SE-PAC: A Self-Evolving PAcker Classifier against rapid packers evolution

Lamine Noureddine
Univ. Rennes, Inria, IRISA
Rennes, France
lamine.noureddine@inria.fr

Annelie Heuser
Univ. Rennes, Inria, CNRS, IRISA
Rennes, France
annelie.heuser@irisa.fr

Cassius Puodzius
Univ. Rennes, Inria, IRISA
Rennes, France
cassius.puodzius@inria.fr

Olivier Zendra
Univ. Rennes, Inria, IRISA
Rennes, France
olivier.zendra@inria.fr

ABSTRACT

Packers are widespread tools used by malware authors to hinder static malware detection and analysis. Identifying the packer used to pack a malware is essential to properly unpack and analyze the malware, be it manually or automatically. While many well-known packers are used, there is a growing trend for new custom packers that make malware analysis and detection harder. Research works have been very effective in identifying known packers or their variants, with signature-based, supervised machine learning or similarity-based techniques. However, identifying new packer classes remains an open problem.

This paper presents a *self-evolving packer classifier* that provides an effective, incremental, and robust solution to cope with the rapid evolution of packers. We propose a composite pairwise distance metric combining different types of packer features. We derive an incremental clustering approach able to identify both (variants of) known packer classes and new ones, as well as to update clusters automatically and efficiently. Our system thus continuously enhances, integrates, adapts and evolves packer knowledge. Moreover, to optimize post clustering packer processing costs, we introduce a new post clustering strategy for selecting small subsets of relevant samples from the clusters. Our approach effectiveness and time-resilience are assessed with: 1) a real-world malware feed dataset composed of *16k* packed binaries, comprising *29* unique packers, and 2) a synthetic dataset composed of *19k* manually crafted packed binaries, comprising *31* unique packers (including custom ones).

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; • **Computing methodologies → Machine learning approaches**.

KEYWORDS

Packers; Malware obfuscation; Features combination; Classification; Clustering; Novelty detection; Incremental learning.

ACM Reference Format:
Lamine Noureddine, Annelie Heuser, Cassius Puodzius, and Olivier Zendra. 2021. SE-PAC: A Self-Evolving PAcker Classifier against rapid packers evolution. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy (CODASPY '21), April 26–28, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3422337.3447848

1 INTRODUCTION

Context. The malware (MW) ecosystem is evolving rapidly, with drastic changes in the frequency and composition of malware. This growing problem has high financial impact. Ransomware damage alone cost more than $5 billion USD in 2017 and was estimated to reach $20 billion USD by 2021, an increment of 400% compared to 2017 and 5700% to 2015 [28].

Effective MW detection and analysis is thus crucial for users and security systems. They can be implemented with analysis techniques that are either static or dynamic. Static analyses examine a binary directly, by disassembly or measuring syntactic properties, with no execution. Dynamic analyses require executing the binary, e.g. in a virtual sandbox environment.

To defeat these analyses, MW authors employ different obfuscation techniques. In particular, they use packers to hinder static analysis by increasing the difficulty to reverse engineer the binaries, which entails higher analysis costs for MW analysts. Packing consists in compressing and encrypting binary data, to produce a new binary that is syntactically different from the original one. Thus unpacking is essential to verify if packed binaries are malicious.

New unknown packers complicate unpacking, since the specific required unpacking function is unknown and generic unpackers are not always effective [19], which makes MW detection and analysis harder. So beside the many well-known packers in use (e.g. UPX, NsPack, ASPack), there is a growing trend for custom packers. The latter are developed either from scratch or partially from available ones (e.g. Vanilla UPX). Their usage has become so widespread that by 2015 Symantec detected their use in over 83% of MW attacks [30]. Research works have also followed this evolution [38].

The detection of new packers using supervised Machine Learning (ML) techniques has shown promising results [19]. It is a binary (packed or non-packed) classification problem and uses large labeled datasets containing a variety of packers to train ML models. However, classifying new packers into families with supervised ML can be unfeasible or unreliable. Indeed, when a packer is just a modified variant of a packer (class) used for training, Supervised Learning (SL) can classify it as related to an existing class; but when the packer (class) is totally new, SL techniques fail. This misclassification can lead to a significant increase in the unpacking costs,

CODASPY '21, April 26–28, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8143-7/21/04…$15.00
https://doi.org/10.1145/3422337.3447848

and create a dataset pollution problem if the wrong predicted labels were then used for (re-)training, which would result in false positives should the signature tools be updated accordingly.

Challenges and Difficulties. Packers evolve as rapidly as MW, constantly bringing new classes or new variants of existing ones [38]. To build an effective MW analysis and detection system, it is thus essential to keep the packer classification system updated. But identifying new packers is difficult, be it manually or automatically. Manual reverse engineering is unfeasible in practice as the number of MW instances grows exponentially. Automatic techniques like signatures-based tools and supervised classification systems cannot properly detect *class novelty.* And although generic unpacking may identify the underlying packer, it may fail if the packer is totally new. Thus, a crucial challenge is to propose an update system able to *automatically* and *effectively* identify *new packer classes.*

Updating the system often consists in retraining from scratch, thus rebuilding the whole model. The drawback of this approach is that at each retraining the knowledge from the previous existing model is omitted. Hence another important challenge is to derive an *incremental* approach where knowledge is constantly *enhanced, integrated, adapted and evolved.* Such a system would eventually lose effectiveness, thus requiring supervisor intervention to completely retrain. But full retraining is very costly, so achieving high robustness (i.e. time-resilience) is crucial to maximize the lifespan of our model, minimize costs and maximize user security.

In this paper, we present a *self-evolving packer classifier* that provides an effective, incremental, and robust solution to cope with the rapid evolution of packers. Our ***self-evolving*** technique consists in predicting incoming packers by assigning them to the most likely clusters, and relies on these predictions to *automatically update* clusters, reshaping them and/or creating new ones. Thus our system *continuously learns* from incoming packers, adapting its clustering to packers evolution over time. Note that the packer *detection* problem (packed, non-packed) is out of the scope of this paper. Our work focuses on packer families classification.

Our main **contributions** are:

- We point out and experimentally show the importance of *constantly updating* the packing classification system.
- We show how to combine different types of features in the construction of a composite pairwise distance metric, in the context of packed binaries.
- To decrease the update time for the incremental clustering procedure, we derive a methodology to extract *representative samples* for each cluster. Interestingly, the number of representatives extracted from each cluster is not fixed, but related to the number of samples in the cluster. We furthermore study this relation experimentally, to derive a good trade-off between effectiveness and update time performance.
- We propose an end-to-end framework, going from features extraction, custom distance metric, to incremental clustering with a self-evolving classifier for packed binaries.
- We support our findings with realistic experiments showing promising results for effectiveness and resilience over time.
- We introduce a new post clustering selection strategy that extracts a reduced subset of relevant samples from each cluster, to optimize the cost of post clustering packer processes.

This paper is structured as follows. Sec. 2 recalls useful background material, and Section 3 related work. Section 4 presents our methodology and Section 5 our post clustering selection strategy. Section 6 details the datasets and ground truth, and Section 7 the evaluation metrics. Section 8 presents the experimental setup and results, which Section 9 discusses. Section 10 concludes.

2 BACKGROUND

This section provides background on packers usage, their impact on a MW analysis workflow, and a brief overview on clustering.

2.1 Usage of Packers in Malware

Packers were designed to fulfill either or both goals of size reduction and protection against reverse engineering [34]. Binary packing takes a target binary (called *payload*) as input and generates a new binary that embeds the original in a packed, “scrambled” form, together with an unpacking routine (*unpacking stub*) that can unpack and execute the original binary in memory. Packers are also called *runtime packers* and sometimes *self-extracting executables.*

For MW detection, the main challenge to analyse an unknown packed binary is to determine whether the payload is malicious or benign (using packing for legitimate purposes, e.g. software integrity protection). This analysis is only possible after partial or complete unpacking of the sample (see Sec. 2.2). Classical signature-based antiviruses (AVs) that heavily rely on syntactical properties and cannot unpack are thus ineffective against packing. So a special concern for MW packers is to alter the syntactic properties of the different instances (i.e. variants) generated from the same MW (family) in order to better defeat AVs. *Polymorphic* codes take different forms at each instance generation (e.g. using encryption). *Metamorphic* codes rewrite themselves at each execution [25].

The protection features sought by MW developers include many different techniques for anti-debugging, anti-disassembly, obfuscation and anti-VM [34]. Various packers will support various protection features. In particular, these anti-analysis features can additionally comprise: multilayer packing, interleaved execution of unpacking and the original program, shifting decode frames, etc. This is referred as the *complexity* of the packer in [38], which proposes to rank packers with a taxonomy including six increasing complexity classes (I to VI) based on the increasing difficulty to unpack the binary, gauged relatively to the multiple anti-analysis techniques that are added to the unpacking stub routine.

2.2 In-depth Scanning

To determine whether a binary sample is malicious, an AV needs to analyze its characteristics. If the AV determines that the sample is packed, a payload extraction is attempted before proceeding with the MW analysis. This is called *in-depth scanning* [1] (see Fig. 1).

The analysis pipeline starts with very lightweight syntactic operations, and may go on with more robust and expensive interventions. If the sample is found to be packed, payload extraction is attempted by a specialized or a generalized unpacker [27]. *Specialized unpackers* are lightweight but require correct prior identification of the packer (and sometimes version). *Generalized unpackers* are intended to handle various packing algorithms, but are more costly, for lower success rates. Therefore, a correct classification of the

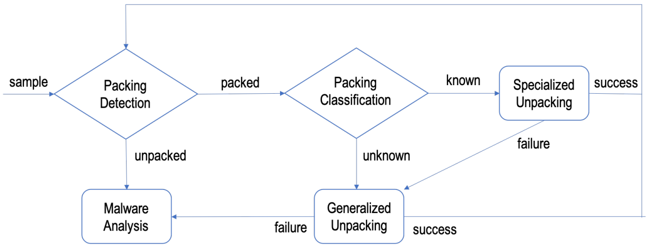

Figure 1: Typical workflow of in-depth scanning

packer (and version) is crucial for a good, cost-effective in-depth scanning, by deciding whether a specialized unpacker can be tried before a generalized unpacker. The classification process must also be efficient, to add limited overhead in the analysis workflow.

