Computers, Materials & Continua
DOI:10.32604/cmc.2022.020202
Article

Tech Science Press

Unified Detection of Obfuscated and Native Android Malware

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: 14 May 2021; Accepted: 30 June 2021

Abstract: The Android operating system has become a leading smartphone platform for mobile and other smart devices, which in turn has led to a diversity of malware applications. The amount of research on Android malware detection has increased significantly in recent years and many detection systems have been proposed. Despite these efforts, however, most systems can be thwarted by sophisticated Android malware adopting obfuscation or native code to avoid discovery by anti-virus tools. In this paper, we propose a new static analysis technique to address the problems of obfuscating and native malware applications. The proposed system provides a unified technique for extracting features from applications and native libraries using a selection algorithm that can extract a small set of unique and effective features for detecting malware applications rapidly and with a high detection rate. Evaluation using large Android malware detection datasets obtained from various sources confirmed that the proposed approach achieves very promising results in terms of improved accuracy, low false positive rate, and high detection rate.

Keywords: Android malware detection; native code; obfuscation; unified feature extraction

1 Introduction

The use of Android devices is rapidly and continuously rising, with the number of applications listed in the Google Play store currently close to three million. The Android operating system (OS) also comprised the largest share of the smartphone market, with an 87.7% market penetration [1]. Smartphones are being equipped with increasingly powerful hardware capable of running various applications. This rapid growth in the Android market and functionality has naturally led to cyber-attacks targeting Android devices through the use of malicious apps, which are evolving quite rapidly in terms of number and sophistication level [2]. Therefore, malware can be considered one of the most important security and privacy threats faced by Android users.

To address the malware threat, it is necessary to develop efficient defense systems for detecting and analyzing potentially dangerous applications in the form of, e.g., detectors that can inform users whether an installed app is malicious or benign. In recent years, the amount of research focused on Android malware detection has increased significantly and has led to the development of a large number of detection systems [3–11]. Unfortunately, the latest generation of malware applications can evade current static analysis techniques using native code [4] or code obfuscation [2].

As a countermeasure to such techniques, we introduce a lightweight and scalable solution. The proposed system employs a lightweight static feature set that is not limited to Java bytecode, native or obfuscated code and has a low computational cost. Although it looks at fewer factors than other systems, the proposed system is highly accurate and has a low false positive rate (FPR).

The results reported in this paper make the following contributions to the detection of Android malware.

- Unified native and obfuscation code-based malware detection: We introduce a static analysis method based on Android Runtime (ART). In this method, ART converts bytecode of Android application to native code in order to construct pure native binary, which is used to extract feature sets from malware applications, and is analyzed using machine learning. This approach has a great advantage to provide consistent and unified analysis of Android application regardless of whether it is bytecode or native application.
- Precise and compact feature selection: We propose an intersectional method for handling feature sets derived from filter-based feature selection. In this process, the feature set is reduced to a very small dimension by removing irrelevant features. In this manner, our system can detect malware within a very short time and with high accuracy.
- Lightweight malware detecting algorithm: Our system can detect Android malware applications within a short period of time from 'feature extraction' to 'detection.' Detection results reveal that the system can outperform its counterparts by a factor of 10–100.

The rest of the paper is organized as follows. Section 2 describes previous work related to this study. Our research motivation and a description of the proposed system's design feature selection and classification procedures are given in Section 3. Section 4 discusses the results of comparative performance evaluation of the proposed and selected conventional approaches. Finally, we present our conclusion in Section 5.

2 Related Work

Static analysis, one of the well-known approaches in Android malware detection, relies on static feature extraction from files residing in the Android Package Kit (APK). Most static analysis systems extract features from bytecode as well as executable, manifest, and other resource files [3–16] without pre-executing the tested application. As a result, static analysis tends to be lighter than other approaches such as dynamic analysis. However, many static analysis techniques can be easily hindered by bytecode obfuscation or native code. As the requirements for Android applications increase in terms of functionality and performance, there is an increased reliance on legacy native libraries or partial implementations in native code that are optimized to and dependent on specific hardware. In addition, code obfuscation is applied not only to the code but also to the resource [17] to prevent illegal reverse engineering.

Drebin [18] is a well-known static analysis approach. It can be hosted on a mobile application and provides detection scores in real time. MUDFLOW [19] mines unusual flows of predefined sensitive data types through the application of features defined in terms of the source and sink of sensitive data flow to machine learning. RevealDroid [20] extracts application programming interface (API), native code executable and linkable format (ELF), and Android metadata as feature sets. It is often discussed in the context of the native code in the Android malware landscape and has a very large selection of features extracted from all possible malware residing areas. Adagio [21] uses calls extracted from the smali code to construct function call graphs for encoding as features for machine learning.

