A Compact Multi-Step Framework for Packing Identification in Portable Executable Files for Malware Static Analysis

Jong-Wouk Kim[1], Yang-Sae Moon[2,3], and Mi-Jung Choi[2,3,4,*]

[1]Department of Computer Science, Kangwon National University, Gangwon 24341, Republic of Korea
[2]Department of Computer Science and Engineering, Kangwon National University, Gangwon 24341, Republic of Korea
[3]Department of Data Science, Kangwon National University, Gangwon 24341, Republic of Korea
[4]IGP. In Bigdata Convergence, Kangwon National University, Gangwon 24341, Republic of Korea
Email: jw.kim@kangwon.ac.kr (J.W.K); ysmoon@kangwon.ac.kr (Y.S.M); mjchoi@kangwon.ac.kr (M.J.C)
*Corresponding author
Manuscript received May 7, 2025; revised June 11, 2025; accepted October 18, 2025; published January 9, 2026

***Abstract*—Packing presents a major challenge in cybersecurity, as it complicates malware analysis and extends the operational lifespan of malicious software. This study addresses the issue by developing a robust framework designed to detect packed executable files and identify the specific packers used. The proposed framework leverages 20 optimally selected features extracted from Portable Executable (PE) files to detect packing and recognize packer signatures. A series of extensive experiments was conducted to determine the most effective combination of classification model and feature set. The extreme gradient boost algorithm was selected based on its superior performance. The proposed model achieved a high detection accuracy of 99.27% and an F1-score of 98.84%, outperforming recent methods in the field. In addition, the study introduces a publicly accessible dataset containing 213,784 PE samples and 125 features to facilitate future research. The framework provides a practical tool for security analysts, improving their ability to identify and respond to PE file-based malware in real-world environments. This study focuses exclusively on a static analysis pipeline; no dynamic execution is performed. We also describe how the framework could interface with sandbox-derived dynamic behavioral signals in future work without extending the current study's scope. Overall, this research contributes a static feature-based approach for packer detection and signature identification, together with a large-scale open dataset that supports ongoing advances in malware classification and analysis.**

***Keywords*—packed malware, packing, feature engineering, machine learning**

I. Introduction

Malicious software, more commonly known as malware, remains a persistent and serious threat in the field of cybersecurity. Despite continuous technological advancements in security measures, detecting and mitigating malware threats continues to pose significant challenges. A notable example occurred during the opening ceremony of the 2018 Pyeongchang Winter Olympics, when Cisco's Wi-Fi network and official website suffered major disruptions. Subsequent internal investigations attributed these incidents to malware attacks, demonstrating the serious harm such programs can inflict [1]. In addition to compromising user privacy and data security, malware can cause substantial financial losses and disrupt critical services. Recognizing these potential damages, organizations are increasingly investing significant resources to defend against malware attacks. Consequently, effective malware detection and analysis are essential for both users and service providers to mitigate risks such as personal information leakage, brand reputation damage, and degraded service quality. The Pyeongchang incident was particularly difficult to detect and mitigate because the malware employed advanced anti-analysis techniques (i.e., packing) to evade traditional security tools. In addition to packing, malware frequently employs anti-debugging, anti-virtualization, timing-based evasions, code obfuscation or self-modifying code, and user interface or device interference, all of which reduce both static and dynamic observability and thereby complicate detection and response [2–4].

Malware developers employ a range of sophisticated anti-analysis techniques, including but not limited to packing, anti-Virtual Machine (VM), and anti-debugging strategies. Anti-VM techniques are designed to detect the nature of the execution environment, such as sandboxes, VMs, or analysis tools. Similarly, anti-debugging measures aim to identify debugging environments (e.g., OllyDbg, WinDbg, and IDA Pro) and thereby delay or hinder the analysis process. These anti-analysis mechanisms both extend the operational lifespan of malware and increase the complexity of its examination. Among these techniques, packing is particularly notable, as it compresses or encrypts the original program code while incorporating additional anti-analysis features to obstruct examination efforts. Fig. 1 illustrates the packing process: the packer compresses or encrypts the binary file and injects an unpacking stub that decompresses or decrypts the original data at runtime. When the packed file is executed in memory, the stub code unpacks the data so that the original program can run, either within the same section or in a separate one.

Malware analysis encompasses several approaches, including automated, static, dynamic, and advanced methods. Automated analysis utilizes Application Programming Interfaces (APIs) or tools such as sandboxing utilities and the VirusTotal API to identify malware behavior and classification. Static analysis relies on specialized tools (e.g, HxD, PEView, and Strings) to examine the code without execution, allowing analysts to extract static features from the raw binary, including strings, headers, and functions. Dynamic analysis, by contrast, involves executing the malware to observe its behavior in real time, with analysts monitoring network communications and system processes using tools such as Wireshark, TCPView, and Process Explorer. Advanced analysis goes even deeper, employing debuggers to scrutinize malware behavior, memory usage, register states, registry interactions, and other intricate operational details [2].

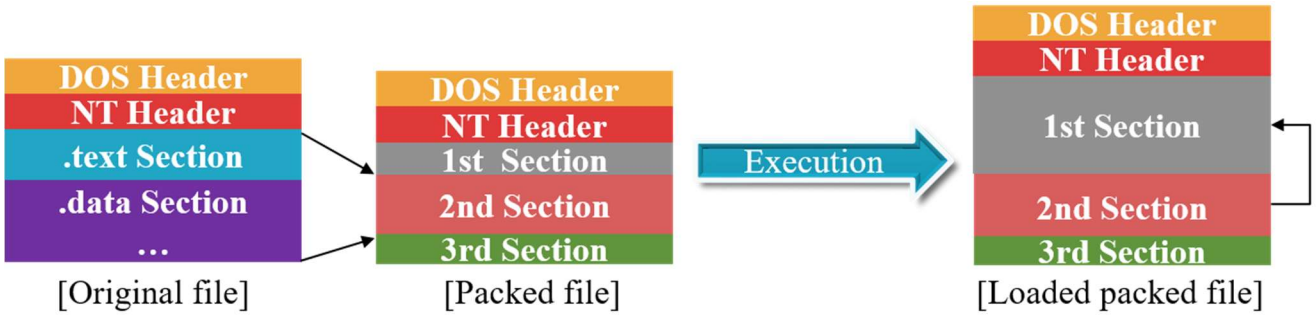


Fig. 1. Overview of packing the original program and executing the malware.

To evade antivirus systems and analytical tools, malware developers employ a variety of anti-analysis techniques. However, the challenge of anti-analysis extends beyond simply prolonging the operational lifespan of a malware. When researchers examine packed malware samples, they frequently extract features introduced by the packer rather than those belonging to the original program [5]. This discrepancy can cause mismatches between analysis results and the malware's true behavior, especially since packed malware can often detect and evade dynamic analysis. As a result, conducting advanced analysis on malware that employs anti-debugging or anti-emulation techniques remains particularly difficult. Numerous researchers have sought to develop models that combine machine language (opcode) sequences with deep learning to improve malware detection [6, 7]. Nonetheless, many of these studies overlook obfuscation methods such as packing and encryption.

Packing not only hinders the detection and analysis of malware but also creates substantial challenges when training Artificial Intelligence (AI)-based malware detection models. Gibert et al. investigated how packing affects static, machine learning-based malware detection and classification systems [8]. Their study revealed that malware detectors may unintentionally rely on specific packing routines as cues for identifying benign or malicious behavior, or for distinguishing between malware families. This dependency can severely undermine a detector's performance and reliability. Kearns and Li [9] demonstrated that maliciously selected errors can appear within the training data of Valiant's probably approximately correct learning framework. Likewise, Auer [10] emphasized that models must include a sufficient proportion of accurate data to achieve valid classification results. When inappropriate or adversarial data are present, poisoning attacks can occur, producing flawed and biased models. Therefore, packed samples must be explicitly incorporated into the training process. In recent years, the proportion of packed files has continued to increase. Rahbarinia *et al.* [11] analyzed three million samples collected in 2014 and found that 58% of malicious files and 54% of benign files were packed using well-known packers. Morgenstern and Pilz [12] reported that 35% of malicious files used custom packers. Pedrero *et al.* [13] classified packers into six categories and discovered that some malware samples underwent multiple layers of custom packing. Furthermore, 65.6% of packers were found to exhibit inconsistent unpacking routines.

In malware analysis, verifying whether a binary is packed is an essential step; if packing is detected, the binary must be unpacked before further examination. To address challenges related to packing, this paper introduces a comprehensive framework that classifies packed malware into three categories: not packed, custom packed, and well-known packed. Not packed samples are those that have not undergone packing by any tool. Custom packed samples are packed using a custom packer whose signature is unknown. Well-known packed samples are those packed with a recognized packer whose signature can be identified. The framework also supports identifying the specific packer used for well-known packed samples. The main contributions of this study are as follows:

1. Proposal of a compact, multi-step framework that integrates feature selection, packed-sample classification, and packer identification, validated through extensive analysis and experimentation.
2. Emphasis on the critical role of packing in malware research, underscoring the importance of detecting and unpacking packed samples.
3. Provision of a publicly accessible dataset to advance future studies and address persistent challenges in the field.

The remainder of this paper is organized as follows: Section II introduces the topics of malware detection, packing detection, and adversarial attacks on machine learning models. Section III details the proposed multi-step framework and the dataset. Section IV presents the experimental setup and evaluates the framework's performance, emphasizing the superior results achieved by the eXtreme Gradient Boost (XGBoost) model trained on preselected features and its successful identification of packers for well-known packed samples. Section V concludes the study and outlines directions for future research.

II. Related Works

A. Adversarial Attacks on AI Models

This section explains the challenges faced when training AI models with insufficient or distorted data. For AI systems to achieve their intended objectives, they must be trained on accurate and representative datasets. However, packing obscures the original structure and content of malware, complicating the training process for malware classifiers. As a result, several studies have emphasized the need to integrate packing detection and unpacking into the model training phase [14, 15]. Without these steps, AI models may inadvertently learn from misleading features introduced by the packer, potentially resulting in unintended adversarial behavior [15].

Goodfellow *et al.* [16] demonstrated that adversarial examples can mislead machine learning models, showing that

such models may inconsistently classify these examples when their training data are contaminated with perturbations. Zügner *et al.* [17] conducted experiments involving small perturbations and causal attacks on graph convolutional networks, focusing on vulnerabilities during the training phase. Their results revealed that even minor interference can significantly reduce classification accuracy, underscoring the susceptibility of many models to deception. To address this issue, they proposed Nettack, an efficient attack method based on incremental calculations that exposes and evaluates a model's resilience to adversarial manipulation.

Adversarial attacks are generally categorized into four types: gradient-based, score-based, decision-based, and transfer-based attacks. Gradient-based attacks, a subset of white-box approaches, assume that the adversary has access to the model's parameters and can obtain output labels for specific inputs. Goodfellow *et al.* [16] introduced the fast gradient sign method to explore the gradient direction along a model's decision boundary. Similarly, Carlini *et al.* [18] developed a powerful attack against defensive distillation and evaluated the effectiveness of possible countermeasures by using the Adam optimizer to generate adversarial examples. A key limitation of this approach is its dependence on the attacker's access to detailed model information to craft such examples.

