Computers, Materials & Continua
DOI: 10.32604/cmc.2022.024540
Article

Tech Science Press

High Performance Classification of Android Malware Using Ensemble Machine Learning

Pagnchakneat C. Ouk[1] **and Wooguil Pak**[2,*]

[1]Department of Computer Engineering, Keimyung University, Daegu, 42601, Korea
[2]Department of Information and Communication Engineering, Yeungnam University, Gyeongsan, Gyeongbuk, 38541, Korea
*Corresponding Author: Wooguil Pak. Email: wooguilpak@yu.ac.kr
Received: 21 October 2021; Accepted: 21 December 2021

Abstract: Although Android becomes a leading operating system in market, Android users suffer from security threats due to malwares. To protect users from the threats, the solutions to detect and identify the malware variant are essential. However, modern malware evades existing solutions by applying code obfuscation and native code. To resolve this problem, we introduce an ensemble-based malware classification algorithm using malware family grouping. The proposed family grouping algorithm finds the optimal combination of families belonging to the same group while the total number of families is fixed to the optimal total number. It also adopts unified feature extraction technique for handling seamless both bytecode and native code. We propose a unique feature selection algorithm that improves classification performance and time simultaneously. 2-gram based features are generated from the instructions and segments, and then selected by using multiple filters to choose most effective features. Through extensive simulation with many obfuscated and native code malware applications, we confirm that it can classify malwares with high accuracy and short processing time. Most existing approaches failed to achieve classification speed and detection time simultaneously. Therefore, the approach can help Android users to keep themselves safe from various and evolving cyber-attacks very effectively.

Keywords: Android malware classification; family grouping; native code; obfuscation; unified feature extraction

1 Introduction

With over 2 billion active Android devices worldwide [1,2], Android is considered as the worldly most popular mobile operating system. The rising popularity of Android system is accompanied by the increasing number of malwares that targets this system. As a result, there are many different kinds of malware variance available on vast Android landscape. Android malware becomes one of the biggest critical issues in Android system. According to [3], there are almost 8,400 malware applications were

found daily and they target on Android device in order to steal the private information for malicious activity. Although Google uses its own scanning tool, i.e., Play Protect security suite for years, malware issues still exist. Recently, McAfee lab found out several obfuscated malwares in Google Store from the same publisher [4]. Furthermore, ESET analyzer also found out that there are three specific malware families can invade from Google Store detection [5]. Finally, there is also native malware that invade Google Store [6] found by Fortinet analyzer.

To keep smartphone users safe from the threat of the malware, the solution to detect and identify the malware variant is needed. Thus, many various approaches using static or dynamic analysis have been proposed [7–12]. In traditional ways, malware developers use flaws of system permission [7–9,13–17] to load malicious function in the malware. Recently, malware is evolving very fast in terms of the total number and the sophistication level [1,4–6]. More seriously, modern malware evades detection from most of the existing solutions by applying code obfuscation and native code [10,11]. Such techniques cause increased detection time or even detection failure for most existing anti-malware solutions [8,9,14,15,18,19]. Thus, we need a new approach to overcome modern malwares.

Additionally, using the obfuscation technique, the malware developer easily generates multiple variants of malware that can cause misdetection in most existing systems [10,11]. Malware developers also manually make variants of original malwares and incapacitate anti-malware programs. Such variants and original malwares can be categorized as the same family. However, most previous literature only focuses on malware detection to inform the user whether it is benign or malware without informing the malware family name.

Malware classification identifies the belonging malware family of the malware. Malware classification has many advantages compared to malware detection [8,20]. First, it helps existing solutions to detect a malware with higher accuracy. Existing detection technique extracts unique signatures from each malware and use the signatures to detect it; thus, they have a high failure probability to detect malware variants created manually or with code obfuscation [10,11]. If we know the family that existing malwares belong to, we can build signature based on common features among malwares in the same family. Such signatures are very robust against obfuscation or manual variations. Second, a family name from classification is very helpful to decide how to deal with unknown malwares. From other known malwares in the same family, we can use the same remedy to remove the malware and recover the damaged system. Third, it also helps to prevent the system from malwares. It is very difficult to configure security policies one-by-one for every single malware to protect the system for all malwares. Instead, if we configure them according to each family, the configuration cost can be low and scalable.

However, most existing anti-malware approaches still focus on malware detection rather than classification. To cope with the issue, we present a new solution based on unified static feature extraction that can classify the malware application, which is not incapacitated to obfuscated and native malwares. This study optimizes the feature set to the small size of the feature dimension. We also suggest an efficient grouping technique for malware families. Additionally, we propose an ensemble learning technique to classify malwares using results from different groups of malware families to achieve high scalability in terms of accuracy and processing time.

In summary of our proposed approach, this study makes the following contribution to state of the art malware classification for Android.

- High accurate group-based classification: This study introduces an efficient way to group the malware families by applying Boosted Random Forest (BRF) to a dataset and recursively combining the largest family with the smallest one. Such a grouping technique greatly helps to increase the classification performance and reduce processing time.

- Ensemble machine learning: We thoroughly analyze 28 malware families including various obfuscated and native code malwares. Based on the result, we determine the size of groups and design the optimal ensemble learning technique with multiple Random Forests (RFs). It provides high classification accuracy compared to latest solutions.
- Lightweight and unified feature selection: This study unifies the feature extraction to seamlessly process both bytecode and native code. After obtaining initial features, it selects only best features by intersecting the initial features. Since the classification results depend on feature selection, we propose a very practical feature extraction and selection algorithm, and therefore, achieving a good performance with a large scale dataset.
- Robust malware classification: Although many existing works fail to detect or classify modern malwares using code obfuscation and native code, our algorithm provides classification results with high accuracy. To achieve this performance. We gathered various malware applications from many available sources that include traditional, obfuscation, and native malware for analysis. Based on the result, we elaborately design our algorithms such as predefined irrelevant features and 2-gram based features to achieve high classification accuracy regardless of malware types.

The rest of our paper is organized as follows. We present some selected related work in Section 2. We introduce and describe our proposed system in Section 3. We analyze evaluation results in Section 4 and conclude it in Section 5.

