Legay, A., Quilbeuf, J.: Detection of mirai by syntactic and behavioral analysis. In: ISSRE. pp. 224–235. IEEE (2018)
44. Sebastio, S., Baranov, E., Biondi, F., Decourbe, O., Given-Wilson, T., Legay, A., Puodzius, C., Quilbeuf, J.: Optimizing symbolic execution for malware behavior classification. Computers & Security p. 101775 (2020)
45. Shafiei, A., Rimmer, V., Tsingenopoulos, I., Desmet, L., Joosen, W.: Position paper: On advancing adversarial malware generation using dynamic features. In: Proceedings of the 1st Workshop on Robust Malware Analysis. pp. 15–20 (2022)
46. Smith, M.R., Johnson, N.T., Ingram, J.B., Carbajal, A.J., Haus, B.I., Domschot, E., Ramyaa, R., Lamb, C.C., Verzi, S.J., Kegelmeyer, W.P.: Mind the gap: On bridging the semantic gap between machine learning and malware analysis. In: Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security. pp. 49–60 (2020)
47. Song, W., Li, X., Afroz, S., Garg, D., Kuznetsov, D., Yin, H.: Mab-malware: A reinforcement learning framework for attacking static malware classifiers. arXiv preprint arXiv:2003.03100 (2020)

48. Treadwell, S., Zhou, M.: A heuristic approach for detection of obfuscated malware. In: IEEE International Conference on Intelligence and Security Informatics, ISI 2009, Dallas, Texas, USA, June 8-11, 2009, Proceedings. pp. 291–299. IEEE (2009)
49. Ugarte-Pedrero, X., Balzarotti, D., Santos, I., Bringas, P.G.: Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In: 2015 IEEE Symposium on Security and Privacy. pp. 659–673. IEEE (2015)
50. Ugarte-Pedrero, X., Santos, I., García-Ferreira, I., Huerta, S., Sanz, B., Bringas, P.G.: On the adoption of anomaly detection for packed executable filtering. Computers & Security **43**, 126–144 (2014)
51. Wu, C., Shi, J., Yang, Y., Li, W.: Enhancing machine learning based malware detection model by reinforcement learning. In: Proceedings of the 8th International Conference on Communication and Network Security. pp. 74–78 (2018)
52. Yan, X., Han, J.: gspan: Graph-based substructure pattern mining. In: 2002 IEEE International Conference on Data Mining, 2002. pp. 721–724. IEEE (2002)
53. Yason, M.V.: The art of unpacking. Retrieved Feb **12**, 2008 (2007)
54. Zakeri, M., Faraji Daneshgar, F., Abbaspour, M.: A static heuristic approach to detecting malware targets. Security and Communication Networks **8**(17), 3015–3027 (2015)