Given these limitations, black-box attacks, where adversaries generate adversarial examples solely through queries, without any prior knowledge of the target model, have attracted growing research interest. However, many service providers impose limits on user queries, forcing attackers to minimize query frequency, since excessive requests over a short period are often flagged as suspicious or hostile. Li *et al.* [19] proposed an adversarial machine learning model for malware detection that leverages opcode *n*-grams. They constructed a dataset of 7927 malicious and 4070 benign samples, extracted opcode *n*-gram sequences using term frequency–inverse document frequency, and generated adversarial features with XGBoost. Although Support Vector Machine (SVM), Deep Neural Network (DNN), and XGBoost models trained on the original opcode *n*-gram sequences performed well on test data, they proved ineffective against the adversarial features.

B. Packed File Detection Using Machine Learning

Determining whether a binary is packed is a crucial preliminary step in malware detection [5]. Research by Osaghae [20] indicated that more than 80% of modern malware samples employ packing, underscoring the importance of addressing this issue through both detection and unpacking. This section reviews prior studies focused on packer detection and classification. Bat-Erdene *et al.* [14] categorized packing algorithms into three types: single-layer packing, re-packing, and multi-layer packing. Using symbolic aggregate approximation representations to classify packing algorithms, the authors analyzed entropy scaling patterns along with symbolic representations. Their experiments, which used 2196 samples and 19 packing algorithms, achieved a precision of 97.7%, an accuracy of 97.5%, and a recall of 96.8%.

Perdisci *et al*. [21] conducted a study to classify packed and non-packed files using nine distinct features, including the number of standard and non-standard sections, as well as the number of executable sections, in combination with pattern recognition techniques. The dataset comprised 5498 Portable Executable (PE) files, of which 2598 were packed malware, 2231 were non-packed benign files, and 669 were packed benign files created using 17 different tools. Their results showed an accuracy exceeding 95% across five classifiers: naïve Bayes, J48, bagged, K-nearest neighbor, and multilayer perceptron.

Zhang *et al.* [22] proposed a novel approach for detecting packed malware using system call analysis. By extracting contextual information from system calls in both benign and malicious samples within a sandbox environment, they trained a deep belief network model that achieved high accuracy. Similarly, Biondi *et al.* [23] identified an effective, efficient, and robust set of 119 features for detecting and classifying packed malware according to their respective packers. By labeling the ground truth in three distinct ways and testing more than 1500 combinations of features and algorithms, they aimed to determine the optimal machine learning model and feature set for maximum performance.

Park *et al.* [24] proposed a framework for inferring the lineage of packed malware. They employed both static and dynamic analysis to extract shared and family-specific features and included steps for identifying different versions of packed malware. The study addressed an agglomerative clustering problem using a dataset of 12,221 files. This approach enabled the matching of packed and unpacked files and the inference of lineage based on both feature sets.

Vasan *et al.* [25] introduced an architecture for classifying packed and unpacked malware using an ensemble of convolutional neural networks. The proposed architecture demonstrated strong performance, even against encrypted malware, while maintaining a low false alarm rate across 9339 malware images. It achieved accuracy rates of 99.0% for unpacked malware and 98.0% for packed malware.

Fleshman *et al.* [15] applied *n*-gram analysis and neural networks to raw byte sequences, demonstrating that machine learning classifiers exhibit greater resilience to evasion techniques such as binary modification, packing, and malicious code injection. Using a dataset of 2 million training samples and 80,000 test samples, they achieved a True Positive Rate (TPR) of 98.7%, significantly exceeding the average TPR of 84.2% reported by four antivirus engines for detecting obfuscation caused by packing.

Alkhateeb *et al.* [26] developed an advanced static analysis framework for identifying malware packers through multilayer feature engineering. They constructed a balanced dataset consisting of packed and benign PE samples for training and validation. Their approach integrated multiple static features, including PE header attributes, entropy measurements, and visual (image-based) representations of executable files. This multilayer feature set substantially improved detection accuracy for both known and unknown packers. The proposed classifier achieved an accuracy of 99.60% for known packers and 91% for unknown packers, underscoring the effectiveness of a comprehensive static analysis strategy. Table 1 presents a comparison of the methods used in previous studies with those of the proposed framework.

Table 1. List of packing detection studies and details

Study	Feature Type	# of Samples	Method
[14]	Entropy	2196	Similarity measurement
[15]	Raw bytes	2,080,000	*n*-gram, neural network
[21]	Static feature	5498	Pattern recognition
[22]	Dynamic feature, (system calls)	5995	Deep Belief Network (DBN)
[23]	Static feature	280,000	Machine learning
[24]	Static feature, Dynamic feature	12,221	Agglomerative clustering
[25]	Static feature	9339	An ensemble Convolutional Neural Network (CNN)
[26]	File header, Image, Entropy	7482	Random forest
[27]	File header	6278	Random forest
[28]	Call graphs	12,905	Graph neural network
Ours	Static feature	213,784	Machine learning

Ramamoorthy *et al.* [27] demonstrated that simple packers such as Ultimate Packer for eXecutables (UPX) can be detected effectively, whereas advanced packers like ElfPack and Kiteshield pose substantial challenges. Their complex packing methods, featuring executable and linkable format section docking and multi-layer encryption, led to lower F1-scores and higher false negative rates when evaluated using a random forest classifier.

Gennaro *et al.* [28] introduced PackHero, a framework designed to identify well-known packers through a graph matching network combined with hierarchical clustering. PackHero extracts call graphs from packed binaries and identifies the corresponding packer by comparing them with labeled graphs in a reference database. Using only 10 samples per packer, PackHero achieved a macro-average F1-score of 93.7% and an accuracy of 98.7%, demonstrating high efficiency even with limited training data.

III. Multi-Step Approach to Classifying Packed Malware

This paper proposes a systematic approach, illustrated in Fig. 2, for classifying and identifying packed samples. The framework is designed to achieve three main objective steps:

Step 1. Preselect pseudo-optimal features that accurately determine whether a sample is packed;

Step 2. Evaluate and identify effective models for classification; and

Step 3. Determine the specific packer used.

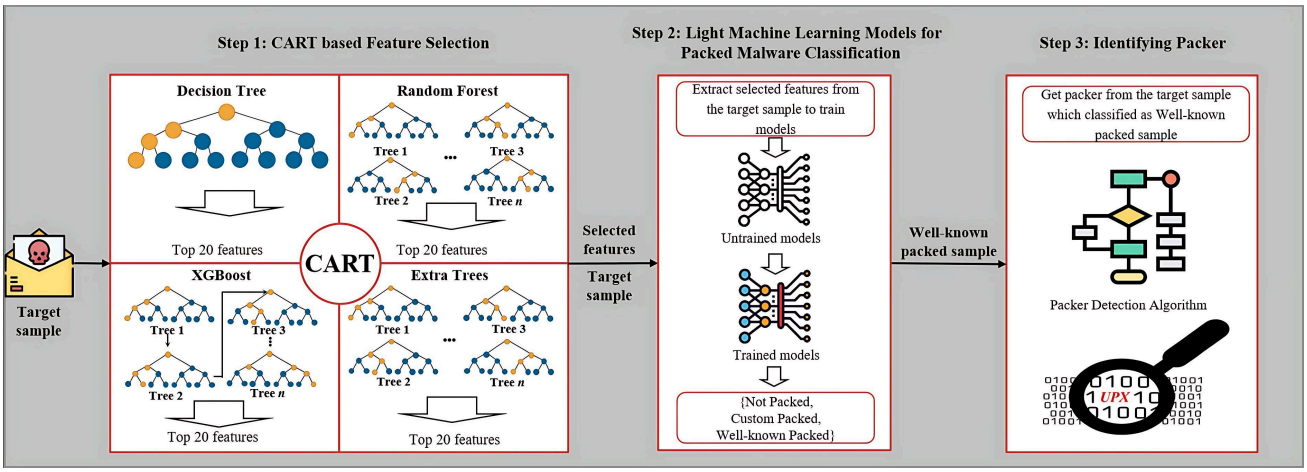


Fig. 2. Multi-step approach to classifying and identifying packed malware.

A. Step 1: Feature Selection

1) Classification and Regression Trees (CART)

The CART algorithm constructs a decision tree that minimizes Gini impurity, thereby selecting the most relevant features for classifying samples into homogeneous groups [29–31]. Using the CART algorithm, based on the feature evaluation approach proposed in a recent study [23], the framework preselects 20 key features. This choice was informed by experiments showing that smaller feature sets (5, 10, or 15) led to suboptimal detection performance. Consequently, 20 features were identified as necessary for effective classification of packed samples. The framework employs several CART-based algorithms, including decision tree, random forest, extra trees, and XGBoost, to rank feature importance and select the top 20. Preliminary testing revealed that classification accuracy dropped notably when fewer features were used, while exactly 20 yielded the best balance of accuracy and efficiency. Adding more than 20 features offered only marginal accuracy gains but substantially increased computational cost.

The Gini impurity for a particular node N_j is calculated using Eq. (1).

$$G(N_j) = \sum_{i=1}^{C} p(i|N_j)\left(1 - p(i|N_j)\right) = 1 - \sum_{i=1}^{C} p(i|N_j)^2 \quad (1)$$

Here, C represents the number of classes (or labels) and p denotes a probability value. The term $p(i|N_j)$ represents the probability that a sample belongs to class (or label) i at node N_j, whereas $1 - p(i|N_j)$ represents the probability that the sample belongs to all other classes (or labels) at the same node. A node with low impurity effectively divides the dataset into homogeneous subgroups, thus indicating a more effective split. That is, a lower Gini impurity value corresponds to a clearer separation of data. By minimizing Gini impurity, the CART algorithm generates nodes that classify samples more accurately. A substantial reduction in Gini impurity, as determined by Eq. (1), reflects the importance of the corresponding node in the tree-building

process. Within the CART algorithm (Algorithm 1), the feature that produces the greatest decrease in Gini impurity is selected at the algorithm's sixth line. The algorithm terminates when any of the following conditions are met:

- No remaining data;
- All data within a node share the same feature value;
- The number of samples within a node falls below a specified threshold; or
- The tree depth exceeds a defined limit.

Algorithm 1. CART (classification and regression trees) algorithm for a single tree

A classification and regression tree (CART) algorithm for a single decision tree *CART* (𝕏, 𝔽)

Input: a training dataset 𝕏, a feature set 𝔽
Output: a tree *T*
Begin
(1) Creates a single tree *T* with a root node;
(2) **if** all stopping criteria are true **then**
(3) *T* has one node with the most common class in 𝕏 *as* label;
(4) **return** *T;*
(5) **Else**
(6) Finds $f \in \mathbb{F}$ that best split 𝕏 using the gini impurity;
(7) Creates a label node with *f;*
(8) **for** values *v* of *f* **do**
(9) *X*= the subset of 𝕏 where *f* has value *v;*
(10) $F = \mathbb{F} - f$;
(11) *CART (X, F);*
(12) Connects a new node (*f, v*) to the parent node;
(13) **end-for**
(14) **end-if**
(15) **end**

In summary, the CART algorithm identifies features that partition data effectively, selecting the most discriminative attributes for classification tasks.

Table 2 presents a simple dataset illustrating the packing status of samples based on the attributes "Entropy of EPS", "Entropy of .text", "Number of standard sections", and "Zero sizes of uninitialized data". Fig. 3 depicts the decision tree used to classify these samples. To determine the most suitable feature for the root node in Fig. 3, the Gini impurity for each feature must be calculated, as shown in Table 3. Among the four features, the attribute "Zero size of raw data" yields the lowest Gini impurity value, allowing the root node to divide the dataset into six Not-Packed and eight Packed samples using the "Zero size of raw data" criterion.

