IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received February 29, 2020, accepted March 25, 2020, date of publication April 6, 2020, date of current version May 7, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2986014

Comparative Analysis of Low-Dimensional Features and Tree-Based Ensembles for Malware Detection Systems

SEOUNGYUL EUH[1], HYUNJONG LEE[1], DONGHOON KIM[2], AND DOOSUNG HWANG[3]
[1]Security Technology Institute, KSign, Seoul 06231, South Korea
[2]Department of Computer Science, Arkansas State University, Jonesboro, AR 72467, USA
[3]Department of Software Science, Dankook University, Yongin 16890, South Korea

Corresponding author: Doosung Hwang (dshwang@dankook.ac.kr)

This work was supported by the Institute for Information and Communications Technology Promotion (IITP) grant funded by the Korean Government (MSIT) (Trust-Based Cyber Security Platform for Smart Facility Environment) under Grant 2019-0-00197.

ABSTRACT Advances in machine learning algorithms have improved the performance of malware detection systems for the last decade. However, there are still some challenges such as processing a large amount of malware, learning high-dimensional vectors, high storage usage, and low scalability in learning. This paper proposes low-dimensional but effective features for a malware detection system and analyzes them with tree-base ensemble models. Expert knowledge and frequency analysis are adapted for relevant feature selection from the collected data set, which contributes to fast low-dimensional feature preparation, low storage usage, and fast learning. We extract the five types of malware features represented from binary or disassembly files. Specifically, the novel WEM (Window Entropy Map) image is designed to represent malware with variable length, and the set of frequently used APIs is analyzed to shorten the processing time. To validate the effectiveness of the selected features, we compare the performance of tree-based ensemble models such as AdaBoost, XGBoost, random forest, extra trees, and rotation trees. The proposed feature can reduce the original feature dimensionality by several tens to hundreds of times and decrease the training time of ensemble models without degrading the malware detection rate when compared to the performance of the whole set of malware features. In accuracy and AUC-PRC evaluation, XGBoost is the highest in rank.

INDEX TERMS Malware detection, feature extraction, tree-based ensemble, AUC-PRC.

I. INTRODUCTION

Malware (malicious software) is any program or file that is intended to damage computers, computer systems, or networks. The malware performs a type of misbehavior that goes against the benefits of users after it is implanted into computers. Types of malware include viruses, worms, trojans, rootkits, spyware, ransomware, etc. Malware infection has been increasing over the years. There have been many endeavors to prevent malware attacks and the spread of infection to other computers, but it is difficult to deal with advanced malware variants, such as polymorphism [1]–[4], packing [5], obfuscation [6], etc.

A malware detection system utilizes known detective patterns to verify whether a new application becomes a threat. The set of detective patterns are collected by analyzing the previously-known malware. The fast growth of malware variants nullifies malware detection systems based on these known patterns. Commercial antivirus software has difficulty detecting new malware variants unless the software is kept up-to-date. To resolve these difficulties, machine learning techniques have been applied in malware detection systems [2], [7]. Static analysis using machine learning algorithms is able to detect some parts of polymorphism or obfuscation code that can appear as patterns in sequence or 2-dimensional image [2], [8].

The general workflow of the machine learning process includes the following steps: data collection, feature extraction, model training, and model selection. The data collection step collects malware and benign files. The feature extraction step decides a suitable representation for malware vectorization and prepares the training dataset. The model training

The associate editor coordinating the review of this manuscript and approving it for publication was Ahmed Farouk.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

step makes use of the training dataset to yield the learned models for choosing the best model. In the model selection step, the best learned model is selected and applies to a real application.

Feature extraction is the important step of machine learning process [7], [9]. It prepares feature vectors for representing the characteristics of malware because it is associated with the general performance of a machine learning algorithm. Leaning performance often depends on the types of extracted features. The step of feature preparation is considered time-consuming in terms of selected feature types. However, the feature vector preparation can be done in parallel.

Malware feature extraction is conducted with static or dynamic analysis [1], [2]. Static analysis examines executable or disassembled files without execution. A PE (Portable Executable) file is composed of headers and sections that tell the dynamic linker how to map the file into memory. IAT (Import Address Table) in PE can be analyzed to utilize DLL information [1], [2], [10]. The chosen features are byte sequence [11], [12], byte entropy [11], *n*-gram, opcode (operational code) [8], [13], API sequence [10], [14], [15], etc.

Dynamic analysis observes malware behavior while malware is being executed in a virtual environment, such as CWSandbox [16] or Cuckoo Sandbox [17]–[19]. In a virtual environment, the malware attack cannot cause damage to a system due to a controlled execution. The virtual environment monitors the changes in network, registry, MFT (Master File Table), and the behavior of processes. Additionally, the virtual environment records log files. Dynamic features are the API call sequence and arguments, monitored processes, registry changes, mutex changes, etc. These kinds of features require computationally intensive operations due to the need of a virtual environment. However, they are able to detect obfuscated malware while static features are vulnerable to it [2], [10], [20].

Machine learning models require persistent malware analysis against increasing malware variants. Malware detection systems are being developed with machine learning techniques to reduce FPR (False Positive Rate) of signature-based malware detection techniques. In recent years, tree-based ensemble algorithms are one of most important methods among all the ensemble algorithms for solving prediction and classification [21], [22]. Decision trees have less impact on learning time than SVM, despite the growth of training data. However, a decision tree with high depth has the disadvantage of being overfit. Ensemble learning algorithms can prevent overfitting with the bias-variance analysis. Thus, the tree-based ensemble algorithms are considered for this study. Through an ensemble approach, the experimental results show that the performance of malware detection systems can be improved by learning numerous classifiers and combining their outputs.

In summary, this paper makes the following contributions:

- We propose updated malware features that minimize the drawbacks, such as variable length size, high dimension, and high storage use. The proposed malware features show that their overall performance is better than the original training feature in terms of training time and accuracy.
- By applying expertise analysis knowledge, frequently used functions, and entropy discretization to the studied malware features, the variable lengths of malware features can turn into fixed lengths. Such features do not require selecting a fixed length and padding feature vectors.
- We perform extensive experiments with tree-based ensemble algorithms. Experimental results show that the tree-based ensemble model is effective and efficient to detect malware in terms of training time and overall performance.

The paper is organized as follows: Section II presents the related works of malware detection methods in terms of datasets, feature vectors, machine learning algorithms, and performance. Section III discusses gram matrix, entropy feature, and API or DLL feature that characterize malware. Section IV presents the experimental results and Section V concludes our work.

II. RELATED WORKS

The evaluation of malware learning models uses known datasets or self-collected datasets. Training datasets are collected from Web databases and composed of binary classification or malware family classification. Table 1 summarizes the malware datasets in terms of class size, malicious and benign size, adopted feature, learning model, and data source.

Various *n*-gram features of opcode are selected to represent malware and benign [8], [13], [23]. Learning models were tested for TF and TF-IDF (Term Frequency - Inverse Document Frequency [35]) features of 2-gram opcode [23]. Random forest achieved the highest accuracy of 0.95 for TF feature and *n*-gram feature was optimal if $n = 2$. The 2-gram and weighted term frequency features were tested for learning models [13]. SVM of polynomial kernel showed the best accuracy of 0.96. A major block of opcode was chosen to avoid a high dimensional feature when including all the opcode, which lowered feature extraction time [8]. The selected opcode was transformed into a square image and tested CNN (Convolution Neural Network [36]). Since the length of the selected opcode differs in malware, a hash function was applied to opcode and transformed hash values into a squared image. The detection accuracy of 0.87 or more was reported for all classes.

The ANN (All-Nearest-Neighbor) algorithm was introduced to detect malware based on sequential patterns from PE headers [26]. The results showed a detection rate of 96.0%. Each training feature was vectorized from PE sections with byte entropy, histogram, and imported functions and meta data [11]. The dimension of each feature was 256 and the neural network structure was $1024 \times 1024 \times 1024 \times 1$. The result ranged from 67.0 to 95.0% and the highest rate was 95.2%. The histogram similarity analysis [37] was adapted

TABLE 1. List of malware detection systems in the studies.