2.3 Clustering

Clustering is the task of dividing a set of elements in subsets (*clusters*) following some criterion. The elements can be treated individually (*incremental clustering method*) or they can be processed in batches (*batch clustering method*) [17]. While the *batch method* attempts to capture the underlying structure of the elements in a compact and efficient way, the *incremental method* suits the scenario where new data continually arrive and recomputing the clusters from scratch becomes infeasible due to the volume of data. In particular, *incremental clustering* allows *incremental learning*, where knowledge is *enhanced, integrated, adapted and evolved.*

In practice, many clustering algorithms exist. They can be primarily categorized into prototype-based, hierarchical-based or density-based [37]. The latter have the particularity of not being sensitive to noise and can deal with different cluster sizes and different cluster shapes. DBSCAN [21] is a typical density-based clustering algorithm widely used in many applications.

Finally, clustering can be evaluated by *intrinsic* or *extrinsic* metrics. *Intrinsic metrics* evaluate with some distance metric whether the elements in the same cluster are close while the elements in different clusters are distant. *Extrinsic metrics* evaluate the quality of a clustering by comparing it against a ground truth, also called gold standard, that represents the expected clustering.

3 RELATED WORK

Packer classification has been moderately tackled in literature. Most studied techniques rely on syntactic signatures, ML, or similarity.

Syntactic signatures are sequences of bytes that characterize a specific packer. Tools like PEiD [33], Yara [16], and DIE [8] statically parse the packed binary and match it against a signature database. Since they rely on priorly generated rules, they are unable to detect new or sometimes slightly modified packers. Furthermore, these tools often need to be manually updated by an analyst who writes signatures as new packers are manually reverse-engineered. This is extremely costly in practice, given the tremendous number of new MW appearing every day, which increases the time during which new packers remain undetected by these tools.

More advanced approaches use ML techniques to classify packers. SL is the most prevalent technique in literature [19, 24, 35, 40]. In SL, the ML algorithms build a classifier model by learning from a set of (syntactic or behavioral) features extracted from packed binaries, and from the corresponding labels provided as ground truth. This packer classifier model is then used to classify unlabeled packed binaries. SL performs well on packers belonging to classes for which the ML model was trained ([19] achieved an F-score up to 0.9999 on more than 280k samples). However, it fails to recognize new packer classes, that were not present during training.

Other works [18, 23, 26, 31, 32] extracted signatures from the unpacking stub code (see Sec. 2.1), then a *similarity metric* regroups packers into families. An unknown packed sample is classified by computing its distances to a referential of packer classes. This approach quantifies the *similarity* wrt. a referential of signatures, unlike traditional signature-based approaches that hinge on direct matching unable to capture small differences. The authors of [31] claim their system can identify new packer classes, but do not validate it experimentally. In [23], new custom packers are correctly classified only if the packer uses a set of well-known obfuscation techniques. Works [18, 26, 31] can identify new packer classes, but rely on pairwise similarities, while we rely on clustering that adapts itself to packers evolution. Like us, [26] analyze a large dataset of manually and wild packed files, while [18, 31] study smaller datasets. Feature extraction in [26, 31] uses recursive traversal disassembly and [18] requires full execution with multiple entropy measurements, whose cost can be prohibitive in our clustering scenario.

Our work is the first we know of that provides a methodology relying on clustering to tackle rapid packers evolution. Moreover, we construct a robust pairwise distance metric that combines dynamically extracted unpacking stub assembly language (ASM) signatures, and a set of statically extracted structural features from the Windows Portable Executable (PE) packed files. This combination is more robust against obfuscation techniques and avoids ML overfitting. Finally, our post clustering selection strategy is new, and provides for each cluster a small subset of relevant samples that can undergo deeper and more costly analyses whose results can be extended to all samples in the cluster.

4 METHODOLOGY

This section starts with a brief overview of the methodology for our self-evolving classifier, before exploring each part in more details.

4.1 Overall Toolchain

Our approach comprises two phases (see Fig. 2). First, the *offline phase* exploits and models all available knowledge, by using available packed (binary) samples as well as packer labels to tailor the generation of clusters. In the second phase, the *online phase*, the system self-evolves by incrementally updating the clusters as new samples, packed with either previously seen or unseen packers, are processed. Both phases are divided into three main steps: feature extraction, distance computation and clustering. Feature extraction and distance computation are identical for the offline and online phase, whereas clustering differs notably. Joining supervised learning in the offline phase and unsupervised learning in the online phase makes the whole system learn in a *semi-supervised* method.

We rely on clustering because it can efficiently cluster similar samples that belong to the same packer family, *regardless whether*

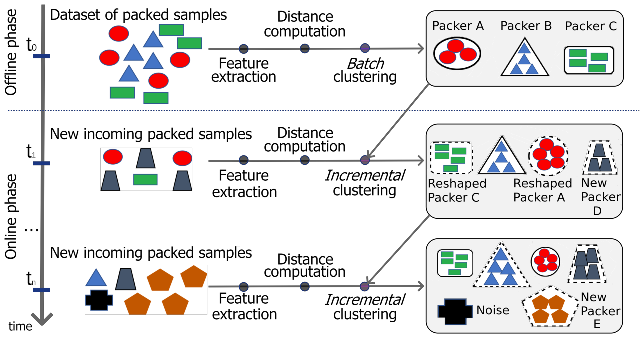

Figure 2: Overall toolchain

the class was previously known, and thus fits our goal of *discovering new unknown packer classes*. In particular, *incremental clustering* provides *incremental learning* where knowledge about packers constantly evolves, by creating new clusters for new families, or reshaping existing clusters according to new variants that can represent different versions, configurations, or polymorphic instances.

Initially, at t_0, **the offline phase** trains the self-evolving packer classifier and tunes parameters that will be used in the online phase. The initial dataset contains packed samples and the corresponding ground truth labels. Various static and dynamic features are extracted from each packed sample, to represent it by a *heterogeneous* features set. To define a unique pairwise distance for pairs of packed samples, each individual feature yields a *partial distance metric* that is averaged with all the others to obtain the final distance. Clustering is then used to group similar packed samples into clusters. The best clustering is found by tweaking the distance between packed samples according to the ground truth, so that the obtained clustering combines *homogeneity* (clusters do not mix packers from different families) with the most *coherent number* of clusters (the closest to the number of packer families in the ground truth). At the end of this phase, the parameters that produce the best clustering, as well as the corresponding clusters, are the first *clustering setup* which serves as baseline for the second phase.

The online phase ($t_1, ..., t_n$) has successive self-evolutions, processing one packed sample at a time, for as long as the model produces accurate results before requiring full retrain or supervisor intervention. In this phase, the same tasks are repeated for features extraction and distances computation. Then starting with the clustering setup obtained from the offline phase, the system self-evolves by relying on incremental clustering that *dynamically* includes the incoming packed samples, thus creating new clusters for new families and reshaping existing clusters with new variants. Since our system classifies new incoming packed samples in real-time, it can be used in production, hence reinforcing the security of the user.

4.2 Features Extraction

We extract 6 groups of packer features chosen according to state-of-the-art [19]: *metadata* (21 features), *sections* (21 features), *entropies* (6 features), *resources* (2 features), *import functions* (5 features), *unpacking stub mnemonic sequences* (1 feature). A selection is done by testing all the possible non-repetitive group combinations. The most effective one is *sections* associated to the *unpacking stub mnemonic sequences*. This combination provides effectiveness as well as robustness against obfuscation techniques that can deceive each group taken separately. It also reduces ML over-fitting.

4.2.1 Unpacking stub mnemonic sequences. We perform a *lightweight dynamic emulation* of the first instructions that follow the binary Entry Point (EP). As instructions are fetched, the corresponding mnemonics[1] are stored in a list, which is the mnemonic sequence feature. The rational is that runtime packers often start executing the unpacking stub routine before reaching the malicious payload Original EP (OEP); so the unpacking stub code can be a characteristic feature [23, 26, 31, 32]. Dynamic emulation has the advantage to thwart unpacking stub codes that use obfuscation techniques to impede static analysis, such as anti-disassembly and metamorphism (see Sec. 2.1). Furthermore, although an attacker can engineer a bunch of code protection that foreruns the unpacking code routine, this would not mislead our step since this bunch of code protection would serve as well to identify the packer.

4.2.2 PE sections. By changing the program structure, packers introduce many artifacts into the sections of the binary. In this work, we extract 21 state-of-the-art features [19] related to sections:

Ten integer features representing the number of sections of the PE packed file that are: *standard/non-standard*; *executable/writable/readable* (and combinations); have raw data size zero; have different virtual and raw data sizes.

Seven boolean features: `.text` section not executable; a non-`.text` section is executable; `.text` section not present; EP not in `.text` or `.tls` section; EP not in a standard section; EP not in an executable section; and address not matching file alignment,

Four ratio based features: ratio of standard sections to all sections; ratio between raw data and virtual size of the section containing the EP; maximum ratio of raw data to virtual size; and minimum ratio of raw data to virtual size.