2 Related Work

In the state-of-art Android malware analysis, it can be classified into two categories: Malware detection and Malware classification. Malware detection is the technique that focuses only on two target classes whether the application is benign or malware. On the other hand, Malware classification looks deeper into each malware characteristics to identify belonging family of malware application. As the result, malware classification provides a more scalable solution to malware analysis on specific malware over malware detection.

For better understanding, this study, we will first discuss related work on the techniques that only focus on detecting malware in Android. We also discuss on the signature-based and machine learning based techniques that use to detect malware application. We then cover about the techniques that focus on malware classification. We also look at ensemble classification and random feature ensemble technique that are used to classify the malware.

2.1 Malware Detection

There are two popular techniques to detect the malware. The first technique is signature-based malware detection; the technique solely depends on a specific pattern such as Control Flow Graph (CFG) [9,21,22] or known malicious functions used in malicious applications. The second technique is machine learning [13,15,16] based malware detection that trains feature set extracted from malware applications with recent machine learning algorithms to detect the malware.

2.1.1 DroidNative

DroidNative [9] is the system that focuses on detecting native malware application. DroidNative generates the Annotated Control Flow Graph (ACFG) [23] and Sliding Window of Difference (SWOD) using Malware Analyze Intermediate Language (MAIL) [24]. It compares generated ACFG

with the built model to identify the malware. Furthermore, it reduces the comparison time using similarity comparison based on machine learning. DroidNative can effectively detect native malware and traditional malware applications. However, DroidNative has no mention of obfuscated malwares; thus, the system may be vulnerable to obfuscated malwares. Furthermore, DroidNative can only perform the detection not classification. Since DroidNative uses CFG, it can detect only pre-existing malwares in the trained model.

2.1.2 DroidSieve

DroidSieve [13] is one of unique systems that can detect obfuscated malware. It uses only static analysis to extract the feature from the application. DroidSieve extracts all possible features for malware analysis, including bytecode, strings, and resource files. Then, it selects only fine-grained features for malware detection. It can easily detect obfuscated malware as well as other traditional malware. It also argues that it can handle native malware. However, it only relies on metadata to detect native one, and therefore it cannot robustly detect native malware applications.

2.2 Malware Classification

The fundamental techniques for classifying malware are literally the same with detecting malware but using the technique without any consideration for classification characteristics achieves usually low accuracy. To increase the classification accuracy, many approaches have been proposed: the random feature set ensemble [25], ensemble different classification algorithms [7], and ensemble supervised model with un-supervised model [8]. However, each literature has its own limitation on malware classifications.

2.2.1 Ensemble Clustering and Classifier

To mitigate the classification issues, ensemble clustering and classifier (EC2) [8] proposed a way to ensemble the results from clustering and classifying processes. EC2 aims to understand the characteristics of the malware family, making more robust on malware's signatures. It uses static and dynamic analyses to extract the feature for the malware classification. Static analysis in EC2 extracts activities, services, and other metadata, whereas dynamic analysis extracts read and write states, SMS, and network. It ensembles six different algorithms including three clustering algorithms and three classification algorithms to obtain the final result. However, EC2 does not include any information about malware variants using obfuscation and native code. Therefore, it can be vulnerable to obfuscated and native malwares. Furthermore, clustering can cause huge overheads to the processing performance of the overall system.

2.2.2 RevealDroid

RevealDroid [7] extracts all possible features from malware applications using only static analysis method. RevealDroid extracts application programming interface (API), native code executable and linkable format (ELF), and Android metadata as feature sets from malwares. Thus, it is able to detect obfuscated or native malwares. RevealDroid builds multiple classifiers according to the number of targets. For example, if the model contains five target families, then RevealDroid will generate five classifiers to detect each family whether it corresponds to the target family. The system focuses on selecting the fine-grained features from the dataset for better scalability and performance.

2.2.3 IagoDroid

IagoDroid [25] is the extended study of RevealDroid that focuses on malware classification. IagoDroid selects random features from the original feature set, and then uses multiple RevealDroid classifiers to classify the random malware on each random feature set. Each classifier uses each own disjoint feature set, and IagoDroid ensembles the results from the classifiers to generate the final result. Then, it compares the ensembled result with the original RevealDroid result to finalize the classification. However, IagoDroid only focuses on reducing misclassification more than increasing the accuracy of the overall system. Thus, it is limited for practical deployment in the real system.

3 Proposed Algorithm

They still have much room for improvement in terms of classification detection and run-time although many malware classification algorithms have been proposed so far. Currently, no systems are available to classify obfuscated malwares. For native malwares, only a few systems exist but they also fail to achieve a high classification accuracy. We note that most approaches do not rely on application code but non-application code such as metadata. The code of the application contains plenty of information which can be directly used for classifying malwares. However, extracting features from the application code is time-consuming [25,26]. Moreover, native malware contains bytecode and native code together, we need to process bytecode and native code, separately. Such an approach makes the classification process more complicated and slower. Therefore, in this study, we focus on unifying separate feature extractions for both bytecode and native library code by converting everything into a native binary. Android Runtime (ART) [25,27] makes it possible to achieve unifying feature extraction since ART can convert Android bytecode into native binary code; thus, we can perform feature extraction using only native code. It will greatly simplify the feature extraction process, and it makes malware classification more efficient in terms of speed and accuracy.

For better classification, some previous literatures on malware classifications use ensemble techniques [7, 8,14]. The ensemble is the process that combines the many algorithms or model to find out the best result for each feature set. However, it has some issues such as insufficient accuracy or long processing time. To cope with the issues, this study proposes a different way to generate the ensemble model for boosting the performance of the classification system. We build groups of the malware based on each label using BRF for the entire dataset then train it using BRF for each group separately. We will explain each procedure in more detail.