Studied	Class	Total	Malware	Benign	Type	Feature	Model	Source
Shabtai [23]	2	26,093	5,677	20,416	static	2-gram of opcode	Random forest	VXHeavens [24]
Santos [13]	2	2,000	1,000	1,000	static	2-gram of opcode	SVM	VXHeavens [24]
Ni [8]	9	10,868	10,868	-	static	opcode	CNN	Malware Challenge [25]
Fan [26]	2	10,307	8,847	1,460	static	executable	Neural network	VXHeavens [24]
Saxe [11]	2	431,926	350,016	81,910	static	executable	CNN	VirusTotal [27]
Han [12]	50	1,000	1,000	-	static	entropy	Histogram analysis	VXHeavens [24]
Sun [28]	12	2,798	2,798	-	static	executable	Random forest	VirusShare [29]
Ahmadi [30]	9	10,868	10,868	-	static	executable	XGBoost	Malware Challenge [25]
Ki [15]	2	23,388	23,080	308	dynamic	API sequence	DNA sequence matching	Malica-project [31] VirusTotal [27]
Hansen [19]	5	270,837	270,000	837	dynamic	API & DLL sequence	Random forest	VXHeaven [24] VirusShare [29]
Burnap [17]	2	1,188	594	594	dynamic	modified resource data	SOFM+distance	VirusTotal [27]
Rieck [16], [32]	24	3,133	3,133	-	dynamic	MIST	Clustering+NN	Malheur [33]
Stiborek [34]	2	231,255	144,229	87,026	dynamic	modified resource data	Random forest	VirusTotal [27]
This work	2	122,963	102,963	20,000	static	opcode, API, DLL, entropy	Tree-based ensemble	Malwares

to detect malware family by comparing the peak points of the byte entropy graph [12]. The malware family detection rate was about 98.0% with threshold 0.75. When the static feature was gathered from bytecode features, disassembly code, and PE features, the performance of random forest was the highest and the F1-ratio was 93.56% [28]. XGBoost learned multiple byte-based features from registry, keyword frequency, n-gram, entropy, and byte images [30]. The accuracy was 0.98 for the entropy feature and 0.99 for the keyword frequency feature.

The DNA sequence matching method was applied to extract LCS (Longest Common Subsequence) from API call sequences [15]. The LCS subsequences were collected from malware only and excluded patterns that appeared in benign. They reported near 99.9% generalization performance. Cuckoo Sandbox was used to capture API call sequences, DLL call sequences, and the presence or absence of string information [19]. IGR (Information Gain Ratio) monitored the feature dimension for feature fusion process. Random forest achieved 0.996 in AUC for malware detection and 0.978 in AUC for malware family classification. Under Cuckoo Sandbox, malware features were represented with resource related data such as file access log, registry key access, process execution, packet log, CPU, and memory [17]. Self-organizing feature map (SOFM) was trained with malware features and used to predict the cluster of test data [29]. The detection accuracy of about 0.9 was the best when the feature map size was 80×80.

Malicious behavior was encoded with resource data such as files, mutexes, registry keys, network traffic, and error messages [34]. Random forest reported TPR (True Positive Rate) of 95.0%. They claimed three limitations of dynamic malware analysis. In absence of malware behavior, dynamic analysis cannot characterize malicious behavior. More and more malware has evolved along with anti-sandbox functionality. Lastly, due to high FPR (False Positive Rate), malware can be installed easily in the system directory without user interactions. To overcome these limitations, FPR should be lowered if training data containing malicious behavior is prepared.

MIST (malware instruction set) encoded malware behavior as a sequence of instructions captured under CWSandbox [32]. Their fusion learning model grouped training data into similar behavior and predicted malware into behavioral classes by the nearest-neighbor algorithm [16]. The results showed that F-measure was 0.94 to 0.99 when the MIST level was 1 or 2.

The malware feature of this study uses static analysis and proposes the modification of opcode, API, DLL, and entropy features. The modification method utilizes dimension reduction to represent a fixed-length malware feature, which is essential for applying machine learning algorithms. Various tree-based ensemble algorithms are chosen to model malware detection systems and their results are analyzed for practicability.

III. MALWARE DETECTION MODEL

The malware detection system consists of three steps as in Figure 1: feature engineering, model learning and model evaluation. The feature engineering step prepares training sets based on byte, opcode, and API data from the collected dataset. `Radare2`[1] is used to generate disassembly code. For the model learning step, tree-based ensemble models are used for our malware detection models. The model evaluation step chooses the best model through cross-validation and confusion matrix.

A. DATA COLLECTION

Malware files were collected from `Malwares`.[2] 18 malware families are listed as shown in Table 2. Adware is the most frequent family, and CoinMiner is the rarest family. `Kaspersky`[3] categorizes malware files according to malware behavior. The collected dataset is composed of 122,963 files including 20,000 benign files. Malicious files are labeled with positive class (+1) and benign files with negative class (−1) for a binary classification problem.

[1] https://rada.re/r
[2] http://www.malwares.com
[3] https://usa.kaspersky.com

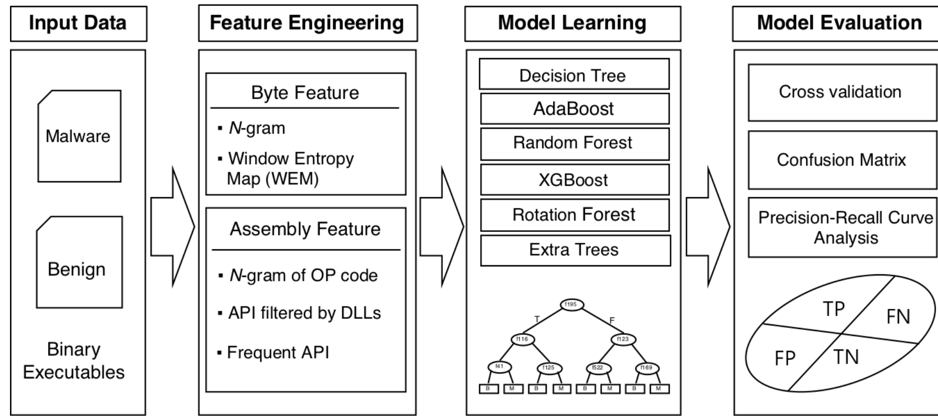


FIGURE 1. Structure of malware detection model.

TABLE 2. Malware family classification.

Family	No. of files	Family	No. of files
Adware	38,162	Malware	29,526
Virus	9,370	Downloader	6,780
Trojware	5,908	Crypto	3,820
Ransomware	1,301	Riskware	3,134
Backdoor	1,072	Bundler	1,058
Packed	871	Injector	772
Worm	371	Dropper	361
HackTool	244	VirTool	128
SpyWare	60	CoinMiner	25
Total			102,963

B. FEATURE ENGINEERING

It is necessary to map training samples to feature vectors for the ensemble algorithm. A vector representation is obtained by defining a numerical measurement method that can replace samples. We focus on static features, mainly from both executable and disassembly code.

1) GRAM MATRIX

Malware M is represented as a sequence of opcodes: $\mathrm{M} =< s_1, s_2, \cdots, s_N >$ and $s_i \in \mathrm{S}$, where N is the total length and S is the set of opcodes. An n-gram presents a contiguous subsequence of n items. By sliding a window of length n over M, the n-gram of the ith index is subsequence $(s_i, s_{i+1}, \cdots, s_{i+n-1})$. The n-gram maps M to a high-dimensional vector space, where each dimension is related to a single gram. The function ϕ returns the n-gram feature for M: $\phi(\mathrm{M}) = \{(g, \mathrm{freq}(g|\mathrm{M}))|g \in \mathrm{G}\}$, where G is the set of unique n-grams and the function freq$(g|\mathrm{M})$ returns the frequency, the probability, or the binary flag of g in M.

The gram features are widely used in natural language processing and DNA sequence analysis [35]. Although n-gram feature provides the effective representation of malware, the exponential growth of feature dimension suffers from time complexity [38]. The total number of unique byte n-grams becomes 2^{8n}. The experimental analysis showed that the appropriate value of n was 2 for opcode features [13], [23].

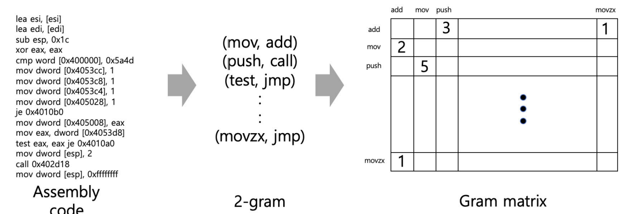


FIGURE 2. Example of 2-gram matrix.

