

Experimental Toolkit for Manipulating Executable Packing

Alexandre D'Hondt^[0000–0001–6160–8624], Charles Henry Bertrand Van Ouytsel^[0000–0001–5720–6569], and Axel Legay^[0000–0003–2287–8925]

Université Catholique de Louvain, Rue Archimede 1, Louvain-la-Neuve, Belgium
{alexandre.dhondt,charles-henry.bertrand,axel.legay}@uclouvain.be

Abstract. Executable packing is a well-known problematic especially in the field of malware analysis. It often consists in applying compression or encryption to a binary file and embedding a stub for reversing these transformations at runtime. This way, the packed executable is more difficult to reverse-engineer and/or is obfuscated, which is effective for evading static detection techniques. Many detection approaches, including machine learning, have been proposed in the literature so far, but most studies rely on questionable ground truths and do not provide any open implementation, making the comparison of state-of-the-art solutions tedious. We thus think that first solving the issue of repeatability shall help to compare existing executable packing static detection techniques. Given this challenge, we propose an experimental toolkit, named Packing Box, that leverages automation and containerization in an open source platform that brings a unified solution to the research community. We present our engineering approach for designing and implementing our solution. We then showcase it with a few basic experiments, including a performance evaluation of open source static packing detectors and training a model with machine learning pipeline automation. This introduces the toolset that will be used in further studies.

Keywords: executable packing · packer detection · packed classification · toolkit · machine learning · static analysis

1 Introduction

Executable packing can be seen as a set of transformations on a binary file (including compression or encryption) that modifies it so that it recovers its logic at runtime. It can be applied to many executable formats but is particularly diversified for the Windows Portable Executable (PE) format. Its use dates from the 1990's already, especially with malicious software for evading detection by antivirus software that typically use signatures and ad hoc heuristics. Packing detection has already been addressed in many researches since the early 2000's. Its main challenge is *performance*, involving static and dynamic analysis. While dynamic techniques can collect more information, they require executing the target binary, which incurs an additional cost. Static techniques, on the other

hand, are faster as the binary is not executed. Many static techniques were presented in the past, relying on entropy [15], signatures [12, 18, 23], control flow graphs [5, 6] or even ad hoc heuristics [8, 13, 14, 25]. Many researches [1, 4, 7, 19] have been published, often bringing excellent results to the forefront but never sharing a solution to repeat experiments and compare methods with each other. A recent survey [17] highlights the different works regarding packing detection and the wide variety of datasets (often custom-made) used in literature, only providing a qualitative analysis of state-of-the-art detection techniques. In this context, it is not easy to compare the results from several studies based on a same dataset. *Repeatability* is thus a major concern. To do this, we need to standardize the process of creating and formatting datasets but also provide a single operations pipeline so that each researcher can reproduce results from other studies for comparison. Moreover, machine learning has become the emerging trend for gaining in accuracy while keeping execution fast enough. Its optimal use, considering supervised learning, involves datasets of collected packed and not packed executables, but also capturing their most relevant characteristics in a trade-off between the number of features and their quality while maintaining sufficient accuracy. *Feature engineering* is thus yet another concern. Furthermore, addressing machine learning requires making datasets that can be considered generalized enough, otherwise the resulting model will not be relevant for every target executable. A major concern then lies in the *relevance of the available datasets* and the *correctness* in the establishment of a ground truth considered for training models.

The above considerations show that there is still room for improvement in packing detection, including tools to find and prove the best ways to do this. While *performance* and underlyingly *feature engineering* in the use of machine learning are the ultimate goals of our research, the primary focus of this work is on designing a toolkit that can ensure *repeatability* and help us build accurate and shareable ground truths and automate the entire machine learning pipeline, in preparation for further experiments to find the best features and models. Therefore, this work does not yet aim to make a scientific contribution, but rather proposes a framework that will constitute the reference toolbox for future studies. Given our focus, the design questions hereafter arise.

- Q1 How can we enhance *repeatability* of the experiments of packing detection ?
- Q2 How can we build *accurate ground truths* ?
- Q3 How can we measure the *performance of state-of-the-art static detectors* ?

To date, as far as we know, there exists no framework that gathers packing techniques and tools yet. For this purpose and in order to address the issue of *repeatability*, our contribution is to propose an experimental toolkit aptly named the Packing Box [11] that automates dataset generation for building accurate ground truths in a shareable format and provides tools for training machine learning models in a convenient and repeatable way. This toolset also includes data visualization utilities and allows to assess the performance of open detectors (e.g. PEiD or DIE).

The remainder of this paper first presents, in Section 2, some background on the topic of packing detection, including general considerations about packing, common static detection techniques and machine learning, together with requirements for our experimental toolkit that we could deduce. In Section 3, it then outlines our solution, including its design and implementation, its capabilities and how it can help us solve our questions. Then in Section 4, we showcase our solution with some experiments focused on the evaluation of the performance of some open static detectors and model training with our automated machine learning pipeline. We finally conclude in Section 5 on what we have achieved and what remains to be further developed and studied.

2 Background

This section aims to present the most important concepts that will be the source material of our toolkit. All along those, we deduce the requirements (REQ) for our design and implementation explained in Section 3.

