PackHero: A Scalable Graph-based Approach for Efficient Packer Identification

Marco Di Gennaro, Mario D'Onghia, Mario Polino, Stefano Zanero, and Michele Carminati

Dipartimento di Elettronica, Informazione e Bioingegneria,
Politecnico di Milano, Milan, Italy
{marco.digennaro,mario.donghia,mario.polino,
stefano.zanero,michele.carminati}@polimi.it

Abstract. Anti-analysis techniques, particularly packing, challenge malware analysts, making packer identification fundamental. Existing packer identifiers have significant limitations: signature-based methods lack flexibility and struggle against dynamic evasion, while Machine Learning approaches require extensive training data, limiting scalability and adaptability. Consequently, achieving accurate and adaptable packer identification remains an open problem. This paper presents PackHero, a scalable and efficient methodology for identifying packers using a novel static approach. PackHero employs a Graph Matching Network and clustering to match and group Call Graphs from programs packed with known packers. We evaluate our approach on a public dataset of malware and benign samples packed with various packers, demonstrating its effectiveness and scalability across varying sample sizes. PackHero achieves a macro-average F1-score of 93.7% with just 10 samples per packer, improving to 98.3% with 100 samples. Notably, PackHero requires fewer samples to achieve stable performance compared to other Machine Learning-based tools. Overall, PackHero matches the performance of State-of-the-art signature-based tools, outperforming them in handling Virtualization-based packers such as Themida/Winlicense, with a recall of 100%.

Keywords: Packer Identification · Graph Similarity · Graph ML.

1 Introduction

Code packing, a widely used *anti-analysis technique* [10], affects the performance of both Machine Learning (ML)-based and traditional signature-based malware detection systems [22]. Packers encrypt or compress executable code, rendering static analysis ineffective [34]. At runtime, an *unpacking stub* embedded in the executable restores the original code by decrypting or decompressing it in memory, allowing the program to execute. This ability to bypass static analysis makes packing particularly appealing to malware authors. The prevalence of *packed* malware can bias ML-based detectors into flagging all packed executables as malicious [1]. However, this assumption is flawed, as many benign programs

(*goodware*) are also packed to protect intellectual property. For instance, Rahbarinia et al. [31] found that 54% of benign programs and 58% of malware samples in their study were processed with known packers, demonstrating a similar distribution.

Recovering packed code might seem feasible through *dynamic analysis*, as executing a packed program can trigger it to unpack itself. However, modern malware increasingly employs dynamic *evasive behaviors* designed to detect analysis environments and prevent the program from exposing its true functionality at runtime [10]. As a result, packed malware with evasive tactics often resists dynamic-based analysis techniques. Additionally, incorporating dynamic analysis mechanisms into commercial AVs presents challenges, such as requiring kernel-level privileges to execute untrusted code [9,1] and introducing significant computational overhead due to the virtualization infrastructure [22]. Alternatively, *static* identification of the specific packer used in a malware sample could allow AVs to retrieve the original code, if possible, by executing a corresponding unpacker when available. Previous works in this area have applied signature-based methods [13] or ML-based algorithms using static features [15,19,33]. While effective for known packers, these approaches demand substantial effort to accommodate new packers or variations of existing ones. This challenge is amplified by the frequent emergence of *custom packers* in novel malware [33], necessitating either extensive manual signature analysis (e.g., with *Detect It Easy* [13]) or complete re-training of ML-based models. Recent work proposed *PackGenome*, a tool that automates YARA rule generation from packed samples to detect packed binaries [16]. While effective on large and heterogeneous datasets, it relies on dynamic analysis, requiring packers to generate custom-packed samples, limiting the integration of newly discovered packers. These limitations motivated our research into new methodologies for code packer identification, focusing on minimizing packer integration effort. The primary challenge lies in achieving a balance between *accuracy*, rapid *adaptability* for integrating newly discovered packers, resilience against dynamic *evasion* techniques, and overall *scalability*.

This paper presents PackHero, a packer identifier that leverages statically extracted Call Graphs from packed programs. PackHero extracts the CG of a given binary and identifies the packer by comparing it with previously labeled graphs in a stored collection. The graph representation is inspired by the work of X. Li et al. [17]. CGs enable a high level of abstraction and reveal that portions of these graphs remain identical or similar for binaries packed with the same packer. To leverage this, we introduce a heuristic to isolate the graph segment corresponding to the unpacking stub, identifying unique patterns shared by binaries processed by the same packer. To solve the graph similarity problem, we use a specific Graph Neural Network (GNN) [11], known as a Graph Matching Network (GMN) [18]. Additionally, PackHero incorporates a hierarchical clustering approach to group similar graphs, enhancing identification accuracy while ensuring constant inference time when integrating new packers.

We evaluate PackHero on a publicly available dataset of packed Windows Portable Executables (PEs) [1], containing both malware and benign samples,

repacked with various commercial and free packers, categorized by complexity according to existing taxonomies [34]. PackHero achieves a macro-average F1-score of 93.7% and an accuracy of 98.7% using only 10 programs per packer during configuration. In its best configuration, utilizing 100 samples per packer, it reaches a macro-average F1-score of 98.3% and an accuracy of 99.8%. The scalability of PackHero, supported by its clustering approach, is validated through comparisons with a non-clustering version in terms of both performance and inference calls to the GMN. Once configured, PackHero requires significantly fewer samples to converge and stabilize than existing ML-based tools, needing just 10 samples versus 40 for the best-performing alternative. Moreover, PackHero features a constant integration cost, whereas the integration cost of other ML-based tools increases linearly with the number of packers recognized. PackHero is a robust alternative to signature-based detection tools, achieving performance aligned with SotA tools. Notably, it performs significantly better against virtualization-based [32] packers like Themida/Winlicense, which employ advanced dynamic evasive behaviors that hinder signature extraction in dynamic analysis-based tools. Specifically, PackHero achieves a perfect recall of 100% on this packer, compared to 92% for DIE and 31% for PackGenome.

In summary, the contributions are the following :

- A hybrid ML and graph signature-based approach for packer identification, enabling automatic and scalable integration of both accessible and non-accessible packers (e.g., custom or closed-source) directly from packed programs throughout the tool lifecycle.
- A heuristic to statically extract a Call Graph of an unpacking routine or a part of it from a packed binary.
- A combination of Graph Matching Network (GMN) with hierarchical clustering to enhance the accuracy and reduce the search space of similar graphs.
- We release PackHero's source code[1] for reproducibility.

2 Background and Motivation

Code packing is a widely used anti-analysis technique [24], where packers, acting as third-party software, transform a program's structure and content, recovering the original software at runtime via a *tail jump* to the original entry point. Initially intended for file compression, most packers now aim to obfuscate and hinder program analysis in legitimate and malicious software. Packers are classified by runtime complexity [34] into six types (I–VI), with most common packers falling within types I–III. Another taxonomy focuses on obfuscation methods, distinguishing *compressors*, *crypters*, and *virtualization-based* packers, such as Themida [32], which translate code into virtual instructions and implement advanced anti-dynamic analysis techniques. In our experiments, we consider packers from types I–III, with Themida representing the VM-based category.