Table 2. Example dataset used to create a single decision tree

Entropy of EPS	Entropy of .text	Number of Standard Sections	Zero size of Raw Data	Label
Mid	Mid	0	True	Not Packed
Mid	Mid	1	False	Not Packed
Low	Mid	0	True	Packed
High	High	0	False	Not Packed
High	Low	0	True	Packed
High	High	1	True	Packed
Mid	Low	0	True	Packed
Mid	Mid	2	True	Not Packed
Low	High	1	True	Packed

Table 3. Features enabling selecting root node data splits

Feature	Gini impurity
Entropy of EPS	0.4642
Entropy of .text	0.4500
Numbers of standard sections	0.4354
Zero size of raw data	0.3365

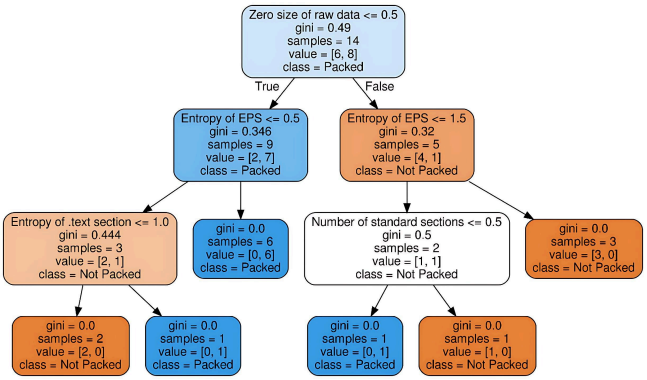


Fig. 3. Decision tree for classifying the example dataset.

2) Feature selection based on CART and impurity

Feature selection based on CART and impurity involves calculating feature importance during tree construction, as described in Algorithm 2 and Eqs. (2)–(5). Eq. (2) evaluates the importance of a node N_j by subtracting its impurity value from the impurity values of its child nodes.

Algorithm 2. Feature selection algorithm for one or many trees

An algorithm to select important features from one or many trees

Input: a set of trees 𝕋, a set of features 𝔽
Output: the top 20 important features
Begin
(1) **for each** tree $T \in \mathbb{T}$ **do**
(2) **for each** node $j \in T$ **do**
(3) **if** *j.leftChild = leaf* **and** *j.rightChild = leaf* **then**
(4) Go to the next node;
(5) **else**
(6) *left = j.leftChild;*
(7) *right = j.rightChild*;
(8) $nodeImportancej = weigthed\ number\ of\ sampels\ in\ j \times impurity\ value\ of\ j$;
(9) $nodeImportancej = nodeImportancej - weighted\ number\ of\ samples\ in\ left \times impurity\ value\ of\ left$;
(10) $nodeImportancej = nodeImportancej - weighted\ number\ of\ samples\ in\ right \times impurity\ value\ of\ right$;
(11) Go to the next node;
(12) **end-for**
(13) **for each** feature $f \in \mathbb{F}$ **do**
(14) $featureImportacne_f = \left(\frac{\sum_{k:\, node\ k\ splited\ by\ f} nodeImportacne_k}{\sum_{j \in all\ nodes} nodeImportance_j}\right) \div \sum_{l \in \mathbb{F}} featureImportacne_l$;
(15) **end-for**
(16) **end-for**
(17) **for each** feature $f \in \mathbb{F}$ **do**

(18) $featureImportacne_f = \frac{\sum_{m\in\mathbb{T}} featutreImportance_f \text{ in tree } m}{Number\ of\ trees\ in\ \mathbb{T}}$;
(19) **return** the top 20 important features
(20) **end**

$$I(N_j) = w_j \times G(N_j) - w_{jleft} \times G(N_{jleft}) - w_{jright} \times G(N_{j\_rig}) \quad (2)$$

$$I(f_{i\in all\ features}) = \frac{\sum_{j:node\ j\ splits\ on\ feature\ i} I(N_j)}{\sum_{k\in all\ nodes} I(N_k)} \quad (3)$$

$$I(f_{i\in all\ features})^{norm} = \frac{I(f_j)}{\sum_{i\in all\ features} I(f_i)} \quad (4)$$

$$I(f_{i\in all\ features})^{all\ trees} = \frac{\sum_{t\in all\ trees} I(f_i)^{norm\ in\ t}}{Number\ of\ trees} \quad (5)$$

Here, $I(N_j)$ denotes the importance of node N_j, and w_j represents its weight, corresponding to the number of samples contained in that node. This procedure determines the relative importance of all nodes in the tree, where a higher $I(N_j)$ value indicates a more influential node. Eq. (3) computes the importance of each feature by dividing the sum of the importance values of nodes split by that feature by the total importance across all nodes.

Eq. (4) then normalizes feature importance within the tree. For ensemble methods such as random forest, Eq. (5) calculates the overall feature importance by averaging the values obtained from all trees in the model.

Eqs. (2)–(5) are applied to calculate the importance of features within the decision tree. To determine the feature importance illustrated in Fig. 3, the following calculations are performed:

- $nodeImportance_{1st}(Zero\ size\ of\ raw\ data) = 14 \times 0.49 - 9 \times 0.346 - 5 \times 0.32 = 2.146$
- $nodeImportance_{2nd}(Entropy\ of\ EPS) = 9 \times 0.346 - 3 \times 0.4444 - 6 \times 0 = 1.782$
- $nodeImportance_{2nd}(Entropy\ of\ EPS) = 5 \times 0.32 - 2 \times 0.5 - 3 \times 0 = 0.6000$
- $nodeImportance_{3r}\ (Entropy\ of\ .text) = 3 \times 0.4444 - 2 \times 0 - 1 \times 0 = 1.332$
- $nodeImportance_{3rd}(Number\ of\ standard\ sections) = 2 \times 0.5 - 1 \times 0 - 1 \times 0 = 1.0000$
- $featutreImportance_{Entropy\ of\ EPS} = (2.394 \div 6.872) \div 1 = 0.3483$
- $featutreImportance_{Entropy\ of\ .text} = (1.332 \div 6.872) \div 1 = 0.1934$
- $featutreImportance_{Number\ of\ standard\ sections} = (1.000 \div 6.872) \div 1 = 0.1455$
- $featutreImportance_{Zero\ size\ of\ raw\ data} = (2.146 \div 6.872) \div 1 = 0.3484$

3) Permutation importance

Permutation importance measures the contribution of each feature by assessing the extent to which shuffling or altering that feature degrades model performance, as described in Algorithm 3. A noticeable drop in a classifier's performance after shuffling a highly influential feature demonstrates its significance, since randomization disrupts the relationship between the feature and the target variable. This approach functions as a model inspection technique applicable to any model trained on tabular data. Its model-agnostic design and capacity for repeated evaluation across multiple permutations are major advantages. However, permutation importance also has limitations. The results can vary across runs, though increasing the number of shuffling iterations helps reduce error variance by broadening the computation base. Moreover, random shuffling may produce unrealistic feature combinations, especially when strong correlations exist among variables, potentially increasing data uncertainty and adversely affecting prediction reliability.

Algorithm 3. Determining permutation importance

An algorithm of permutation importance *permutationImportance*($\mathbb{X}, \mathbb{F}, \mathbb{M}$)
Input: a test dataset $\mathbb{X}$, a feature set $\mathbb{F}$, a set of trained machine learning models $\mathbb{M}$
Output: scored feature set
begin
(1) for feature $f \in \mathbb{F}$ **do**
(2) Shuffles the records of f in the $\mathbb{X}$;
(3) Measures and records performance;
(4) **end-for**
(5) Measures the scored of a feature set $\mathbb{F}$;
(6) return scored feature set;

B. Step 2: Machine Learning Models for Classifying Packed Malware

In Step 2, 10 models are trained using the 20 most important features identified in Step 1 through empirical analysis. During feature evaluation, sets of 5, 10, 15, 20, and 25 top-ranked features were compared for classification performance. The highest accuracy rates, exceeding 99.8%, were achieved with 20 and 25 features. Because both configurations produced similarly high results, the selection of 20 features was deemed the most practical, balancing predictive performance with model simplicity. To determine the algorithm with superior performance in classifying packed malware into three categories, multiple classification models implemented in scikit-learn [32] and Keras [33] were tested. The goal is to classify packed samples with known packers into appropriate categories, recognizing the possibility of misclassification when sample sizes are limited. Accordingly, the model categorizes samples as not packed, custom packed, or well-known packed, and is trained on a dataset $\mathbb{X}$ with preselected features $\mathbb{F}$ and labels $\mathbb{Y}$, ultimately selecting the best-performing classifier as outlined in Algorithm 4.

Algorithm 4. Training machine learning models

Implying Machine Learning Classifier *Cls*($\mathbb{X}$, $\mathbb{F}$, $\mathbb{Y}$, $\mathbb{M}$)
Input: training dataset $\mathbb{X}$, pre-selected features $\mathbb{F}$, label $\mathbb{Y}$, machine learning models $\mathbb{M}$
Output: trained machine learning models $\mathbb{T}$
Begin
(1) **for each** model $M \in \mathbb{M}$ **do**
(2) *Classifier* ← supervised learning by model M with $\mathbb{X}$, $\mathbb{F}$, and $\mathbb{Y}$;
(3) Append *Classifier* in $\mathbb{T}$;

(4) **end-for**
(5) **return** $\mathbb{T}$
(6) **end**

Through supervised learning, all models $\mathbb{M}$ infer a function or decision boundary from labeled training data. Each model is trained using the preselected features from Step 1 along with the corresponding ground truth labels. After training, the models are evaluated on a separate test dataset, and Step 2 concludes by selecting the model that demonstrates the highest overall performance.

C. Step 3: Identifying the Packer Using Signatures

In Step 3 of the proposed framework, packer identification is performed for samples classified as well-known packed in Step 2. This process employs the `pefile` library [34] to detect packer signatures within the PE sample, as described in Algorithm 5. The framework inspects the Entry Point Section (EPS) of the binary file to locate the packer's signature. If multiple signatures stored in a comprehensive database are detected in the EPS, the framework returns the most recently matched packer signature.

Algorithm 5. Identifying the packer

Packer detection *Packer(PE, sigDB)*
Input: a PE file, packer signatures
Output: packer
begin
(1) *signatures* = load the signatures and packers from *sigDB;*
(2) *data* = scan the entry point of *PE;*
(3) **for** (*sig*) ∈ *signatures* **do**
(4) **if** *sig* in *data* **then return** *packer*
(5) **end-for**
(6) **return** *None*
(7) **end**

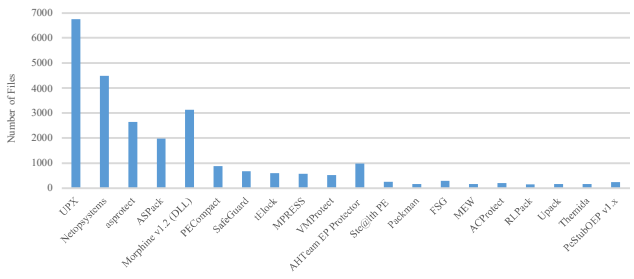


Fig. 4. Distribution of well-known packers (top 20).

The signature database, containing more than 4200 entries sourced from `PEiD` [35], serves as a core component of the detection process. Fig. 4 presents a subset of the packer signatures cataloged in the database, where the first line lists the packer and its version information, and the second line displays its hexadecimal signature. The `ep_only` attribute indicates whether a signature appears exclusively in the EPS, taking the value true or false. A key distinction between this framework and `PEiD` lies in their detection methodologies: whereas `PEiD` primarily identifies packers based solely on signature matching, the proposed framework integrates multiple techniques to detect packing and determine the corresponding packer signature.