3.1 System Overview

Fig. 1 shows the overview of the proposed approach. It consists of four phases: Data Preprocessing, Feature Engineering, Family Grouping, and Ensemble Model Generation. In data preprocessing, application bytecode and native library are converted into native binary code using ART and disassembly [11,16,28]. In Feature Engineering, we perform feature extraction from the applications converted in the pre-processing. We then remove the irrelevance features and perform multiple feature selections to choose common features from the feature sets. We intersect the multiple filter-method based-feature selections together to obtain the finely chosen features set. Then, we categorize malware families into small groups with different characteristic based on the result of BRF. Lastly, we ensemble each result of BRF for each group to obtain the result for the target family. Now, we will explain each step in detail.

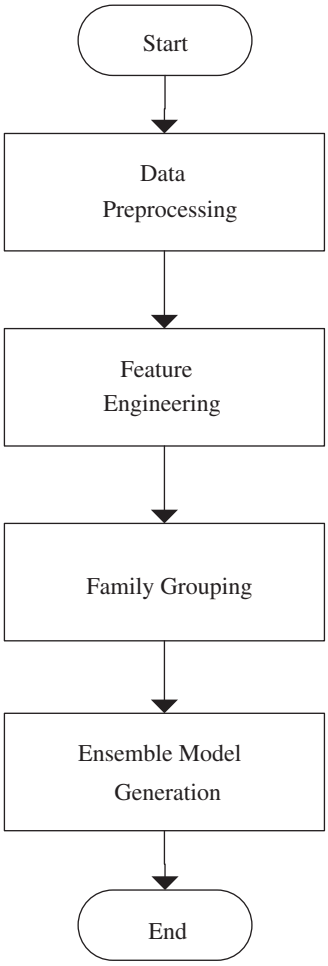

Figure 1: Overview of building the classification model for our proposed approach

3.2 Data Preprocessing

In the preprocessing stage, we first extract bytecode and native library from the Android malware applications. Starting from Lollipop, Android introduced a new function called ahead of time (AOT) compilation at the installation time [26]. Using Android runtime (ART), we convert extracted bytecode into native binary code. As a result, ART makes it possible for the system to unify the process of feature extraction. The system disassembles native code obtained from both bytecode and native library into a readable assembly text file for feature extraction.

In this study, we use the disassembly technique proposed by [29]. First, the system decodes everything on the native binary into an assembly including data as well as code. We should note that it may generate incorrect instructions especially when we try to disassemble data. Second, we decode everything on the native binary into assembly without the data. This process can sometimes fail to find valid embedded machine instructions in data. Thus, we combine the first and the second assembly techniques to remove the data by comparing the instructions between both assemblies.

3.3 Feature Engineering

We remove the irrelevance instructions [9] from the dataset obtained from the data preprocessing. Irrelevance instructions mean instructions used too widely or used for only specific purposes, so they are not regarded as ones used for malware operation. For example, INVD is the instruction used to invalidate internal caches, consequently, inappropriate for malware analysis. During the preprocessing process, we remove such irrelevance instructions to increase the performance and reduce overall feature dimension in feature extraction.

In this study, the proposed system uses 2-gram, i.e., 'pair of segment and instruction' or 'consecutive instructions' to generate the feature set. In addition, we use all loadable code segments [30] to generate the features. Using all the segments is essential to analyze all operations of a target application.

Tab. 1 shows all segments used for feature extraction in this study. All loadable code segments include .rodata, .oatexec, and .oatdata. In original ELF, .rodata contains read-only data without any machine instruction; however, .rodata in Android ELF contains two sub-segments: .oatexec and .oatdata [30], which are able to embed machine instructions. Thus, we should include segments such as .rodata in our feature extraction process.

Table 1: List of segments used to generate features on the proposed system

Segment	Description
.rodata	Read-Only data: In traditional ELF, it contains only the string without any instructions. Android ELF .rodata contains .oatexec and .oatdata that include the instruction of embedded library.
.text	Code part: It contains all the code instructions of the application.
.dynstr	Dynamic string: It holds strings needed for dynamic linking. They represent the names associated with symbol (dynsym) table entries.
.dynsym	Dynamic symbol: It holds the dynamic linking symbol table. It includes string data used in an application linked in .dynstr.
.hash	It holds a symbol hash table.
.oatdata	It contains OAT headers and embedded original Dalvik executable file (DEX) file.
.oatexec	It contains generated native codes.

Let us take an example as Fig. 2. Assuming that we have .rodata and .text segments in disassembled binary first, our system selects .rodata and removes irrelevance instructions from this segment, i.e., *wbinvd*. Then, the system starts to count the instruction in .rodata segment; there are two 'add' instruction in this segment, and it counts that instruction by combining it with the segment as 'rodata_add:2'. After counting all the instruction in the segment, it starts to combine instruction with instruction; if the subsequence instruction of previous 'add' is 'pop', the system counts sequence of the combined instructions as 'add_pop:1'. Once all the subsequence instructions in .rodata are counted, the system continues the same process with the next segment .text. The last instruction from .rodata is used to combine with the very first instruction of .text segment. From previous work, it is known that there are instruction combinations mainly used only in malicious applications. By using information about these combinations as features, normal apps and malicious apps can be distinguished. Long n-gram are mainly used in existing studies, but short n-grams are used in the proposed method. When code obfuscation is applied to malicious applications, the order of the instructions is changed severely,

which makes features based on long n-gram not working well. On the other hand, the proposed method uses short n-gram, making the features based on short n-grams robust even for code obfuscation. Fig. 2 illustrates the feature extraction in our proposed system.

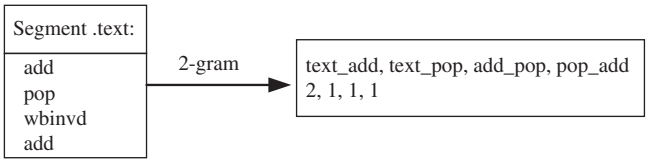

Figure 2: Example of 2-gram based feature generation using segments and instructions

To increase the classification accuracy and reduce the overall feature dimension size, the system applies feature selection to the obtained dataset [31]. We decided to use a filter-method to select the features since it chooses features depending on the general characteristics of the data independent of any dataset without involving any training model [32]. Thus, it leads to a faster feature selection pipeline. Each filter-method selects a different dimension of feature set; thus, to obtain the final feature set, we intersect the feature sets together to get the final feature set.