TABLE 3. List of the 32 most frequent opcodes.

Opcode	Frequency	Ratio	Opcode	Frequency	Ratio
add	1,238,931,582	29.71%	and	49,905,245	1.20%
mov	550,713,111	13.21%	or	47,802,366	1.15%
push	344,826,210	8.27%	ret	46,823,025	1.12%
call	152,105,975	3.65%	jne	46,707,670	1.12%
pop	144,945,283	3.48%	adc	36,244,576	0.87%
inc	129,443,348	3.10%	xchg	35,173,936	0.84%
cmp	105,531,460	2.53%	sbb	28,406,613	0.68%
xor	84,539,136	2.03%	imul	22,739,049	0.55%
dec	76,137,878	1.83%	jb	19,823,052	0.48%
je	71,461,717	1.71%	out	17,729,151	0.43%
lea	69,198,544	1.66%	in	16,759,175	0.40%
test	66,718,491	1.60%	jae	14,442,147	0.35%
int	65,810,779	1.58%	outsd	12,954,199	0.31%
jmp	61,262,817	1.47%	outsb	12,149,289	0.29%
sub	56,204,546	1.35%	popal	11,861,022	0.28%
nop	52,559,582	1.26%	movzx	11,812,719	0.28%

The dimension reduction method of malware features was adapted by finding out the frequent relevant opcodes from malware and benign files [13], [39]. Table 3 lists the selected opcodes along with frequency rate. The number of selected opcodes is 32 (i.e., 2^5), so a 2-gram training feature is presented with a 32×32 matrix.

Gram matrix can solve the drawback that the feature vectors of n-gram have different lengths according to file size. It is also represented as a sparse vector which has the benefit of high-dimensional features [40]. Figure 2 is the example of a gram matrix of a binary code.

2) WEM (WINDOW ENTROPY MAP) FEATURE

Entropy has been used to detect malware by measuring the degree of uncertainty from binary files [8], [11], [30]. The maximum entropy appears when the probabilities for all symbols are the same. On the contrary, if the bytes occur with high probabilities, the entropy value will be smaller. Considering

a binary file as hexadecimal time series data, the frequency rate of each byte is mapped to the entropy value measuring the degree of uncertainty. WEM generates a two-dimensional entropy feature from malware.

A byte entropy histogram is computed through sliding a byte window of size ω over hexadecimal sequences with a step size of τ bytes. Let T be the total number of sliding windows in M. Then, M is represented by $\mathrm{M} = < \mathrm{W}_1, \mathrm{W}_2, \cdots, \mathrm{W}_T >$, where W_k is the kth window. The bin entropy is computed by the Shannon entropy, $\mathrm{H(M)} = [h_{k,j}]_{T\times 256}$, where $h_{k,j}$ is the jth bin entropy of W_k. The window entropy map $\mathrm{B} = [b_{l,j}]_{L\times 256}$ is presented through level-wise variation of H(M) and L becomes a quantizing resolution on entropy values. For W_k, the level index l of $b_{l,j}$ is based on the accumulation $c_{k-1,j}$ of the jth bin, where the entropy accumulation is $\mathrm{C} = [c_{k,j}]_{T\times 256}$ and $c_{k,j}$ is the sum of the jth bin entropy from $\mathrm{W}_1, \mathrm{W}_2, \cdots, \mathrm{W}_{k-1}$. The bin entropy $h_{k,j}$ is augmented to $b_{l,j}$ after index l is computed by quantizing $c_{k,j}$ with the predefined Δ. Algorithm 1 is a pseudo code for building the WEM from H(M).

Algorithm 1 Building the WEM Feature of H(M)

```
1:  function BuildWEM(H, T, L, Δ)          ▷ T, L, and Δ are constants.
2:      Declare C = [c_{k,j}]_{T×256} and B = [b_{l,j}]_{L×256}
3:      c_{k,j} = 0 for k = 1, ···, T and j = 1, ···, 256
4:      b_{l,j} = 0 for l = 1, ···, L and j = 1, ···, 256
5:      c_{1,j} = h_{1,j} for j = 1, ···, T          ▷ Initialization
6:      for k = 2, ···, T do                        ▷ Accumulation
7:          for j = 1, ···, 256 do
8:              c_{k,j} = c_{k-1,j} + h_{k,j}
9:          end for
10:     end for
11:     for k = 1, ···, T do                        ▷ Construct WEM
12:         for j = 1, ···, 256 do
13:             l = min(L, ceil(c_{k,j}/Δ))
14:             b_{l,j} = b_{l,j} + h_{k,j}
15:         end for
16:     end for
17:     return B
18: end function
```

When $L = 1$, WEM is similar to the byte entropy feature [11] because $\mathrm{B} = [b_j]_{256}$ is the aggregation of the jth bin entropy over M. Other methods studied use feature vectors that result in changes in entropy or generating hash values in sliding windows of malware and benign [12], [41]. But WEM is focused on the bin entropy which takes advantage of a fixed feature map and level-wise variation regardless of different file size. In addition, as L increases, the quantization level captures the scatter relation of the jth among sliding windows.

Figure 3 shows the 2-dimensional grayscale images (32×16) of WEM for selected malware and benign ($\omega = 1024$, $\tau = 256$, $L = 2$, $\Delta = 0.2$). Figure 3 (a) to (d) are the examples of malware, such as Downloader, Packed, Trojware, and Ransomware, while (e) are the examples of benign.

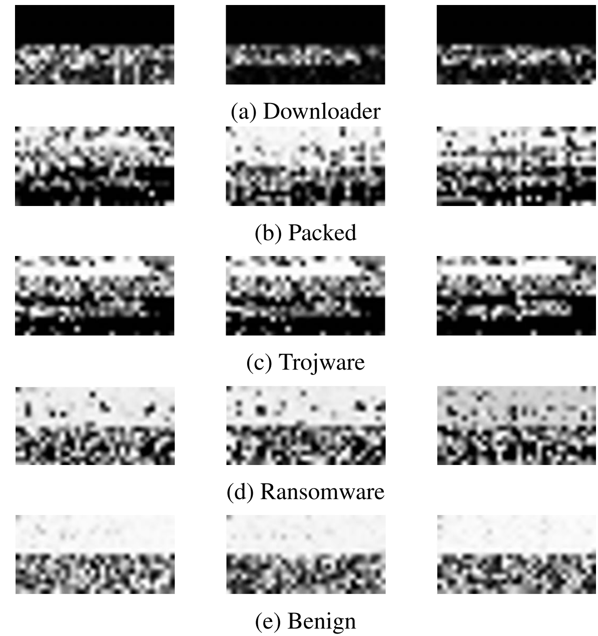


FIGURE 3. Examples of WEMs.

The variants of Downloaders, Trojware, or Ransomware are shown with their own patterns are shown with their own pattern, so the visualization of WEM can be used to identify the members within the same malware family. The examples of Packed malware are liable to change but have their own patterns. The examples of benign also have their own patterns that slightly differ among them.

3) API FEATURES

The analysis of API call sequences provides the information on how malware codes will run or what they will do. Malware or benign are classified by the simple statistics of the frequency of the called APIs. The static analysis is applied to extract API names from PE section and constitute API sequences. The extracted APIs are included within Microsoft Windows DLLs (Dynamic-Link Libraries) or user defined DLLs. API sequences follow the order in which API functions appear in disassembly code.

From the collected database, the number of unique APIs is 147,200: 49,158 for malware and 121,023 for benign. The number of APIs occurring in both malware and benign is 18,056. The API feature of a single binary file must have 18,056 dimensions if all API functions are included.

The frequency of an API is computed by checking whether the feature is selected or not. An API is selected as a final feature if the entropy of the frequency rate is more than 6×10^{-6}. Therefore, the number of selected APIs is 122 for our database. Vyas *et al.* selected 92 APIs extracted from PE section of 1,100 malicious files (i.e., Virus, Backdoor, Worm, and Trojan) and 2,600 benign files [42]. Infosec Institute reported 131 APIs that were commonly encountered by malware analysis [43]. Table 4 shows the list of APIs frequently used in malware. From three studies, we choose

TABLE 4. Comparison of frequent APIs.

Reference	No. of APIs
Rushabh *et al.* [42]	92
Inforsec Institute [43]	131
Selected APIs	122
Total of unique APIs	195

TABLE 5. Categorical comparison of frequent API functions.