4.3 Composite Pairwise Distance Metric

Since extracted features are *numeric* (relating to PE sections) or *string sequences* (mnemonic sequences), we need to derive a unique pairwise distance metric able to combine both types. To this end, we apply a Gower distance [22], a composite metric overcoming the issue of mixed data type variables by being computed as the average of partial distances that range in $[0, 1]$. Our work has two different partial distance metrics: *Manhattan* distance for numeric features and *Tappered Levenshtein* distance for mnemonic sequences. These two distances are then normalized, and their average computed, thus providing our composite distance metric. Formally:

$$Gower(i, j) = \frac{1}{p} \sum_{k=1}^{p} NormD_{ij}^{(f_k)} \qquad (1)$$

i and j are the indices of two samples in the dataset, $NormD_{ij}^{(f_k)}$ is the normalized partial distance metric applied wrt. the data type of the k^{th} feature f, and p is the number of features. Each packed sample is represented by 22 features (a string and 21 numbers).

[1] In assembly language, mnemonics specify an opcode that represents a complete and operational machine language instruction.

The Manhattan distance provides partial distance metrics for numeric features. The normalized partial distance of a numeric feature f between two samples of indices i and j is the ratio between the absolute difference of observations $x_i^{(f)}$ and $x_j^{(f)}$ and the absolute maximum range $R^{(f)}$ observed for f among all samples:

$$NormManh_{ij}^{(f)} = \frac{|x_i^{(f)} - x_j^{(f)}|}{|R^{(f)}|} \tag{2}$$

The Tapered Levenshtein Distance provides partial distance metrics for ASM mnemonic sequences [32]. It has the advantage of quantifying the similarities between two ASM mnemonic sequences, in contrast to straightforward comparison that cannot capture small differences in the sequence. Furthermore, the *tapered* version proportionally decreases the weight of each element as they appear later in the sequence, punishing more differences in the beginning of the sequence and less in the end. In our work, the intuition behind *tapering* is that most of the times the unpacking stub routine is located directly at the binary EP, and thus the order in which the instructions appear is important. Nonetheless, since the length of the unpacking stub is a priori unknown, the focus is to fetch as few instructions as possible to attain the optimal balance between efficiency and information gain for packer classification. The normalized partial distance of a string mnemonic sequence f between two packed samples i and j, is formally defined as:

$$NormTapLev_{ij}^{(f)} = \frac{\sum_{k=0}^{L^{(f)}_{(S_i,S_j)}-1} W^{(f)}_{(S_i,S_j)}(k)(1 - \frac{k}{L^{(f)}_{(S_i,S_j)}})}{C} \tag{3}$$

where S_i and S_j are the respective mnemonic sequences of the two packed samples whose indexes are i and j. $L^{(f)}_{(S_i,S_j)}$ is the maximum length between S_i and S_j. $W^{(f)}_{(S_i,S_j)}(k)$ is a *factor* which is equal to 0 if the ASM mnemonic of S_i and S_j are equal at the position k, or equal to 1 otherwise. Finally, C is the maximum length of mnemonic sequence extraction, used to normalize the distance into $[0, 1]$.

Example 1. *Let C be set to 50, and S_i and S_j the following ASM mnemonic sequences, extracted from packed samples i and j:*

$$S_i = \{push, mov, push, push, push, mov, push, mov, sub\},$$
$$S_j = \{push, mov, mov, push, add, mov, add, mov, sub\},$$

then $L^{(f)}_{(S_i,S_j)} = 9$ *and* $NormTapLev_{ij}^{(f)} = 0 + 0 + (1 - (2/9)) + 0 + (1 - (4/9)) + 0 + (1 - (6/9)) + 0 + 0 = 1.67/50 = 0.033$

4.4 Clustering: Batch and Incremental

In both offline and online phases we use DBSCAN [21] as clustering algorithm, because it: (*i*) does not require the a priori number of clusters, which fits our ambition of discovering new packers; (*ii*) can find arbitrarily-shaped clusters, since the packers may be different in terms of complexity (see Sec. 2.1), packing techniques, or how they overlay the binary; (*iii*) has the notion of noise, which can have multiple interpretations in our context: rare (singleton) packers, outliers, or packers using very sophisticated obfuscation techniques and thus hard to group under clusters; (*iv*) gives control on parameters (*eps*[2] and *minPts*[3]), which allows parameter tuning in the offline phase.

4.4.1 *Scattered representative points.*

To ease locating the clusters in the hyperplane while reducing comparisons complexity during the incremental update process, we select for each cluster a subset of its points *to represent its geometry.* To this end, we target the *most scattered points* of the cluster, which we call *Scattered Representative Points* (SRPs), see Fig. 3.

Finding the most scattered points of a cluster is akin to the *farthest neighbors traversal* problem: given a set of N points, find the X points that are the farthest apart from each other. Heuristics exist in literature to solve this problem [20]. In our work, we trade-off between precision and performance, with a *greedy approach* : first take the two farthest points, then incrementally select points for which the sum of distances to the already selected points is the greatest, repeating this until a specific number of representative points (n_{rp}) have been selected. This n_{rp} is equal to $\lceil K\sqrt{N_c}\rceil$, where N_c is the number of elements of cluster c, K is a positive constant (that can be experimentally tuned), and $\lceil . \rceil$ is the round up number. This means that each cluster has a different n_{rp}, at least equal to 1, that grows dynamically with the number of elements of the cluster.

4.4.2 *Clustering in the offline phase.*

In this phase, DBSCAN parameter (*eps*) is tuned to achieve the best clustering wrt. the provided ground truth. The clusters are evaluated according to their *homogeneity* and *number* wrt. the number of packer families classes in the ground truth. The trade-off between these two constraints is established via clustering metric AMI (see Sec. 7).

The intuition behind the offline phase is to derive a generalized *eps* that is *learned* from the already available labeled samples. This experimental *eps* should work for a big variety of packers and give the incremental clustering step the ability to handle *new unseen packers*, hence the importance of providing a well-varied packer training set at the beginning. In practice, this variety represents various packer complexities, and techniques, e.g. self-installers, cryptors, compressors, protectors and virtualizers.

Parameter *minPts* is left as supervisor choice for the desired minimum number of samples in clusters. Assuming that some unknown packers (families) can be spread with very few elements (e.g. only 3), the value of this parameter should be low so that our clustering encompasses small clusters (in addition to larger ones).

4.4.3 *Incremental clustering in the online phase.*

The *online* phase is responsible for updating the existing clustering as new samples come in. To this end, we customize an incremental version of DBSCAN [36] including a specific update policy and a set of optimizations techniques. This *update policy* covers three cases, checked in the following order:

A: The new sample joins the nearest cluster when at least *minPts* sample, including the new one, are within radius *eps*.

B: A new cluster is formed when at least *minPts* unclassified samples (noise), including the new one, are within radius *eps*.

C: The new sample remains unclassified when not A nor B.

[2] eps specifies the radius of a neighborhood wrt. some point.

[3] $minPts$ is the minimum number of points within radius eps required to form a cluster.

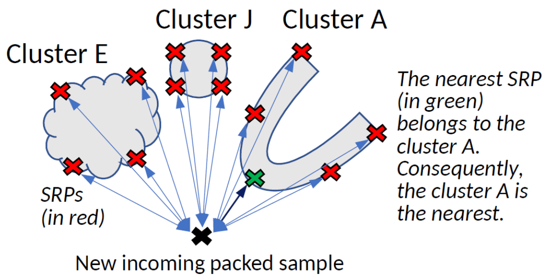
Cluster J
Cluster A
Cluster E
The nearest SRP (in green) belongs to the cluster A. Consequently, the cluster A is the nearest.
SRPs (in red)
New incoming packed sample

Figure 3: Nearest cluster search in incremental update

A naive update would require computing distances between the new (sample) point and all the points in the clusters, resulting in a complexity of $O(M \cdot N_c)$ per update, where M is the number of existing clusters and N_c the number of points in cluster c. However, using the *SRPs* in order to select the nearest cluster reduces the computation to $O(M \cdot n_{rp})$, in our case $O(M \cdot \sqrt{N_c})$ (see Sec. 4.4.1). The idea to optimize the computations is to get gradually closer to the hyperplane region where the potential points for case A are located. Therefore, in the first step, we compute the distances to the set of SRPs of each cluster, as illustrated in Fig. 3, to find the nearest cluster. In the second step, we delimit the region of points around the nearest SRP for which distances have to be computed. By exploiting the *triangle inequality*, using the distances already computed between the nearest SRP and the points in the cluster, we identify the set of points that are *certainly closer* or *further* than *eps* from the new point. These points can thus be directly *accepted* or *rejected* (without distance computation) within the update policy. Then, only the remaining points (those we can not decide using the triangle inequality) would require distance computation.

If the new sample joins the nearest cluster (case A), the set of SRPs of the cluster may have to be updated. Such update, if performed frequently (e.g. at each cluster modification), would make the incremental process costly. Therefore, to be efficient while still capturing the evolving geometry of the cluster, we limit the update of SRPs to be recomputed only when n_{rp} increases.

5 POST CLUSTERING SAMPLES SELECTION

After some amount of updates, an analyst may want to select a subset of relevant samples[4] from a cluster for further packer analysis.

Our strategy consists in representing the cluster by multiple connected regions, from which we select what we call *Post Clustering Relevant Samples* (PCRS). The diameter of these regions is set up by the analyst and allows for quicker or more detailed view on the clusters. The selected samples are ranked according to their region density in order to provide a measure of their *relevance* in the cluster. We call this measure *density marker*.