Assume that we have two feature sets A and B selected from an original feature set using filter-methods A and B, respectively. Both filter-methods can result in different feature dimension sizes. For example, feature set A contains "pushf_add, add_add, add_pop, pop_pushf, pushf_add, pop_add" while feature B contains "add_add, add_pop, pop_add, mov_call, call_lret". The system starts to intersect two feature sets by keeping only the same features that exist in both feature sets. After intersection, final features, i.e., "add_add, add_pop, pop_add" are used to train the model. Fig. 3 shows how the feature set is intersected.

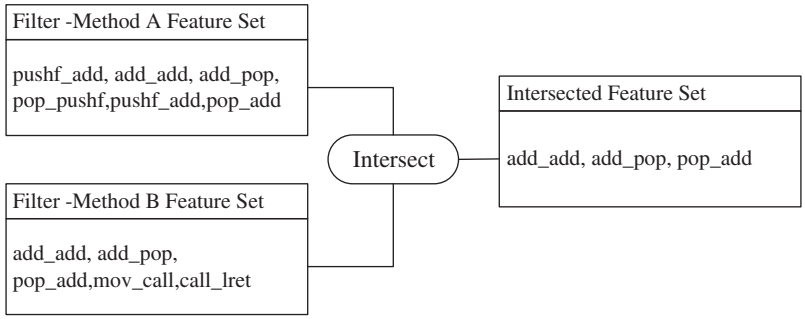

Figure 3: Filter-method based feature set intersection process

3.4 Family Grouping

To group malware families, we can consider two different techniques. The first technique is to cluster the whole instance using *k*-means algorithm. The algorithm uses unsupervised learning to group the instance with similar characteristics together using centroid. However, this technique causes the performance overhead to the classification system, thus we do not consider this clustering technique.

The second technique is to group the families based on both the family size and the misclassified results of each family from BRF. We group the malware dataset based on family size since it has less overhead compared to the clustering technique. Before the grouping process, we obtain classification results using BRF model trained with all families, and initially, each family corresponds to each group. In the grouping process, the system selects the largest and smallest groups. For the largest group, the system checks whether the most misclassified group is the same as the smallest in the BRF applied classification results. If it is the same, it tries to the same procedure with the next smallest group. If it differs, the two groups are merged into a larger group. This process is repeated until the total family size in the largest group reaches to the predefined maximum family size in a group. If it reaches to the maximum size, the system selects the next largest group and repeats the grouping procedure. Fig. 4 shows the overall grouping process.

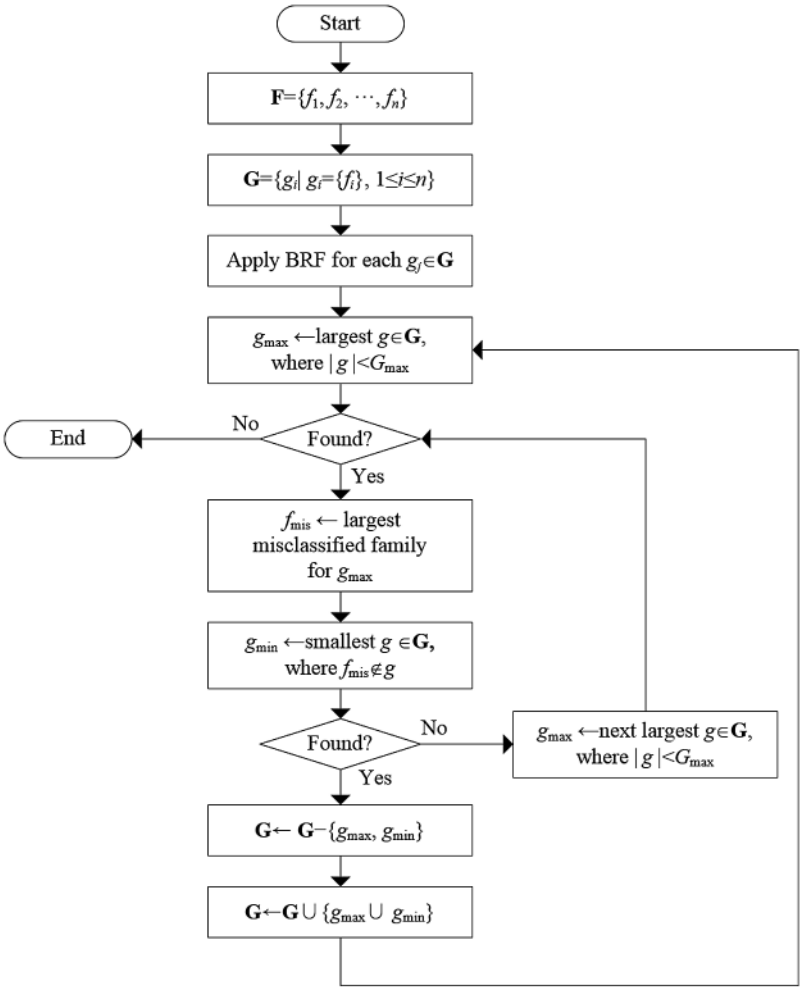

Figure 4: Malware family grouping process, where f_i is the set of the i-th malware family that includes all malwares belonging to the family as elements, $G_{\max}$ is the maximum number of families in a group, and $|\cdot|$ is the cardinality of the set

For a better explanation for the grouping process, we assume that we have eight malware families as shown in Tab. 2, where the maximum family size in a group is four. The system selects *geinimi* and

boxer as the largest and the smallest groups, respectively. However, the most misclassified family of *geinimi* is *boxer*, and thus, the system cannot merge *geimini* with *boxer*. The system chooses the next smallest group, i.e., *Fakerun* and merges it with *geinimi*. On the other hand, *boxer* will be merged with the next largest group, i.e., *Fakeinst*. This step is repeated until each group contains four families. Tab. 3 shows the result of the example dataset in Tab. 2.