Category	Count	Category	Count
Network	38	Hook	2
Device	1	Synchronization	3
Keyboard	3	File I/O	21
System	18	Windows	2
Services	7	Registry	30
Crypto	3	DLL	5
Screenshot	2	Persistent	1
PrivilegeEscalation	2	Process	25
Memory	15	Thread	17
Total			195

the 195 unique APIs as the final API feature. The selected APIs are analyzed with the API categories as in Table 5.

In the collected dataset of 122,963 files, the number of distinct DLLs is 4,244, consisting of 1,276 malware files and 3,415 benign files. Thus, the number of DLLs imported for malware is less than that of the benign. The high dimension of API or DLL features is reduced by selecting the features which are frequently used in malware.

A dynamic link library (DLL) in Microsoft Windows operating system calls functions by importing other DLL files. For example, malicious codes utilize some functions in `kernel32.dll` that handles memory, process, and thread. Window executable files are not necessary to call functions in `ntdll.dll` directly because `kernel32.dll` imports `ntdll.dll`. Malicious codes often include DLL functions implemented by developers. However, DLL files implemented by developers are excluded because common APIs provided by system software are only considered to detect malware.

Table 6 shows the top 20 DLLs with more than 0.2% in frequency. Based on the selected DLLs whose frequency is above 0.3%, we transform their functions defined within DLLs into feature vectors. For the collected database, the number of DLLs is 2,054 and the number of distinct DLL APIs is 3,842. Therefore, the dimension of DLL API features consists of 3,842 DLL APIs.

C. EVALUATION METHOD

Our malware detection system was evaluated with a confusion matrix as shown in Table 7. Accuracy and error rates are used to measure the performance of malware detection systems. However, they cannot give an overview of the range of performance with varying thresholds. When a single threshold divides test examples into malware or benign class, it is not obvious how the right threshold value should be chosen. Therefore, threshold-free measures, such as Receiver Operating Characteristic (ROC) and Precision Recall Curve (PRC) plots [44], are selected when comparing the generalization performance of malware detection system.

$$\text{accuracy} = \frac{TP + TN}{TP + FN + FP + TN}$$
$$\text{precision} = \frac{TP}{TP + FP}$$
$$\text{recall} = \frac{TP}{TP + FN}$$

ROC plots reveal a tradeoff between specificity and sensitivity while PRC plots present a tradeoff between precision and recall [44], [45]. Recall is the fraction of correctly predicted examples among malware predictions. An integral score of the area under the ROC (AUC-ROC) represents the performance of a classification model. Similarly, AUC-PRC becomes a useful metric for comparing a classifier along with precision and recall by shifting the decision threshold of the classifier.

Class imbalance phenomenon appears in the collected dataset whose ratio of benign to malware class is about 1 to 5. For class imbalance problems, an estimate of the number of wrong predictions among the positively classified instances is of great importance [46]. The investigation of AUC-ROC in an imbalance problem can be misleading in connection with the reliability of malware detection analysis, owing to a different interpretation on FPR [45]. From our point of view, PRC is more appropriate than ROC because PRC plots reflect the fraction of correctly classified examples among ones predicted as malware. Therefore, the PRC analysis is chosen as a direct and intuitive measure for our malware detection system.

IV. EXPERIMENTS

The feature extraction methods and tree-based ensemble models are implemented with Python and operate as pipelined modules. Tree-based ensemble models are more flexible, and less data-sensitive, than a single tree model [47]. The chosen ensemble algorithms are random forest [48], AdaBoost [49], XGBoost [50], extra trees [51], and rotation forest [52]. The decision tree algorithm is chosen as a base classifier.

We consider the feature extraction an *embarrassingly parallel* problem in which each feature type can be processed independently for every file. Our malware detection system embeds the tree-based ensemble models of the `scikit-learn` framework [53] and the modules of the mentioned training features. The learned models are analyzed with performance measures such as accuracy, precision and recall, and AUC-PRC.

A. DATA PREPARATION

A parallel processing platform based on Hadoop Distributed File System (HDFS) consists of 10 computers. These computers are low cost and uses less memory than the latest high-end personal computers. All the nodes run on Linux Mint

TABLE 6. List of the 20 most frequent DLLs.

Rank	Name	Frequency	Ratio	Description
1	KERNEL32.DLL	10,809,740	42.41%	Windows operating system kernel library
2	USER32.DLL	7,275,994	28.55%	Windows user interface API library
3	GDI32.DLL	1,700,784	6.67%	Graphics Device Interface functions for output to video displays and printers
4	OLEAUT32.DLL	1,024,992	4.02%	OLE object handling library
5	ADVAPI32.DLL	987,559	3.87%	Library for manipulating the registry
6	MSVBVM60.DLL	753,254	2.96%	Component of Windows that enables users to run app. Written in VB programming language
7	OLE32.DLL	383,281	1.50%	Core OLE library
8	COMCTL32.DLL	328,508	1.28%	Standard Windows controls library (File open, save, save as dialogs, progress bars, list view, etc.)
9	WS2_32.DLL	238,104	0.93%	Window socket library
10	MSVCRT.DLL	208,510	0.82%	Standard C library
11	SHELL32.DLL	180,670	0.71%	Windows Shell Library
12	QT5CORE.DLL	175,906	0.69%	C++ application development framework
13	WININET.DLL	96,284	0.38%	Internet-related library
14	GDIPLUS.DLL	91,964	0.36%	Resource library that belongs to Windows GDI(Graphic Display Interface)
15	WSOCK32.DLL	81,197	0.32%	Windows Sockets API
16	SHLWAPI.DLL	68,631	0.27%	Library which contains functions for UNC and URL paths, registry entries and color settings
17	MFC42.DLL	63,653	0.25%	Module that contains the MFC functions used by applications created in Visual Studio
18	VERSION.DLL	52,658	0.21%	Module that contains API functions used for Windows version checking
19	CLARUN.DLL	51,916	0.20%	Clarion 8.0 RuntimeLibrary
20	C60RUNX.DLL	49,969	0.20%	Commands for starting Clarion 6.3 build Clarion 6.300.00

TABLE 7. Confusion matrix.

True class	Predicted	
	Malware (+1)	Benign (-1)
Malware (+1)	True Positive (TP)	False Negative (FN)
Benign (-1)	False Positive (FP)	True Negative (TN)

18.2 Sonya. The master node works as a slave node at the same time. The tested feature vectors are as follows:

1) 2-gram is a sequence vector in hexadecimal.
2) 2-gramM is a gram matrix made of frequent opcodes from disassembly files.
3) API-DLL is prepared from frequent Windows APIs within imported DLL files.
4) API is made up of frequent Windows APIs in assembled codes.
5) WEM is a two-dimensional matrix with binary entropy values in a binary file ($L = 2$).

Figure 4 compares the number of distinct APIs along with the number of files. The number of APIs (ALL-API) increases rapidly up to 20,000 files, and the number of APIs is about 14,000. Later, as the number of files increases, the number of APIs grows slowly. From the collected dataset, the dimension size of ALL-API is approximately 18,056. This high dimensionality is reduced by the frequency analysis, so the API-DLL dimension is 3,842 and the API dimension is 195. If the number of files exceeds 100,000, the change in feature dimension for the API and API-DLLs is much smaller than for ALL-API. This result provides the insight on how many malware and benign files are collected for a malware detection system.

Figure 5 compares the processing time with the number of workers on a log scale for the y-axis. Processing time is reduced quickly by 4 workers. However, there is little change in the processing time hereafter. The processing time of WEM requiring a lot of entropy calculations is the longest, and the preparation time of 2-gramM, API-DLL, and API is about 4.4 times faster. We observed that the processing time was not

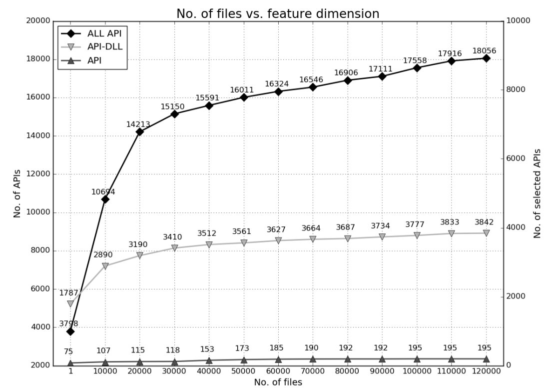


FIGURE 4. Comparison of the number of files versus feature dimension.