[1] `https://github.com/necst/packhero`

2.1 Packer Identification

Packer identification is a multi-label classification task aimed at determining the specific packer used to compress or obfuscate a program. This capability allows AV tools to statically unpack programs, thereby enhancing malware detection [22]. In contrast, *packer detection* identifies whether a program is packed, often employing static methods such as similarity comparisons [14,28] or entropy analysis [12,2]. However, these methods are less effective against *low-entropy* packers [22]. This paper focuses on packer identification and categorizes existing approaches into two main families: signature-based methods, which rely on manually or automatically generated signatures, and pattern recognition techniques, predominantly driven by ML-based algorithms.

Signature-based Methods. Packers often leave specific artifacts that can be used to create signature databases. *Detect It Easy (DIE)* is a well-known tool for packer identification via signature matching [13], outperforming tools like PEiD [27] with its open architecture that allows users to add JavaScript-like scripts for packer detection. However, it requires the manual creation of signature scripts for new packers and their variants, making it challenging to integrate new packers, especially with limited analyzable packed samples. A key limitation of signature-based detection is the need to analyze many samples to identify invariant byte sequences that can be used as signatures. To address this, researchers have explored automating the signature extraction process. Raff et al. propose a method to automatically generate YARA rules [30], a format for defining malware characteristics [3]. Nevertheless, code packing can still easily defeat these rules, similar to other signature schemes. To the best of our knowledge, the State-of-the-art (SotA) tool for signature generation in packed programs is PackGenome [16]. Inspired by biological processes, PackGenome identifies significant instructions in the first unpacking layer (the only statically visible one). It uses Intel *Pintool* [21] to monitor packed programs in a controlled environment, recording and labeling instructions that write "unpacked" instructions. By analyzing multiple executions of programs packed with the same packer and applying similarity metrics, PackGenome extracts *packer-specific* "genes" to generate YARA rules. However, this approach relies on dynamic analysis, making it vulnerable to evasive techniques that hinder the extraction of relevant genes, as empirically confirmed in our experimental evaluation (Subsection 4.5). Accurate packer identification often requires generating a large number of signatures. For instance, PackGenome recommends using the actual packer to create extensive variations of the unpacking stub. However, this approach is impractical in real-world scenarios where malicious software frequently employs custom packers that are inaccessible to analysts. Additionally, the limited availability of samples for such packers makes it infeasible to build a comprehensive signature database.

ML-based Packer Identification. The second family of identification methods relies on pattern recognition algorithms, particularly Machine Learning techniques. Proposed approaches include constructing randomness profiles for packed samples [33], applying binary diffing [15], extracting features from the topology of CGs [19], and evaluating the similarity of Consistently-Executing Graphs

Table 1: Packer identifiers and compliance with requirements R1–R4.

Work	Analysis Type	Approach	Code Replicable		R1	R2	R3	R4
PackGenome [16]	Static + Dynamic	Signature	✓	✓	✓	✓		✓
Detect It Easy (DIE) [13]	Static	Signature	✓	✓	✓		✓	✓
Randomness Profiles [33]	Static	ML		✓	✓	✓	✓	
Binary Diffing [15]	Static	ML		✓	✓	✓	✓	
2SPIFF [19]	Static	ML		✓	✓	✓	✓	
CEG [17]	Static	Signature			✓	✓	✓	
PackHero (our approach)	Static	ML + Signature	✓	✓	✓	✓	✓	✓

(CEGs) [17]. S. Li et al. observed that while most packers significantly affect binary entropy, individual packers exhibit distinctive randomness patterns [33]. They used sliding windows [8] to build randomness profiles, training a k-nearest neighbor classifier. Kim et al. employed an SVM classifier with binary diffing measures as kernels, achieving the best performance using the *longest common substring* computed from the first 15 bytes at each program's entry point [15]. This method leverages the similarity of initial instructions in unpacking stubs from the same packer, but its effectiveness is lowered by code obfuscation [16,37]. Hao et al. represented packed programs as CGs and trained an SVM classifier using topological features (e.g., entry point indegree) and general file information like size or section count [19]. While we draw on this idea to represent packed programs through CGs, our methodology directly uses graphs, offering better generalization and results. X. Li et al. [17] proposed a similar approach by comparing graphs using a Weisfeler-Lehman shortest path kernel. Instead of employing CGs, they introduce CEGs to simulate static execution points by traversing procedures, locating branch instructions, and forming flow paths. However, they do not address the challenge posed by the increasing number of graphs that must be compared during identification due to the introduction of new packers. To the best of our knowledge, none of these works have released their code. However, three of the four approaches were straightforward to implement, enabling their comparison with our method (results in Subsection 4.4). The fourth, CEG, relies on a heuristic for graph extraction, making reimplementation challenging without significant assumptions. Therefore, it was excluded from our study.

2.2 Motivation

Given the limitations of current works, we define key requirements for a novel packer identifier. Table 1 shows that existing SotA approaches only partially meet these requirements, highlighting the need for our solution.

Requirement 1 - High Identification Accuracy. It must achieve high accuracy and low false positives across diverse packer types.

Requirement 2 - Efficient Packer Integration. It must efficiently integrate packers using a limited number of real-world samples, addressing challenges such as inaccessible packers and variations within a single packer family. It should rapidly update its identification capabilities without requiring extensive or frequent retraining. To achieve this, the system should leverage robust algorithms

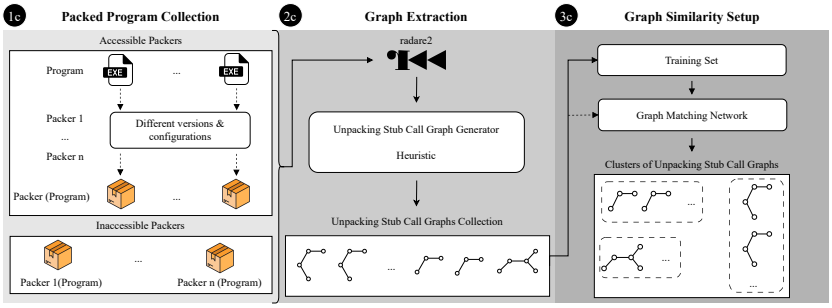

Fig. 1: PackHero Configuration Workflow.

that generalize from existing data, enabling the detection of new variations by integrating them using limited samples with minimal manual intervention.

Requirement 3 - Dynamic Evasive Behavior Management. It must effectively handle evasive behaviors encountered during the collection of wild samples for integration. Wild samples may employ dynamic evasion techniques or be so damaged or corrupted that execution is impossible [35], thereby complicating dynamic signature extraction performed by state-of-the-art tools [16]. To address this challenge, a static analysis-based approach can mitigate these issues by extracting valuable information and artifacts independently of execution.

Requirement 4 - Scalability. It must handle a growing number of packers without performance degradation, integrating new ones seamlessly without major architectural changes or resource demands. In other words, it must handle many/several packer families in parallel.

3 PackHero

PackHero is a *packer identifier* that determines the specific packer used for a given *packed* program. Its approach mirrors the workflow of signature-based detection mechanisms but uses graph "signature" to represent packed programs, with matches determined by similarity rather than exact matching. PackHero leverages a specialized Graph Neural Network (GNN) called Graph Matching Network (GMN) [18]. It operates on *Call Graphs (CGs)* extracted using heuristics. CGs, which represent the invocation relationships between functions in an executable program [6], are chosen for their compact structure and high level of abstraction. Compared to other binary graph representations (e.g., Control-Flow Graph (CFG) and Data Dependence Graph (DDG)), CGs enable an efficient resolution of the similarity problem with GMNs. We divide our approach into two main phases: *configuration* and *inference*.