IV. Experimental Results and Discussion

A. Experimental Setup

For this study, a total of 213,784 samples were obtained from VirusShare. These samples were labeled using a component of the framework introduced in a previous study, outlined as Algorithm 6 [36]. In this experiment, we adopted a three-category labeling scheme for packed binaries: custom packed, well-known packed, and not packed. Custom packed samples refer to binaries that exhibit either unusually high or low entropy [36], or lack a recognizable EPS, and in both cases, do not correspond to any known packer signatures. In contrast, well-known packed samples are those that match at least one known packer signature. This labeling method is based on an entropy- and EPS-based approach, which has been widely used in prior research [36].

Algorithm 6. Labeling Algorithm

Labling algorithm *Labeling(A PE file)*
Input: A PE file
Output: *Not Packed* or *Custom Packed* or *Well-known Packed*
begin
(1) **if** No EP Section **then return** *Custom Pakced;*
(2) **else if** Packed Signature Found **then return** *Well-known Packed;*
(3) **else if** "WRITE" characteristic **and** EPS Entropy ∈ *Packing-Range* **then**
(4) **Return** *Custom Packed;*
(5) **else return** *Not Packed;*
(6) **end if**
(7) **end**

Of the collected samples, 87,499 were classified as custom packed, characterized by binaries with obscured EPS or elevated entropy values without identifiable packer signatures. Additionally, samples that contained a valid entry point with the write attribute and an EPS entropy value within the packing range [36] but lacked a signature were also labeled as custom packed. The write attribute is critical for packed binaries, as it enables the reconstruction of the import address table, original code, and data. A total of 25,955 samples were identified as well-known packed, distinguished by recognizable EPS structures and packer signatures. The remaining 100,544 samples were labeled as not packed.

```
[ASPack 1.05b by]
signature = 75 00 E9
ep_only = true

[ASPAck 1.061b]
signature = 90 90 75 00 E9
ep_only = true

[ASPack 1.08]
signature = 90 90 90 75 01 90 E9
ep_only = true

[UPX -> www.upx.sourceforge.net]
signature = 60 BE ?? ?? ?? 00 8D BE ?? ?? ?? FF
ep_only = true

[UPX 0.50 - 0.70]
signature = 60 E8 00 00 00 00 58 83 E8 3D
ep_only = true

[yoda's Crypter 1.3 -> Ashkbiz Danehkar]
signature = 55 8B EC 53 56 57 60 E8 00 00 00 00 5D 81 ED 6C 28 40 00 B9 5D 34 40 00
ep_only = true
```

Fig. 5. Examples of packer signatures stored in the signature database.

Fig. 5 illustrates the distribution of packers within the well-known Packed category. The packer UPX was the most frequently detected, appearing in 6748 samples, followed by Netopsystems, found in 4482 samples. In addition, 35 packed malware samples were found to use distinctive, less common packers. Detailed insights regarding the well-known packed samples are provided in Table A1.

For the experimental phase, all 213,784 PE-based malware samples were used, with the dataset divided into a 70–30 split for training and testing, respectively. Model performance was evaluated using accuracy, F1-score, and area under the receiver operating Characteristic Curve (AUC). Each metric was computed from the confusion matrix using True Positive (TP), False Positive (FP), True Negative (TN), and False Negatives (FN) characterizations. Accuracy is defined using Eq. (6):

$$Accuracy = \frac{TP+T}{TP+FP+TN+FN} \quad (6)$$

The F1-score represents the harmonic mean of precision and recall, and is calculated as shown in Eqs. (7)–(9).

$$Precision = \frac{TP}{TP+FP} \quad (7)$$

$$Recall = \frac{TP}{TP+F} \quad (8)$$

$$F1 - Score = \frac{2\times Precision\times Recall}{Precision+R} \quad (9)$$

The AUC is constructed by varying the decision threshold and plotting the TPR against the FP rate. The AUC value is then computed as the area beneath this curve.

Hyperparameters relevant to the evaluation of feature importance and classifier performance are summarized in Tables 4 and 5. Table 4 lists the hyperparameters for each model used to assess feature importance via the CART algorithm and permutation importance. To ensure consistent comparison, the random state for all models was fixed at zero during feature importance computation. Furthermore, the evaluation of feature importance through permutation was performed using the same models and hyperparameter settings described in Table 4. Table 5 presents the models employed for data classification along with their corresponding hyperparameters.

Table 4. Model hyperparameters for calculating feature importance (CART, permutation importance)

Purpose	Model	Hyperparameter	
Feature importance scores of CART	Decision tree	Random state	0
	Extra trees	Random state	0
	Random forest	Random state	0
	XGBoost	Random state	0
		Objective	multi:softprob
Permutation importance scores	Decision tree	Random state	0
	Extra trees	Random state	0
	Random forest	Random state	0
	XGBoost	Random state	0
		objective	multi:softprob

Table 5. Model hyperparameters used to classify packed samples

Model	Hyperparameter	
Linear SVM	max_iter	10000
SVM	kernel	rbf
Logistic regression	max_iter	10000
Naïve Bayes	Default	
kNN (k = 3)	k	3
DNN	Hidden layer	32, 64, 128, 256, 128, 64, 32, 16, 8
	Learning rate	1×10^{-5}
	Epsilon	1×10^{-8}
	Epoch	50
	Batch	16
Decision tree	Random state	0
Random forest	Random state	0
Extra trees	Random state	0
XGBoost	Random state	0
	Objective	multi:softprob

B. Experimental Results

1) Step 1: comparing feature importance scores

The framework identified 20 pseudo-optimal features from a total of 125, following the approach proposed by Biondi *et al.* [23] and our previous research, using a dataset of 213,784 (as Table 6) malware samples. This feature selection process was conducted with four algorithms: decision tree, extra trees, random forest, and XGBoost. The features selected by each model showed slight variations, as illustrated in Fig. 6, reflecting differences in how each algorithm evaluates feature importance. Notably, the entropy of the EPS consistently emerged as the most significant feature in most models.

Table 6. Training and testing datasets

Dataset	Group			Total
	Custom-Packed	Well-Known Packed	Not Packed	
Train Set	61,249 (28.6%)	18,075 (8.5%)	70,379 (32.9%)	149,703 (70.0%)
Test Set	26,250 (12.3%)	7746 (3.6%)	30,162 (14.1%)	64,158 (30.0%)

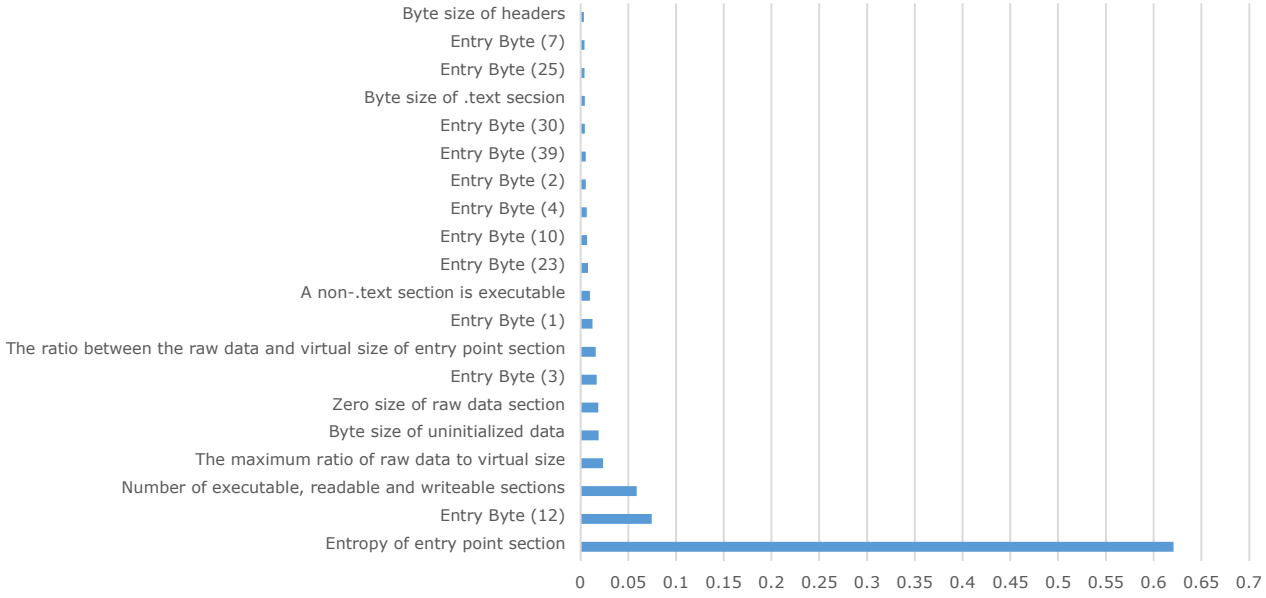
As shown in Fig. 6(a) and (e), the decision tree model assigned particularly high importance to the entropy of the EPS compared with the other models. This variation arises from the methodology of ensemble-based algorithms, which construct multiple trees and average feature importance values across them, as described in line 19 of Algorithm 2. Consequently, models that build multiple trees tend to assess feature importance more accurately by integrating results from a broader range of features.

2) Step 2: comparison of machine learning models

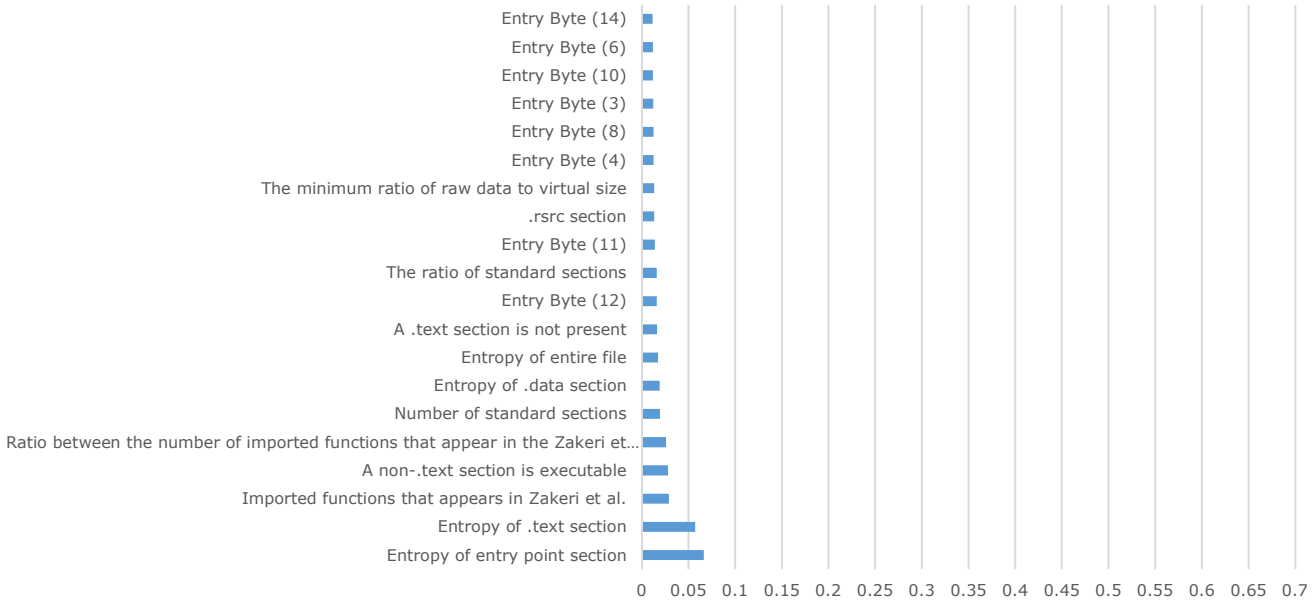
Table 7 presents the results of comparative experiments, including findings from recent work. The table provides a comprehensive evaluation of each experimental scenario using performance metrics such as accuracy, F1-score, and AUC, enabling a thorough comparative analysis. The nine feature selection models fall into three groups. The CART-based group (Decision tree, Extra tree, Random Forest, and

XGBoost) ranks features by impurity reduction and thus tends to emphasize packing-sensitive structure indicators (e.g., entry-point section entropy). The permutation importance group estimates predictive utility by measuring validation-score drop when a feature is shuffled, mitigating correlation bias. The literature baseline (Biondi *et al.* [23]) using a predefined feature set. Notably, both normal and linear SVMs exhibited weaker performance than other classifiers. This limitation is attributed to the `max_iter` parameter settings described in Table 4. The absence of an upper bound on `max_iter` extended computation times beyond 24 h, leading to inadequate model convergence due to the large dataset size. As a result, SVM-based models experienced performance degradation primarily because of the dataset's scale and constraints imposed by the iteration parameter. Future research could address this issue by applying data reduction techniques or optimizing hyperparameters to improve model efficiency.