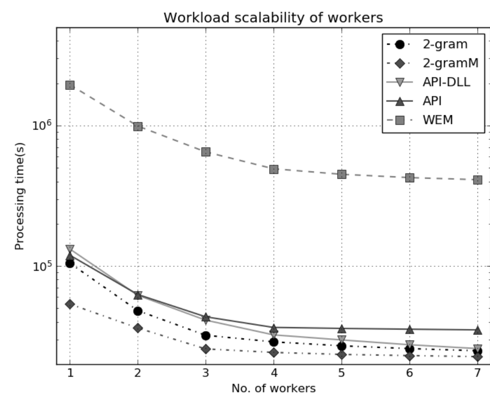


FIGURE 5. Compare of processing time.

improved with more than 6 working nodes. It is the processing latency that causes communication overhead in the cluster.

Table 8 compares feature extraction time as the number of cluster nodes increases. The 2-gram is represented by 61,952 dimensions but its feature vector is highly sparse. The proposed feature analysis decreases the dimensionality of 2-gramM, 2-gramAPI, API-DLL, and API except WEM. While both WEM and API have about 500 or less dimensions, API-DLL has about 3,800 dimensions.

TABLE 8. Time comparison for feature extraction.

Feature	No. dim.	Processing Time(sec)				
		1 node	2 nodes	3 nodes	4 nodes	5 nodes
2-gram	61,952	104,823	48,147	32,127	28,842	27,036
2-gramM	1,024	288,387	168,732	115,761	88,668	72,783
API-DLL	3,842	132,099	61,899	41,181	32,343	29,772
API	195	119,343	62,685	43,491	36,531	35,955
WEM	512	1,953,195	991,593	649,293	492,138	449,625

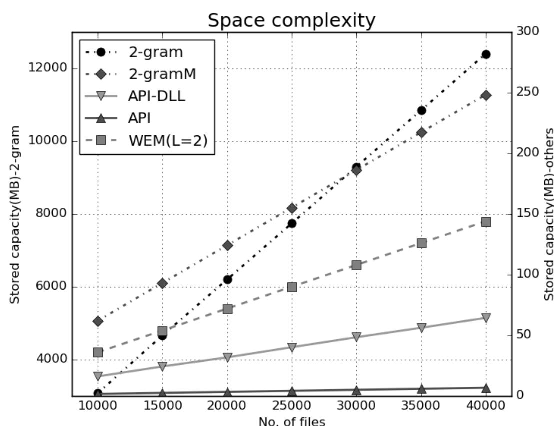


FIGURE 6. Comparison of space complexity.

Figure 6 compares the space complexity with the number of files. The storage size grows linearly as the number of files increases. The 2-gram feature requires the highest storage amount while the API feature needs the smallest amount.

B. PREDICTION ANALYSIS BY PROXIMITY MEASURE

After learning the ensemble model for the training dataset, we compare the similarity or similarity by calculating the proximity between two instances [54], [55]. The proximity measure of two instances is the ratio of the number of identical terminal nodes to the total number of terminal nodes within the ensemble. As the proximity value closes to 1, their classes are predicted at the same terminal node and the classification of the trained model is similar. However, if it is close to zero, the terminal nodes reached by the two instances are different.

Proximity measurements depend on the depth and number of trees in the ensemble. Proximity is used to analyze the results of tree-based ensembles in which instances are trained, even if the instance dimensions are larger [55]. The proximity matrix can be visualized by projecting each instance into d-dimensional space where the proximity value between any pair of instances is considered their distance. Proximity for the entire training instance constitutes 2-dimensional matrix, and Multidimensional Scaling (MDS) visualizes the dataset by orthogonally transforming the proximity matrix onto two eigenvectors with the highest eigenvalues [56].

Figure 7 shows an example of a proximity plot for the WEM feature. The plot was generated by using the MDS implementation of `scikit-learn`. The dataset consists of 2,000 instances per class and the proximity value is projected onto 2-dimensional space after a random forest model.

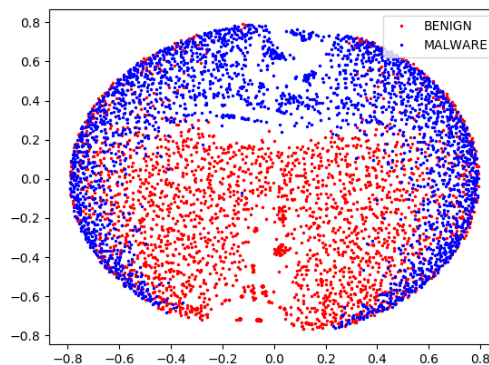


FIGURE 7. Proximity plot for the prepared dataset presented by WEM.

Benign instances have a distribution in which clusters are clearly concentrated in the center region. Malware instances appear in a distribution that surrounds benign clusters, with some instances intermingled. It is appropriate to set a high tree depth for decision trees for benign instance classification in intermingled regions. In general, higher tree depths in decision trees are more likely to result in overfitting. Therefore, the ensemble learning model is more suitable than the single decision tree model for this malware detection problem.

C. PERFORMANCE COMPARISON

The experiments were conducted with 10 separate 5-fold cross-validations. For each validation, we randomly split the data into five equally sized sets; four sets were for training and the other set was held out for testing. Each dataset was tested 50 times and evaluated on average.

The hyperparameters of decision tree model were monitored with the pre-selected 40,000 training examples(20,000 per class). In learning a decision tree, the measure to split a node is the Shannon entropy, the maximum depth of a decision tree is 15, the minimum number of instances to split at an internal node is 2, and the minimum number of instances at a terminal node is 2. The number of decision trees is 50 for learning ensemble models.

Table 9 compares the performance of the selected features in terms of dimension, accuracy, and training time. The ensemble models are higher than the decision tree in training and testing accuracy. We rank the performance of the ensemble models against feature types and calculate the average rank. This compares the functional efficiency and robustness among malware features.

Table 9 and Figure 8 show the training and test performance comparison. The ensemble model outperforms the

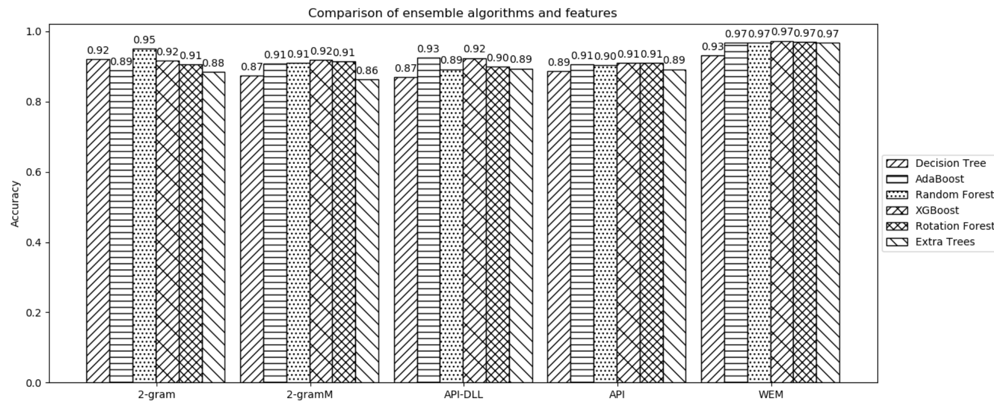


FIGURE 8. Comparison of ensemble algorithms and features.

TABLE 9. Accuracy comparison of tree-based ensemble per feature.