3.1 Configuration

As depicted in Fig. 1, this phase involves three main step.

1c Collecting Packed Programs. The first step consists of collecting programs for the packers we want to integrate into the tool. It is important to distinguish between an *accessible* and *non-accessible* packer. The former enables the use of the actual code packer to generate packed samples, including all possible versions and configurations. This case is, therefore, ideal. Hence, we consider the "non-accessible packers" scenario as the general case.

2c Graphs Extraction. PackHero extracts a Call Graph for each collected program. Our implementation relies on radare2 [29] to analyze and extract the CGs. Each vertex of a CG consists of 12 features extracted using radare2 (shown in Table 2). Furthermore, a heuristic designed to filter the unpacking stub part of the CG is applied to simplify the topology of each graph (details in Subsection 3.3). PackHero collects the generated CGs into a Database (DB) of graphs.

3c Graph Matching Network Training. PackHero identifies intrinsic similarities between extracted CGs using a Graph Matching Network (GMN) [18], a specialized Graph Neural Network (GNN). The GMN processes pairs of graphs and outputs a numeric vector (*embedding*) for each graph. These embeddings result from information propagation between the two graphs, differing from traditional embedding techniques [11] that compute embeddings solely from individual graphs. To train the GMN, we label graph pairs as "similar" if they originate from the same packer and "dissimilar" otherwise. The network is trained to minimize the distance between embeddings of similar graph pairs while maximizing the distance for dissimilar pairs. The loss function is defined as $L(G_1, G_2) = \mathbb{E}_{(G_1,G_2,l)}\left[\max\{0, \gamma - l(1 - \cos(G_1, G_2))\}\right]$, where $l \in \{-1, 1\}$ is the label associated with the pair of graphs $< G_1, G_2 >$,γ is a margin parameter and *cos* is the cosine similarity [38] between the two graphs. In this case, the *cos* is intended as the cosine similarity between the two embeddings of size 256 extracted from the two graphs via the GMN. Lastly, $\mathbb{E}$ is the empirical risk we want to minimize, which can be done through stochastic gradient descent. The overall GMN design follows the original implementation of the paper that introduced it [18]. Finally, we propose a clustering approach to stabilize the number of matches required to identify each packer and improve the overall performance of the framework. Therefore, we also store the clusters and their respective medoids. Finally, we compute a cluster-specific threshold $t_c = \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} cos(G_i, G_j) - \sigma$. Namely,

Table 2: Node features.

Feature	Description
type	Whether it is an internal function, a library imported function, or an entry point containing function.
size	The size of the function in bytes.
real size	The function size in bytes, including any padding.
is pure	Indicates whether the function has any side effects such as modifying external variables or writing to files.
calling conventions	The number of calling conventions used by the function.
number of basic blocks	The number of basic blocks in the function.
number of instructions	The total number of instructions in the function.
number of local variables	The number of local variables declared within the function.
number of arguments	The number of arguments of the function.
edges	The number of edges between basic blocks.
indegree	The in-degree of the function in the call graph.
outdegree	The in-degree of the function in the call graph.

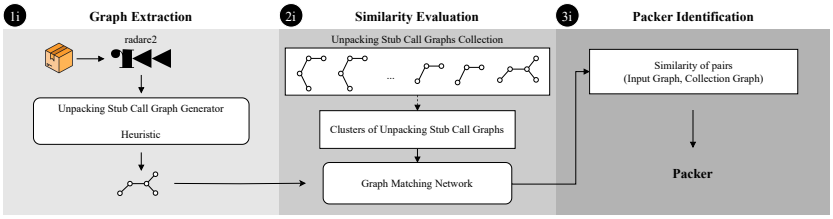

Fig. 2: PackHero Inference Workflow.

the average cosine distance between pairs of graphs belonging to the cluster minus the standard deviation. Such a threshold will then be used at inference time.

3.2 Inference

This phase comprises three steps, depicted in Fig. 2.

1i Graph Extraction. PackHero must first obtain the CG specific to the unpacking stub extracted through the previously mentioned heuristic.

2i Similarity Evaluation. The second step consists of evaluating the similarity between the embeddings computed by the GMN for the input graph and the graphs in the DB. Comparing the input graph against all graphs in the DB may be computationally expensive and decrease the general identification performance. Hence, PackHero computes the cosine similarity between the input graph and the medoids associated with each computed cluster to select the "closer" clusters. In other words, each PackHero identification corresponds at least to m GMN inferences, where m is the number of clusters. PackHero selects clusters represented by medoids with a positive cosine similarity with the input graph. Once the clusters are selected, PackHero evaluates the similarity between the input graph and each graph contained in the selected clusters.

3i Packer Identification. Now, up to m clusters are identified as potential matches, and the similarity between the input graph and all graphs within these m clusters is computed. PackHero identifies the packer with the highest score $s_p := \frac{\sum_{C \in \mathcal{C}_p} \sum_{G_c \in C} \mathbf{1}(\cos(G_{\text{input}}, G_c) \geq t_c)}{\max_{p \in \mathcal{P}} \sum_{C \in \mathcal{C}_p} |C|}$, where G_{input} is the graph extracted from the input program, C_p the set of selected clusters for a packer p, G_c indicates a graph in cluster C, and t_c the threshold for cluster C. $\mathbf{1}(\cos(G_{\text{input}}, G_c) > t_c)$ is a membership function that outputs 1 if the cosine similarity $\cos(G_{\text{input}}, G_c)$ is greater or equal to the threshold t_c, and 0 otherwise. Moreover, $\mathcal{P}$ is the set of all included packers in the selected clusters. Lastly, $\sum_{C \in \mathcal{C}_p} |C|$ is the cardinality of samples in the selected clusters from a packer p. If no cluster is sufficiently "close" to the input CG, PackHero labels the packer as "unknown".

3.3 Extracting the CG of the Unpacking Stub

The Call Graph (CG) is a widely adopted structure [23]. It is also used in security-related tasks such as malware detection [20]. Our approach is based on a principle of "same packer, similar CGs" [19]. To the best of our knowledge,

Algorithm 1 Extract Unpacking Stub Call Graph

```
1: F: Set of all functions with call references, E: Set of entry points in the binary
2: procedure UnpackingStubCG(F, E)
3:     G ← directed global call graph from F, C ← ∅
4:     for e ∈ E do
5:         if G.hasEdges(e) then
6:             C ← C ∪ getConnectedComponent(G, e)
7:         end if
8:     end for
9:     G ← C ≠ ∅ ? C : getComponent(G, E) ∪ E
10:    if G.isEmpty() then
11:        G ← E ∪ externalLibraries()
12:    end if
13:    return G
14: end procedure
```

we are the first to exploit this similarity directly. The adoption of CGs offers several advantages. Their structure is straightforward to obtain [6]. Moreover, unlike other binary graph representations, a CG represents the program at a higher level of abstraction. This makes it a compact yet information-rich program representation, which is particularly well-suited for a GMN [18].