Table 2: Example of malware families for grouping

Family name	Size	The largest misclassified family based on BRF
Fakeinst	200	Zitmo
Fakerun	20	Adrd
Tigerbot	100	Yzhc
Zitmo	70	Adrd
Adrd	30	Zitmo
Boxer	15	Fakerun
Geinimi	300	Boxer
Yzhc	19	Adrd

Table 3: Final grouping results for the example dataset in Tab. 2

Group 1	Group 2
Geinimi	Fakeinst
Zitmo	Tigerbot
Fakerun	Adrd
Yzhc	Boxer

Through experiments, we can see that the classification performance of BRF can be changed according to the number of families in the training dataset. Thus, the number is an important factor of the proposed algorithm. In addition, which families are included in the training dataset can affect performance. If the dataset is composed of very similar families, the classification accuracy will be inevitably low. To avoid this situation and improve the classification performance, the proposed method allocates malware families which have low mis-classification rate into the same training dataset, called group.

3.5 Ensemble Model Generation

We train an RF model with each group. RF is a well-known ensemble model of decision tree where every tree is built with randomly selected features of the dataset [33]. RF has a unique capability to train a large-scale dataset with a high training speed. Therefore, we choose RF to trains each group malware families to build the model. For simplicity, we use the same settings for all model. The features and the number of trees for training model in BRF are the same for each model. We use 10-fold cross-validation in every model.

Finally, we ensemble trained group models to finalize the result. The ensemble approach has some variations. Some approaches ensemble the results from different learning algorithms while others

ensemble results from different random feature sets. In this study, we ensemble results from each model of groups with the same algorithms and the same feature set. We apply an application to all group models, and then choose the family with the highest scores as the classification result. Fig. 5 shows integrated procedures of building our classification machine-learning model.

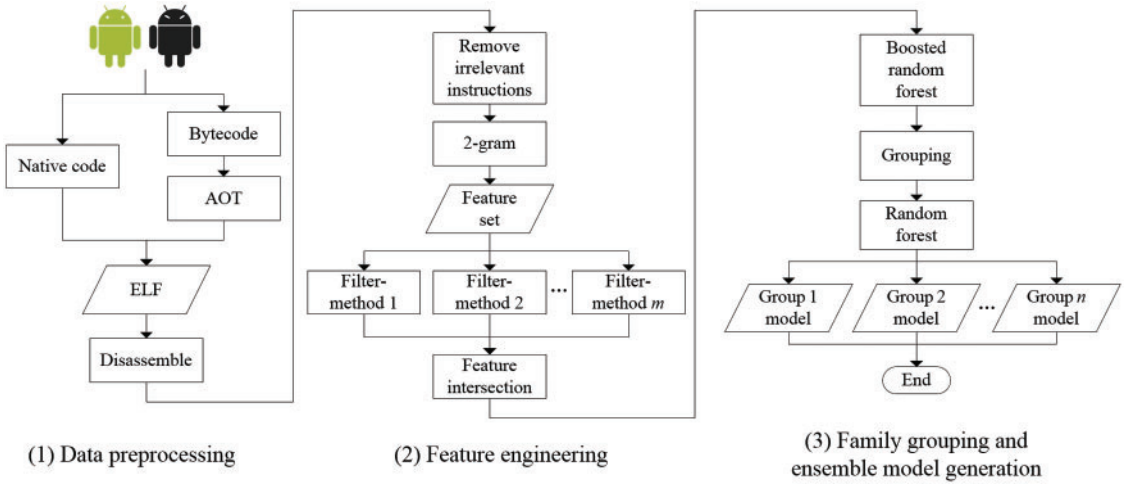

Figure 5: Flowchart of the proposed system

4 Performance Evaluation

4.1 Evaluation Environment

To evaluate the performance of classification for our system, we conducted extensive experiments to compare with existing work. First, we briefly explain the evaluation dataset for our experiment. Second, we discuss metrics used to evaluate the system. Finally, we compare the result of our system with previous literature. All evaluations were conducted in a desktop equipped with one Intel Core i7-4790 K 4 GHz, 16 GB RAM, and two 256 GB SSDs configured to Redundant Array of Independent Disks (RAID) 0.

The dataset for the experiment was gathered from many different sources: Marvin, Information Security Centre of Excellence (ISCX), Drebin, and PRAGuard. There are three types of malware in our dataset: Un-obfuscated, obfuscated, and native ones. The dataset contains originally 43 malware families. However, very small families with one or two malwares only were removed since they have too few samples to be applied for cross-validation. Finally, 28 families are contained in the dataset. The whole dataset consists of 15,890 malware applications, where the sizes of obfuscated and native malware are 7,560 and 1,390, respectively. We retained 70% of each dataset for training and 30% for the testing procedures.

We compare the classification model in term of F-Score and run-time per sample. F-Score is defined as the harmonic mean of precision and recall as shown in (1). The number of malware application has been growing very fast; thus, it is critical to perform high-speed malware classification with a large scale malware. Therefore, we also evaluate our system in term of run-time per sample.

Run-time per sample is the time that system spent from feature extraction to identify the target, so it is very important metric since it indicates the scalability of the system.

$$\text{F-Score} = 2.\frac{\text{Precision.Recall}}{\text{Precision} + \text{Recall}}, \tag{1}$$

$$\text{where Precision} = \frac{\text{True positive}}{\text{True positive} + \text{False positive}} \text{ and Recall} = \frac{\text{True positive}}{\text{True positive} + \text{False negative.}}$$

4.2 Evaluation Results

4.2.1 Classification Accuracy

The main challenge for our proposed algorithm to increase the performance of malware classification is to determine that how many and which filter-method algorithms should be used. It is also important to find how many families should belong to one group. To find such optimal values, we tried on filter-method combinations as many as we can. We also measured the performance as the number of families in a group increases. Through these experiments, we can find the best filter-methods and the family number in a group. For filter-method, we considered six algorithms such as Pearson Correlation, Spearman correlation, Mutual information, Kendall correlation [34], Fisher, and Chi Square.

We measured the classification accuracy using F-Score as mentioned previously. The F-Score is highest when the number of filter-methods and the number of families in a group are four and three, respectively, as depicted in Fig. 6. The highest score is 97.16% and it is very high compared to existing classification algorithms.