Method		2-gram	2-gramM	API-DLL	API	WEM	Avg.	Rank
No. of dimensions		61,952	1,024	3,842	195	512		
Decision Tree	Train	0.990	0.964	0.895	0.920	0.987	0.951	5
	Test	0.921	0.874	0.870	0.887	0.932	0.897	6
	Time	25.098	10.787	12.024	0.905	10.288	11.820	1
	Rank	2	4	5	3	1		
AdaBoost	Train	1.000	1.000	0.971	0.955	1.000	0.985	1
	Test	0.899	0.908	0.925	0.906	0.967	0.921	3
	Time	187.764	808.499	1407.063	82.802	852.205	667.667	5
	Rank	5	3	2	4	1		
Random Forest	Train	0.990	0.973	0.917	0.930	0.995	0.961	4
	Test	0.951	0.910	0.892	0.904	0.967	0.925	2
	Time	814.556	342.878	372.315	28.045	323.527	376.264	4
	Rank	2	3	5	4	1		
XGBoost	Train	1.000	0.991	0.954	0.947	1.000	0.978	2
	Test	0.916	0.918	0.924	0.911	0.973	0.928	1
	Time	51.732	345.282	899.574	59.170	253.506	321.853	3
	Rank	4	3	2	5	1		
Rotation Forest	Train	0.992	0.982	0.926	0.943	0.997	0.968	3
	Test	0.906	0.914	0.899	0.911	0.970	0.920	4
	Time	176.730	1906.404	13427.562	137.419	892.813	3308.186	6
	Rank	4	2	5	3	1		
Extra Trees	Train	0.958	0.915	0.913	0.917	0.991	0.939	6
	Test	0.884	0.864	0.894	0.892	0.967	0.900	5
	Time	9.287	179.120	652.072	23.467	50.488	182.887	2
	Rank	4	5	2	3	1		
Average rank		3.4	3.6	3.2	3.8	1		

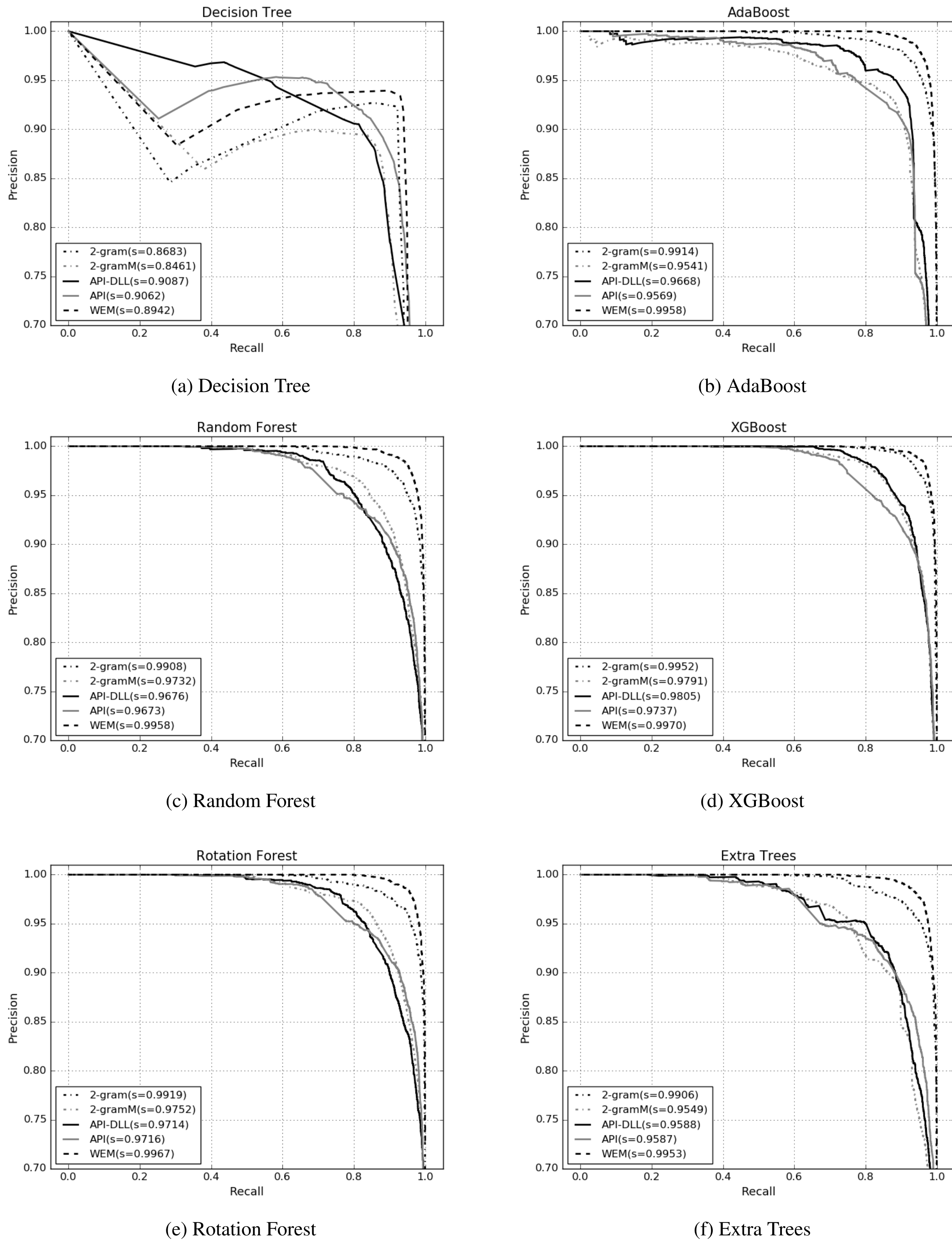
decision tree. In comparison of learning accuracy, AdaBoost is the best compared to other ensembles, but the extra trees is the lowest. AdaBoost has a training accuracy of 0.985 and the extra trees of 0.939. The learning evaluation of XGBoost is 0.978, the second highest after AdaBoost. The next higher order is rotation forest, random forest, decision tree, and extra trees.

In the test performance comparison, XGBoost and random forest show the classification rate of more than 0.925. Except for rotation forest, the test performance of all ensemble models is over 0.92. The accuracy of rotation forest is 0.9, which is similar to that of decision tree.

WEM is the best in the performance comparison among feature types. Next, it shows the low performance with API-DLL, 2-gram, 2-gramM, and API features. However, WEM shows the best results in training and test performance when compared to other features. API and API-DLL, which consist of functions that are frequently used in malware, are represented by lower dimensional vectors than 2-grams, but do not show much difference in time and performance. The test performance of the WEM of the ensemble model is more than 0.967. In particular, XGBoost is highest with 0.97 classification.

Our results suggest that XGBoost is superior to all algorithms. AdaBoost performs best in the experiment of API feature, and both the decision tree and XGBoost are almost equal for API-DLL. In the performance analysis, the difference between training data and training data based on feature extraction is not significant. Both 2-gram and WEM are better than 2-gramM, API, and API-DLL. However, the fastest ensemble algorithm is extra trees followed by random forest, XGBoost, AdaBoost, and rotation forest in order.

In comparison of training time, the training time of rotation forest runs longest than other ensembles due to PCA (Principle Component Analysis) computation. It is analyzed that the training time of the ensemble model is appropriate, but it requires tens times more training time than decision tree. The PCA compuation for 2-gram with more



(a) Decision Tree (b) AdaBoost

(c) Random Forest (d) XGBoost

(e) Rotation Forest (f) Extra Trees

FIGURE 9. Performance comparison with AUC-PRC.

than 60,000 dimensions was not possible, so we selected 4,201 attributes through the random forest feature selection. Therefore, the 2-gram training time (176.73 sec) of rotation tree is not a proper comparison with the training time of other ensembles. In other words, when training high-dimensional features in rotation tree, it is difficult to transform them by PCA.

Table 10 and Figure 9 compare precision, recall, and AUC-PRC score to assess the effect of malware features versus ensemble models. The AUC-PRC scores differ

TABLE 10. AUC-PRC comparison of tree-based ensembles.

Method		2-gram	2-gramM	API-DLL	API	WEM
Decision Tree	Precision	0.922	0.877	0.877	0.889	0.934
	Recall	0.921	0.870	0.860	0.884	0.930
	AUC-PRC	0.868	0.846	0.909	0.906	0.894
AdaBoost	Precision	0.916	0.954	0.933	0.907	0.974
	Recall	0.879	0.901	0.916	0.906	0.960
	AUC-PRC	0.968	0.954	0.925	0.957	0.996
Random Forest	Precision	0.944	0.924	0.892	0.899	0.965
	Recall	0.958	0.894	0.892	0.910	0.969
	AUC-PRC	0.991	0.973	0.968	0.967	0.996
XGBoost	Precision	0.924	0.931	0.932	0.912	0.972
	Recall	0.906	0.904	0.915	0.909	0.974
	AUC-PRC	0.977	0.979	0.980	0.974	0.997
Rotation Forest	Precision	0.922	0.925	0.898	0.911	0.969
	Recall	0.888	0.902	0.901	0.913	0.972
	AUC-PRC	0.973	0.975	0.971	0.972	0.997
Extra Trees	Precision	0.904	0.915	0.920	0.883	0.967
	Recall	0.860	0.803	0.863	0.904	0.967
	AUC-PRC	0.962	0.955	0.959	0.959	0.995

significantly between decision tree and ensemble algorithm. The decision tree show AUC-PRC scores higher than about 0.9 for API-DLL and API features, but low for WEM, 2-gram, and 2-gramM. However, the AUC-PRC score of other ensembles is a higher AUC-PRC score than the decision tree for all feature types.