To better represent the logic behind a packer, it is necessary to filter the graph to get the unpacking stub. To systematically obtain this filtered CG, we design a heuristic shown in Algorithm 1. The intuitions behind it are: (i) the unpacking stub, or part of it (case of a multilayer packer), must be the first part of the code to be executed, and (ii) except for further obfuscation of the unpacking stub, a part of this routine is always statically visible. Therefore, the heuristic extracts the unpacking stub by exploiting the concept of connected components in undirected graphs, i.e., a subgraph where each pair of nodes is connected via a path [7]. Notice that CGs are directed graphs, but the algorithm requires undirected ones, thereby we convert the CGs into undirected graphs. Given the packer could disrupt links between functions, it should create multiple connected components in the CG. Thus, the idea is to extract the connected component containing the program entry point. At the same time, some packers affect the program entry point to make the analysis harder. For instance, analyzing Call Graphs extracted from binaries packed with ASPack [5], we noticed the common part among all the graphs was a second connected component in addition to the single entry function node, which appears to be isolated. Thus, when the entry function is not connected to any other node, a second connected component is maintained in the graph along with the entry function. Otherwise, if the graphs have no edges (UPX [36]), we keep only the program entry functions and any functions from external libraries. As the heuristic suggests, we do not consider a fixed number of functions for each graph. In our experimental evaluation, the average number of functions in the unpacking stubs is ≈ 3.

3.4 Graphs Clustering

Integrating a new packer requires collecting additional graphs. Without clustering, identifying a packer involves matching against *all* graphs in the DB, increasing inference time as new packers are added. To ensure scalability, we

Algorithm 2 Packer Call Graphs Clustering

```
DB: a collection of unpacking stub call graphs, M: GMN trained model P: Mapping from graphs to their respective packers
procedure GetClustering(DB, M, P)
    C ← ∅                                                    ▷ Initialize clustering result
    for each unique packer p ∈ P do
        DB_p ← {G ∈ DB | P(G) = p}
        D ← initialize empty distance matrix for DB_p
        for G_i, G_j ∈ DB_p, i ≠ j do
            d ← cosineSimilarity(M(G_i, G_j))
            D.update(d)                                      ▷ Update distance matrix with similarity
        end for
        C_p ← hierarchicalClustering(D)
        for C_j ∈ C_p do
            C_j.representative ← medoid(C_j)
        end for
        C.update(C_p)
    end for
    return C
end procedure
```

introduce a clustering approach that reduces inference time and improves identification performance, as demonstrated in our Experimental Validation. Each cluster contains graphs from only a single packer, allowing the identification of potential sub-groups within the same packer. This *packer unicity* is ensured by constructing the distance matrix in an intra-packer manner, as expressed in Algorithm 2. This approach can mitigate variations in unpacking stubs due to different configurations or versions of the same packer [16]. PackHero employs hierarchical clustering with a single linkage merge criterion, using a distance matrix derived from the trained GMN as input. The silhouette score [38] determines the optimal number of flat clusters for each packer. Finally, PackHero computes a medoid for each cluster, representing the graph with the minimal sum of dissimilarities to all other graphs in the cluster [38].

4 Experimental Validation

We evaluate PackHero through the following four research questions:

RQ1. What is the minimum number of programs required for PackHero's configuration to recognize packers effectively? In addition, once configured, is PackHero able to recognize different packers?

RQ2. How does the clustering-based approach impact PackHero's performance and scalability compared to its non-clustering version?

RQ3. Given an already configured PackHero, how many samples does it require to successfully integrate a new, unseen packer, and how does this integration compare to other ML-based tools?

RQ4. Does PackHero perform better than signature-based tools?

4.1 Experimental Setup

Dataset. We use the *lab* dataset from H. Aghakhani et al.[1], created by repacking Portable Executables (PEs) from benign and malicious Windows x86 soft-

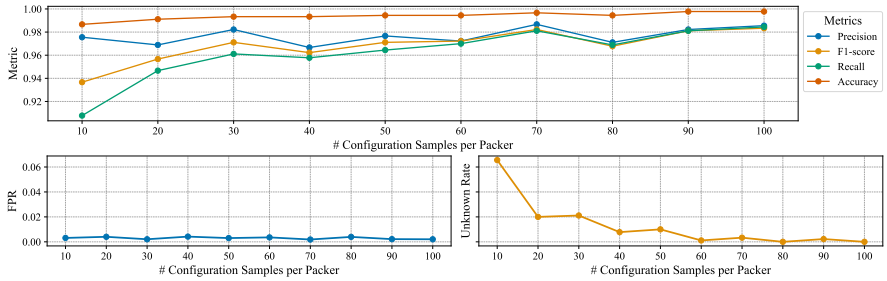

Fig. 3: Metrics trends varying the number of programs to configure PackHero.

ware collected from a commercial anti-malware vendor and the EMBER dataset[4]. The samples were repacked using nine widely recognized packers: kkrunchy, MPRESS, Obsidium, PECompact, PELock, Petite, tElock, Themida, and UPX. The dataset includes different packer families. Following a SotA taxonomy [34], it covers Type-I (e.g., UPX), Type-III (e.g., PECompact), and VM-based packers (e.g., Themida). To replicate Experiment II from the original work, we apply the same undersampling strategy, resulting in 15,353 samples per packer. We randomly select 10% of the undersampled dataset while preserving the distribution of malware, benign programs, and packers. This subset, referred to as the *lab-10* dataset, excludes 10 outliers (CGs with more than 500 nodes). In Section 4.2, we test PackHero with the *RGD* dataset from PackGenome [16], which consists of three manually constructed programs, compiled from 2-5 lines of C code and packed with several versions and configurations of 20 off-the-shelf packers. To assess transferability, we select only the *RGD* packers also present in *lab-10*.

Hyperparameter Tuning. We optimize the hyperparameters of the GMN by maximizing intra-packet similarity using a grid search approach.

Evaluation Metrics. We evaluate PackHero using metrics [38] such as precision, recall, F1-score, accuracy, False Positive Rate (FPR), and the unknown rate, which indicates how often PackHero fails to recognize a packer. We also measure the *Average Number of Inference Calls*, representing the average calls PackHero makes to the GMN during identification.

4.2 Effective Identification (RQ1)

Configuration. PackHero requires a configuration phase starting with the step of collecting programs. If the packer is accessible, i.e., we can use it to craft new samples, the configuration becomes trivial and we can obtain as many programs as we need. However, if the packer is not accessible, we need to collect packed programs in the wild. These programs can be malware or benign, and it is uncertain whether we can find them in large quantities. Therefore, we test PackHero in a scenario with a limited number of programs, defining the number of programs needed for each packer to achieve a good average performance. From the *lab-10*, we select 100 programs for each packer to configure PackHero, maintaining the original distribution of malware and benign programs. We use

the remaining programs for testing. Starting from the 100 programs for each packer, we gradually eliminate 10 programs and create 10 different collections of gradually smaller sizes. We use these 10 collections to configure PackHero. Then, we test our approach, configured with different "training" sizes, using the same test dataset. The metrics we use to evaluate PackHero in this experiment are precision, recall, F1-score, accuracy, FPR, and the unknown rate. Each plot in Fig. 3 shows the macro-average results, i.e., the average results among the 9 packers in the dataset. Looking at the precision, F1-score, recall, and accuracy, the tool does not perform badly even with only 10 programs per packer. Furthermore, starting from 30 samples per packer, PackHero maintains all metrics above 0.96. In addition, the plot shows precision and recall converge in the long run. We also notice that from the configuration of 70 samples for packers, PackHero achieves a good balance between precision and recall, which means a good tradeoff between False Positives (FPs) and False Negatives (FNs). As regards the FPR and the unknown rate trends, they follow all the other metrics. The FPR is overall low and always below 0.00418, i.e., 0.41% of FPs. We also observe higher unknown rates for lower "training sizes", which means PackHero becomes more confident in his choices as the "training" size increases. The plot fluctuations are because the experiment was done with a single run due to the computational cost of training and testing with 10 different configurations. However, a single run places PackHero in a realistic scenario with limited samples and no ability to select the most suitable ones for tool configuration.