Packing can be defined as a set of transformations that modify the layout of a binary file (an image file, typically `.exe`, or an object file, e.g. a DLL) without affecting its logic, adding a stub that allows either to revert these transformations when executing the binary for restoring its original logic in memory or to execute an equivalent one. In particular, this allows to hide the logic when the binary resides on the disk, therefore preventing certain static detection systems like antiviruses from viewing the real payload (i.e. causing evasion from detection). But it can also be used to reduce size or even to protect the target binary against piracy (e.g. by adding a prompt at startup to require a license key). By nature, packing is thus not limited to malware.

We distinguish seven types of distinct transformations used jointly by packers. *Bundling* makes a single executable with multiple files and is the typical behavior for portable versions of executables produced by tools like Cameyo, ThinApp or even Enigma Virtual Box. *Compression* allows to reduce the binary’s size, sometimes for obfuscating some of its regions. Most packers are compressors, including some very popular ones like UPX and UPack, typically using LZMA, LZ77 or Deflate as compression algorithms. *Encoding* is a reversible operation that changes a given character set to another, possibly of different size. Base64 and a simple XOR with a 1-byte key of value 1 are examples of encodings that could be used. *Encryption* allows to obfuscate parts of the binary and to protect the binary from piracy. It is reversible and requires a key, either embedded or requested at runtime. *Mutation* alters the executable’s code so that it uses a modified instruction set and architecture, e.g. using metamorphism (that is, the executable mutates its code at each execution). *Protection* makes reverse engineering harder by using anti-debugging, anti-tampering or other tricks that prevent analysis tools to properly do their job (e.g. Themida). *Virtualization* involves embedding a virtual machine that allows to virtualize executable’s instructions for the sake of isolating it from the underlying operating system. This can be used for portability (e.g. BoxedApp Packer and Enigma Virtual Box) but

for obfuscation as well (e.g. Molebox). While the aforementioned taxonomy is not the primary focus of this work, it shows the variety of packing operations and the potential for classifying packers per *categories*.

(REQ01) Our experimental toolkit should allow to deal with the categories from the aforementioned taxonomy. This involves supporting multi-label classification.

Many previous researches focused on telling whether a binary is packed or not (binary classification) [13, 15] while others aimed to go further and classify packers according to their names (e.g. UPX or Themida) or *families* (as of the term used by Biondi *et al.* [4]) [7, 23]. This term of *family* is used as a single product may yield variants in function of the version and point in time it was released. Moreover, a packer may have input options that can shape the layout of the packed executable differently. This is also a challenge if we want to be able to train models that can detect a whole family or a category (e.g. compressors) while the packer has very dissimilar versions and a panel of available options.

(REQ02) Our toolkit should handle different packer versions and configurations.

(REQ03) It should also allow to handle either binary or multi-class classification.

Among the techniques for packing detection, static ones consist of actions performed without ever executing the binary (on the contrary of dynamic techniques). Some of them rely on characteristics of the executable formats and others on computed values. In the current literature, we find various methods relying on entropy (such as Bintropy [15] or REMINDER [13]), signatures (based on opcodes [23], byte sequences [18] or metadata [12]), control flow graphs (relying on similarity [5] and decompilation [6]) or heuristics (based on a vector of structural or header features and using a distance metric [8] or a risk score [14, 25]). Most of the aforementioned studies focused on malware analysis and Windows Portable Executables, often considering a single static technique. Most of them achieved good results on specific datasets and showed few or no comparison of performance with other approaches, often relying on popular signature-based open tools like PEiD¹, i.e. using them to validate a reference ground truth.

(REQ04) Our toolkit shall handle more than simply Portable Executables (PE) from Windows, i.e. Executable and Linking Format (ELF) from Unix and if possible Mach Objects from MacOS.

(REQ05) It should implement as many readily available static detection tools and techniques as possible. This will allow for further comparison of the state-of-the-art tools and techniques and it could also greatly help analyze weaker techniques like those based on entropy as shown by Mantovani *et al.* [16].

Even though many researches exhaustively detail the functioning of their approaches, it is rare that an open implementation is provided, therefore mak-

¹ <http://www.secretashell.com/codomain/peid>

ing the task of comparing approaches very tedious and time-consuming as it is necessary to re-implement tools relying on these approaches.

(REQ06) Our toolkit shall be open and transparent to other researchers.

Various sources of executables can be found in the wild, especially related to malware, but few provides packer labels or they have their labels determined based on proprietary or unpublished detection algorithms. It appears from many studies that some open datasets do not survive over time, making performance testing even more tedious with approaches from studies using more recent datasets. Among the interesting works for offering datasets, we note EM-BER [2] (2018, large collection of features from PE files), PackingData² (2019, small collection of native PE files from Windows packed with 19 packers – reference dataset of Choi *et al.* [7]) and its extension [10] (2021, 6 more packers added) but also a new similar set with ELF samples³ (2022, native Linux files packed with 6 different packers). There are also older datasets that were used in some studies and that are now unmaintained like Malfease⁴ (2008), VX Heaven⁵ (from 2010 to 2017) or even ViruSign⁶ (stopped in 2020). Some more recent datasets are still actively maintained and usable, like Malware Bazaar⁷ and VirusShare⁸.

(REQ07) Our toolkit shall allow easy ingestion of items from external data sources for replicating datasets from related studies.