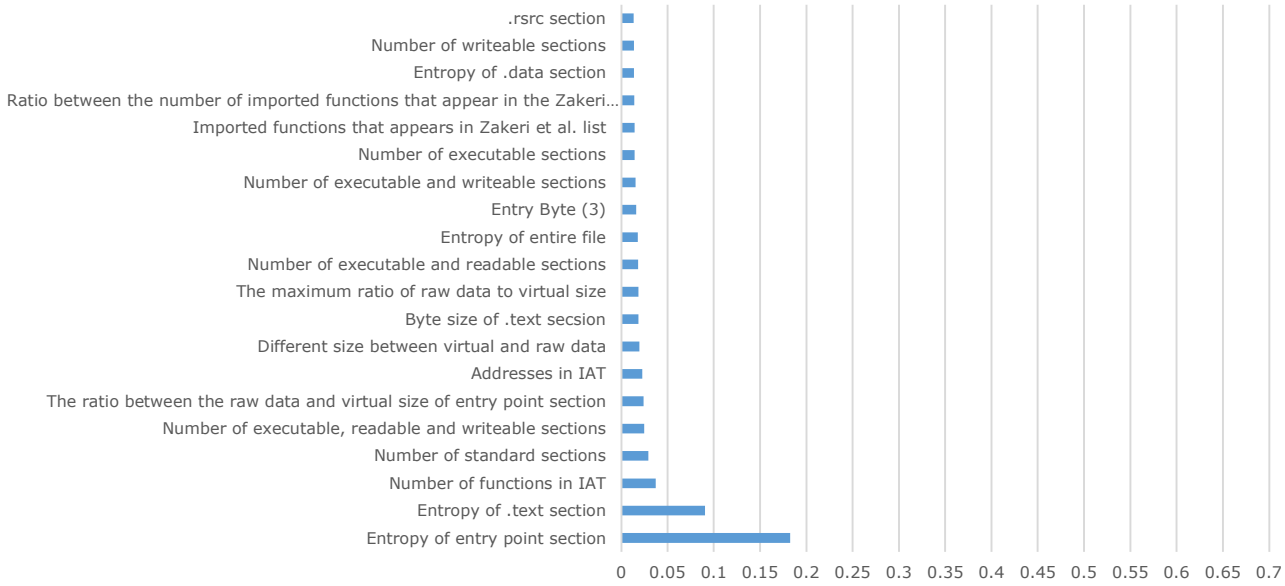
Among all experimental configurations, the XGBoost algorithm, trained on the top 20 most important features identified through the permutation importance method, achieved the highest overall performance. This superior outcome is attributed to XGBoost's capability for parallel tree boosting, which enables efficient and precise handling of diverse classification challenges. The proposed framework's XGBoost model outperformed the results of Biondi *et al.* [23] across all evaluation metrics while utilizing significantly fewer features. In Biondi *et al.*'s study [23], certain among the 119 features were found to introduce noise, diminishing performance compared with the current framework. Furthermore, XGBoost's boosting mechanism improves accuracy by assigning greater weight to misclassified samples, thereby enhancing overall model robustness relative to alternative approaches.



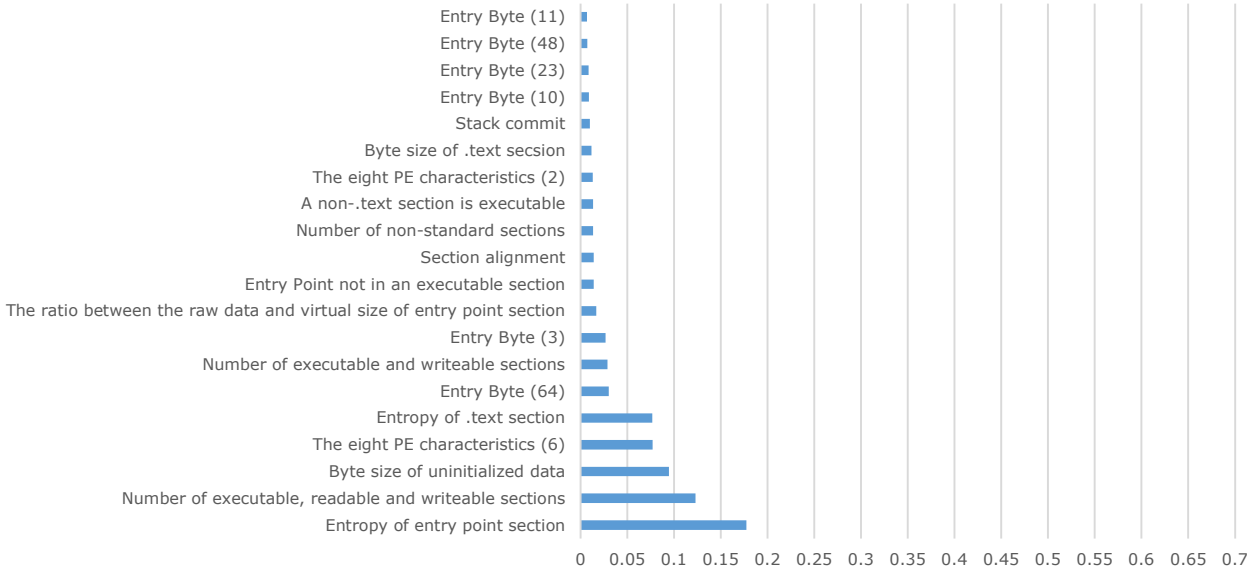
(a)



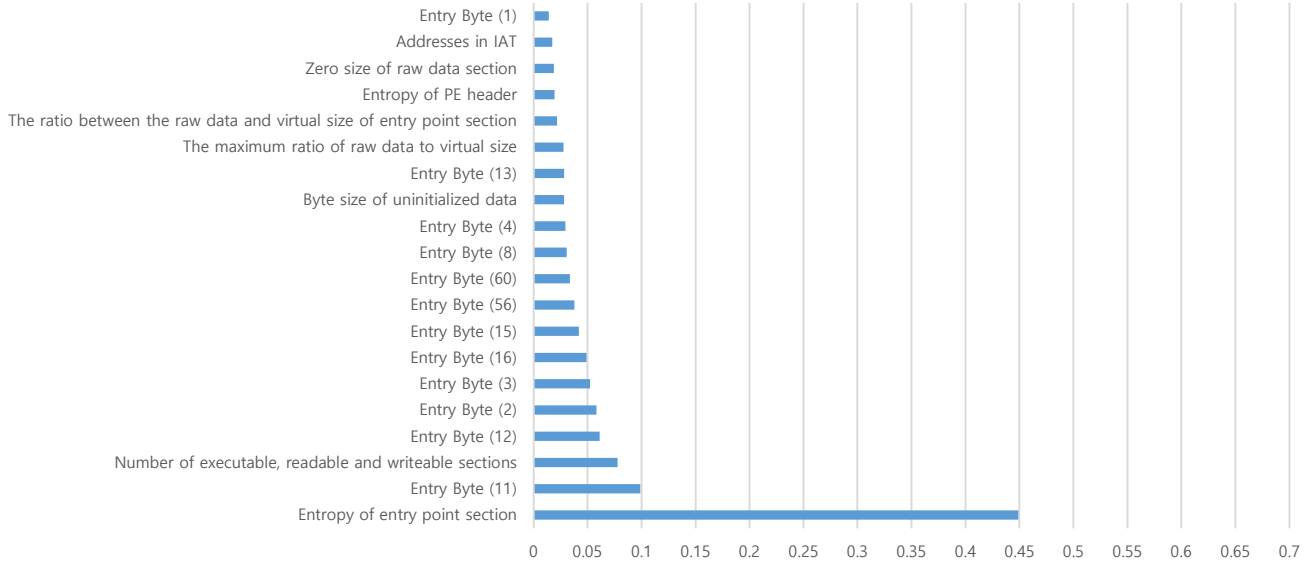
(b)


.rsrc section
Number of writeable sections
Entropy of .data section
Ratio between the number of imported functions that appear in the Zakeri...
Imported functions that appears in Zakeri et al. list
Number of executable sections
Number of executable and writeable sections
Entry Byte (3)
Entropy of entire file
Number of executable and readable sections
The maximum ratio of raw data to virtual size
Byte size of .text secsion
Different size between virtual and raw data
Addresses in IAT
The ratio between the raw data and virtual size of entry point section
Number of executable, readable and writeable sections
Number of standard sections
Number of functions in IAT
Entropy of .text section
Entropy of entry point section
0 0.05 0.1 0.15 0.2 0.25 0.3 0.35 0.4 0.45 0.5 0.55 0.6 0.65 0.7

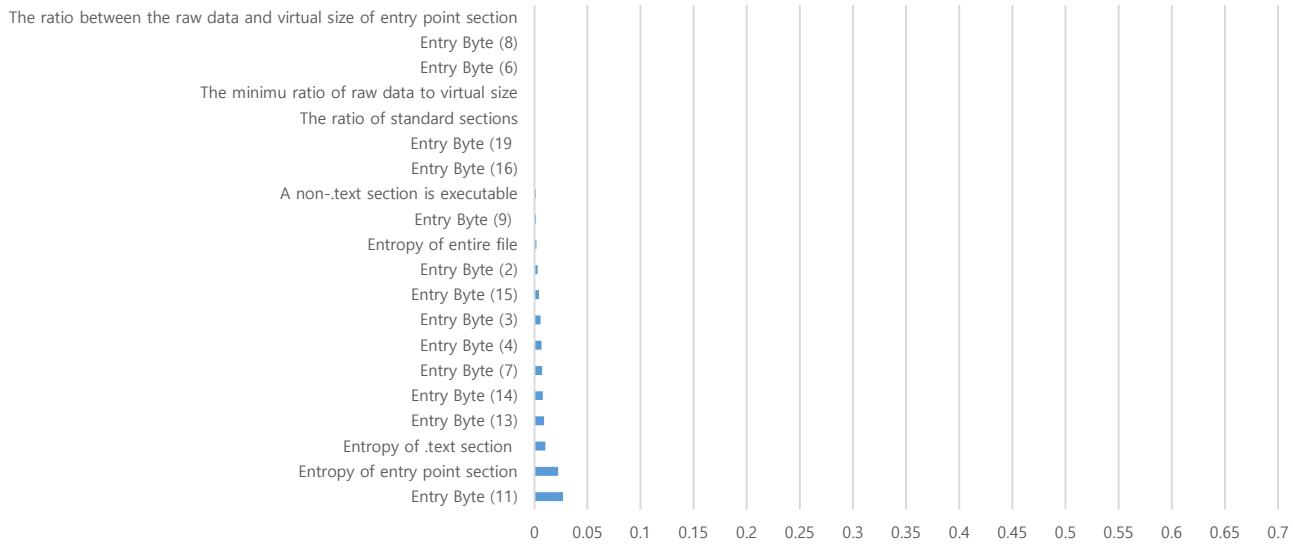
(c)