Effectiveness on different packers. Here, we zoom in on the results obtained from the best configuration in the previous experiment, namely the one with 100 samples for packers. The test set is the remaining part of the dataset. Table 3 shows that PackHero performs very well on each packer in the dataset. We have a near-maximum accuracy in general and equally good results in all other metrics for other packers. The only exception is tElock, which is found to have a higher FPR than the others. This result has a chain effect on precision and F1-score. An answer can be found by looking at the clusters' composition. In particular, tElock produces two clusters of 1 and 99 samples. In its current version, PackHero merges single-sample clusters with the nearest one. As a result, for tElock, we obtain a single cluster consisting mostly of similar samples, along with one slightly different sample, which lowers intra-cluster similarity. This leads to a reduced threshold (≈ 0.10 lower than others) and less confidence in

Table 3: PackHero performance on lab-10. UR denotes the Unknown Rate.

Packer	#Samples	UR	FPR	Prec	Rec	F1	Acc
kkrunchy	1435	0.0	0.0001	1.00	0.99	1.00	1.00
MPRESS	1435	0.0	0.0018	0.99	0.98	0.98	1.00
Obsidium	1435	0.0	0.0031	0.98	0.98	0.98	0.99
PECompact	1434	0.0	0.0012	0.99	1.00	0.99	1.00
PELock	1435	0.0	0.0002	1.00	0.96	0.98	1.00
Petite	1434	0.0	0.0001	1.00	0.99	0.99	1.00
tElock	1432	0.0	0.0107	0.92	0.98	0.95	0.99
Themida	1433	0.0	0.0007	0.99	1.00	0.99	1.00
UPX	1432	0.0	0.0006	1.00	0.98	0.99	1.00
macro-avg	-	**0.0**	**0.0021**	**0.986**	**0.984**	**0.983**	**0.998**

the choice. This observation suggests that treating single-element clusters as outliers and excluding them could enhance PackHero's performance.

Different versions and configurations. The *lab-10* dataset includes only one configuration and version per packer. To evaluate transferability, we test PackHero's best configuration (*lab-10*) on the *RGD* dataset [16], which contains multiple versions and configurations for each packer—except for tElock, which is not included in the PackGenome evaluation. Table 4 presents the configurations identified by PackHero for each version in *RGD*. An "identified configuration" occurs when PackHero recognizes all samples, while non-identified configurations show a 0% identification rate, likely due to differences in the unpacking stub. Overall, PackHero generalizes across 16 out of 19 different versions, despite being configured with only a single version and configuration per packer.

Answer to RQ1. The number of packed programs required to configure PackHero depends on the desired performance level. With just 10 samples per packer, PackHero achieves a minimum macro-average F1 score of 93.7% and accuracy of 98.7%. Increasing the sample size to 30 can further improve recall and F1-score while maintaining high precision and accuracy. Table 3 demonstrates PackHero's ability to effectively identify multiple packers from different families. Given the dataset's composition, PackHero successfully integrates and recognizes packers of varying complexity in both packed malware and benign programs. Specifically, based on the taxonomy by Ugarte-Pedrero et al. [34], PackHero performs well on Type-I (UPX), VM-based (Themida), and Type-III (PECompact) packers.

4.3 Clustering Effectiveness (RQ2)

We evaluate the impact of the clustering approach on PackHero's performance and scalability. To do this, we replicate the experiment from Section 4.2 without the clustering layer: PackHero evaluates similarity with all graphs in the DB and computes packer-specific thresholds instead of cluster-specific ones. Fig. 4 illustrates the performance gap between PackHero with and without clustering. This gap is more pronounced for smaller training set sizes and narrows as the training set size increases. Without clustering, the unknown rate consistently drops to 0, as the *unknown* classification (explained in Section 3) depends on the similarity step involving clusters' medoids. However, the absence of clustering increases False Positives. These results demonstrate that incorporating clustering

Table 4: PackHero performance on RGD. A configuration is considered identified if all samples from that configuration are correctly classified.

Packer	**Versions (#Identified Configurations/#Configurations)**
kkrunchy	v0.23alpha **(0/2)**, v0.23alpha2 **(1/1)**
MPRESS	v1.27 **(1/1)**, v2.18 **(1/1)**, v2.19 **(1/1)**
Obsidium	v1.5 **(7/7)**
PEcompact	v3.02.2 **(18/19)**, v3.11 **(10/12)**
PElock	v1.06 **(0/5)**
Petite	v2.4 **(5/5)**
Themida	v2.37 **(8/9)**, v2.39 (*Winlicense*) **(8/9)** v3.04 **(0/5)**
UPX	v1.00 **(4/4)**, v1.20 **(8/8)**, v1.25 **(4/4)**, v2.00 **(5/5)**, v3.09 **(8/8)**, v3.96 **(8/8)**

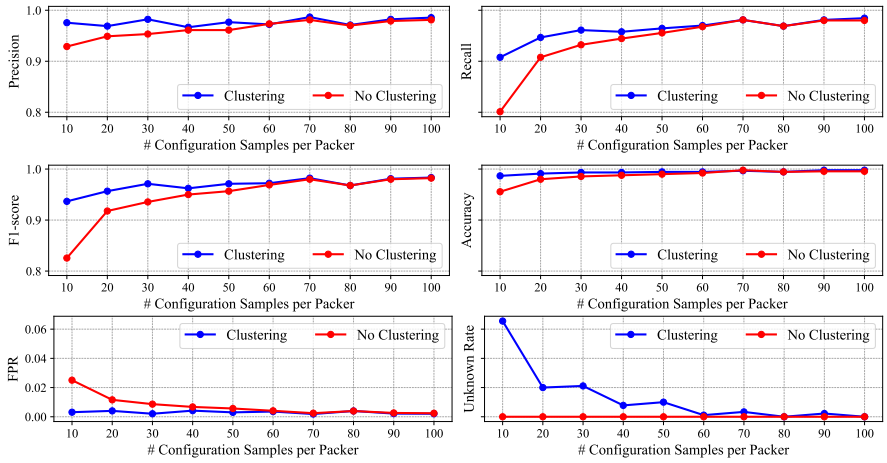

Fig. 4: Comparison between PackHero with and without clustering.

into PackHero's workflow is effective and that the chosen medoids are good representatives of subgroups within the same packer.