Our PCRS selection strategy (see Fig. 4) assumes that any shape generated by DBSCAN consists of connected *core points*[5]. Core points that are close (within *eps* radius) are relatively similar. A core point can be selected to represent the points (including core points) comprised within its region. More precisely, for given a cluster C, for each of its core points P_i our procedure visits, we identify the list of core points $\mathcal{P}_r^{(P_i)}$ directly reachable within radius r (selected by the analyst), i.e. $\mathcal{P}_r^{(P_i)} = \{P_{j,j \neq i} | d(P_i, P_j) \leq r\}$. The density marker of P_i is computed as $\rho_r(P_i) = \frac{|\mathcal{P}_r^{(P_i)}|}{|C|}$. Our traversal procedure visits a core point P_j only if it does not belong to the list $\mathcal{V}$ of previously visited core points, nor to their reachable core points list i.e. $P_j \notin \{\mathcal{V} \cup \mathcal{P}_r^{(P_k)} | P_k \in \mathcal{V}\}$. At the end of the procedure, the cluster is represented by the list of visited core points $\mathcal{V}$, for which samples with higher ρ_r are more *relevant* for further analysis.

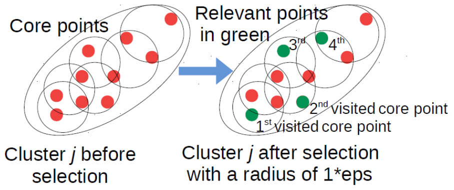

Figure 4: PCRS selection strategy. Note that the selection of the first core point to visit is done randomly.

6 DATASETS AND GROUND TRUTH

In our experimentation, we rely on two datasets: a MW feed and a synthetic dataset. We detail the origin, the collection period and the ground truth construction for each dataset.

6.1 Malware Feed

This first dataset consists of *real* packed binaries selected from a MW feed (unpacked and packed MW binaries) from Feb. 2017 to Oct. 2017 provided by Cisco. As our contribution focuses on packers classification, we select from the feed the binaries for which there is a consensus on the packer family label in the packing detection tools (DIE, Yara, PEiD, and a Cisco proprietary tool) used to build the ground truth. A consensus is met when at least three of these tools agree on the packer family (regardless of the packer version). With this 3-consensus, we selected 16 159 binaries out of the whole feed dataset of 281 344 binaries. Although it discarded many potentially packed binaries, the 3-consensus provides a higher quality ground truth, hence higher reliability for training and evaluation. Table 1 summarizes selected packer families, as well as the packer version indicated by DIE to illustrate a possible version labeling.

The complexity classes (see Sec. 2.1) of most of these packers range from I to III, like more than 85% of the world-wide packers evaluated in [38]. This means our packers are quite representative in terms of complexity. They also use diverse packing techniques: self-installers, compressors, cryptors, protectors, virtualizers.

6.2 Synthetic Dataset

We created a second dataset of 18 798 packed binaries[6]. On a freshly installed 32-bit Windows 7, we collected 694 PE clean binaries mainly from the *system 32* folder and packed them with 31 public, commercial, professional and custom packers. The fact these binaries are cleanware and not MW is not a problem, since we focus mainly on packers, which are not all malicious.

[4] Not to be confused with SRPs.

[5] P is a *core point* if at least $minPts$ points (including P) are within a radius eps of it.

[6] For access to this dataset, please contact the authors.

Table 1: Malware feed. *Packers in blue italics are specific to this dataset*. Packers in black belong to families *common* to both MW feed and synthetic datasets. "v?" is unspecified version. "x" is one or multiple sub-versions.

Packers	(version, # samples)	Total	Packers	(version, # samples)	Total
ActiveMARK	(v?, 2), (v5.x, 1)	3	*NsPacK*	(v?, 1), (v3.x, 13)	14
ASPack	(v1.08.x, 2), (v2.1, 13), (v2.12-2.42, 5818)	5833	Packman	(v1.0, 23)	23
ASProtect	(v1.0, 1), (v1.23-2.56, 174)	175	*PCGuard*	(v4.06, 3), (v5.0x, 1)	4
AutoIt	(v3, 1036)	1036	*Shrinker*	(v3.5, 3)	3
ExeStealth	(v2.74, 1)	1	*PEPACK*	(v1.0, 6)	6
eXPressor	(v1.3, 11), (v1.4.5.x, 1)	12	*PESpin*	(v1.3x, 6)	6
FishPE	(v1.3, 3)	3	Petite	(v2.1, 2), (v2.2, 7)	9
FSG	(v1.33, 2), (v2.0, 11)	13	RLPack	(v1.15-1.18, 1)	1
InnoSetup	(v?, 51), (v1.12.9, 1), (v1.3.x, 3), (v2.0.x, 6), (v3.0.x, 7), (v4.0.x, 4), (v4.1.4, 1), (v4.2.x, 4), (v5.0.x, 4), (v 5.1.x, 40) (v5.2.x, 35), (v5.3.x, 73), (v5.4.x, 45), (v5.5.x, 660)	934	PECompact	(v19x, 1), (v20x, 14), (v2.7x, 12), (v2.80x, 1), (v2.9x.x, 13), (v3.0x.x, 1185)	1226
InstallShield	(v?, 1), (v7.01.x, 1), (v9.00.x, 1), (v10.50.x, 1)	4	Themida	(v1.8.x-1.9.x, 12)	12
MEW	(v1.1-1.2, 217)	217	UPack	(v0.1x/0.20/0.21/0.24, 1), (v0.24-0.27/0.28, 19), (v?, 21)	41
MoleBox	(v2.3.3-2.6.4, 7)	7	WinRAR	(v?, 35)	35
NeoLite	(v1.0, 2)	2	UPX	(v0.6x, 2), (v0.7x, 5), (v1.0x, 40), (v1.2x, 2157), (v1.9x, 14), (v2.0x, 114), (v2.9x, 6), (v3.0x, 1599), (v3.9x, 609)	4549
NSIS	(v?, 361), (v1.x, 5) (v2.0x, 47), (v2.1x, 20), (v2.2x, 28), (v2.3x, 33), (v2.4x, 977), (v2.5x, 43), (v3.0x, 397), (v9.99, 4)	1916	WinZip	(v3.1, 66)	66
			Wise	(v?, 8)	8
MW feed			All packers		16159

Table 2: Synthetic dataset. *Packers in blue italics are specific to this dataset.* Packers in black belong to families common to both MW feed and synthetic datasets.

Packers	(version, # samples)	Total	Packers	(version, # samples)	Total
Armadillo	(v2.52, 628)	628	MEW	(v1.1, 634)	634
ASPack	(v2.36, 633)	633	*mPress*	(v2.19, 593)	593
Custom Packer 1 [4]	(v1.0, 125)	125	NeoLite	(v2.0, 617)	617
Custom Packer 2 [9]	(v1.0, 22)	22	PackMan	(v1.0, 640)	640
Custom Packer 3 [7]	(v1.0, 277)	277	PECompact	(v3.03.23, 670)	670
Custom Packer 4 [3]	(v1.0, 648)	648	*PELock*	(v2.08, 621)	621
Custom Packer 5 [10]	(v1.0, 655)	655	*PENinja*	(v1.0, 666)	666
Custom Packer 6 [2]	(v1.0, 635)	635	Petite	(v2.4, 625)	625
Custom Packer 7 [14]	(v1.0, 651)	651	RLPack	(v1.21, 645)	645
Custom Packer 8 [6]	(v1.0, 651)	651	*telock*	(v0.98, 595)	595
Custom Packer 9 [5]	(v1.0, 547)	547	Themida	(v2.4.5.0, 612)	612
Custom Packer 10 [15]	(v1.0, 655)	655	UPack	(v0.39, 664)	664
eXPressor	(v1.8.0.1, 668)	668	UPX	(v3.91, 579) (v3.95, 579)	1158
ezip	(v1.0, 597)	597	WinRAR	(v5.60, 694)	694
FSG	(v2.0, 630)	630	WinZip	(v5.0, 689)	689
			YodaCryptor	(v1.2, 653)	653
Synthetic dataset			All packers		18798

Packers "Custom Packer i", $i \in [1..10]$, come from public repository Github. Their families are not found by PEiD, Yara nor DIE. Although we could just use known packers (e.g. UPX, Armadillo) to simulate the arrival of new packer classes after training, these extra Custom Packers more concretely simulate the arrival of *new* packers. Indeed, in practice MW can simply be packed with a very recent custom packer, taking advantage of the relatively unknown packer class to evade MW detection systems before these are updated.

For all 31 packers default settings were used when packing the binaries. Packing failed in some cases because the binaries were too small to be packed, or their PE structure could not be modified. Table 2 summarizes the number of samples of each packer family and the version used. Github references are given for custom packers.

This dataset is quite representative as well, because the packers complexity classes range from I to III (except Armadillo that has the class IV [38]), and their packing techniques are quite different.

7 PERFORMANCE METRICS

Both extrinsic and intrinsic metrics are used to evaluate clusters.

7.1 Extrinsic Metrics

Our ground truth packed samples are labeled by their packer family. Each family comprises many variants (different versions in Table 1; this could also be different configurations or polymorphic instances). Our feed dataset has no absolute ground truth, since labeling tools do not agree on versions. Variants may be far apart wrt. *eps*, so we do not punish a clustering procedure splitting a family into different clusters, provided sub-clusters contain elements of the same family.

In this context, the *homogeneity score* indicates how much a cluster contains samples belonging to a single family (class). Let T be the ground truth classes, and C be the predicted clusters by the clustering algorithm, then the homogeneity score h is given by:

$$h = 1 - H(T|C)/H(T) \tag{4}$$

where $H(\cdot)$ is entropy and $H(\cdot|\cdot)$ conditional entropy. $h \in [0, 1]$, low values indicating low homogeneity. h does not punish dividing one class in smaller clusters, so high homogeneity is easy to achieve with a large number of clusters. h is 1 if every element is clustered into its own size-1 cluster, so we use this metric to evaluate our clustering *homogeneity* only, not its global extrinsic quality.