Given these data sources, it is frequent that labels are not standardized and even not accurate or verifiable. This issue can be addressed by building a superdetector [4, 12, 24] gathering multiple detectors and aggregating their results through a decision heuristic handling their normalized outputs. Based on ad hoc heuristics (e.g. a majority vote), we can then filter out labels that are not considered fully reliable and obtain a ground truth that is assumed to be of good quality. Another approach for building an accurate ground truth is to manually pack native binaries [7]. This is de facto the most reliable way to build a dataset but also the most time-consuming. Presently, as far as we know, there does not exist any solution yet that automates this process.

(REQ08) Our toolkit shall include as many detectors as possible and define a superdetector for reproducing experiments based on sets of detectors.

(REQ09) Our experimental toolkit shall automate packers to allow for bulk-creation of datasets of packed samples.

² <https://github.com/chesvectain/PackingData>

³ <https://github.com/packing-box/dataset-packed-elf>

⁴ <https://web.archive.org/web/20141221153307/http://malfease.oarci.net>

⁵ <https://web.archive.org/web/20170817143838/http://vxheaven.org>

⁶ <https://www.virusign.com>

⁷ <https://bazaar.abuse.ch/browse>

⁸ <https://virusshare.com>

Machine learning tends to become the best way to tie multiple techniques together, using their results as features to improve the models. While simpler techniques like custom heuristics use feature vectors with simple distance metrics [8], machine learning algorithms offer far more possibilities in shaping models and leveraging features. Currently, *Supervised learning* is the area most covered by the literature, which exploits feature sets of different sizes [1,3,4,20,22]. *Semi-supervised learning* is relatively few covered [21,26] and *Unsupervised learning* is even less so and has only recently been addressed [19]. Today, it appears that supervised learning was already extensively tackled while not identifying best features yet, except in the study of Bertrand Van Ouytsel *et al.* [3]. It should also be noted that no implementation to automate the learning pipeline and compare model results has yet been provided.

(REQ10) Our toolkit shall automate the ML pipeline and cover supervised, semi-supervised and unsupervised learning as well.

3 Toolkit

This section explains the design and implementation choices with regard to the requirements mentioned in Section 2.

Given our requirements, our toolkit needs to rely on many existing libraries. While the framework could be designed upon several host operating systems, our objective is to provide a single platform and not a set of toolkits for different executable formats. Moreover, given the nature of some of the required libraries, i.e. for dealing with the different formats, and the fact that we may be working with malware samples, it involves, as a good practice, isolation from the host. Therefore, as virtualization may be too heavy for our needs in terms of performance and maintenance, containerization can provide an elegant solution. We thus plan our design starting with a container with several layers: its operating system, the libraries for handling our scope (at a minimum including the machinery for running Windows PE and Unix ELF formats) and our toolset.

Given our design, we choose Docker⁹ as the containerization technology and we host our project on GitHub (REQ06). Figure 1 depicts our layered architecture. At the lower layer, we rely on Ubuntu 22.04 with no desktop interface, thus only relying on a command-line interface. We choose this OS as it is particularly convenient for adding libraries that can extend our scope. Furthermore, for our intermediate layer, this OS supports the well-known frameworks Wine¹⁰ for executing PE binaries and Mono¹¹ for .NET executables. Of course, it natively supports running ELF binaries and can also run Mach Objects with Darling¹², a translation layer to run MacOS software on Linux (REQ04). For our upper layer, we select Python as our programming language as it is shipped with Ubuntu and

⁹ <https://www.docker.com>

¹⁰ <https://www.winehq.org>

¹¹ <https://www.mono-project.com>

¹² <https://www.darlinghq.org>

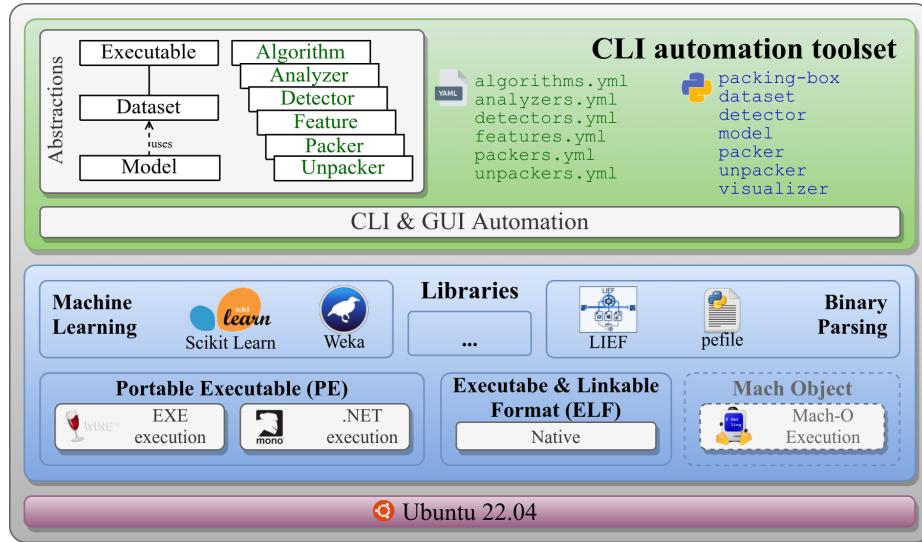


Fig. 1. Packing Box's architecture

it has a large support for libraries that can parse binaries and that implement ML algorithms. For building our toolset, we use LIEF¹³ as our binary parsing library and select Scikit Learn¹⁴ as our primary ML framework. We also support Weka¹⁵ (Java-based) via Python with a dedicated package that provides a wrapper for its execution. We call our resulting Docker image "Packing Box" [11].