Table 5 shows the average number of inference calls to the GMN during the identification, i.e., the average number of matched graphs needed to identify a packer from a program. Removing the clustering approach, the similarity with the input graph is evaluated with the entire collection. Thus, the metric is always equal to the size of the entire collection. In contrast, the values we empirically obtain with the use of the clustering approach show are close to the ideal number of inference calls to identify the packer. The ideal number of inference calls in the presence of the clustering approach is represented by the sum of m (number of clusters) and the number of programs per packer stored in the DB. Thus, we can state that, involving the clustering approach, the PackHero's number of inference calls does not depend on the number of packers but only on the number of samples for each packer stored in the DB. To further emphasize the significance of the results shown in Table 5, it is important to consider the inference time for our GMN. Indeed, while this network's expressive power surpasses that of its alternatives, it comes with the trade-off of increased temporal complexity. In this experiment, the average single inference time recorded was $1.76ms$. The

Table 5: Average number of inference calls made by PackHero to the Graph Matching Network (GMN) during packer identification of a single sample.

Configuration	**#Clusters**	**Clustering**		**No Clustering**
		Ideal Values	**Real Values**	
10	14	24.00	24.14 ± 1.33	90.00 ± 0.00
20	13	33.00	33.02 ± 0.17	180.00 ± 0.00
30	17	47.00	47.03 ± 1.19	270.00 ± 0.00
40	16	56.00	60.59 ± 4.68	360.00 ± 0.00
50	13	63.00	64.31 ± 2.65	450.00 ± 0.00
60	14	74.00	82.68 ± 7.48	540.00 ± 0.00
70	11	81.00	83.71 ± 3.84	630.00 ± 0.00
80	16	96.00	107.54 ± 13.35	720.00 ± 0.00
90	13	103.00	107.30 ± 3.57	810.00 ± 0.00
100	17	117.00	124.47 ± 7.78	900.00 ± 0.00

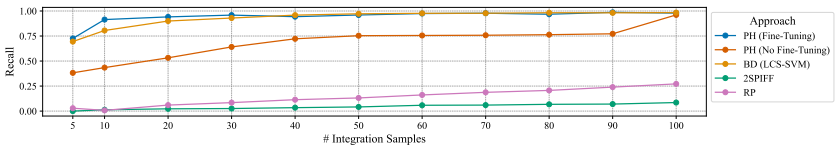

Fig. 5: Recall trend increasing the number of integration samples; PackHero (Fine-Tuned and Not) vs ML-based Tools.

time difference observed in the tests conducted on 9 packers is not substantial between the version with and without clustering. However, in a scenario with 200 packers and 100 samples in the DB for each packer, the time required for identification would be $\approx 35s$ without clustering but only $21ms$ with it.

Answer to RQ2. The clustering approach improves PackHero's performance, especially with limited samples per packer, while its effectiveness remains unaffected by the number of recognized packers, ensuring scalability.

4.4 Integration of New Packers (RQ3)

As discussed in Subsection 4.2, to demonstrate the integration is always feasible we have to deal with a scenario in which we need to collect programs in the wild to integrate the new packer. Thus, we have to face the possibility that these programs are not numerous. We have already tested PackHero in a scenario with few samples for its configuration. Here, we aim to evaluate how many samples PackHero needs to "integrate" a new packer with the entire system already configured. The process of integration corresponds to the "configuration" of PackHero described in Subsection 1. Since the workflow includes a Graph Matching Network (GMN), we can avoid re-training the model from scratch. Indeed, given that we use a Neural Network (NN), it can be updated through fine-tuning, i.e., partially re-training on new samples. At the same time, PackHero may even allow us to avoid fine-tuning the model altogether. Specifically, a new packer can be integrated into PackHero without re-training the GMN, simply by adding its corresponding graphs to the DB. However, this approach is feasible only if the collected graphs for the packer are sufficiently homogeneous. Currently, we assume that manual intervention was previously performed on the packer samples to be integrated, which we assume are always correctly labeled.

We evaluate PackHero with and without fine-tuning the GMN. We train a GMN for each packer, excluding it from the training set, which consists of 100 samples per remaining packer. Then, we integrate samples from the "unseen" packer. In the version without fine-tuning, we add the new packer's graphs directly to the DB. In the fine-tuned version, we fine-tune the GMN using the new graphs before integration. To compare PackHero with SotA tools, we replicate the experiment using three ML-based approaches from Table 1: Randomness Profiles [33], Binary Diffing [15] (best-performing version: LCS-SVM), and 2SPIFF [19]. As implementations of these tools are unavailable, we reimplement them to the best of our ability and validate the implementations by comparing the achieved accuracy on the remaining packers in this experiment. For a

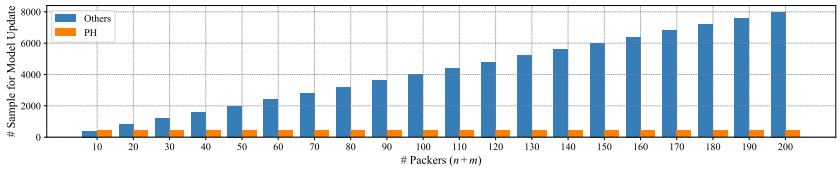

Fig. 6: # Samples involved in the integration ($m = 10$ new packers, $l = 40$ samples per packer, n already integrated packers).

fair evaluation, we train them using a stratified 80-20 split of the *lab-10* dataset. Therefore, for each ML-based tool, we train each model by excluding one packer, using 80% of the dataset. This subset includes the 100 samples per packer used to train PackHero. However, none of these approaches utilize a NN. Therefore, we are required to entirely re-train their models each time we want to integrate a new packer. Ultimately, both our method and the other approaches are tested using the test set from the 80-20 split, which corresponds to the stratified 20% of *lab-10*, equal to 307 samples for each packer. Results specific to each packer are obtained by testing the samples from that particular packer on the updated tools. By evaluating separately for each packer, the metric we use is recall, which is equivalent to accuracy in this experimental setup. In Fig. 5, we show the average recall trends for PackHero with and without fine-tuning and compare them against all other ML-based packer identifiers. As the figure shows, 2SPIFF and Randomness Profiles (RP) perform very badly, even in the best configuration. In contrast, both the fine-tuned PackHero and Binary Diffing (BD) with LCS-SVM achieve very good performance and consistency using a small number of samples. Starting from the 40 samples, their performance is aligned, except for small fluctuations due to the single run. However, as the plot shows, PackHero reaches a high recall before BD. Indeed, using 5, 10, 20, and 30 samples to integrate the new packer, PackHero performs better. Similarly, PackHero without fine-tuning also exhibits good performance, although not as good as the other two methods. However, the average performance hides the results for single-packers. Indeed, removing two packers out of nine (Obsidium and Petite) the non-fine-tuned PackHero achieves performance very close to the fine-tuned version and BD starting from 50 integration samples. This result shows that fine-tuning the GMN is unnecessary for low-heterogeneity graphs, saving computation.

Additionally, all ML-based approaches require complete retraining to integrate new packers, a scalability issue PackHero avoids. To demonstrate this, we simulate packer integration during the tool lifecycle. Assuming integration cost depends on the number of samples used, we define the cost function $f(n, m, l)$, where n is the number of known packers, m the new packers, and l the samples per packer. For other tools, $f(n, m, l) = (n + m) \cdot l$, while PackHero 's cost is $f(n, m, l) = m \cdot l$, as it only depends on new packers. Fig. 6 simulates $m = 10$ and $l = 40$, showing PackHero 's constant integration cost, unlike other tools, where cost grows linearly with the number of recognized packers. This result demonstrates that PackHero scales effectively in realistic scenarios where new packers must be integrated over time.