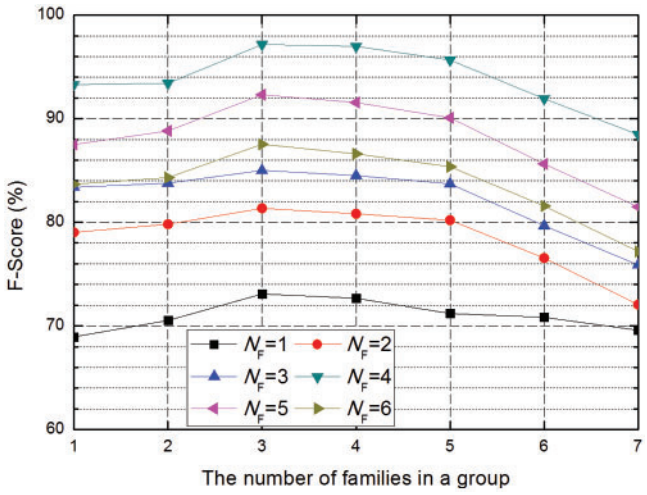

Figure 6: Comparison of F-Scores according to the number of filter-methods (N_F) with the maximum number of families in a group increasing (G_{max})

To ensure our system against malware classification in Android, we compared this study to two previous literatures on malware classification: RevealDroid and IagoDroid. We decide to choose RevealDroid and IagoDroid in our comparison because they are only practically available malware classification systems that can detect and classify obfuscated and native malware [35].

RevealDroid is the system that supports both malware detection and classification. RevealDroid focuses on the manifest, library, API call, Reflective, and native call to extract all possible features from every malware application. RevealDroid produces many classifiers to perform malware classification. For example, if there are *n*-target families in the dataset, RevealDroid will produce *n* classifiers for each target family. Thus, using our dataset, RevealDroid generates 28 classifiers based on a regression tree for each family. However, RevealDroid was designed to focus on malware detection more than classification thus it achieves moderate F-Score. Through the experiment, our system outperforms RevealDroid in terms of F-Score, achieving around 16% higher score. Fig. 7 show the F-Score comparison between our proposed and RevealDroid.

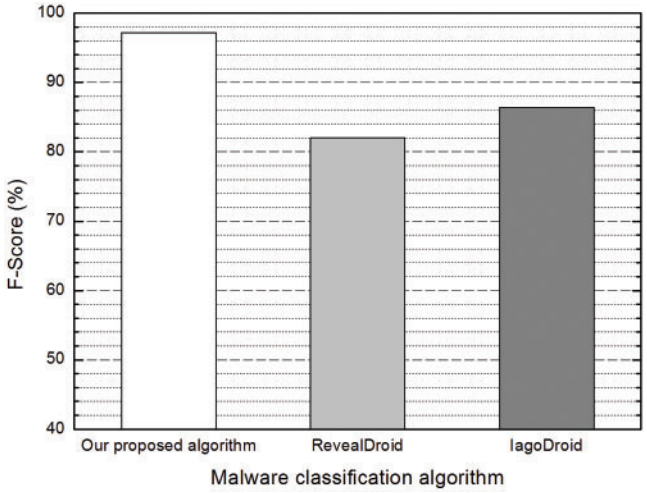

Figure 7: Comparison of the classification accuracy between our proposed algorithm and related work

IagoDroid is the extended version of RevealDroid; it was designed to support better malware classification than RevealDroid. IagoDroid generates many RevealDroid classifiers by randomly selecting the features among original features, and ensembles the result from multiple classifiers to obtain the result. Since IagoDroid shows different F-Scores according to the number of classifiers, we configured the number as the best value, i.e., four to achieve the highest F-Scores. However, we outperform IagoDroid with the best number of classifiers roughly 10% in term of F-Score. Fig. 7 also shows the F-Score comparison results between our proposed and IagoDroid.

In addition, there are 4,767 obfuscated apps included in the test dataset. For obfuscated apps, RevealDroid and IagoDroid have 77% and 81% on F-score, whereas the proposed method has 96%. Therefore, it shows a tendency similar to the performance for the total test dataset.

Tab. 4 shows the best combination of filter-methods for intersection. For highest performance, we choose four filter-method algorithms: Chi Square, Mutual information, Kendall correlation, and Spearmen correlation. Tab. 5 also shows the name of families belong to each group when we achieve the best score.

Table 4: Intersected filter-methods when the F-Score is highest according to the number of filter-methods

The number of filter-methods	Intersected filter-methods achieving the best F-Score
1	Kendall
2	Mutual, Chi Square
3	Spearman, Mutual, Chi Square
4	Spearman, Mutual, Kendall, Chi Square
5	Pearson, Spearman, Mutual, Kendall, Fisher
6	Pearson, Spearman, Mutual, Kendall, Fisher, Chi Square

Table 5: Grouping result when the highest score is achieved

Group ID	The list of family names belonging to each group
1	Fakeinst, Fakerun, Tigerbot
2	Droidkungfu, Smgreg, Batterydoctor
3	Plankton, Rootsmart, Lotoor
4	Opfake, Adrd, Boxer
5	Ginmaster, Yzhc, Kmin
6	Anserver, Imlog, Pjapps
7	Geinimi, Gappusin, Nickyspy
8	Mobiletx, Droiddreamlight, Zitmo
9	Basebridge, Sendpay, Iconosys, Golddream

4.2.2 Classification Time

The classification time is a very important metric since it determines whether the algorithm can be used in a real environment. We also measured the classification time according to the number of filter-methods and the number of families in a group, where filter-methods with the highest F-Score are selected. As shown in Fig. 8, the time generally increases with the number of filter-methods and the number of families in a group. It achieves moderate classification time when it achieves the best F-Score, i.e., four filter-methods and three families in a group.

We also compared our algorithm with RevealDroid and IagoDroid in terms of the classification time. Our system shows 6.5 times faster than RevealDroid, and 11 times faster than IagoDroid. Since IagoDroid internally utilizes many RevealDroid classifiers and ensembles the result from multiple classifiers, it shows the worst classification time. For our algorithm, it achieves fastest classification though efficient feature selection algorithm. Fig. 9 show the classification time results between our proposed algorithm and related work.