Since our goal is to provide researchers with a convenient interface for using our toolkit, we pay attention to the integration of new elements that should not require profound changes to the container. To this end, we abstract the raw material from our toolkit, considering elements that require no tunability and others that do. For the first type, we define *Executable* to be handled in a *Dataset* that can itself be processed by a *Model*. For tunable items that should provide the extensibility of our toolkit (REQ05), i.e. packers or detectors, we define an abstraction that holds attributes that we can customize through YAML¹⁶ configuration files. This way, one dictionary per abstraction (see the above layer in Figure 1) is defined. Among the attributes, we define metadata including the item's source, applicable executable formats and packing categories (REQ01), interesting references and status. We also define the installation and execution steps and also variants (REQ02). Thanks to these, our toolset can include custom libraries that will contain the logic to handle simple but also more complex items. For instance, as our container is headless (using Ubuntu with no desktop

¹³ <https://lief-project.github.io>

¹⁴ <https://scikit-learn.org>

¹⁵ <https://waikato.github.io/weka-site>

¹⁶ Yet Another Markup Language

environment), dealing with GUI-based Windows packers requires more complex processing relying on the emulation of a desktop environment. Thanks to the *Packer* abstraction, it is easy to apply plenty of packers to input samples for generating datasets according to the *Dataset* abstraction (REQ09). Moreover, we also define the *Algorithm* abstraction for ML algorithms including static and cross-validation parameters (REQ10). In addition, the learning pipeline supports either binary or multi-class classification (REQ03).

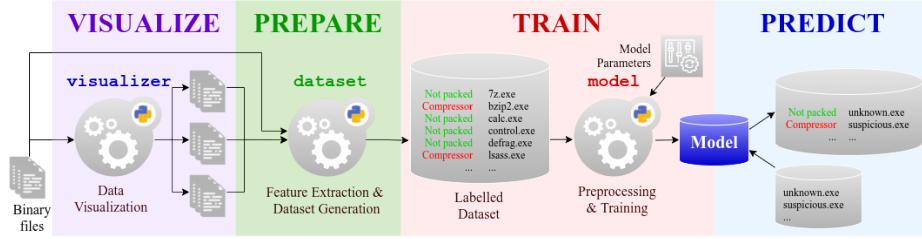


Fig. 2. Machine Learning pipeline and the automation toolset

While running the Docker image, the user ends up in a terminal that makes all the necessary tools available. It includes **packing-box**, an administrative utility for setting up and testing new items. **detector** is the tool that relies on the *Detector* abstraction and allows to use multiple detectors at once as a super-detector (REQ08) or even to apply mass detection and to output statistics for a given dataset. **packer** and **unpacker** are the tools that rely on the *Packer* and *Unpacker* abstractions and allow to test the packing and unpacking processes or also allows to apply bulk packing/unpacking. Other tools are readily available to start with experiments, including **visualizer**, **dataset** and **model** that automate the steps of the machine learning pipeline (REQ10) as depicted on Figure 2. At the *VISUALIZE* step, **visualizer** helps sort the input samples or even complete packer labels or discard outliers. At the *PREPARE* step, **dataset** provides commands for creating, merging or splitting all kinds of datasets or even ingesting samples from a data source (REQ07) and simplify the preparation for model training. Then at the *TRAIN* step, **model** allows to train models and select the best parameters with a grid search based on the selected algorithm as defined in the related YAML file. At the *PREDICT* step, **model** also allows to further test the trained model on other datasets to evaluate its performance.

dataset is the tool that relies on the *Dataset* abstraction to apply mass packing based on the packers defined in the related YAML configuration. Currently, it considers samples as successfully packed if and only if the hash of the original version has been changed in the packed version and is not known as a failure hash (e.g. hash of an empty binary file, with only headers and no section). As we only aim to work statically at this time, we do not use dynamic techniques to verify that the packed binary still runs as intended, hence the importance of studying the behavior of a packer when integrating it through its YAML defi-

dition. Attention shall be paid to reviewing the packer’s documentation (if any) and testing edge cases to see if there exists failure hashes. With mass packing automated (REQ09), we can build accurate ground truths (Q2) as we make them based on prior knowledge because we know how we pack the binaries.

Data aims to be easily exported and shared with other users. This is ensured by setting a workspace folder that sorts datasets and models in separate sub-folders, enforcing their structure. A dataset folder consists of its metadata, its features dictionary of descriptions, its data and, if relevant, its sample files. In the case when a dataset is converted to its fileless version, features are computed and stored with the dataset’s data and the file samples are then dropped to save space. A model folder contains its model dumped in the JobLib¹⁷ format. The **experiment** tool can be used to set up a dedicated workspace for a particular experiment. This way, specific datasets and models can be saved in the experiment’s folder but configuration files can also be tuned in this specific context, without affecting the base configurations from the main workspace. It is worth being noted that the **dataset** tool can also export a dataset folder to multiple formats including ARFF (dataset format used in Weka – this allows for interoperability with Weka while using the same interface as Scikit Learn), CSV and a custom-made format relying on XSLX called DataSet File Format (DSFF) [9].