The Normalized Mutual Information (NMI) trades-off between homogeneity of clusters and their number. We use the Adjusted Mutual Information (AMI), an *adjusted-for-chance* version of NMI highly recommended in the clustering literature [39], defined by:

$$AMI(T, C) = \frac{I(C, T) - E[I(T, C)]}{\sqrt{H(C) * H(T)} - E[I(T, C)]} \tag{5}$$

where $I(\cdot)$ is the mutual information and $E[\cdot]$ is the expectation. $AMI \in [0, 1]$, higher values indicate more *homogeneous clusters* and/or a number of clusters closer to the number of packers families in T. AMI is 1 when T and C are identical and 0 when any commonality is due to chance. This metric punishes clustering with large number of clusters, since such clustering has a high $H(C)$.

7.2 Intrinsic Metrics

We use DBCV (Density-Based Clustering Validation)[29], that can validate arbitrarily-shaped clusters. This metric computes the density within a cluster (*density sparseness*), and the density between clusters (*density separation*). For a clustering $C = \{C_i\}, 1 \leq i \leq l$:

$$DBCV(C) = \sum_{i=1}^{l} |C_i|/|O|\, V_c(C_i) \qquad (6)$$

where $|O|$ is the number of samples, and $V_c(C_i)$ is the validity index of cluster C_i, $1 \leq i \leq l$, defined as:

$$V_c(C_i) = \frac{\min\limits_{1 \leq j \leq l, j \neq i} \big(DSPC(C_i, C_j)\big) - DSC(C_i)}{\max\left(\min\limits_{1 \leq j \leq l, j \neq i} \big(DSPC(C_i, C_j)\big), DSC(C_i)\right)}$$

where $DSPC(C_i, C_j)$ is the density separation between clusters C_i and C_j, and $DSC(C_i)$ is the density sparseness of cluster C_i.

$DBCV(C) \in [-1, 1]$, higher values indicate a clustering with high density within clusters and/or low density between clusters.

8 EXPERIMENTAL EVALUATION

This section starts with an overview of software and hardware implementations, then presents the experimental evaluation we performed on our offline and online phases, and the results obtained.

All experiments were performed in Python 3.6.8 on a server with four 14-core processors at 2GHz with 128 GB of RAM.

The selection of the most effective features groups combinations was based on the best AMI score obtained in the offline phase.

We used the framework Radare2 [12] to emulate the unpacking code execution and get the trace of ASM mnemonic sequences. Execution is stopped when the ASM sequence length reaches 50 mnemonics, a tradeoff between relevant information (unpacking stub code) and cost of extraction. In [32], up to 30 mnemonics are statically extracted. We slightly extended this length to 50 to improve the relevant information quality. The average time needed to extract the ASM sequence is 0.23 second per sample.

We computed the PE sections features from the PE header of the packed file using a C++ PE parser able to handle (packed) MW samples [11]. The average time needed to extract these features is 0.014 second per sample.

We largely modified the online implementation [36] of the incremental DBSCAN to fit our methodology. For the batch version of DBSCAN and the evaluations metrics, we used Scikit-learn [13].

8.1 Scenarii Definition

The two datasets share common packers, but also have their own (see Tables 1 and 2). So we designed two scenarii to exhibits the behavior of our system when facing the arrival of known packers as well as new, unknown ones:

MF/S: we use the MW feed as training set in the offline phase, then the Synthetic dataset as test in the online phase.

S/MF: we use the synthetic dataset as training set in the offline phase, then the MW feed as test in the online phase.

8.2 Offline Phase

This phase supervises the creation of clusters with each training set, given the ground truth, in each scenario. The goal is to fine-tune *eps* according to the *AMI* score, the best found *eps* will then be used all along the online phase against the testing set, in each scenario.

Table 3: Summary of results of the offline phase

Training set	AMI	h	# clusters	best *eps*	*minPts*
MW feed	0.941	0.987	38	0.08	3
Synthetic	0.985	0.993	43	0.06	3

Table 4: Impact of n_{rp} on the effectiveness and update time per sample

K	Scenario			Scenario		
	MF/S			S/MF		
	AMI	h # clusters	update time (s)	AMI	h # clusters	update time (s)
10^{-2}	0.933	0.960 82	1.097	0.934	0.981 95	1.066
10^{-1}	0.936	0.959 80	1.190	0.935	0.981 95	1.125
1	0.945	0.960 76	3.245	0.937	0.981 91	3.035
10	0.948	0.960 73	22.950	0.937	0.981 91	22.378
100	0.948	0.960 73	51.196	0.937	0.981 91	49.544

In both scenarii, we set the value of *minPts* to 3 (see Sec. 4.4.2). For *eps*, we tune it over $[0.01, 2]$ with 0.0025 increments.

Table 3 summarizes for each training set the best results achieved wrt. AMI score. Homogeneity h and number of clusters are given to explain the AMI score and detail the obtained clustering. The best *eps* found slightly differs between the two training sets, due to the bias caused by the different variety of packers in each training set. The resulting clusters and their contents are presented in the column labeled "offline phase" in Table 6.

8.3 Online Phase

In this phase, we first study the impact of n_{rp} on the effectiveness and update time of our system. Then, we evaluate the robustness of our solution over time. Finally, we test our PCRS selection strategy and discuss how it optimizes the cost of post clustering analysis tasks.

8.3.1 *Scattered Representative Points.*

Here, we study the impact of n_{rp} (see Sec. 4.4.1) on the *effectiveness* and *update time* of our model. Thus, we vary K in $\lceil K\sqrt{N_c}\rceil$ and then select a K that provides effectiveness while keeping our solution reasonably fast. For both scenarii, we try K among $\{10^{-2}, 10^{-1}, 1, 10, 100\}$, in order to largely vary n_{rp}. With our dataset, when $K = 10^{-2}$, n_{rp} is 1 (the minimum possible) for all clusters; when $K = 100$, n_{rp} equals the number of samples (the maximum possible) for all clusters.

Table 4 presents the obtained results. The update time is the average time in seconds to update the clustering when a new sample arrives, without considering features extraction time.

Impact on effectiveness. In Table 4, we observe in both scenarii that the higher the K, the slightly higher the AMI score, until stabilizing at $K = 10$. The AMI score is controlled by the ratio between homogeneity and number of clusters. Homogeneity stays stable when K increases, but the number of clusters decreases then stabilizes, so the AMI score increases then stabilizes. Thus, reaching

$K = 10$, the n_{rp} becomes sufficient for our model to achieve its best effectiveness. When SRPs are too few to well represent the geometric shape of some or all clusters, the model does not always find the nearest cluster (see Fig. 3). Thus, the new sample stays "unclassified" or contributes to creating a new cluster, instead of joining the nearest existing cluster.

Impact on update time. In Table 4, the more K increases the more update time increases (drastically). For $K = 10^{-2}$, average update time is around one second. For $K = 100$, it is around 50 seconds, because many more comparisons with SRPs are performed to find the nearest cluster. This confirms the paramount importance of SRPs to optimize computation in the update process.

Optimal K selection strategy. The optimal K trades-off between effectiveness and update time. This work aims to be *highly effective* and *reasonably fast*, so we discard all K values leading to an update time above 1.5 second. We then select the K with the highest AMI score in the solutions left. $K = 10^{-1}$ appears as the optimal solution for both our scenarii, and is thus used in our next experiments.

8.3.2 *Effectiveness and robustness of SE-PAC.*

Here, we evaluate in more details the effectiveness of our **SE-PAC** (**S**elf-**E**volving **PA**cker **C**lassifier) system on the various update uses-cases (see Sec.4.4.3). Then we study how this effectiveness evolves over time in order to gauge the robustness of the model. The values of *eps*, *minPts*, and K are selected as previously described.

Time-flow of incoming samples. We simulate the arrival of the test packers over several months. Each month, a number of specific and common packers (see Tables 1 and 2) appear in each scenario. The *specific packers* represent *new packer classes*, and the *common packers* represent *variants*. The specific packers arrive in a random order, one specific packer appearing each month. The experimental test period covers 17 months (17 specific packers) for MF/S and 15 months (15 specific packers) for S/MF. The samples of each specific packer are equally distributed from the arrival month of their packer till the end of experiment: in MF/S, 125/17 samples of Custom Packer 1 appear in month 1, the rest is then equally distributed over the 16 months left; 22/16 samples of Custom Packer 2 appear in month 2, the rest is distributed over the 15 months left. The samples of common packers are equally distributed through the whole experimental period: in MF/S, packers ASPack and UPX appear monthly with a quantity of 633/17 and 1158/17 respectively.

Table 5 summarizes the final results regarding AMI, homogeneity, number of clusters, and DBCV obtained after the whole update process. Fig. 5 to 8 present the monthly evolution of those metrics. We now explain and discuss these results.

AMI, homogeneity, and number of clusters evolution. In MF/S, AMI stays high and quite constant over time. In S/MF it slightly decreases (see Fig. 5), because over time S/MF forms higher number of clusters (which the AMI metric punishes, see Sec. 7.1) than MF/S (see Fig. 7). Indeed, the scenario S/MF is more likely to classify an incoming packer to a new cluster, because the *eps* range (= 0.06) is smaller, so the incoming samples of some test packers may not be grouped under one cluster but form new additonal clusters instead. Therefore, higher homogeneity in S/MF is maintained since different packer families are more likely not to mix (see Fig. 6).

Table 5: Summary of the final results

Scenario	AMI	h	# clusters	DBCV
MF/S	0.936	0.959	80	0.285
S/MF	0.935	0.981	95	0.575

DBCV evolution. The significant decrease of the DBCV score (see Fig. 8) was expected, since SE-PAC tends to find for some packers multiple but very close clusters, which thus strongly decreases their density separation. Being multiple, they strongly impact the global mean of the intrinsic clustering quality DBCV(C). This score is worse in MF/S than in S/MF, because the best *eps* (= 0.08) is higher in MF/S, hence the model tends to have lower density within clusters and higher density between clusters.