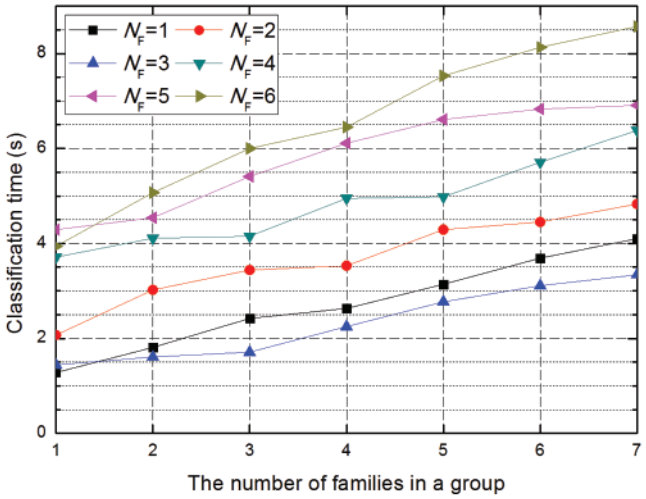

Figure 8: Comparison of average classification time according to the number of filter-methods (N_F) with the maximum number of families in a group (G_{max}) increasing

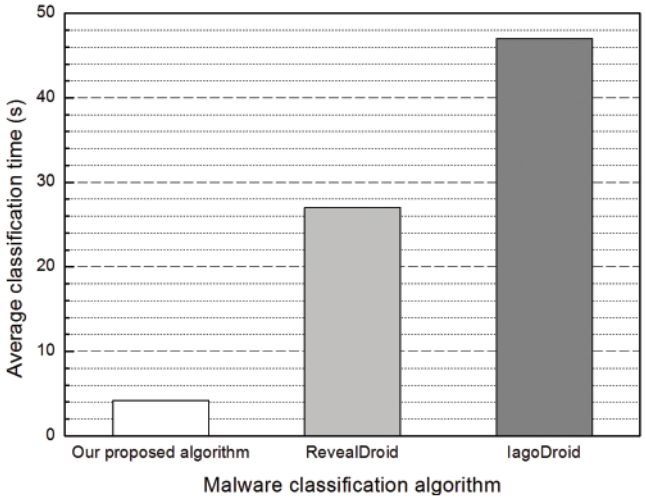

Figure 9: Comparison of average classification time between our proposed algorithm and related work, where the total number apps is 15,890

Fig. 10 shows the total run-time according to the various filter-method numbers and the total family numbers in a group. The run-time includes consumed time from feature selection, grouping, learning, and classification with 15,890 applications. The run-time increases in proportion to the number of families in a group. With three families in a group, we can achieve the best F-Score and very short run-time.

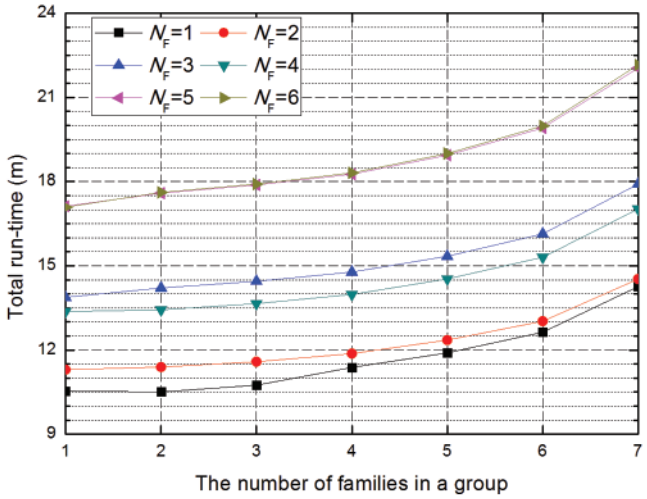

Figure 10: Comparison of the total run-times according to the number of filter-methods (N_F) with the number of families in a group (G_{max}) increasing, where the run time includes learning and classification time

5 Conclusion

Although Android malware is one of the most serious threats in Android landscape, it becomes more difficult to classify recent obfuscate and native code malwares even with the latest solutions. To overcome this issue, we developed malware type independent unified feature extraction, small but effective feature selection, fast BRF based grouping, and accurate ensemble machine learning with multiple RFs. These techniques are integrated into one high-performance classification algorithm and it can classify Android malwares regardless they are obfuscated or native code-based malwares. Through extensive simulation with many types of malware, we could prove its high performance such as high accuracy and fast classification speed. Malware classification is known as more difficult than malware detection. Thus, many existing works have not been able to improve the detection accuracy and fast classification speed simultaneously. Due to such a high performance of the algorithm, we expect that our approach can help Android users to keep them from various and evolving cyber-attacks.

Funding Statement: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT)(NRF-2019R1F1A1062320).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] *Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2017*, 2018. [Online]. Available: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems.

[2] B. Popper, *Google announces over 2 billion monthly active devices on Android*, 2017. [Online]. Available: https://www.theverge.com/2017/5/17/15654454/android-reaches-2-billion-monthly-active-users.