The Packing Box [11], thanks to the aforementioned functionalities, solves the issue of *repeatability* (Q1), bringing clarity in the conditions that lead to the best models but also allowing for portability of research results.

4 Experiments

For this paper, we focus our experiments on two aspects: performance evaluation of the integrated detectors and training machine learning models. The part related to feature engineering will be addressed in a further study. In the remainder of this section, we use the list of commands from Table 1 aimed to execute the experiments of this study. "DS" is our target dataset for the sake of conciseness but, in our scope, it is in particular **dataset-packed-pe** [10].

ID	Phase	Command	Purpose
CMD01	Setup	git clone https://github.com/packing-box/docker-packing-box cd docker-packing-box docker build -t packing-box . docker run -it -h packing-box -v `pwd`: /mnt/share packing-box	From the host, clone the Git repository, change the directory to it, build the box and then run it.
CMD02	/	detector ds -d die -m [-b] [-w]	Test DIE against DS, using binary classification and/or weak mode
CMD03	PREPARE	dataset update test-upx -s DS/not-packed -l DS/labels.json	Update the target dataset (creating it if not existing) with the not-packed samples from the given source
CMD04	PREPARE	dataset update test-upx -s DS/packed/UPX -l DS/labels.json	Same as above but for UPX-packed samples
CMD05	PREPARE	dataset make test-upx -n 300 -f PE -p upx --pack-all	Make 300 UPX-packed PE samples to be added to the target dataset

¹⁷ <https://joblib.readthedocs.io>

CMD06 /	<code>dataset show test-upx</code>	Show the metadata and statistics of the given dataset
CMD07 <i>TRAIN</i>	<code>model train test-upx -a rf</code>	Train a Random Forest on a dataset
CMD08 /	<code>model show test-upx.pe_854.rf_f138</code>	Show the metadata and statistics of the given model, including a ranking of the best features
CMD09 <i>TEST</i>	<code>model test test-upx.pe_854.rf_f138 test-mix</code>	Test the newly trained model on another dataset

Table 1: Commands used during the experiments

Our reference dataset is **dataset-packed-pe** [10], which is a sanitized (from outliers found via visualization) and augmented (with 5 new packers) fork of PackingData [7] consisting of 432 not packed samples, and ~ 130 samples for each of the following packers: Alienzyze, Amber, ASPack, BeroEXEPacker, Enigma VirtualBox, Eronana’s Packer, Exe32pack, EXpressor, FSG, JDPack, MEW, Molebox, MPRESS, Neolite, NSPack, Packman, PECompact, PETite, RLPack, Telock, Themida, UPX, WinUPack, Yoda’s Crypter and Yoda’s Protector.

4.1 Detectors Performance

For this experiment, it is worth being noted that many studies do not provide an open implementation (as addressed in Section 2, leading to REQ06). Therefore, it is difficult at this point to compare the performance of those techniques. However, our design allows to easily integrate these but this is currently a work in progress. At this time, we could already include various open source static detectors that can be found on the Web. Most of them rely on simple but yet relevant heuristics based on entropy or structural features of executable formats (mostly PE), including DIE¹⁸, Manalyze¹⁹, PEFrame²⁰ and RetDec²¹. We also reimplemented one open source detector (which is unmaintained), PyPackerDetect²², and a few techniques addressed in the literature, including Bintropy²³ [15], PEID²⁴ (used in almost all studies for building ground truths) and REMINDER²⁵ [13]. The characteristics of these detectors are presented in Table 2. This shows the supported formats for each detector and whether it supports multi-class classification (that is, either it only output whether the target is packed or not, or the name of the packer used), weak mode (meaning that the detector outputs formal detection decisions but also suspicions that are used in a simple plurality vote; when using this mode, we also consider for instance section names of the binary found as an indication of packing) and if it is selected in our superdetector (resulting from the analysis of Table 3).

¹⁸ <https://github.com/horsicq/DIE-engine>

¹⁹ <https://github.com/JusticeRage/Manalyze>

²⁰ <https://github.com/guelfoweb/peframe>

²¹ <https://github.com/avast/retdec>

²² <https://github.com/packing-box/pypackerdetect>

²³ <https://github.com/packing-box/bintropy>

²⁴ <https://github.com/packing-box/peid>

²⁵ <https://github.com/packing-box/reminder>

Detector	Formats	Multiclass			Comments
		Weak	Mode	Superdetector	
Binentropy	PE, ELF, MSDOS	N	N	N	Python CLI version of the tool of Lyda <i>et al.</i> [15] that integrates a plot capability.
DIE	All	Y	N	Y	Most diversified detector we could find, supporting PE, ELF and Mach-O; it contains many fine-grained signatures and heuristics and is used in many studies.
Manalyze	PE, MSDOS	Y	Y	N	Open source tool relying on PEiD's signatures DB, Yara rules and some heuristics (e.g. pattern matching on some packer names), sometimes used in studies.
PEFrame	PE, MSDOS	Y	Y	N	Open source tool that relies on PEiD's signatures DB and some heuristics, found in only one study.
PEiD	PE, MSDOS	Y	N	Y	Python CLI version of the original tool holding a sanitized database (aggregating many sources from GitHub repositories with signatures unrelated to packing removed) that we implemented for use with the Packing Box.
PyPacker-Detect	PE, MSDOS	Y	Y	Y	Fork of the unmaintained open source tool that relies on PEiD's signatures DB and a few simple heuristics.
REMINDER	PE, ELF, MSDOS	N	N	N	Python CLI version of the tool of Han <i>et al.</i> [13], using a simple heuristic relying on the writable property and the entropy of the EP section.
RetDec	All	Y	N	Y	Open source retargetable machine-code decompiler supporting many executable file format and using custom signatures and header-based heuristics.