However, these techniques tend to be vulnerable to code obfuscation and native malware. To overcome the weaknesses, Alam et al. [4] introduced DroidNative as a defense against native code malware. It detects malicious code hidden in native libraries using its own extended static analysis technique involving the use of 'Malware Analyze Intermediate Language' (MAIL) to generate 'Annotated Control Flow Graphs' (ACFGs) and 'Sliding Windows of Difference' (SWOD) to assess suspicious applications. ACFG is an extended version of CFG techniques, which applies graph-based representations of all paths that are traversable through a program during its execution [22]. By comparing the ACFG and SWOD results, native code malware can be detected with 93% accuracy at a 2.7% FPR and traditional malware sets can be predicted with 99.48% accuracy. MAIL also reduces detection time by using machine-learning techniques to create a 'similar detector' for detecting code that is highly similar to known malware data. The size of the generated ACFGs is large, so it requires expensive storage. For example, for 2,000 APKs, the total data size reaches 14 GB.

DroidSieve was the first application to focus on obfuscated malware [3]. According to Maiorca et al. [17], it can be applied not only to bytecode but also to native code or strings and even to assets. It uses static analysis to detect and classify obfuscated malware into multiple groups, which improves the accuracy and performance of the detection process. After extracting features from these groups, DroidSieve applies feature selection to reduce the number of features and improve detection performance. Using the Extra Tree algorithm, DroidSieve achieves a 99.15% detection rate for datasets containing both non-obfuscated and obfuscated malware and a 99.03% detection rate, along with a 0% FPR, for datasets containing only obfuscated malware. The system is considered to be one of the most accurate malware detection systems that apply only static analysis. However, instead of obtaining features from native binary, DroidSieve checks only for the presence of the ELF and header in the native code. Therefore, it can have a low detection rate for native malware.

Marvin [5] provides a new approach that combines static and dynamic analysis. It can extract hundreds of thousands of features from a huge dataset within a few hours and achieves 98.24% accuracy and 0.04% FPR. Marvin's capabilities illustrate how the combination of both analysis types improves performance. The system further leverages machine learning to detect malware and can produce risk scores for even unknown tested applications. Iterative classifier fusion system (ICFS) [14] adopts a similar combined extraction approach in which static analysis is used to request and extract permission from the APK and dynamic analysis is used to extract CFG and Dalvik bytecode. The two-tier ICFS training procedure applies three different classification algorithms, and the application also includes a feature selection procedure based on the use of the wrapper subset evaluator (WSE) combined with particle swarm optimization (PSO) algorithms tailored to each feature and running on a different dataset (as WSE is a technique used to perform feature selection for specific training algorithms, each algorithm is given its own feature set). PSO is a relative population-based stochastic global optimization algorithm [23] that, upon obtaining feature sets from WSE, combines their features through a double-union operation. While Marvin and ICFS have high accuracies and short detection times, there is no indication in the related literature that either can detect obfuscated code or native code; correspondingly, neither can be assumed to be particularly effective in detecting malicious applications with these types of malware.

VILO [24] is a Windows-based (non-Android) malware application that extracts and then permutes features from assemblies using an n-permutation technique under machine instruction. In preparing this study, the authors considered using n-permutation instead of n-gram to build features, as permutation can reduce the feature vector space and using a nearest-neighbor algorithm to produce features results in 95% accuracy in variant malware detection. However, as there are significant differences between Android and Windows malware applications, such a feature builder would have significant limitations in application to Android platforms.

3 Proposed Algorithm

We propose a method for implementing an NIDS that can process packets received in real time and determine whether an attack has occurred. The proposed algorithm generates the latest features by updating the feature table for each session whenever a packet is received, and it determines whether an attack has occurred using the features. As shown in Fig. 1, the proposed system is configured to simultaneously increase both classification speed and accuracy by utilizing two classifiers.

As mentioned earlier, most Android malware detection systems use static [3,5,12,15] or dynamic [5,15] analysis feature extraction or CFG-based comparison [19,21] to detect malwares. However, because most CFG-based systems use exact pattern matching, they can be easily thwarted by simply using code obfuscation. This problem also occurs in a dynamic context through the injection of malicious code into the native binary. To address these problems, the authors applied a different approach in which potential malware features are extracted from the native code and in the Android application, enabling the detection of even those malware applications that use code obfuscation or native binary injection.