[3] C. Lueg. *8,400 new Android malware samples every day*, 2017. [Online]. Available: https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day.
[4] F. Ruiz, *Obfuscated malware discovered on Google play*, 2016. [Online]. Available: https://securingtomorrow.mcafee.com/mcafee-labs/obfuscated-malware-discovered-google-play.
[5] B. Barth, *Three more Android malware families invade Google play store*, 2017. [Online]. https://www.scmagazine.com/three-more-android-malware-families-invade-google-play-store/article/707693.
[6] K. Lu, *Deep Analysis of Android rootkik malware using advanced anti-debug and anti-hook, part I: debugging in the scope of native layer*, 2017. [Online]. https://www.fortinet.com/blog/threat-research/deep-analysis-of-android-rootnik-malware-using-advanced-anti-debug-and-anti-hook-part-i-debugging-in-the-scope-of-native-layer.html.
[7] J. Garcia, M. Hammad and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of android malware," *ACM Transactions on Software Engineering Methodology*, vol. 26, no. 3, Article 11, pp. 11–29, 2018.
[8] T. Chakraborty, F. Pierazzi and V. S. Subrahmanian, "EC2: Ensemble clustering and classification for predicting android malware families," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, pp. 1–16, 2017.
[9] S. Alam, Z. Qu, R. Riley, Y. Chen and V. Rastogi, "DroidNative: Automating and optimizing detection of android native code malware variants," *Computers & Security*, vol. 65, pp. 230–246, 2017.
[10] D. Arp, M. Spreitzenbarth, M. Huebner, H. Gascon and K. Rieck, "Drebin: Efficient and explainable detection of android malware in your pocket," in *Proc. the 21th NDSS Annual Network and Distributed System Security Symposium*, San Diego, California, USA, Feb. 2014.
[11] *Android. ART and Dalvik*. [Online], 2021. Available: https://source.android.com/devices/tech/dalvik.
[12] S. J. Hussain, U. Ahmed, H. Liaquat, S. Mir, N. Jhanjhi *et al.,* "IMIAD: Intelligent malware identification for android platform," in *Proc. Int. Conf. on Computer and Information Sciences (ICCIS)*, Wuhan, China, pp. 1–6, Apr. 2019.
[13] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto *et al.,* "DroidSieve: Fast and accurate classification of obfuscated android malware," in *Proc. the Seventh ACM on Conf. on Data and Application Security and Privacy*, New York, NY, USA, pp. 309–320, Mar. 2017.
[14] J. Abawajy and A. Kelarev, "Iterative classifier fusion system for the detection of android malware," *IEEE Transactions on Big Data*, vol. 5, no. 3, pp. 282–292, 2019.
[15] M. Lindorfer, M. Neugschwandtner and C. Platzer, "MARVIN: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *Proc. IEEE 39th Annual Computer Software and Applications Conf.*, Taichung, pp. 422–433, 2015.
[16] R. Riley. *A script to use the Android ART compiler to generate x86 binaries from apk files*, 2015. [Online]. Available: https://gist.github.com/rriley/ce38ab20532c1d7a2667.
[17] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu *et al.,* "Vetting undesirable behaviors in android apps with permission use analysis," in *Proc. the ACM SIGSAC Conf. on Computer & Communications Security*, New York, NY, USA, pp. 611–622, 2013.
[18] P. Faruki, V. Laxmi, A. Bharmal, M. S. Gaur and V. Ganmoor, "AndroSimilar: Robust signature for detecting variants of android malware," *Journal of Information Security and Applications*, vol. 22, pp. 66–80, 2015.
[19] Y. Aafer, W. Du and H. Yin, "DroidAPIMiner: Mining api-level features for robust malware detection in android," in *Proc. Int. Conf. on Security and Privacy in Communication Networks*, Sydney, Austrailia, vol. 127, pp. 86–103, 2013.
[20] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov and G. Giacinto, "Novel feature extraction, selection and fusion for effective malware family classification," in *Proc. the 6th ACM Conf. on Data and Application Security and Privacy*, New York, NY, USA, pp. 183–194, 2016.
[21] *Control flow graph* [Online], 2021. Available: https://en.wikipedia.org/wiki/Control_flow_graph.

[22] M. Fan, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 1890–1905, Aug. 2018.
[23] S. Alam, I. Traore and I. Sogukpinar, "Annotated control flow graph for metamorphic malware detection," *Computer Journal*, vol. 58, no. 10, pp. 2608–2621, 2015.
[24] S. Alam, R. N. Horspool and I. Traore, "MAIL: Malware analysis intermediate language: A step towards automating and optimizing malware detection," in *Proc. the 6th ACM Int. Conf. on Security of Information and Networks*, New York, NY, USA, pp. 233–240, 2013.
[25] A. Calleja, A. Martín, H. D. Menéndez, J. Tapiador and D. Clark, "Picking on the family: Disrupting android malware triage by forcing misclassification," *Expert Systems with Applications*, vol. 95, pp. 113–126, 2018.
[26] F. Á. Anilú, J. A. Carrasco-Ochoa, G. Sánchez-Díaz and J. F. Martínez-Trinidad, "Decision tree based classifiers for large datasets," *Computación y Sistemas*, vol. 17, no. 1, pp. 95–102, 2013.
[27] V. Rastogi, Y. Chen and X. Jiang, "DroidChameleon: Evaluating android anti-malware against transformation attacks," in *Proc. of the 8th ACM SIGSAC Symp. on Information, Computer and Communications Security*, ACM, New York, NY, USA, pp. 329–334, 2013.
[28] P. Sabanal, *Hiding Behind ART*, 2014. [Online]. Available: http://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp.pdf.
[29] B. Schwarz, S. Debray and G. Andrews, "Disassembly of executable code revisited," in *Proc. Working Conf. on Reverse Engineering*, Richmond, VA, USA, pp. 45–54, 2002.
[30] Nairobi-embedded. "ELF sections & sections and linux VMA mappings, 2017. " [Online]. Available: http://nairobi-embedded.org/040_elf_sec_seg_vma_mappings.html.
[31] S. S. Rathore and A. Gupta, "A comparative study of feature-ranking and feature-subset selection techniques for improved fault prediction," in *Proc. the 7th ACM India Software Engineering Conf.*, New York, NY, USA, 10, no. 7, pp. 1–10, Feb. 2014.
[32] P. Somol, B. Baesens, P. Pudil and J. Vanthienen, "Filter- versus wrapper-based feature selection for credit scoring," in *International Journal of Intelligent Systems*, vol. 20, no. 10, Oct. 2015.
[33] L. O. Hall, N. Chawla and K. W. Bowyer, "Decision tree learning on very large data sets," in *Proc. IEEE Int. Conf. on, Systems, Man, and Cybernetics*, San Diego, CA, USA, vol. 3, pp. 2579–2584, 1998.
[34] A. Stepanov, "On the kendall correlation coefficient," *arXiv: Statistics Theory*, July 2015.
[35] H. Cai, N. Meng, B. Ryder and D. Yao, "Droidcat: Effective android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2019.