Answer to RQ3. PackHero achieves strong results in integrating a new "unseen" packer with as few as 10 samples. Among 9 tested packers, PackHero outperforms the three other selected ML-based tools for integration sample counts less than 30. Furthermore, a simulation comparing retraining and fine-tuning shows that PackHero 's integration cost function remains constant as new packers are added, unlike other ML-based approaches, whose integration costs grow linearly with the number of recognized packers.

4.5 Signature-Based Tool Comparison (RQ4)

In this experiment, we compare the performance of PackHero with State-of-the-art signature-based methods (Detect It Easy (DIE) and PackGenome). DIE [13] is currently the best-performing and most signature-rich packer identifier. Here, we use its latest available version (v3.10). PackGenome [16] is the best tool for automating the extraction of signatures for packer detection, generating YARA [3] rules that can be later used to identify packers. The authors of PackGenome have already compared their framework against DIE, but we include the results for both approaches for completeness. To compare the three frameworks, we use the False Positive Rate (FPR) and recall. We focus on recall because it is the most representative metric when comparing tools designed not just for identification but also for detection (as for DIE and PackGenome). We extract all results from the *lab-10* dataset, removing only the 100 samples for each packer used to configure PackHero. We test DIE using its command-line version, while we test PackGenome by loading the YARA rules provided in the original work. Since DIE does not include PELock signatures and PackGenome's authors did not perform experiments on PELock and tElock, we discard these two packers for this comparison. In Table 6, we show the comparison results. Starting with the recall, the three tools perform very similarly, although the two signature-based tools generally exhibit the highest recall. Attention must particularly be directed towards Themida, which poses significant challenges for both DIE and PackGenome, as depicted in the table. An interesting observation is that the matched signatures from DIE are related to the same version of Winlicense/Themida, specifically Themida with Trial/Licensing options [25]. Despite PackGenome including signatures for the same version of this packer, it shows a recall of 0.31. This result motivates the entire work. Indeed, Themida/Winlicense employs advanced (dynamic) evasive behaviors in its packed programs. Furthermore, it is the VM-based packer used during our evaluation. As explained in

Table 6: PackHero vs DIE vs PackGenome on lab-10.

		Recall			**FPR**		
	#Samples	**PH**	DIE	PG	**PH**	DIE	PG
kkrunchy	1435	0.99	1.0	1.0	0.0001	0.0	0.0
MPRESS	1435	0.98	1.0	1.0	0.0018	0.0008	0.0034
Obsidium	1435	0.98	1.0	1.0	0.0031	0.0002	0.0000
PECompact	1434	1.0	1.0	1.0	0.0012	0.0	0.0005
Petite	1434	0.99	0.99	0.99	0.0001	0.0	0.0
Themida	1433	1.0	0.92	0.31	0.0007	0.0	0.0499
UPX	1432	0.98	1.0	1.0	0.0006	0.0002	0.0011

Section 2.1, PackGenome extracts YARA rules by tracing instructions during their execution. Consequently, it is likely to struggle with the evasive behaviors introduced by Themida into the binary during the packing process. Additionally, PackGenome appears to face challenges due to the inherent nature of this packer. Indeed, the result suggests that both the signature itself and its automatic extraction encounter difficulties when dealing with this packer family. Finally, looking at FPR, PackGenome confirms its issues with Themida/Winlicense but shows in-line results for the other packers. PackHero demonstrates a low FPR on average, while DIE generates the fewest FPR.

Answer to RQ4. PackHero matches SotA signature-based tools in accuracy and significantly outperforms them on VM-based packers with advanced evasive behaviors like Themida/Winlicense, demonstrating our approach's effectiveness.

5 Limitations and Future Work

PackHero relies on heuristics to extract filtered CGs. Unpacking stubs play a crucial role in the analysis but other code segments might also contribute to the CG's structure. Even if this work demonstrates that a few statically visible functions are often sufficient to determine the packer's identity, the exact number of functions considered for each CG remains an open aspect. PackHero directly exploits disassembly and function identification, which are inherently challenging problems, especially in the context of malware due to obfuscation techniques, indirect branch resolution, and evasive behaviors. Furthermore, PackHero is potentially vulnerable to adversarial attacks that manipulate the CG to evade identification. An adversary could, for instance, obfuscate the CG by inserting bogus functions, modifying calls, or hiding call targets, significantly complicating packer identification. Additionally, different dynamic evasive behaviors implemented by malware could further impact the accuracy of the extracted CG. Hence, future work may study heuristics to resist adversarial attacks by evaluating CG obfuscation to identify which aspects of our heuristics and features are most susceptible to evasion. In our study, we selected radare2 as the disassembler due to its ease of use and integration. However, recent research has demonstrated that several other open-source disassemblers outperform radare2 performance [26]. This reliance, while currently effective, necessitates further investigation, particularly in scenarios involving adversarial manipulation of the unpacking stub or CG structure. Finally, PackHero currently focuses on *packer identification* but does not determine whether a sample is packed (*detection*). Preliminary analyses revealed a notable number of False Positives when analyzing non-packed samples, indicating the need for further improvements in this area. Therefore, future work will also address the *packer detection* problem.

6 Conclusion

This paper introduced PackHero, a packer identifier that leverages a heuristic to extract a Call Graph (CG) representing the unpacking routine of a program. Us-

ing a clustering approach to enhance performance and reduce the search space, PackHero evaluates the similarity between the extracted CG and labeled CGs stored in a DB, employing a Graph Matching Network (GMN) to compute these similarities and identify the packer. Evaluated on a public dataset of packed benign and malicious programs re-packed multiple times, PackHero meets all key requirements for a novel packer identifier: high accuracy, efficient packer integration, evasive behavior management, and scalability. Relying exclusively on static analysis, PackHero integrates new packers effectively, achieving strong performance with as few as 10 samples, while eliminating the limitations of dynamic analysis, particularly against dynamic evasive behaviors. For some packers, it avoids fine-tuning the GMN, and when fine-tuning is needed, it converges faster than other ML-based tools. Its integration cost remains constant throughout its lifecycle, unlike other methods, where costs grow linearly with the number of packers recognized. PackHero performs comparably to signature-based tools, the current best-performing solutions for packer identification, and significantly outperforms SotA approaches on Themida, a VM-based packer employing advanced dynamic evasive behaviors.

Acknowledgements. This work was partially supported by Project FARE (PNRR M4.C2.1.1 PRIN 2022, Cod. 202225BZJC, CUP D53D23008380006, Avviso D.D 104 02.02.2022) and Project SETA (PNRR M4.C2.1.1 PRIN 2022, Cod. P202233M9Z, CUP F53D23009120001, Avviso D.D 1409 14.09.2022) under the Italian NRRP MUR program, and by Project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan, all funded by the European Union - NextGenerationEU.