Entry Byte (11)
Entry Byte (48)
Entry Byte (23)
Entry Byte (10)
Stack commit
Byte size of .text secsion
The eight PE characteristics (2)
A non-.text section is executable
Number of non-standard sections
Section alignment
Entry Point not in an executable section
The ratio between the raw data and virtual size of entry point section
Entry Byte (3)
Number of executable and writeable sections
Entry Byte (64)
Entropy of .text section
The eight PE characteristics (6)
Byte size of uninitialized data
Number of executable, readable and writeable sections
Entropy of entry point section
0 0.05 0.1 0.15 0.2 0.25 0.3 0.35 0.4 0.45 0.5 0.55 0.6 0.65 0.7

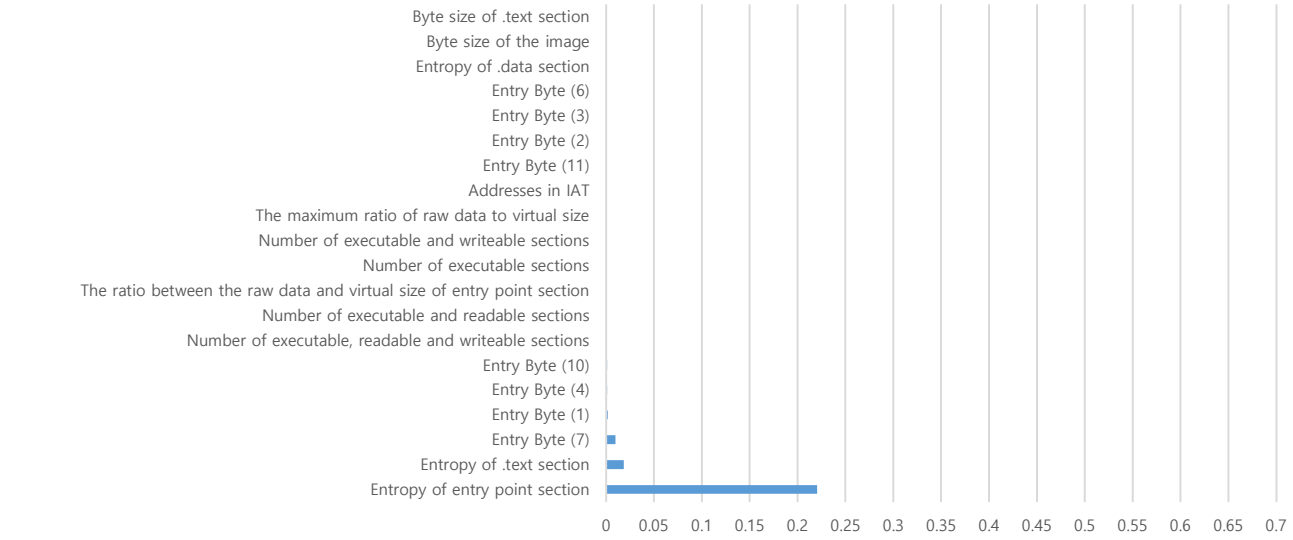
(d)


Entry Byte (1)
Addresses in IAT
Zero size of raw data section
Entropy of PE header
The ratio between the raw data and virtual size of entry point section
The maximum ratio of raw data to virtual size
Entry Byte (13)
Byte size of uninitialized data
Entry Byte (4)
Entry Byte (8)
Entry Byte (60)
Entry Byte (56)
Entry Byte (15)
Entry Byte (16)
Entry Byte (3)
Entry Byte (2)
Entry Byte (12)
Number of executable, readable and writeable sections
Entry Byte (11)
Entropy of entry point section
0 0.05 0.1 0.15 0.2 0.25 0.3 0.35 0.4 0.45 0.5 0.55 0.6 0.65 0.7

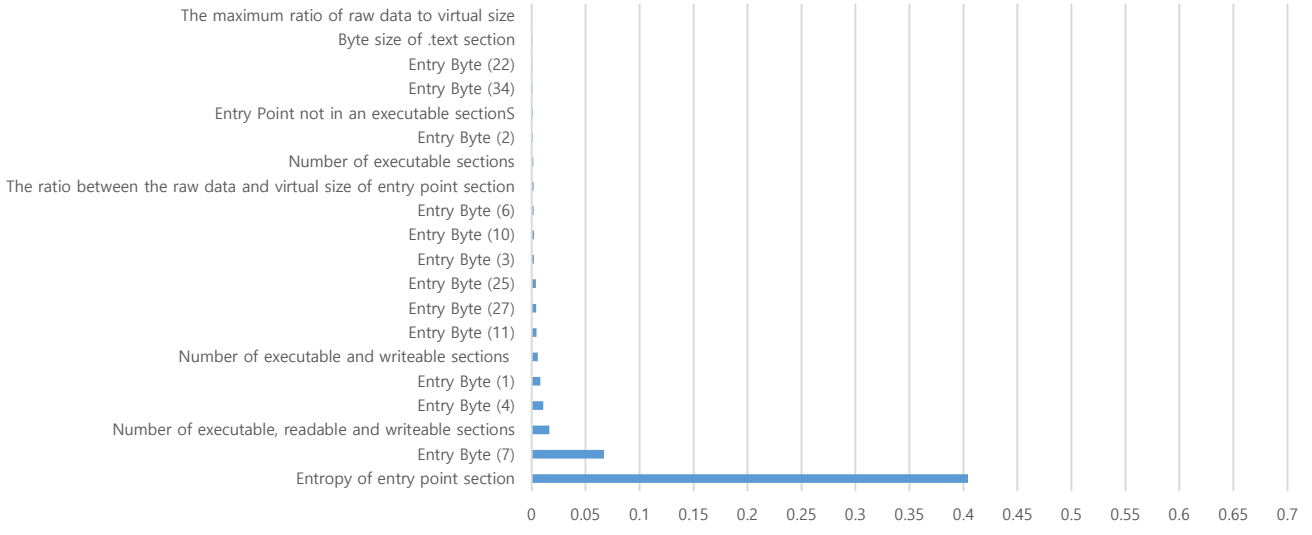
(e)



(f)



(g)



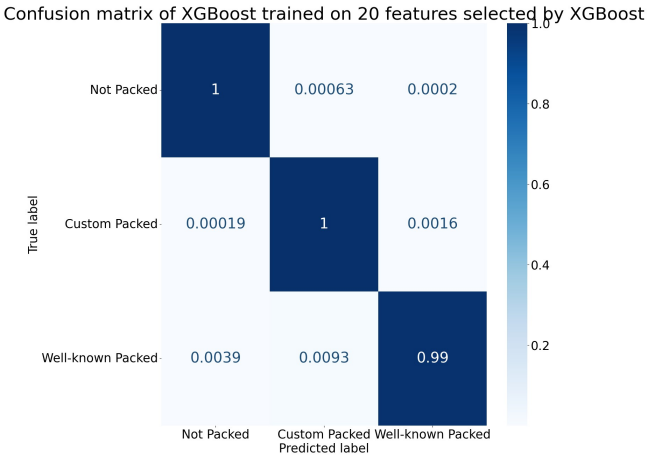
(h)

Fig. 6. Distribution of well-known packers (top 20). (a) Feature importance scores of CART with decision tree; (b) Feature importance scores of CART with extra trees; (c) Feature importance scores of CART with random forest; (d) Feature importance scores of CART with XGBoost; (e) Permutation importance scores with decision tree; (f) Permutation importance scores with extra trees; (g) Permutation importance scores with random forest; (h) Permutation importance scores with XGBoost.

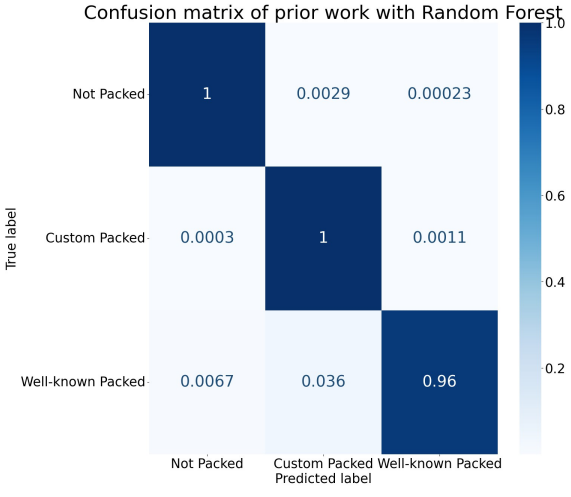
Table 7. Classifying results of each feature selection and classifying model