Table 6 presents the final results obtained by SE-PAC after the offline and online phases, considering the content of clusters found and the DBCV score, for each packer family. We explain and discuss the results of this table next.

Misclassifications. They are marked in *italics*: e.g. in MF/S, one sample of Custom packer 2 is wrongly classified with both samples of Custom packer 10 and one sample of AutoIt, in cluster 47. Since *eps* tuning trades off between correct classifications and number of clusters, it may lead to misclassifications, which are reported in the offline phase . We chose the best AMI score. Experiment shows that scenario S/MF generates a smaller *eps* in the offline phase, resulting in less misclassifications during offline and online phases.

Specific packers. In both scenarii, despite some misclassifications, for most specific packers, including the custom ones, new clusters are created. This shows our system is able to identify new packers.

Common packers. They either joined their respective existing packer family clusters or formed new ones, or both of them. For example, in MF/S all samples of ASPack joined their family cluster 2, while all samples of eXPressor formed the new cluster 39. For FSG, many samples joined existing cluster 6, while the others formed new clusters 25, 40 and 56. Some common test packers did not join their existing family clusters mainly because their *versions* differ greatly (wrt. *eps*) from the one used in the training set. For example, in MF/S samples of Themida version 2.4.5.0 (see Table 2) used as test formed the new cluster 45, instead of joining their packer family cluster 19 that hosts a different version of the same packer. The same happened to packers eXPressor, Petite, WinRAR and WinZip.

Finally, multiple sub-clusters were formed for some packers families, because of: (*i*) Different *versions*: e.g. see UPX, NSIS and InnoSetup in Table 1; (*ii*) Obfuscation: e.g. YodaCryptor generates polymorphic instances of its unpacking code by using the *instruction substitution* technique that *tampers* quite arbitrarily the mnemonic sequence without affecting its behavior, which makes the grouping of ASM sequences harder, causing additional clusters, and noise.

8.3.3 *PCRS selection.*

This experiment evaluates how our PCRS selection strategy (see Sec. 5) performs on the previously obtained clusters.

Table 6: Cluster contents and DBCV score for each packer family, in both scenarii, after both offline and online phases. "Not learned" in the offline phase column indicates training does not include the packer, so the packer is considered *specific*, thus new, when used as test packer. "No score" means there is no cluster to evaluate. Cluster ID "-1" means noise. Results in *italics* are misclassifications.

Packers	**Content of clusters: (Cluster ID – # samples)**				**DBCV score per cluster: (Cluster ID – score)**	
	Scenario MF/S		**Scenario S/MF**		**Scenario MF/S**	**Scenario S/MF**
	Offline phase	**Online phase**	**Offline phase**	**Online phase**	**Online phase**	**Online phase**
ActiveMARK	(-1 – 3)	(-1 – 3)	**Not learned**	(-1 – 3)	No score	No score
ASProtect	(4 – 173), (-1 – 2)	(4 – 173), (-1 – 2)	**Not learned**	(62 – 82), (73 – 9), (81 – 81), (-1 – 3)	(4 – 0.6)	(62 – -0.1), (73 – -0.1), (81 – -0.1)
AutoIt	*(25 – 1)*, (36 – 1027), (37 – 4), (-1 – 4)	*(25 – 1)*, *(36 – 1027)*, (37 – 4), *(47 – 1)*, (-1 – 3)	**Not learned**	*(42 – 1027)*, (90 – 3), (-1 – 6)	(25 – 0.6), (36 – 0.8), (37 – 0.2), (47 – -0.2)	(42 – 0.6), (90 – 1)
ExeStealth	(-1 – 1)	(-1 – 1)	**Not learned**	(-1 – 1)	No score	No score
FishPE	*(20 – 3)*	*(20 – 3)*	**Not learned**	*(37 – 3)*	(20 – -1)	(37 – 0.7)
InnoSetup	(32 – 870), (33 – 38), (34 – 8), (35 – 3), (-1 – 15)	(32 – 870), (33 – 38), (34 – 8), (35 – 3), (-1 – 15)	**Not learned**	(56 – 610), (57 – 260), (61 – 38), (80 – 8), (84 – 3), (-1 – 15)	(32 – 0.4), (33 – 1), (34 – 0.8), (35 – 1)	(56 – 0.1), (57 – 0.9), (61 – 1), (80 – 0.7), (84 – 1)
InstallShield	*(7 – 4)*	*(7 – 4)*	**Not learned**	(74 – 4)	(7 – 0.2)	(74 – 0.9)
MoleBox	(8 – 7)	(8 – 7)	**Not learned**	(85 – 7)	(8 – 0.8)	(85 – 0.9)
NsPacK	(9 – 9), (10 – 5)	(9 – 9), (10 – 5)	**Not learned**	(82 – 9), (92 – 5)	(9 – 0.8), (10 – 0.7)	(82 – 0.8), (92 – 0.7)
NSIS	*(7 – 3)*, (29 – 12), (30 – 1730), (31 – 130), (-1 – 41)	*(7 – 3)*, (29 – 12), (30 – 1730), (31 – 130), (-1 – 41)	**Not learned**	(65 – 6), (66 – 205), (67 – 112), (68 – 39), (69 – 52), (70 – 399), (71 – 17), (72 – 10), (76 – 22), (77 – 993), (78 – 8), (79 – 6), (-1 – 47)	(7 – 0.2), (29 – 0.4), (30 – 0.2), (31 – 0.5)	(65 – 0.9), (66 – -1), (67 – 0), (68 – -1), (69 – 0.2), (70 – 0.2), (71 – 0.5), (72 – 0.6), (76 – 0.5), (77 – -1), (78 – 1), (79 – 0.7)
PEPACK	(17 – 5), (-1 – 1)	(17 – 5), (-1 – 1)	**Not learned**	(88 – 5), (-1 – 1)	(17 – 0.7)	(88 – 0.7)
PESpin	(27 – 3), (-1 – 3)	(27 – 3), (-1 – 3)	**Not learned**	(94 – 3), (-1 – 3)	(27 – 0.9)	(94 – 0.958)
PCGuard	(26 – 3), (-1 – 1)	(26 – 3), (-1 – 1)	**Not learned**	(93 – 3), (-1 – 1)	(26 – 1)	(93 – 1.0)
Shrinker	(28 – 3)	(28 – 3)	**Not learned**	(60 – 3)	(28 – 0.6)	(60 – 0.8)
Wise	*(7 – 8)*	*(7 – 8)*	**Not learned**	(83 – 8)	(7 – 0.2)	(83 – 0.6)
Armadillo	**Not learned**	(38 – 627), (-1 – 1)	(0 – 628)	(0 – 628)	(38 – 0.5)	(0 – 0.6)
Custom Packer 1	**Not learned**	(65 – 123), (-1 – 2)	(5 – 123), (-1 – 2)	(5 – 123), (-1 – 2)	(65 – 1.0)	(5 – 1.0)
Custom Packer 2	**Not learned**	*(47 – 1)*, (54 – 16), (77 – 5)	(6 – 22)	(6 – 22)	(47 – -0.2), (54 – -0.5), (77 – -0.3)	(6 – 0.9)
Custom Packer 3	**Not learned**	(48 – 277)	(7 – 277)	(7 – 277)	(48 – 1)	(7 – 0.9)
Custom Packer 4	**Not learned**	(53 – 646), (-1 – 2)	(8 – 648)	(8 – 648)	(53 – 0.3)	(8 – 0.5)
Custom Packer 5	**Not learned**	(62 – 651), (-1 – 4)	(9 – 654), (-1 – 1)	(9 – 654), (-1 – 1)	(62 – 0.6)	(9 – 0.6)
Custom Packer 6	**Not learned**	(58 – 635)	(10 – 635)	(10 – 635)	(58 – 0.8)	(10 – 0.8)
Custom Packer 7	**Not learned**	(49 – 646), (78 – 3), (-1 – 2)	(11 – 648), (12 – 3)	(11 – 648), (12 – 3)	(49 – 0.7), (78 – 1)	(11 – 0.7), (12 – 0.9)
Custom Packer 8	**Not learned**	(66 – 648), (-1 – 3)	(13 – 648), (-1 – 3)	(13 – 648), (-1 – 3)	(66 – 0.9)	(13 – 0.9)
Custom Packer 9	**Not learned**	(55 – 547)	(14 – 547)	(14 – 547)	(55 – 0.9)	(14 – 0.9)
Custom Packer 10	**Not learned**	*(47 – 655)*	(15 – 655)	(15 – 655)	(47 – -0.2)	(15 – 1)
ezip	**Not learned**	(52 – 597)	(17 – 597)	(17 – 597)	(52 – 0.8)	(17 – 0.8)
mPress	**Not learned**	(64 – 593)	(22 – 593)	(22 – 593)	(64 – 1)	(22 – 0.9)
PELock	**Not learned**	(50 – 617), (79 – 3), (-1 – 1)	(30 – 617), (31 – 3), (-1 – 1)	(30 – 617), (31 – 3), (-1 – 1)	(50 – 0.8), (79 – 0.7)	(30 – 0.8), (31 – 0.8)
PENinja	**Not learned**	(63 – 661), (-1 – 5)	(32 – 663), (-1 – 3)	(32 – 663), (-1 – 3)	(63 – 0.6)	(32 – 0.6)
telock	**Not learned**	(61 – 595)	(35 – 595)	(35 – 595)	(61 – 0.5)	(35 – 0.5)
YodaCryptor	**Not learned**	(67 – 27), (68 – 259), (69 – 30), (70 – 79), (71 – 55), (72 – 59), (73 – 79), (74 – 4), (75 – 3), (76 – 22), (-1 – 36)	(2 – 427), (3 – 27), (4 – 3), (-1 – 196)	(2 – 427), (3 – 27), (4 – 3), (-1 – 196)	(67 – 1), (68 – -0.2), (69 – -0.2), (70 – -0.3), (71 – -0.3), (72 – -0.2), (73 – -0.2), (74 – 0.2), (75 – 0.2), (76 – -0.2)	(2 – -0.1), (3 – 1), (4 – 0.1)
ASPack	(0 – 5702), (1 – 9), (2 – 117), (3 – 3), (-1 – 2)	(0 – 5702), (1 – 9), (2 – 748), (3 – 3), (-1 – 4)	(1 – 633)	(1 – 750), (43 – 4), (44 – 9), (45 – 5282), (46 – 416), (91 – 3), (-1 – 2)	(0 – 0.5), (1 – 0.4), (2 – 0.7), (3 – 0.8)	(1 – 0.4), (43 – 0.9), (44 – 0.5), (45 – 0.4), (46 – 0.4), (91 – 0.8)
eXPressor	(5 – 11), (-1 – 1)	(5 – 11), (39 – 668), (-1 – 1)	(16 – 668)	(16 – 668), (58 – 11), (-1 – 1)	(5 – 1), (39 – 1)	(16 – 1), (58 – 1)
FSG	*(6 – 13)*	*(6 – 624)*, *(25 – 1)*, (40 – 7), (56 – 4), (-1 – 7)	(18 – 611), (19 – 7), (20 – 4), (-1 – 8)	(18 – 624), (19 – 7), (20 – 4), (-1 – 8)	(6 – 0.3), (25 – 0.6), (40 – 0.9), (56 – 0.9)	(18 – 0.1), (19 – 0.9), (20 – 0.5)
MEW	*(6 – 217)*	*(6 – 851)*	(21 – 634)	(21 – 851)	(6 – 0.3)	(21 – 0.9)
NeoLite	(-1 – 2)	(41 – 614), (59 – 4), (-1 – 1)	(23 – 612), (24 – 4), (-1 – 1)	(23 – 614), (24 – 4), (-1 – 1)	(41 – 0.6), (59 – 0.9)	(23 – 0.7), (24 – 0.9)
Packman	*(11 – 23)*	*(11 – 661)*, (-1 – 2)	(25 – 640)	(25 – 663)	(11 – -0.1)	(25 – 0.1)
PECompact	(12 – 1016), (13 – 16), (14 – 140), (15 – 16), (16 – 4), (-1 – 34)	(12 – 1475), (13 – 16), (14 – 140), (15 – 16), (16 – 4), *(20 – 4)*, (42 – 43), (60 – 5), (-1 – 193)	(26 – 407), (27 – 40), (28 – 3), (29 – 8), (-1 – 212)	(26 – 486), (27 – 41), (28 – 3), (29 – 38), (47 – 838), (48 – 17), (49 – 140), (53 – 16), (63 – 16), (75 – 6), (86 – 24), (87 – 4), (89 – 7), (-1 – 260)	(12 – -0.7), (13 – 0.7), (14 – 1), (15 – 1), (16 – 1), (20 – -1), (42 – 0.9), (60 – 0.2)	(26 – -0.9), (27 – 0.9), (28 – 1), (29 – 0.3), (47 – 1), (48 – 1), (49 – 1), (53 – 0.6), (63 – 1), (75 – 1), (86 – -0.7), (87 – 1), (89 – 1)
Petite	(18 – 6), (-1 – 3)	(18 – 6), (43 – 625), (-1 – 3)	(33 – 625)	(33 – 625), (64 – 6), (-1 – 3)	(18 – 0.7), (43 – 9)	(33 – 0.9), (64 – 0.8)
RLPack	(-1 – 1)	(44 – 645), (-1 – 1)	(34 – 645)	(34 – 645), (-1 – 1)	(44 – 0.9)	(34 – 0.9)
Themida	(19 – 11), (-1 – 1)	(19 – 11), (45 – 612), (-1 – 1)	(36 – 612)	(36 – 612), (59 – 11), (-1 – 1)	(19 – 0.9), (45 – 0.9)	(36 – 0.9), (59 – 0.9)
UPack	(22 – 19), (23 – 21), (-1 – 1)	(22 – 19), (23 – 21), *(24 – 648)*, (46 – 16), (-1 – 1)	(40 – 648), (41 – 16)	(40 – 669), (41 – 16), (55 – 19), (-1 – 1)	(23 – 1), (24 – 1), (46 – 0.9)	(40 – 1), (41 – 0.9), (55 – 1)
UPX	*(11 – 5)*, *(20 – 4487)*, (21 – 55), (-1 – 2)	*(11 – 5)*, *(20 – 5066)*, (21 – 55), (51 – 6), (57 – 573), (-1 – 2)	(37 – 1152), (38 – 6)	*(37 – 5638)*, (38 – 6), (50 – 4), (51 – 55), (-1 – 4)	(11 – -0.1), (20 – -1), (21 – 0.8), (51 – 0.7), (57 – -1)	(37 – 0.7), (38 – 0.6), (50 – 0.8), (51 – 0.8)
WinRAR	(24 – 30), (-1 – 5)	*(24 – 30)*, *(36 – 694)*, (-1 – 5)	(39 – 694)	(39 – 694), (54 – 30), (-1 – 5)	(22 – 1), (23 – 1), (36 – 0.8)	(39 – 1), (54 – 0.7)
WinZip	*(25 – 66)*	*(25 – 66)*, *(36 – 689)*	(42 – 689)	*(42 – 689)*, (52 – 66)	(25 – 0.6), (36 – 0.8)	(42 – 0.6), (52 – 0.6)