References

1. Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., Kruegel, C.: When Malware is Packin' Heat; Limits of Machine Learning Classifiers Based on Static Analysis Features. In: Proceedings of Symposium on Network and Distributed System Security (NDSS) (Feb 2020)
2. Al-Anezi, D.M.M.K.: Generic packing detection using several complexity analysis for accurate malware detection. International Journal of Advanced Computer Science and Applications **5**(1) (2014). https://doi.org/10.14569/IJACSA.2014.050102
3. Alvarez, V.M.: Yara. https://virustotal.github.io/yara/ (2024), accessed: 2024-04-15
4. Anderson, H.S., Roth, P.: Ember: an open dataset for training static pe malware machine learning models. arXiv preprint arXiv:1804.04637 (2018)
5. ASPack Software: ASPack Software - Application for compression, packing and protection of software. http://www.aspack.com/ (2024), accessed: 2024-04-15
6. Callahan, D., Carle, A., Hall, M., Kennedy, K.: Constructing the procedure call multigraph. IEEE Transactions on Software Engineering **16**(4), 483–487 (1990). https://doi.org/10.1109/32.54302
7. Diestel, R.: Graph Theory. Springer, 5th edn. (2017)
8. Ebringer, T., Sun, L., Boztas, S.: A fast randomness test that preserves local detail. In: Proceedings of the 18th Virus Bulletin International Conference. pp. 34–42. Virus Bulletin Ltd (2008)

9. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. **44**(2) (mar 2008). https://doi.org/10.1145/2089125.2089126
10. Galloro, N., Polino, M., Carminati, M., Continella, A., Zanero, S.: A Systematical and longitudinal study of evasive behaviors in windows malware. Computers & Security **113**, 102550 (Feb 2022). https://doi.org/10.1016/j.cose.2021.102550
11. Hamilton, W.L.: Graph Representation Learning. Synthesis Lectures on Artificial Intelligence and Machine Learning **14**(3), 1–159 (2020), publisher: Morgan and Claypool
12. Hamrock, J., Lyda, R.: Using entropy analysis to find encrypted and packed malware. IEEE Security & Privacy **5**(02), 40–45 (mar 2007). https://doi.org/10.1109/MSP.2007.48
13. Horsicq: Detect it easy. https://github.com/horsicq/Detect-It-Easy (2024), accessed: 2024-04-15
14. Jacob, G., Comparetti, P., Neugschwandtner, M., Kruegel, C., Vigna, G.: A static, packer-agnostic filter to detect similar malware samples. In: International Conference on Detection of intrusions and malware, and vulnerability assessment. vol. 7591 (01 2010). https://doi.org/10.1007/978-3-642-37300-8_6
15. Kim, Y., Paik, J.Y., Choi, S., Cho, E.S.: Efficient svm based packer identification with binary diffing measures. In: 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC). vol. 1, pp. 795–800 (2019). https://doi.org/10.1109/COMPSAC.2019.00117
16. Li, S., Ming, J., Qiu, P., Chen, Q., Liu, L., Bao, H., Wang, Q., Jia, C.: PackGenome: Automatically Generating Robust YARA Rules for Accurate Malware Packer Detection. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 3078–3092. CCS '23, Association for Computing Machinery (2023). https://doi.org/10.1145/3576915.3616625
17. Li, X., Shan, Z., Liu, F., Chen, Y., Hou, Y.: A Consistently-Executing Graph-Based Approach for Malware Packer Identification. IEEE Access **7**, 51620–51629 (2019). https://doi.org/10.1109/ACCESS.2019.2910268
18. Li, Y., Gu, C., Dullien, T., Vinyals, O., Kohli, P.: Graph Matching Networks for Learning the Similarity of Graph Structured Objects. In: Chaudhuri, K., Salakhutdinov, R. (eds.) Proceedings of the 36th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 97, pp. 3835–3845. PMLR (Jun 2019)
19. Liu, H., Guo, C., Cui, Y., Shen, G., Ping, Y.: 2-SPIFF: a 2-stage packer identification method based on function call graph and file attributes. Applied Intelligence **51**(12), 9038–9053 (2021). https://doi.org/10.1007/s10489-021-02347-w
20. Liu, Z., Wang, R., Japkowicz, N., Gomes, H.M., Peng, B., Zhang, W.: Segdroid: An android malware detection method based on sensitive function call graph learning. Expert Syst. Appl. **235**(C) (Jan 2024), https://doi.org/10.1016/j.eswa.2023.121125
21. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. SIGPLAN Not. **40**(6), 190–200 (jun 2005). https://doi.org/10.1145/1064978.1065034
22. Mantovani, A., Aonzo, S., Ugarte-Pedrero, X., Merlo, A., Balzarotti, D.: Prevalence and impact of low-entropy packing schemes in the malware ecosystem. Proceedings 2020 Network and Distributed System Security Symposium (2020)
23. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco, CA (1997)

24. Muralidharan, T., Cohen, A., Gerson, N., Nissim, N.: File packing from the malware perspective: Techniques, analysis approaches, and directions for enhancements. ACM Comput. Surv. **55**(5) (dec 2022). https://doi.org/10.1145/3530810
25. Oreans Technologies: Winlicense. https://www.oreans.com/WinLicense.php, accessed: 2024-07-08
26. Pang, C., Yu, R., Chen, Y., Koskinen, E., Portokalidis, G., Mao, B., Xu, J.: Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 833–851 (2021)
27. PEiD: Peid. https://www.aldeid.com/wiki/PEiD (2024), accessed: 2024-04-15
28. Perdisci, R., Lanzi, A., Lee, W.: Classification of packed executables for accurate computer virus detection. Pattern Recognition Letters **29**(14), 1941–1946 (2008). https://doi.org/10.1016/j.patrec.2008.06.016
29. radare2: radare2: Unix-like reverse engineering framework and command-line tools (2024), https://github.com/radareorg/radare2, accessed: 2024-04-15
30. Raff, E., Zak, R., Lopez Munoz, G., Fleming, W., Anderson, H.S., Filar, B., Nicholas, C., Holt, J.: Automatic yara rule generation using biclustering. In: Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security. p. 71–82. AISec'20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3411508.3421372
31. Rahbarinia, B., Balduzzi, M., Perdisci, R.: Exploring the Long Tail of (Malicious) Software Downloads. In: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 391–402 (2017). https://doi.org/10.1109/DSN.2017.19
32. Rolles, R.: Unpacking virtualization obfuscators. In: Proceedings of the 3rd USENIX Conference on Offensive Technologies. p. 1. WOOT'09, USENIX Association, USA (2009)
33. 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. pp. 370–390. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
34. Ugarte-Pedrero, X., Balzarotti, D., Santos, I., Bringas, P.G.: SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In: 2015 IEEE Symposium on Security and Privacy. pp. 659–673 (2015). https://doi.org/10.1109/SP.2015.46
35. Ugarte-Pedrero, X., Graziano, M., Balzarotti, D.: A close look at a daily dataset of malware samples **22**(1) (Jan 2019)
36. UPX: UPX – the ultimate packer for executables. https://upx.github.io/ (2024), accessed: 2024-04-15
37. Yadegari, B., Debray, S.: Symbolic execution of obfuscated code. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. p. 732–744. CCS '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2810103.2813663
38. Zaki, M.J., Meira Jr, W.: Data mining and machine learning: fundamental concepts and algorithms. Cambridge University Press, 2 edn. (2020)