Table 2: Base characteristics of in-scope detectors

Detector	Accuracy	Time	Complexity	Observations
Binentropy	58,71%	0,5–10,5s	$\mathcal{O}(n)$	Solely relies on entropy (i.e. block entropy).
DIE	84,25%–85,81%	~450ms	$\mathcal{O}(\log(n))$	Narrow detection rate interval (<2%), null FPR, strong signatures combined with some heuristics.
Manalyze	45,17%–94,83%	~350ms	$\mathcal{O}(\log(n))$	Very broad detection rate interval (~50%), high FPR, relying on a short list of known section names and some simplistic heuristics.
PEFrame	57,42%–87,52%	1,39–27,6s	$\mathcal{O}(n)$	Flawed; only relies on signatures and entropy.
PEiD	69,29%–71,63%	1,69–2,37s	$\mathcal{O}(n)$	Narrow detection rate interval (<3%), low FPR, many signatures are missing from the database.
PyPacker-Detect	79,69%–94,65%	0,71–1,36s	$\mathcal{O}(n)$	Decent FPR, provides good results and its heuristics cover cases not covered by the PEiD signatures.
REMINDER	47,55%	72–190ms	$\mathcal{O}(n)$	Poor detection rate, only relies on the writable property of the EP section and its entropy.
RetDec	70,71%–75,76%	0,32–0,84s	$\mathcal{O}(n)$	Narrow detection rate interval (~5%), low FPR, relying on custom signatures and some heuristics covering unmatched cases.

Table 3: Performance test results for in-scope detectors

In order to measure accuracy, we divided our experiments into 4 classes, taking 2 parameters into account: whether classification is binary (packed or not) or multi-class (per packer name) and if detection is strong (the detector outputs a formal decision) or weak (the output provides detection traces but also suspicions that may yield a decision that is less robust). Accuracy metrics were collected based on the execution of each detector on all the subsets (not packed and 25 packers) from our dataset [10]. With the detections rates of the 4 classes, considering weighted averages per class, we then considered the lower and upper bounds among the four resulting metrics. We used the `detector` tool from the Packing Box for bulk detection. CMD02 illustrates how to get the accuracy for a given dataset with a single detector. This command takes the following arguments: the target folder of the input dataset, `-d` is used to specify the integrated `detector` to be selected, `-m` enables `metrics` acquisition (i.e. accuracy), `-b` enables `binary` classification and `-w` enables the `weak` mode. While using a Bash script to run every detector on each subset of our reference dataset [10], we obtain the accuracy for the 4 different classes.

Processing time and complexity were evaluated by running the detectors (using a script relying on the `detector` tool) against samples from our dataset [10], only keeping the results for file sizes from 0 to 10 megabytes and then considering the lower and upper bounds. Taking the results of this evaluation, we applied curve fitting to get the best fit among constant, logarithmic, linear, linear-algorithmic and quadratic complexities. Results are shown in Table 3.

For the performance evaluation (Q3), Table 3 shows the collected metrics and our related observations. We note that a few detectors have a stable detection (including DIE, PEiD and RetDec) while others do not (i.e. Manalyze, PEFrame and PyPackerDetect). We also see that only a few ones perform in less than 1 second (the best performers being DIE, Manalyze and PEPack and the worst ones being PEFrame, REMINDER and Bintropy). Moreover, we point out that most of the detectors have a complexity in $\mathcal{O}(n)$ which means they parse the entire binary. While this result is logical for Bintropy (as it requires to compute the entropy of the whole binary), for some others it is not. PEiD, which was used for matching signatures from the Entry Point (EP), surprisingly holds this complexity while we expect it to perform in $\mathcal{O}(1)$ because it should only require a constant calculation to find the EP and a pattern matching that is bounded to a constant maximum length. However, this result comes from the underlying library that systematically parses the entire file. The same issue applies to REMINDER too. It is also worth being noted that even the best detector, DIE, only performs at around 86%. This is due to the specificity of the dataset in use, which contains multiple recent custom packers found on GitHub like Alienze (which relocates the EP but does not actually perform packing in its demo version), Amber and Eronana’s Packer. From these results, we see that DIE is the best performer in terms of accuracy, followed by PyPackerDetect. It shall be noted however that PyPackerDetect holds a non-negligible False Positives Rate (FPR). Regarding time complexities, DIE remains the best. In terms of processing time, also considering the FPR and accuracy range, DIE stays at the top.

In terms of candidates for the superdetector, Bintropy and Manalyze can be excluded due to their accuracy. Moreover, PeFrame has a particularly high cost in processing time and can then be discarded. Among the best performers, we thus keep DIE, PEiD, PyPackerDetect and RetDec.