Feature Selection Model	Classifying Model	Accuracy	F1-score	AUC
Decision tree (CART)	Linear SVM	0.2981	0.2772	0.5195
	SVM	0.7113	0.6457	0.7876
	Logistic regression	0.4654	0.2254	0.6915
	Naïve Bayes	0.6884	0.6364	0.8536
	k-Nearest Neighbors (kNN) (k = 3)	0.9047	0.9032	0.9633
	Multilayer perceptron (MLP)	0.7011	0.6991	0.8072
	Decision tree	0.9893	0.9829	0.9898
	Random forest	0.9943	0.9911	0.9998
	Extra trees	0.9868	0.9844	0.9994
	XGBoost	0.9954	0.9927	0.9998
Extra tree (CART)	Linear SVM	0.4610	0.3825	0.6600
	SVM	0.5844	0.5595	0.5371
	Logistic regression	0.7793	0.7367	0.9019
	Naïve Bayes	0.6529	0.6092	0.8397
	kNN (k = 3)	0.9433	0.9444	0.9787
	MLP	0.8999	0.8999	0.9531
	Decision tree	0.9837	0.9747	0.9823
	Random forest	0.9888	0.9833	0.9990
	Extra trees	0.9853	0.9799	0.9981
	XGBoost	0.9887	0.9827	0.9993
Random forest (CART)	Linear SVM	0.4294	0.4006	0.5669
	SVM	0.638	0.4696	0.7478
	Logistic regression	0.4850	0.2727	0.7024
	Naïve Bayes	0.7101	0.6635	0.8608
	kNN (k = 3)	0.8708	0.8603	0.9474
	MLP	0.6557	0.6135	0.8322
	Decision tree	0.9792	0.9643	0.9845
	Random forest	0.9879	0.9791	0.9991
	Extra trees	0.9848	0.9768	0.9987
	XGBoost	0.9868	0.9767	0.9991
XGBoost (CART)	Linear SVM	0.6055	0.4803	0.6480
	SVM	0.5245	0.4800	0.5975
	Logistic regression	0.4793	0.2167	0.4987
	Naïve Bayes	0.6783	0.6471	0.8436
	kNN (k = 3)	0.9060	0.9020	0.9551
	MLP	0.6660	0.6659	0.8281
	Decision tree	0.9851	0.9745	0.9889
	Random forest	0.9904	0.9836	0.9994
	Extra trees	0.9876	0.9811	0.9988
	XGBoost	0.9902	0.9830	0.9995
Decision tree (Permutation importance)	Linear SVM	0.4368	0.3681	0.5845
	SVM	0.6407	0.5761	0.7251
	Logistic regression	0.5824	0.3896	0.7216
	Naïve Bayes	0.7167	0.6777	0.8458
	kNN (k = 3)	0.9348	0.9329	0.9753
	MLP	0.8251	0.8245	0.9379
	Decision tree	0.9865	0.9780	0.9899
	Random forest	0.9916	0.9864	0.9998
	Extra trees	0.9840	0.9808	0.9993
	XGBoost	0.9911	0.9861	0.9996
Extra tree (Permutation importance)	Linear SVM	0.2874	0.2657	0.5250
	SVM	035605	0.4803	0.6386
	Logistic regression	0.7000	0.6784	0.8581
	Naïve Bayes	0.6803	0.6410	0.8408
	kNN (k = 3)	0.9444	0.9461	0.9798
	MLP	0.8880	0.8884	0.9646
	Decision tree	0.9863	0.9800	0.9897
	Random forest	0.9915	0.9883	0.9995
	Extra trees	0.9864	0.9832	0.9993
	XGBoost	0.9920	0.9892	0.9995
Random forest (Permutation importance)	Linear SVM	0.5031	0.4789	0.4821
	SVM	0.5445	0.5219	0.6077
	Logistic regression	0.5873	0.3886	0.5571
	Naïve Bayes	0.6585	0.6183	0.8109
	kNN (k = 3)	0.8760	0.8663	0.9504
	MLP	0.7104	0.7083	0.7666
	Decision tree	0.9891	0.9818	0.9992
	Random forest	0.9933	0.9888	0.9996
	Extra trees	0.9899	0.9857	0.9995
	XGBoost	0.9943	0.9904	0.9996
XGBoost (Permutation importance)	Linear SVM	0.4613	0.4006	0.5669
	SVM	0.5571	0.4908	0.4319
	Logistic regression	0.6015	0.5904	0.6480
	Naïve Bayes	0.6582	0.6398	0.7983
	kNN (k = 3)	0.9042	0.8988	0.9498
	MLP	0.7946	0.7899	0.8819
	Decision tree	0.9913	0.9858	0.9993
	Random forest	0.9945	0.9911	0.9996
	Extra trees	0.9878	0.9859	0.9995
	XGBoost	0.9967	0.9946	0.9998
Biondi *et al.* [23]	Decision tree	0.9891	0.9812	0.8787
	Random forest	0.9928	0.9885	0.9998

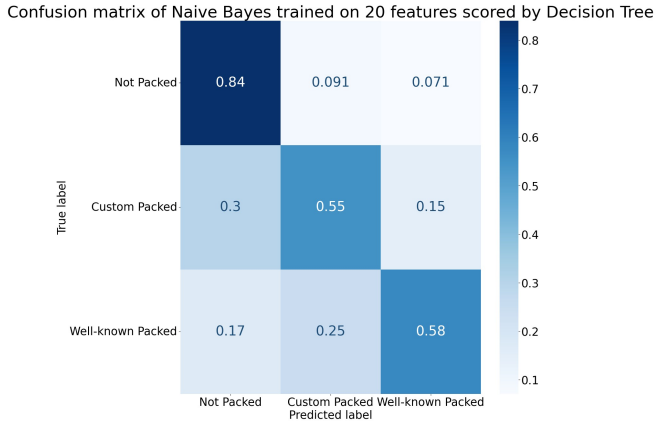
Fig. 7 presents confusion matrices for selected cases, with Fig. 7(a) depicting the best-performing scenario among all experiments. The model accurately classified not-packed and custom-packed samples and achieved a more precise classification of well-known packed samples compared with the results reported by Biondi *et al.* [23]. The relatively weaker classification performance for well-known packed samples is primarily attributed to the limited number of available data points. Moreover, Fig. 7(a) provides a visual representation of the best model's performance across experimental scenarios, offering deeper insight into its classification effectiveness.



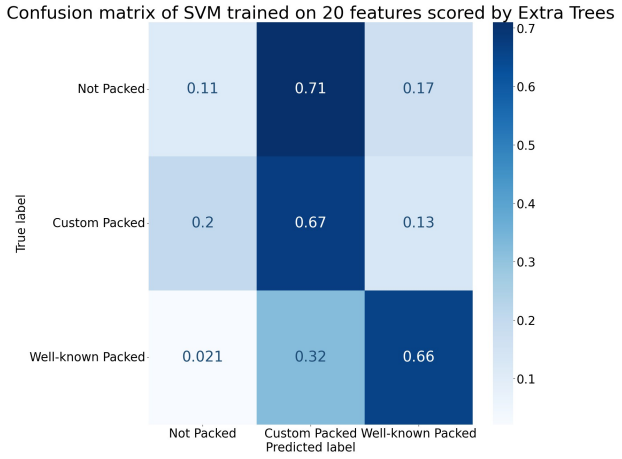
(a)



(b)



(c)



(d)

Fig. 7. Confusion matrix of different scenarios: (a) XGBoost classifier that learned the preselected features by the permutation importance and XGBoost; (b) Prior work by Biondi *et al.* [23]; (c) Naïve Bayes classifier that learned the preselected features by the CART and Decision Tree; (d) SVM classifier that learned the preselected features by CART and Extra Trees.

3) *Step 3: packer identification*

In Step 3 of the proposed framework, packer identification is performed using `pefile` [34], a Python module widely recognized for its cross-platform capabilities in analyzing PE files. Leveraging the best-performing model, the framework successfully identified packers for 7,685 samples. These results were then compared with those obtained from `PEiD` [35], a well-established tool for packer identification. A perfect match was observed between the results produced in Step 3 and those identified by `PEiD`, emphasizing the reliability and accuracy of the proposed method. This complete alignment is attributable to the framework's use of the same packer database employed by `PEiD`. Fig. 8 presents the 20 most frequently identified packers during Step 3, illustrating the most prevalent packers within the analyzed dataset.

C. *Limitations and Future Work*

Establishing an accurate ground truth is essential to ensure the effectiveness of the classification model. Building upon our previous work, future improvements in packing detection techniques could further enhance the accuracy of ground truth construction. A more robust ground truth could be achieved by incorporating advanced methods such as dynamic entropy measurement and AI-based packing detection, which surpass the static labeling approach employed in this study. While the proposed framework demonstrates high accuracy and efficiency using static features, it does not yet account for adversarial or obfuscated malware. These evasive techniques remain an open challenge for future research. Moreover, the framework currently lacks mechanisms to address class imbalance among the packing categories. Future studies could apply sampling strategies or class-weighted learning to improve detection performance for minority classes.

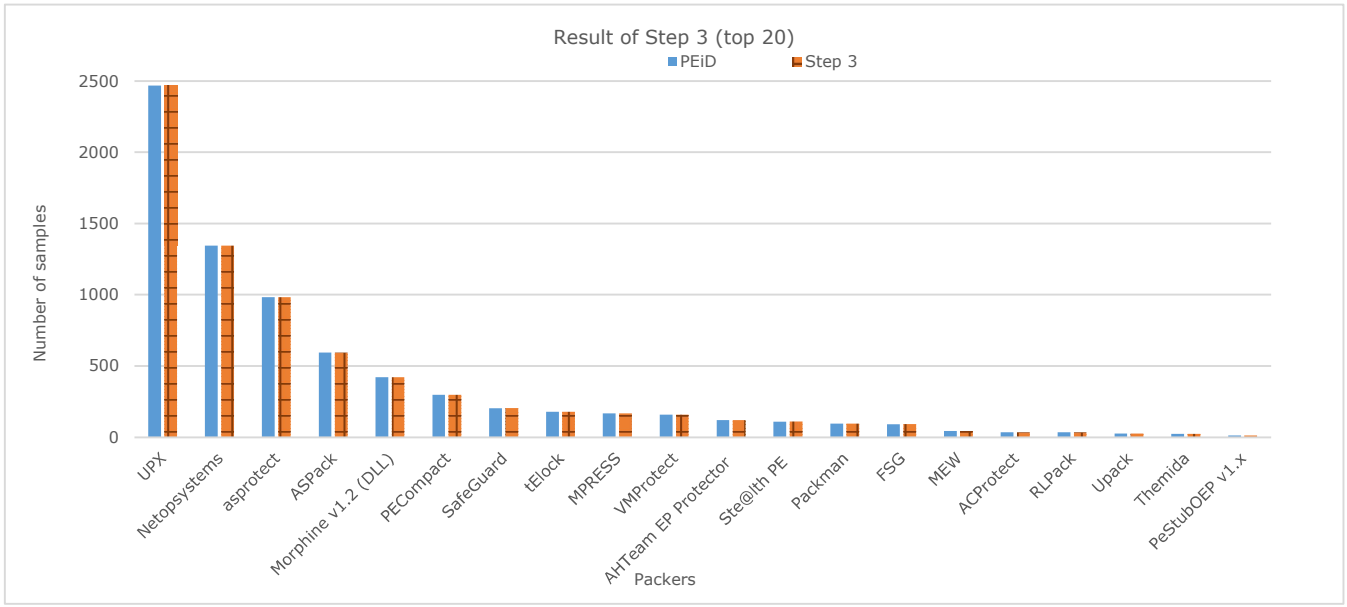


Fig. 8. PEiD and step 3 comparison results.

The present research also focuses exclusively on single-layer packing and does not extend to repacking or multi-layer packing. Repacking involves repeatedly compressing a binary using the same packer, whereas multi-layer packing applies multiple packers in succession. Accurately establishing ground truth will remain a central objective in our future work. Additionally, expanding the framework to address repacking and multi-layer packing will help mitigate the limitations identified in this study.

In summary, the experimental results confirm that the proposed compact framework, particularly the permutation importance-based XGBoost model, substantially improves

packing identification accuracy compared with existing state-of-the-art approaches, demonstrating strong practical applicability for cybersecurity analysts.

The current framework relies exclusively on static analysis features, which may be vulnerable to sophisticated packing techniques designed to evade such detection. To overcome these limitations, future work will focus on integrating dynamic analysis to complement static indicators, thereby enhancing the framework's overall robustness and adaptability to evasive malware behaviors.

V. Conclusion

This study makes a significant contribution by introducing a large, publicly available, labeled dataset of packed malware samples—an essential resource not previously accessible to the research community. It also presents a multi-step framework that integrates static analysis with machine learning to detect and classify packing in malware. The framework identifies 20 key features from a pool of 125 proposed in prior studies using feature selection algorithms, and categorizes packed malware into three classes (i.e., well-known packed, custom packed, and not packed).

In Step 1, the framework employs decision tree, random forest, extra trees, and XGBoost models to rank feature importance based on Gini impurity and permutation importance. From this analysis, 20 features were selected for optimal classification performance, as illustrated in Fig. 6(h). Notably, the XGBoost model trained on these features via permutation importance outperformed all other models, including those used in earlier research (Table 7 and Fig. 7a). Ensemble-based models also demonstrated superior accuracy compared with single-tree approaches.