The proposed system also adopts the VILO feature extraction algorithm to obtain better performance and accuracy; instead of n-permutation, however, it applies n-gram to extract all possible uses of each feature. It also uses unified feature extraction to handle both of bytecode and native code, simultaneously.

3.1 Motivation

Starting at version 5.0, i.e., Lollipop, Android replaced Dalvik virtual machine (DVM) with ART, which introduced a new feature called Ahead of Time (AOT) compilation to transform the Android bytecode into native binaries at installation time using the on-device tool, *dex2oat* [12,25]. ART suggests the possibility of new static analysis approaches. Currently, Android applications can be built using either pure bytecode or the combination of native code and bytecode. Although the latter case generally requires separate analyses of the bytecode and native code, ART can be used to generate native binary from either bytecode or native bytecode. In this manner, ART can unify and simplify the malware detection task.

3.2 System Design

Most static analysis techniques rely on API calls [16], bytecode [14], code structure [15], permission [13,14], or intent [5]. Here, we apply a different approach in which ART is used in place of existing static analysis techniques to translate bytecode into native code for disassembling and extracting assembly code into features. Fig. 1 shows the overview of our system starting pre-processing to detection. In pre-processing, the entire Android app is converted into native code so that features can always be created in the same way. In feature extraction, an n-gram is applied to generate a feature candidate group, and only features that are identically selected from six feature selection algorithms are used for machine learning.

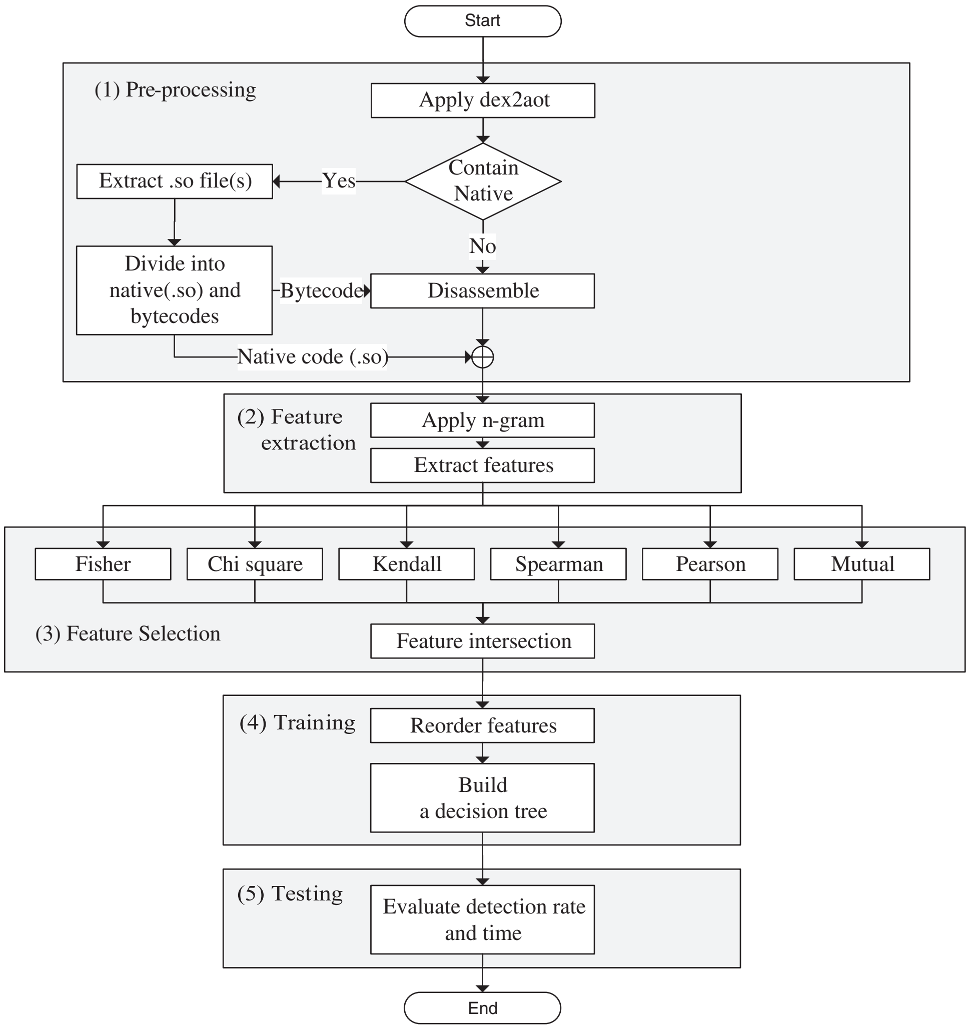


Figure 1: Overview of proposed detection process

3.2.1 Pre-Processing

In the pre-processing step, the basic operations provided by Android Open Source Project (AOSP) are processed to check the APK file for the presence of native code. An application containing native binary has shared object files in APK with a ".so" extension; after obtaining this file, ART is used to convert it into AOT.