The ensemble algorithms in Figure 9 do not show much difference according to the type of malware feature. In most ensemble algorithms, WEM's AUC-PRC scores higher than others. Therefore, it is analyzed that AUC-PRC of WEM is more suitable for classification than other types of malware feature. The ensemble model in API-DLL and API has lower precision than recall. On the other hand, the experimental results of the ensemble model in 2-gram, 2-gramM, and WEM show that the recall is lower than the precision. The decision tree has similar precision in all malware features. From the experimental results, the application of the ensemble algorithm for a malware detection system requires a *precision-recall tradeoff* analysis.

D. COMPARISON WITH OTHER WORKS

It is difficult to compare our result with the previous studies because there is not enough information about the used datasets, feature information and machine learning algorithms. Also, most experimental results are not comparable because PRC-based analysis is not presented, and some studies are related to in-depth learning approaches and ensemble applications based on both static and dynamic features.

The proposed API and WEM features are compared to the previous dimension reduction studies and the entropy representation in association with malware detection. The training data consists of 40,000 instances, with 20,000 instances selected for each class. The comparison algorithms are SVM, deep neural network (DNN), and random forest of the relevant work.

The API reduction method is compared to both Rushabh *et al.* [42] and Infosec [43] ones that are referred in Table 4. SVM chose RBF kernel ($\gamma = 5$, $C = 10.0$), and random forest was set to the parameter adopted in Subsection IV-C. In Table 11, the proposed API feature shows high performance regardless of the learning algorithms. Random forest

TABLE 11. Comparison of reduced API features.

Method		Rushabvh	Infosec	API
No. of dimensions		92	131	195
SVM (RBF)	Train	0.912	0.868	0.923
	Test	0.845	0.818	0.901
	AUC-PRC	0.886	0.839	0.939
	Time	336.810	299.613	343.121
Random Forest	Train	0.854	0.804	0.921
	Test	0.832	0.791	0.911
	AUC-PRC	0.929	0.905	0.939
	Time	1.213	1.246	2.001

TABLE 12. Comparison of byte entropy and WEM.

Method		Byte entropy	WEM
No. of dimensions		256	512
SVM (RBF)	Train	0.984	0.979
	Test	0.859	0.864
	AUC-PRC	0.913	0.926
	Time	1,134.138	2,083.561
Deep Neural Network	Train	0.818	0.899
	Test	0.808	0.876
	AUC-PRC	0.837	0.903
	Time	589.530	1137.793
Random Forest	Train	0.931	0.960
	Test	0.889	0.911
	AUC-PRC	0.943	0.981
	Time	40.570	80.901

can learn hundreds of times faster than SVM and exhibit high performance. However, it is difficult to analyze API dimension reduction methods for efficient malware detection, because the time of data collection and the dimension of feature vectors are very different.

The proposed WEM feature is compared with the byte entropy [11], and the learning algorithm is SVM, DNN, and random forest. The DNN architecture consists of 4 layers ($256 \times 256 \times 128 \times 2$) including an input layer. The DNN structure was modified from the architecture that tested the byte entropy feature. The dropout regularization was applied between layer 2 and 3 by dropping 10% of connections. The activation function of the hidden layer units is PReLU (parametric rectified linear unit) and that of the output units is sigmoid. The learning algorithm is Adam with the learning rate of 0.001 and the total of 3,000 epochs. Table 12 is the performance comparison between byte entropy and WEM. WEM has 512 dimensions and byte entropy expresses the characteristics of malicious code in 256 dimensions. The WEM feature is superior to the byte entropy feature in all learning algorithms. However, SVM and DNN require much higher training time than random forest. The experimental result of SVM is expected to be overfitted in WEM and byte entropy, so additional experiments are required according to various parameter settings. The learning time of SVM and DNN is expected to increase in proportion to the number of learning data.

The advantage of our approach is that the use of purely static features allows rapid analysis for a malware detection system in terms of feature reduction, fast computation and generalization performance. Therefore, the proposed approach will be very applicable and effective when applying to EDR (Endpoint Detection & Response) systems.

V. CONCLUSION

This paper analyzed malware features for static analysis and compares tree-based ensemble algorithms. A modified malware feature representation has been proposed to minimize the drawbacks such as variable length, high-dimensional representation and high storage usage in commonly used malware feature representations. The proposed malware features showed better generalization of ensemble algorithms in terms of training time and performance than the original training features. The modified malware features take advantage of frequently used functions, expertise knowledge, or entropy discretization. Therefore, the malware features do not require fixed length selection or padding that appears when training data is prepared in a manner appropriate to feature vectorization for machine learning.

The experimental analysis indicates that the tree-based ensemble model is effective and efficient for malware classification in relation to training time and generalization performance. In addition, our approach can quickly analyze a large amount of malware in terms of low-dimensional features and fast learning. The low-dimensional feature representation using WEM, API, and API-DLL can be an alternative to guarantee high generalization performance when static analysis is applied for malware detection. Further studies that provide the evidence of ensemble model prediction and the learning algorithm for combined malware features are required.

REFERENCES

[1] M. Sikorski and A. Honig, *Practical Malware Analysis: Hands-On Guide to Dissecting Malicious Software*. San Francisco, CA, USA: No Starch Press, 2012.

[2] C. LeDoux and A. Lakhotia, "Malware and machine learning," in *Intelligent Methods for Cyber Warfare*. Cham, Switzerland: Springer, 2015, pp. 1–42.

[3] J. Drew, T. Moore, and M. Hahsler, "Polymorphic malware detection using sequence classification methods," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2016, pp. 81–87.

[4] M. Alazab, "Profiling and classifying the behavior of malicious codes," *J. Syst. Softw.*, vol. 100, pp. 91–102, Feb. 2015.

[5] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Comput. Surveys*, vol. 46, no. 1, pp. 1–32, Oct. 2013.

[6] K. Murad, S. N.-U.-H. Shirazi, Y. B. Zikria, and N. Ikram, "Evading virus detection using code obfuscation," in *Proc. Int. Conf. Future Gener. Inf. Technol.* Berlin, Germany: Springer, 2010, pp. 394–401.

[7] M. Wagner, F. Fischer, R. Luh, A. Haberson, A. Rind, D. A. Keim, W. Aigner, R. Borgo, F. Ganovelli, and I. Viola, "A survey of visualization systems for malware analysis," in *Proc. Eurograph. Conf. Visualizat. (EuroVis)-STARs*, 2015, pp. 105–125.

[8] S. Ni, Q. Qian, and R. Zhang, "Malware identification using visualization images and deep learning," *Comput. Secur.*, vol. 77, pp. 871–885, Aug. 2018.

[9] A. Zheng and A. Casari, *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. Newton, MA, USA: O'Reilly Media, Inc., 2018.

[10] C.-T. Lin, N.-J. Wang, H. Xiao, and C. Eckert, "Feature selection and extraction for malware classification," *J. Inf. Sci. Eng.*, vol. 31, no. 3, pp. 965–992, 2015.

[11] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *Proc. 10th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2015, pp. 11–20.

[12] K. S. Han, J. H. Lim, B. Kang, and E. G. Im, "Malware analysis using visualized images and entropy graphs," *Int. J. Inf. Secur.*, vol. 14, no. 1, pp. 1–14, Feb. 2015.

[13] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Inf. Sci.*, vol. 231, pp. 64–82, May 2013.

[14] R. Islam, R. Tian, L. M. Batten, and S. Versteeg, "Classification of malware based on integrated static and dynamic features," *J. Netw. Comput. Appl.*, vol. 36, no. 2, pp. 646–656, Mar. 2013.

[15] Y. Ki, E. Kim, and H. K. Kim, "A novel approach to detect malware based on API call sequence analysis," *Int. J. Distrib. Sensor Netw.*, vol. 11, no. 6, Jun. 2015, Art. no. 659101.

[16] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *J. Comput. Secur.*, vol. 19, no. 4, pp. 639–668, Jun. 2011.

[17] P. Burnap, R. French, F. Turner, and K. Jones, "Malware classification using self organising feature maps and machine activity data," *Comput. Secur.*, vol. 73, pp. 399–410, Mar. 2018.