4.2 Model Training

This experiment aims to briefly showcase learning pipeline automation. Other experiments for identifying categories, best features and models are left for future works. Following the learning pipeline of Figure 2, we address dataset generation and model training, that is, the *PREPARE*, *TRAIN* and *TEST* phases.

For data preparation, CMD03 and CMD04 are used to respectively create and update the dataset with input samples and take the following arguments: a dataset name (e.g. `test-upx`), `-s` specifies the source folder to get samples from and `-l` specifies the JSON file containing the labels to be used. Note that, if no labels dictionary (with file hashes as keys and labels as values) is provided, samples will be considered not labelled. CMD05 shows the `make` command that takes the following arguments: `-n` gives the maximum number of samples, `-f` specifies the executable format to be selected from the source folder, `-p` specifies the packer to be used and `--pack-all` enables packing every sample (otherwise, samples are packed or not randomly so that the dataset gets balanced). Creating a second dataset with CMD03 and CMD04, we can reuse this last command to add for instance Neolite and Packman samples. This set of commands completes the *PREPARE* phase. For our experiment, we end up with two datasets: one consisting of 854 samples whose 422 are UPX-packed (this can be inspected using CMD06) and the second consisting of 785 samples including 116 Neolite, 115 Packman and 122 UPX-packed samples. We can then train our model.

CMD07 aims to train a model using the Random Forest algorithm based on the first dataset we just made. The `model` tool takes the following arguments: the target dataset is specified to train the model on and `-a` specifies the algorithm to be selected. The result of this command displays evaluation metrics as shown in Table 4, including the accuracy, precision, recall, F-measure, Matthews Correlation Coefficient (MCC) and Area Under the Curve (AUC)²⁶. Note that the default for the train set split is 80%, hence we get both train and test metrics in the training results. We obtain a perfectly fitted model that gets automatically named `"test-upx_pe_854_rf_f138"` according to the reference dataset and its related size, the learning algorithm's short name and the number of applicable features (that is, excluding the features with a null variance). We can inspect our new model with CMD08, i.e. for getting the ranking of the best features according to their importance. At this point, we completed the *TRAIN* phase.

CMD09 shows how to address the *TEST* phase by applying our new model to our second dataset. The `model` tool takes the following positional arguments: the automatically generated name of the trained model (shown after training it) and the name of the dataset to test the model on. We get 83.18% accuracy on

²⁶ https://scikit-learn.org/stable/modules/model_evaluation.html

our new dataset, meaning the model trained based on UPX samples cannot fully explain samples packed with Neolite and Packman. As the precision is 100%, it means that our model does not yield False Positives. However, the recall shows that it yields many False Negatives, which means that it does not use relevant information for detecting samples packed with another packer than UPX. In a few commands, we thus followed the complete learning pipeline. By playing with input datasets, it is now easy to study other classifications, i.e. packer categories.

	Accuracy	Precision	Recall	F-Measure	MCC	AUC
Train (test-upx)	100%	100%	100%	100%	100%	100%
Test (test-upx)	100%	100%	100%	100%	100%	100%
Test (test-mix)	83.18%	100%	62.61%	77%	69.25%	100%

Table 4: Metric values for training and testing a simple model based on a dataset consisting of not packed and UPX-packed samples, then tested on a separate dataset including UPX, Neolite and Packman samples

5 Conclusion

In this paper, we first stated the problem of executable packing and emphasized the need for *repeatability* in experiments for its detection. We set the background for this topic, enumerated techniques of static detection as currently presented in the literature and parsed some aspects that could help us deduce requirements for our experimental toolkit. We then described our design and implementation of the Packing Box, including the capabilities it offers especially for making experiments repeatable, promoting results sharing (Q1) and building accurate ground truths (Q2). Finally, we conducted some experiments to showcase the functionalities and how the entire machine learning pipeline could be traversed in just a few terminal commands thanks to our toolset. We took advantage of our toolkit to study the performance (Q3) of open source static detectors, as well as a number of detectors proposed in the literature which we have re-implemented.

We designed a platform for studying executable packing, not especially focused on malware analysis but well on packing in general and not limited to only Windows PE but also applicable to other executable formats like ELF. This brings an open source unified solution to the research community that addresses the many related practical challenges, including the need for a solution that allows to compare results from multiple studies in a quantitative way (unlike qualitative such as the survey of Muralidharan *et al.* [17]), allowing to compare approaches based on common data. With the Packing Box, we showed that it significantly enhances experiment *repeatability* as it fully integrates assets (i.e. detectors and packers) and it automates the complete learning pipeline (from dataset generation to model training, testing and comparison). Through packers integration, we showed how it allows to generate accurate ground truths starting from a set of cleanware samples. Through detectors integration, we demonstrated how to mass-process a dataset we created to evaluate the performance of state-of-the-art detectors, allowing us to select the best ones for a superdetector. With

this experimental toolkit in hand, we propose a complete solution for creating and sharing datasets, comparing, improving and sharing machine learning models for static packing detection and even managing and sharing our experiments.