In Step 2, the framework achieved strong classification performance, with an accuracy of 99.27%, an F1-score of 98.84%, and an AUC of 99.96%, when distinguishing between the three packing categories. In Step 3, the framework successfully identified packers associated with well-known packed samples, confirming its practical utility for analysts seeking to detect and characterize packed malware, especially in cases with limited packer data.

Overall, the proposed approach highlights the importance of addressing packing and obfuscation techniques in the development of machine learning models for malware detection. By offering a step-by-step method that not only classifies packed binaries but also identifies their associated packers, this framework provides a solid foundation for future advancements in automated malware analysis.

Despite its demonstrated performance, the current implementation focuses solely on single-layer packing. Future work will extend the framework to handle repacking and multi-layer packing, incorporating a specialized unpacking module and strategies to counter anti-debugging techniques. For deeply nested or heavily obfuscated samples, integrating dynamic analysis (e.g., API tracing, sandbox execution, or lightweight debugging) will be critical to improving resilience. However, dynamic analysis introduces engineering challenges, including the need to neutralize anti-debugging tactics (e.g., process environment block or heap flag checks) and to randomize sandbox environments to defeat anti-VM probes.

To address these issues, future research will pursue a hybrid, multi-layered detection architecture that combines static and dynamic analyses. In this envisioned design, static screening will serve as the primary detection stage, triggering sandbox execution under model uncertainty and integrating both static and behavioral scores through late fusion. This adaptive framework aims to enhance robustness against packing-enabled evasion and contribute to more reliable malware detection in real-world cybersecurity operations.

Appendix A: Distribution of Well-Known Packer

Table A1. Distribution of well-known packers

Packer	# of Files
UPX	6748
Netopsystems	4482
Morphine v1.2 (DLL)	3131
asprotect	2645
ASPack	1981
AHTeam EP Protector	987
PECompact	878
SafeGuard	675
tElock	596
MPRESS	571
VMProtect	516
FSG	297
Ste@lth PE	250
PeStubOEP	240
ACProtect	200
Packman	169
Themida	164
Upack	161
MEW	158
RLPack	145
kryptor 6	132
Enigma	93
NsPack	85
Armadillo	74
Petite	57
Fish Pe Packer	42
InstallShield	38
카 v1.0 -> bbb	33
PE Diminisher	29
ExeShield	26
Safengine Shielden	19
MoleBox	18
ZProtect	15
PKLITE32	13
Hide&Protect	13
W32.Jeefo	12
Crunch/PE	12
HuiGeZi	11
NeoLite v2.0	11
UNKNOWN	10
PESpin	10
PE Pack	10
eXPressor	9
CreateInstall	9
KGB SFX	8
PEBundle	7
Obsidium	7
Embed PE	7
yoda's Crypter	6
PEEncrypt	6
Crinkler	6
Cexe	6
Program Protector	5
PEQuake	5
PC Guard for Win32	5
EXE Cryptor	5
ZealPack	4
WWPack	4
kkrunchy	4
GP-Install	4
Krypton	4

yoda's Protector	3
PECrc32 0.88	3
Feokt	3
EZIP	3
CRYPToCRACk's PE Protector	3
SVKP	3
XPack V0.98	2
VIRUS-I-Worm.KLEZ	2
Silicon Realms Install Stub	2
PE Crypt32	2
Pack Master	2
North Star PE Shrinker	2
GameGuard	2
EXEStealth	2
EXE Shield	2
BeRoEXEPacker	2
dUP2	2
Crypto-Lock	2
Blade Joiner	2
ANDpakk2	2
XComp	2
SafeDisc/SafeCast	2
nPack	2
XJ/XPAL -> LiNSoN	1
Virogen Crypt v0.75	1
AHPack	1
VBOX v4.2 MTE	1
Unnamed Scrambler	1
SuperDAT	1
RPoly crypt	1
RCryptor	1
PeX v0.99	1
NTPacker	1
KByS	1
HPA	1
HA Archive	1
Gleam	1
eXcalibur	1
DBPE vx.xx	1
BlackEnergy DDoS Bot Crypter	1
* PseudoSigner	1
Stone's PE Encrypter	1
Spalsher	1
Software Compress	1
SoftSentry	1
SimplePack	1
Sality.Q	1
Reg2Exe 2.24	1
PUNiSHER	1
PE-Armor	1
JDPack	1
iPBProtect v0.1.3	1
hying's PEArmor	1
Freshbind	1
ExeJoiner	1
EXE32Pack	1
D1S1G	1
Crunch/PE	1

Conflict of Interest

The authors declare no conflicts of interest.

Author Contributions

J. W. Kim wrote and prepared the manuscript, and experimented the overall framework. Y. S. Moon supervised and conceptualized of the framework. M. J. Choi reviewed and edited the manuscript. All authors had approved the final version.

Funding

This research was supported by a grant of the Korea Health Technology R&D Project through the Korea Health Industry Development Institute (KHIDI), funded by the Ministry of Health & Welfare, Republic of Korea (grant number: HI23C0733).

References

[1] Cisco. (Feb. 2019). Defending against today's critical threats. *Cisco Cybersecurity Series 2019.* [Online] Available: https://www.cisco.com/c/dam/global/en_uk/assets/pdfs/en_cybersecurityseries_thrt_01_0219_r2.pdf

[2] J. W. Kim, J. Bang, and M. J. Choi, "Defeating anti-debugging techniques for malware analysis using a debugger," *Advances in Science, Technology, and Engineering Systems Journal*, vol. 5, no. 6, pp. 1178–1189, Dec. 2020.

[3] X. Chen, J. Andersen, Z. M. Mao *et al.*, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *Proc. 2008 IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN)*, 2008, pp.117–186.

[4] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-VM technologies," *Black Hat,* vol. 1, pp. 1–27, July 2012.

[5] T. Muralidharan, A. Cohen, N. Gerson, and N. Nissim, "File packing from the malware perspective: Techniques, analysis, approaches, and directions for enhancements," *ACM Computing Surveys*, vol 55, no. 5, pp. 1–45, Dec. 2022.

[6] A. I. Elkhawas and N. Abdelbaki, "Malware detection using opcode trigram sequnce with SVM," in *Proc. the 2018 26th International Conference on Software, Telecommunications, and Computer Networks (SoftCOM)*, 2018, pp. 1–6.

[7] H. Zhang, X. Xiao, F. Mercaldo *et al.*, "Classifiaction of ransomware families with mahcine learning based on N-gram of opcodes," *Future Generation Computer Systems*, vol. 90, pp. 211–221, Jan. 2019.

[8] D. Gibert, N. Ttosis, C. Patsakis, Q. Le, and G. Zizzo, "Assessing the impact of packing on static machine learning-based malware detection and classification systems," *Computer & Security,* vol. 156, 2025.

[9] M. Kearns and M. Li, "Learning in the presence of malicious errors," in *Proc. 20th Annual ACM Symposium on Theory of Computing*, 1988, pp. 267–280.

[10] P. Auer, "Learning nested differences in the presence of malicious noise," *Theoretical Computer Science*, vol. 185, no. 1, pp. 159–175, Oct. 1997.

[11] B. Rahbarinia, M. Balduzzi, and R. Perdisci, "Exploring the long tail of (malicious) software downloads," in *Proc. the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017, pp. 391–402.

[12] M. Morgenstern and H. Pilz, "Useful and useless statistics about viruses and anti-virus programs," in *Proc. CARO Workshop*, 2010, pp. 653–656.

[13] X. U. Pedrero, D. Balzarotti, I. Santos *et al.*, "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *Proc. 2015 IEEE Symppsium on Security and Privacy*, May 2015, pp. 659–673.

[14] M. Bat-Erdene, T. Kim, H. Park, and H. Lee, "Packer detection for multi-layer executables using entropy analysis," *Entropy*, vol. 19, no. 3, 125, 2017.

[15] W. Fleshman, E. Raff, R. Zak *et al.*, "Static malware detection & subterfuge: Quantifying the robustness of machine learning and current anti-virus," in *Proc. 13th International Conference on Malicious and Unwanted Software (MALWARE)*, 2018, pp. 1–10.

[16] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *Proc. International Conference on Learning Representations (ICLR)*, 2015, pp. 1–11.

[17] D. Zügner, A. Akbarnejad, and S. Günnemann, "Adversarial attacks on neural networks for graph data," in *Proc. 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 2847-2856.

[18] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Proc. 2017 IEEE Symposium on Security and Privacy*, 2017, pp. 39–57.

[19] X. Li, K. Qiu, C. Qian, and G. Zhao, "An adversarial machine learning method based on opcode N-grams feature in malware detection," in *Proc. 2020 IEEE 5th International Conference on Data Science in Cyberspace (DSC)*, 2020, pp. 380–387.

[20] E. O. Osaghae, "Classifying packed programs as malicious software detected," *Information Technology & Electrical Engineering*, vol. 5, pp. 22–25, Dec. 2016.

[21] R. Perdisci, A. Lanzi, and W. Lee, "Classification of packed executables for accurate computer virus detection," *Pattern Recognition Letters*, vol. 29, no. 14, pp. 1941–1946, Oct. 2008.
[22] Z. Zhang, C. Chang, P. Han *et al.*, "Packed malware variants detection using deep belief networks," in *Proc. MATEC Web of Conferences*, vol. 309, 2020.
[23] F. Biondi, M. A. Enescu, T. Given-Wilson *et al.*, "Effective, efficient, and robust packing detection and classification," *Computers & Security*, vol. 85, pp. 436–451, Aug. 2019.
[24] L. H. Park, J. Yu, H. K. Kang *et al.*, "Birds of a feature: Intrafamily clustering for version identification of packed malware," *IEEE Systems Journal*, vol. 14, no. 3, pp. 4545–4556, 2020.
[25] D. Vasan, M. Alazab, S. Wassan *et al.*, "Image-based malware classification using ensemble of CNN architectures (IMCEC)," *Computer & Security*, vol. 92, 101748, 2020.
[26] E. Alkhateeb, A. Ghorbani, and A. Lashkari, "Identifying malware packers through multilayer feature engineering in static analysis," *Information*, vol. 15, no. 2, 102, 2024.
[27] J. Ramamoorthy, N. K. Shashidhar, and C. Varol, "Packers and features: Efficacy of static analysis for packed linux malware," in *Proc. 2025 13th International Symposium on Digital Forensics and Security*, 2025, pp. 1–6.
[28] M. D. Gennaro, M. D'Onghia, M. Polino *et al.*, "PackHero: A scalable graph-based approach for efficient packer identification," in *Proc. International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2025, 253–274.
[29] N. Z. Zacharis, "Classification and regression trees (CART) for predictive modeling in blended learning", *International Journal of Intelligent Systems and Applications (IJISA)*, vol. 10, no. 3, pp. 1–9, 2018.
[30] S. L. Crawford, "Extensions to the CART algorithm," *International Journal of Man-Machine Studies*, vol. 31, no. 2, pp. 197–217, 1989.
[31] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and Regression Trees*, Florida, USA: CRC Press, 1984.
[32] F. Pedregosa, G. Varoquaux, A. Gramfort *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
[33] Keras. (Jan. 2021). Developer guides. [Online]. Available: https://keras.io/guides/
[34] E. Carrera. (Dec. 2020). Pefile. [Online]. Available: https://github.com/erocarrera/pefile
[35] H. Neil. (2008). PEiD. [Online]. Available: https://github.com/wolfram77web/app-peid
[36] M. J. Choi, J. Bang, J. Kim *et al.*, "All-in-one framework for detection, unpacking, and verification for malware analysis," *Security and Communication Networks*, vol. 2019, 5278137, 2019.