[18] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surveys*, vol. 44, no. 2, pp. 1–42, Feb. 2012.

[19] S. S. Hansen, T. M. T. Larsen, M. Stevanovic, and J. M. Pedersen, "An approach for detection and family classification of malware based on behavioral analysis," in *Proc. Int. Conf. Comput., Netw. Commun. (ICNC)*, Feb. 2016, pp. 1–5.

[20] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. 23rd Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2007, pp. 421–430.

[21] O. Sagi and L. Rokach, "Ensemble learning: A survey," *WIREs Data Mining Knowl. Discovery*, vol. 8, no. 4, p. e1249, 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.1249

[22] X. Dong, Z. Yu, W. Cao, Y. Shi, and Q. Ma, "A survey on ensemble learning," *Frontiers Comput. Sci.*, vol. 14, no. 2, pp. 241–258, Apr. 2020, doi: 10.1007/s11704-019-8208-z.

[23] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting unknown malicious code by applying classification techniques on OpCode patterns," *Secur. Informat.*, vol. 1, no. 1, p. 1, Dec. 2012.

[24] *VX Heaven*. Accessed: Oct. 9, 2018. [Online]. Available: http://83.133.184.251/virensimulation.org/

[25] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge," 2018, *arXiv:1802.10135*. [Online]. Available: http://arxiv.org/abs/1802.10135

[26] Y. Fan, Y. Ye, and L. Chen, "Malicious sequential pattern mining for automatic malware detection," *Expert Syst. Appl.*, vol. 52, pp. 16–25, Jun. 2016.

[27] *Virus Total*. Accessed: Sep. 21, 2018. [Online]. Available: https://www.virustotal.com/

[28] B. Sun, Q. Li, Y. Guo, Q. Wen, X. Lin, and W. Liu, "Malware family classification method based on static feature extraction," in *Proc. 3rd IEEE Int. Conf. Comput. Commun. (ICCC)*, Dec. 2017, pp. 507–513.

[29] *VirusShare-Because Sharing is Caring*. Accessed: Oct. 9, 2018. [Online]. Available: https://virusshare.com

[30] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, "Novel feature extraction, selection and fusion for effective malware family classification," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy (CODASPY)*, 2016, pp. 183–194.

[31] IMDEA Software Institute. *MALICIA PROJECT Malware in Cybercrime*. Accessed: Sep. 21, 2018. [Online]. Available: http://malicia-project.com

[32] P. Trinius, C. Willems, T. Holz, and K. Rieck, "A malware instruction set for behavior-based analysis," Inst. Comput. Sci., Univ. Mannheim, Mannheim, Germany, Tech. Rep. TR-2009-07, 2009.

[33] *Malheur: Automatic Analysis of Malware Behavior*. Accessed: Oct. 9, 2018. [Online]. Available: http://www.mlsec.org/malheur/

[34] J. Stiborek, T. Pevný, and M. Rehák, "Multiple instance learning for malware classification," *Expert Syst. Appl.*, vol. 93, pp. 346–357, Mar. 2018.

[35] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*. New York, NY, USA: Cambridge Univ. Press, 2011.

[36] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[37] V. V. Strelkov, "A new similarity measure for histogram comparison and its application in time series analysis," *Pattern Recognit. Lett.*, vol. 29, no. 13, pp. 1768–1774, Oct. 2008.

[38] C. Wressnegger, G. Schwenk, D. Arp, and K. Rieck, "A close look onn-grams in intrusion detection: Anomaly detection vs. Classification," in *Proc. ACM workshop Artif. Intell. Secur. (AISec)*. New York, NY, USA: ACM, 2013, pp. 67–76, doi: 10.1145/2517312.2517316.

[39] P. O'Kane, S. Sezer, and K. McLaughlin, "Detecting obfuscated malware using reduced opcode set and optimised runtime trace," *Secur. Informat.*, vol. 5, no. 1, p. 1, Dec. 2016, doi: 10.1186/s13388-016-0027-2.
[40] K. Rieck, T. Krueger, and A. Dewald, "Cujo: Efficient detection and prevention of drive-by-download attacks," in *Proc. 26th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2010, pp. 31–39.
[41] Z. Ren and G. Chen, "EntropyVis: Malware classification," in *Proc. 10th Int. Congr. Image Signal Process., Biomed. Eng. Informat. (CISP-BMEI)*, Oct. 2017, pp. 1–6.
[42] R. Vyas, X. Luo, N. McFarland, and C. Justice, "Investigation of malicious portable executable file detection on the network using supervised learning techniques," in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manage. (IM)*, May 2017, pp. 941–946.
[43] *Windows Functions in Malware Analysis—Cheat Sheet—Part 1*. Accessed: Oct. 5, 2018. [Online]. Available: https://resources.infosecinstitute.com/windows-functions-in-malware-analysis-cheat-sheet-part-1/#gref
[44] J. Davis and M. Goadrich, "The relationship between precision-recall and ROC curves," in *Proc. 23rd Int. Conf. Mach. Learn. (ICML)*. New York, NY, USA: ACM, 2006, pp. 233–240, doi: 10.1145/1143844.1143874.
[45] T. Saito and M. Rehmsmeier, "The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets," *PLoS ONE*, vol. 10, no. 3, 2015, Art. no. e0118432.
[46] C. X. Ling and V. S. Sheng, *Class Imbalance Problem*. Boston, MA, USA: Springer, 2010, p. 171, doi: 10.1007/978-0-387-30164-8_110.
[47] R. Blaser and P. Fryzlewicz, "Random rotation ensembles," *J. Mach. Learn. Res.*, vol. 17, no. 4, pp. 1–26, 2016. [Online]. Available: http://jmlr.org/papers/v17/blaser16a.html
[48] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001, doi: 10.1023/A:1010933404324.
[49] R. E. Schapire and Y. Freund, *Boosting: Foundations and Algorithms*. Cambridge, MA, USA: MIT Press, 2012.
[50] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2016, pp. 785–794.
[51] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Mach. Learn.*, vol. 63, no. 1, pp. 3–42, 2006.
[52] J. J. Rodriguez, L. I. Kuncheva, and C. J. Alonso, "Rotation forest: A new classifier ensemble method," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, no. 10, pp. 1619–1630, Oct. 2006.
[53] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011.
[54] L. Breiman, "Manual on setting up, using, and understanding random forests v3. 1," Statist. Dept., Univ. California, Berkeley, CA, USA, Tech. Rep., 2002, vol. 1, p. 58.
[55] G. Louppe, "Understanding random forests: From theory to practice," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Univ. Liège, Liège, Belgium, 2014.
[56] A. Buja, D. F. Swayne, M. L. Littman, N. Dean, H. Hofmann, and L. Chen, "Data visualization with multidimensional scaling," *J. Comput. Graph. Statist.*, vol. 17, no. 2, pp. 444–472, Jun. 2008, doi: 10.1198/106186008X318440.

SEOUNGYUL EUH received the B.S. and M.S. degrees from Ajou University, South Korea. He is the Managing Director of KSign and pursuing the Ph.D. degree with the Department of Software Science, Dankook University, South Korea. His research interests include machine learning, parallel processing, and threat intelligence.

HYUNJONG LEE received the B.S. and M.S. degrees from Dankook University, South Korea. He is currently a Researcher with the Security Technology Institute, KSign, South Korea. His research interests include machine learning, parallel processing, and image processing.

DONGHOON KIM received the Master of Science degree from Auburn University, USA, and the Ph.D. degree from North Carolina State University, USA. He is currently an Assistant Professor with the Department of Computer Science, Arkansas State University, USA. During his Ph.D. degree, he worked with the SAS Institute and IBM Cloud Computing Team, as a Research Intern. Previously, he was a Software Engineer with Samsung Electronics, South Korea. His research interests include cloud computing security, high-performance computing, and software engineering. He serves as an Associate Editor for JIPS journal.

DOOSUNG HWANG received the Ph.D. degree from Wayne State University, USA. Previously, he was a Senior Researcher with the Electronics and Telecommunications Research Institute (ETRI), South Korea, and worked on learning algorithm design and intelligent systems, such as expert systems, image recognition, speech recognition, and parallel computing. He is currently a Professor with the Department of Software Science, Dankook University, South Korea. His research interests include machine learning, high-performance computing, image processing, and inductive logic programming.