We already planned a few enhancements in terms of functionalities but also in our experiments as well;

- *Packer categories*: We mentioned categories at the beginning of Section 2 but we did not present any experiment focused on detecting them as this is still a work in progress. We plan to train many models based on many datasets of our own in the hope of identifying sets of features that could better apply to one category or another.
- *Feature engineering*: We already enhanced our toolkit to support custom features handling through another declarative YAML configuration file. Furthermore, we added the logic to compose new features from the ones currently extracted (e.g. for PE files, via `pfeats` [4]). Throughout our review of the literature, we gathered state-of-the-art features and established our own set in what we called the *Feature Book*. We thus plan to implement this to further refine our features set for our next experiments.
- *State-of-the-art techniques*: Among the techniques mentioned in Section 2, we plan to implement a few more to extend our scope and provide a comparison to measure how much better our ML approach could be.

Acknowledgements. Charles-Henry Bertrand Van Ouytsel is FRIA grantee of the Belgian Fund for Scientific Research (FNRS-F.R.S.). The authors are funded by the CyberExcellence project (RW, Convention 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. p. 20 (2020)
2. Anderson, H.S., Roth, P.: EMBER: An open dataset for training static PE malware machine learning models (2018)
3. Bertrand Van Ouytsel, C.H., Given-Wilson, T., Minet, J., Roussieau, J., Legay, A.: Analysis of machine learning approaches to packing detection (2021)
4. Biondi, F., Enescu, M.A., Given-Wilson, T., Legay, A., Noureddine, L., Verma, V.: Effective, efficient, and robust packing detection and classification (2019)
5. Cesare, S., Xiang, Y.: A fast flowgraph based classification system for packed and polymorphic malware on the endhost. In: 24th International Conference on Advanced Information Networking and Applications. IEEE (2010)
6. Cesare, S., Xiang, Y., Zhou, W.: Malwise - An effective and efficient classification system for packed and polymorphic malware (2013)
7. Choi, M.J., Bang, J., Kim, J., Kim, H., Moon, Y.S., Díaz-Verdejo, J.: All-in-one framework for detection, unpacking, and verification for malware analysis (2019)
8. 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)

9. D'Hondt, A.: Dataset file format (DSFF) (2022), <https://github.com/packing-box/python-dsff>
10. D'Hondt, A.: Dataset of packed PE files (2022), <https://github.com/packing-box/dataset-packed-pe>
11. D'Hondt, A.: Packing Box – Study executable packing easy (2022), <https://github.com/packing-box/docker-packing-box>
12. Hai, N.M., Ogawa, M., Tho, Q.T.: Packer identification based on metadata signature. In: Proceedings of the 7th Software Security, Protection, and Reverse Engineering. p. 11. SSPREW-7, ACM (2017)
13. Han, S., Lee, K., Lee, S.: Packed PE file detection for malware forensics. In: 2nd International Conference on Computer Science and Its Applications. IEEE (2009)
14. Lim, C., Nicsen: Mal-EVE: Static detection model for evasive malware. In: 10th International Conference on Communications and Networking in China. IEEE (2015)
15. Lyda, R., Hamrock, J.: Using entropy analysis to find encrypted and packed malware (2007)
16. Mantovani, A., Aonzo, S., Ugarte-Pedrero, X., Merlo, A., Balzarotti, D.: Prevalence and impact of low-entropy packing schemes in the malware ecosystem. In: Network and Distributed Systems Security Symposium. NDSS (2020)
17. Muralidharan, T., Cohen, A., Gerson, N., Nissim, N.: File packing from the malware perspective: Techniques, analysis approaches, and directions for enhancements. ACM Computing Surveys (2022)
18. Naval, S., Laxmi, V., Gaur, M.S., Vinod, P.: SPADE: Signature based Packer DETECTION. In: Proceedings of the First International Conference on Security of Internet of Things. pp. 96–101. SecurIT '12, ACM (2012)
19. Noureddine, L., Heuser, A., Puodzius, C., Zendra, O.: SE-PAC: A Self-Evolving Packer Classifier against rapid packers evolution. In: Proceedings of the 11th ACM Conference on Data and Application Security and Privacy. pp. 281–292. CODASPY '21, ACM (2021)
20. Perdisci, R., LANZI, A., Lee, W.: McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables. In: 2008 Annual Computer Security Applications Conference (ACSAC). pp. 301–310 (2008)
21. Santos, I., Nieves, J., Bringas, P.G.: Semi-supervised learning for unknown malware detection. In: International Symposium on Distributed Computing and Artificial Intelligence. Springer (2011)
22. Shafiq, M.Z., Tabish, S.M., Farooq, M.: PE-Probe: Leveraging packer detection and structural information to detect malicious portable executables. In: Proceedings of the Virus Bulletin Conference (2009)
23. Shin, D., Im, C., Jeong, H., Kim, j., Won, D.: The new signature generation method based on an unpacking algorithm and procedure for a packer detection. In: International Journal of Advanced Science and Technology. IJAST (2011)
24. Sun, L., Versteeg, S., Boztaş, S., Yann, T.: Pattern recognition techniques for the classification of malware packers. In: Steinfeld, R., Hawkes, P. (eds.) Information Security and Privacy. Springer (2010)
25. Treadwell, S., Zhou, M.: A heuristic approach for detection of obfuscated malware. In: International Conference on Intelligence and Security Informatics. IEEE (2009)
26. Ugarte-Pedrero, X., Santos, I., Bringas, P.G., Gastesi, M., Esparza, J.M.: Semi-supervised learning for packed executable detection. In: 5th International Conference on Network and System Security. IEEE (2011)