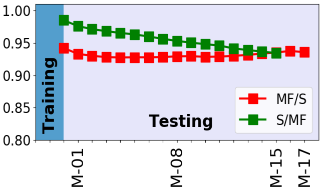

Figure 5: AMI evolution

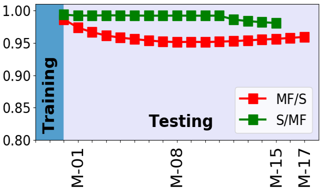

Figure 6: h evolution

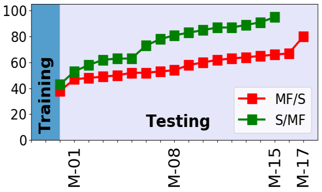

Figure 7: # clusters evolution

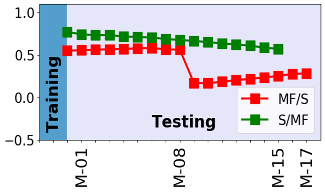

Figure 8: DBCV evolution

Table 7: # of PCRS

Radius r	# of PCRS	
	MF/S	S/MF
1 * eps	220	257
1.5 * eps	105	102
2 * eps	96	94

Table 8: Overview of some clusters after PCRS selection (MF/S)

Cluster ID	(PCRS rank – density marker)
12	(1 – 57%), (2 – 32%), (3 – 4%), (4 – 3%), (5 – 1%), (6 – 1%), (7 – 0.4%) , (8 – 0.3%), (9 – 0.3%), (10 – 0.1%) , (11 – 0.1%), (12 – 0.1%) (13 – 0.1%), (14 – 0.1%) , (15 – 0.1%) (16 – 0.1%) , (17 – 0.1%)
78	(1 – 50%), (2 – 21%) , (3 – 14%), (4 – 7%), (5 – 7%)
66	(1 – 100%)

The total number of samples inside all these clusters is 34613 in MF/S and 34304 in S/MF. Selection radius r is given by $r = \alpha * eps$, where *eps* is the best *eps* found for each scenario, and α is evaluated over $\{1, 1.5, 2\}$.

Table 7 presents the results of our PCRS selection strategy on all the previously found clusters, with different radii r. For $r = 1 * eps$, the number of PCRS is 0.6% (220/34613) of the total number of samples in clusters in MF/S, or one PCRS for 157 samples, and 0.7% (257/34304) in S/MF, or one PCRS for 133 samples. This ratio tends to stabilize when the selection radius is enlarged, the number of PCRS *converging* towards the number of clusters found.

Table 8 shows the results in MF/S of our PCRS selection strategy on three individual clusters, with $r = 1 * eps$. The PCRS are ranked by density marker. Clusters 12, 78 and 66 correspond respectively to PECompact and custom packers 7 and 8. The difference in number and distribution of the PCRS logically lies on how the samples of a cluster are scattered in the hyperplan. For example, cluster 12 needs 17 PCRS, because PECompact uses a random key to generate polymorphic instances of the unpacking stub.

9 DISCUSSION

In this section, we first give further insights on our results, then we discuss the possibility of human interaction when using our approach over a large time frame. Finally, we examine experimental properties influencing our results validity, and limitations.

Findings and Insights. Our results show that our incremental system stays valid over time. Indeed, keeping high homogeneity at the cost of a few additional clusters is favored. In that sense, it appears that the S/MF scenario is the most suitable. Reducing *eps* would reduce the cost of supervisor intervention and ensure longer user protection.

Furthermore, in spite of a sharp decrease of the intrinsic quality of our model (see Fig. 8), extrinsic quality remains effective (see Fig. 5). Indeed, our update policy does not autonomously merge close clusters, which prevents potential upcoming misclassifications, at the cost of a few additional clusters.

Custom packers were accurately classified (see Table 6). This validates the efficiency of the chosen features to represent packers. This also hints that these packers are simply inspired from well-known packers. In parallel, the difficulty to group some samples gives insights on whether the packer was originally developed for obfuscation purposes, like YodaCryptor or PECompact, or for mere compression and encoding like WinRAR and WinZip. This could be further exploited by an analyst for *threat intelligence*.

Our PCRS selection strategy may reduce post clustering complexity by a factor of up to 157 (see Sec. 8.3.3), an important optimization for *concrete* tasks like packer analysis and/or unpacking. Therefore, *with minimal cost, packer classification, thus unpacking, systems can be updated*, which means updating signature-based tools, ground truth labels, specialized unpacking functions, and generic unpacking heuristics.

Our methodology of cluster evolution is fully portable, and our feature extraction can be adapted to other executable file formats.

When and How to Retrain? After a large amount of incremental updates, a human intervention may be useful, hence the question of when and how to retrain.

One approach is to exploit the extrinsic metrics that use the available ground truth to detect misclassifications. We distinguish two cases: (i) *misclassification of known packers*, where the error can be detected (see for example our experiment MF/S with the mix between WinZip and WinRAR); (ii) *misclassification of unknown packers*, where the error is not detectable as long as the samples inside the clusters are not further analyzed and labeled.

Another approach relies on intrinsic metrics, like DBCV, that can indicate that some packers are very close, and thus prevent potential upcoming misclassifications.

Based on the metrics above, a trigger policy might be adopted to rise an alarm for retraining, which means adjusting *eps*, clustering from scratch, then continuing the incremental clustering.

Threats to Validity. Our clustering updates do not autonomously merge or split clusters. This provides our system a high resilience over time, by prohibiting the autonomous merging of close different packers, at the cost of a few additional clusters. Such operations can be done manually by (re-)labeling the involved clusters if necessary, which means giving the same ID for clusters that would merge, and adding new IDs for clusters that would result from a split.

Noise does not impact our approach, because noise points are not discarded but kept to be re-clustered or left unclassified. In practice, noise can be hard-to-group samples due to obfuscation, or rare (singletons) packers. Further post clustering analysis could reveal their exact nature.

Limitations and Future Work. The tapered Levenshtein distance we applied on our ASM sequences remains fragile against *highly* polymorphic engines that generate very different variants of the same unpacking stub code. If these variants are few, the number of clusters generated remains acceptable; otherwise, our clustering system will produce a very high number of clusters. Future work should consider the extraction of more semantic features, and thus the use of a semantic distance metric, in order to better mitigate such high and complex polymorphism.

10 CONCLUSIONS

This paper presented a *self-evolving packer classifier* that deals with the issue of rapid evolution of packers. We proposed a composite pairwise distance metric that is constructed from the combination of different types of packer features. We derived an incremental clustering approach able to identify both (variants of) known packers (families) and new ones, as well as automatically and efficiently update the clusters.

We evaluated our solution on two datasets: MW feed, and synthetic. The results showed that our classifier is effective and robust over time in identifying both known and new packer families. Indeed, our approach constantly enhances, integrates, adapts and evolves packer knowledge, making our classifier effective for longer times.

Furthermore, our work includes a new post clustering strategy that selects a subset of relevant samples from each cluster found, to optimize the cost of post clustering processes.

We thus believe our work can help security companies, researchers and analysts to quickly, effectively and continually update their packer classification, and thus unpacking, systems to ensure a better continuity of security for users over time.

ACKNOWLEDGMENTS

The authors want to thank Cisco for providing the MW feed and stimulating discussions; Stefano Sebastio for his valuable discussions and his help in structuring this paper in its early stages; Stéphane Ubeda for his constructive insights; the four anonymous reviewers for the thoughtful comments in their reviews that helped us significantly improve this paper.

REFERENCES

[1] 2007. BITDEFENDER ANTIVIRUS TECHNOLOGY. White paper. https://www.bitdefender.com/files/Main/file/BitDefender_Antivirus_Technology.pdf Access: Oct. 2020.

[2] 2016. PE Toy. https://github.com/qy7tt/petoy. Access: Oct. 2020.

[3] 2017. PePacker. https://github.com/SamLarenN/PePacker. Access: Oct. 2020.

[4] 2018. Amber. https://github.com/EgeBalci/Amber. Access: Oct. 2020.

[5] 2018. Simple-PE32-Packer. https://github.com/z3r0d4y5/Simple-PE32-Packer. Access: Oct. 2020.

[6] 2019. theArk. https://github.com/aaaddress1/theArk. Access: Oct. 2020.

[7] 2019. Writing a simple PE Packer in detail. https://github.com/levanvn/Packer_Simple-1. Access: Oct. 2020.

[8] 2020. Detect-It-Easy. https://github.com/horsicq/Detect-It-Easy. Access: Oct. 2020.

[9] 2020. Origami. https://github.com/dr4k0nia/Origami. Access: Oct. 2020.

[10] 2020. PE-Packer. https://github.com/czs108/PE-Packer. Access: Oct. 2020.

[11] 2020. PeLib. https://github.com/avast-tl/pelib. Access: Oct. 2020.

[12] 2020. Radare2. https://rada.re/n/. Access: Oct. 2020.

[13] 2020. scikit-learn: Machine Learning in Python. https://scikit-learn.org/. Access: Oct. 2020.

[14] 2020. Silent-Packer. https://github.com/SilentVoid13/Silent_Packer. Access: Oct. 2020.

[15] 2020. xorPacker. https://github.com/nqntmqmqmb/xorPacker. Access: Oct. 2020.

[16] 2020. YARA. https://github.com/VirusTotal/yara. Access: Oct. 2020.

[17] Margareta Ackerman and Sanjoy Dasgupta. 2014. Incremental clustering: The case for extra clusters. In *Advances in Neural Information Processing Systems*. 307–315.

[18] Munkhbayar Bat-Erdene, Hyundo Park, Hongzhe Li, Heejo Lee, and Mahn-Soo Choi. 2017. Entropy analysis to classify unknown packing algorithms for malware detection. *International Journal of Information Security* 16, 3 (2017), 227–248.

[19] Fabrizio Biondi, Michael A Enescu, Thomas Given-Wilson, Axel Legay, Lamine Noureddine, and Vivek Verma. 2019. Effective, efficient, and robust packing detection and classification. *Computers & Security* 85 (2019), 436–451.

[20] Erhan Erkut, Yilmaz Ülküsal, and Oktay Yenicerioğlu. 1994. A comparison of p-dispersion heuristics. *Computers & operations research* 21, 10 (1994), 1103–1113.

[21] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Kdd*, Vol. 96. 226–231.

[22] John C Gower. 1971. A general coefficient of similarity and some of its properties. *Biometrics* (1971), 857–871.

[23] Nguyen Minh Hai, Mizuhito Ogawa, and Quan Thanh Tho. 2017. Packer identification based on metadata signature. In *Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop*. 1–11.

[24] Kesav Kancherla, John Donahue, and Srinivas Mukkamala. 2016. Packer identification using Byte plot and Markov plot. *Journal of Computer Virology and Hacking Techniques* 12, 2 (01 May 2016), 101–111. https://doi.org/10.1007/s11416-015-0249-8

[25] Xufang Li, Peter KK Loh, and Freddy Tan. 2011. Mechanisms of polymorphic and metamorphic viruses. In *2011 European intelligence and security informatics conference*. IEEE, 149–154.

[26] Xingwei Li, Zheng Shan, Fudong Liu, Yihang Chen, and Yifan Hou. 2019. A consistently-executing graph-based approach for malware packer identification. *IEEE Access* 7 (2019), 51620–51629.

[27] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. 2007. Omniunpack: Fast, generic, and safe unpacking of malware. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 431–441.

[28] Steve Morgan. 2019. Global Ransomware Damage Costs Predicted To Reach $20 Billion (USD) By 2021. https://cybersecurityventures.com/global-ransomware-damage-costs-predicted-to-reach-20-billion-usd-by-2021 Access: Oct. 2020.

[29] Davoud Moulavi, Pablo A Jaskowiak, Ricardo JGB Campello, Arthur Zimek, and Jörg Sander. 2014. Density-based clustering validation. In *Proceedings of the 2014 SIAM international conference on data mining*. SIAM, 839–847.

[30] Balaji Prasad. 2016. Cloak and Dagger: Unpacking Hidden Malware Attacks. https://www.symantec.com/blogs/expert-perspectives/unpacking-hidden-malware-attacks Access: Oct. 2020.

[31] Moustafa Saleh, E Paul Ratazzi, and Shouhuai Xu. 2017. A control flow graph-based signature for packer identification. In *MILCOM 2017-2017 IEEE Military Communications Conference (MILCOM)*. IEEE, 683–688.

[32] Mike Sconzo. 2015. I am packer and so can you. DEF CON. https://youtu.be/jCIT7rXX8y0 Access: Oct. 2020.

[33] Mike Sconzo. 2020. Packerid. https://github.com/sooshie/packerid. Access: Oct. 2020.

[34] Michael Sikorski and Andrew Honig. 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software* (1st ed.). No Starch Press, San Francisco, CA, USA.

[35] Li Sun, Steven Versteeg, Serdar Boztaş, and Trevor Yann. 2010. Pattern recognition techniques for the classification of malware packers. In *Australasian Conference on Information Security and Privacy*. Springer, 370–390.

[36] Chrysostomos Symvoulidis. 2020. An Incremental DBSCAN approach in Python for real-time monitoring data. https://github.com/csymvoul/Incremental_DBSCAN. Access: Oct. 2020.

[37] PN Tan, M Steinbach, and V Kumar. 2006. Chapter 8 Cluster analysis: basic concepts and algorithms. *Introduction to data mining, 6th edn. Peason Addison Wesley, Boston* (2006), 486–568.

[38] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. 2015. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 659–673.

[39] Nguyen Xuan Vinh, Julien Epps, and James Bailey. 2010. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *Journal of Machine Learning Research* 11, Oct (2010), 2837–2854.

[40] Jixin Zhang, Kehuan Zhang, Zheng Qin, Hui Yin, and Qixin Wu. 2018. Sensitive system calls based packed malware variants detection using principal component initialized MultiLayers neural networks. *Cybersecurity* 1, 1 (2018), 10.