3.2.2 Feature Extraction

The feature extraction stage uses ART to obtain assembly codes from the bytecode and native library extracted through pre-processing. Using n-gram technique, the assembly codes are converted into a candidate feature set for machine learning. In the context of the proposed approach, the n-gram technique involves the extraction of contiguous sequences of *n* items from a given sequence of machine instruction. In [26], four-gram was used to combine sequences of six initial bytes into a sequence that could store a very large amount of data. The proposed method also extracts very large sets of unique features from each dataset; however, since the number of datasets is large, only two-gram is considered. Note that the dimension of a feature set can vary; as the overall feature dimension is very large, the dimension of the feature is optimized by using pre-feature selection to remove irrelevant features.

- Disassemble

As shown in Fig. 1, the disassembler disassembles the converted AOT file into the readable text files segment, address, raw bytes, and instruction [24,26,27]. Fig. 2 shows each part of a disassembled output file. According to Schwarz et al. [28], there are two standard techniques for disassembling native binary files: linear sweep and recursive traversal. Linear sweep is a straightforward approach that decodes all contents within a segment into machine codes. However, as data and codes are mixed under linear sweep, it can produce incorrect results. Recursive traversal functions identically to linear sweep except that it does not disassemble data. As a result, recursive traversal can sometimes fail to disassemble when it does not find valid machine instruction. To take advantage of the respective advantages of the two techniques, we developed a hybrid disassembly algorithm that combines the best qualities of each approach [28]. The hybrid disassembly algorithm compares each machine instruction obtained from recursive traversal and linear sweep and skips to next instruction if it is identified as data.

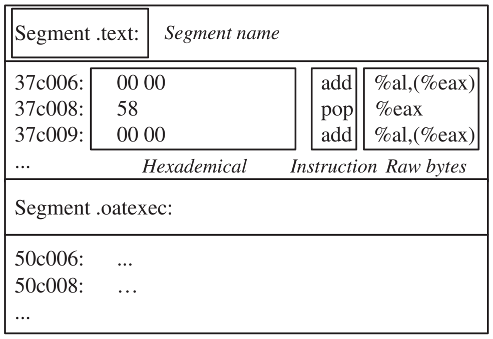


Figure 2: Disassembled AOT in text format

All disassembly information can be used as features for malware detection [24]. Although most systems [24,26,27] use hexadecimal, raw bytes, or the entirety of the disassembled data, we choose only instructions and segments as the feature sets for extraction, as others were considerably more expensive in term of the file size and optimization overhead as shown in Fig. 3.

- Feature Extraction

In the feature extraction step, the system extracts instructions, groups them into segments, and then saves them into a file, as shown in Fig. 4. The primary purpose of extracting segments is to count how many segments contain a given instruction, as more segments correspond to more features.

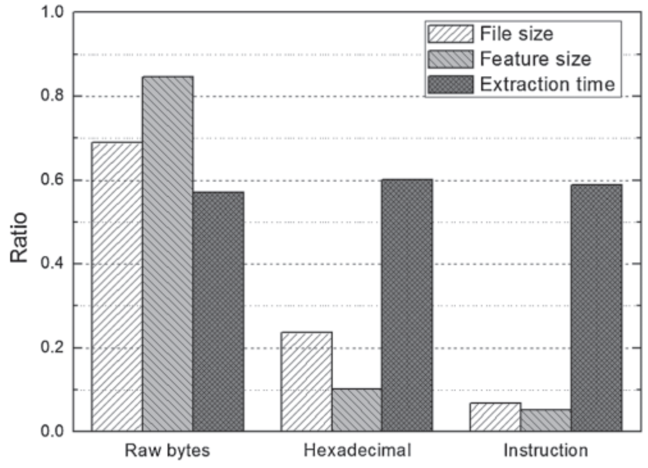
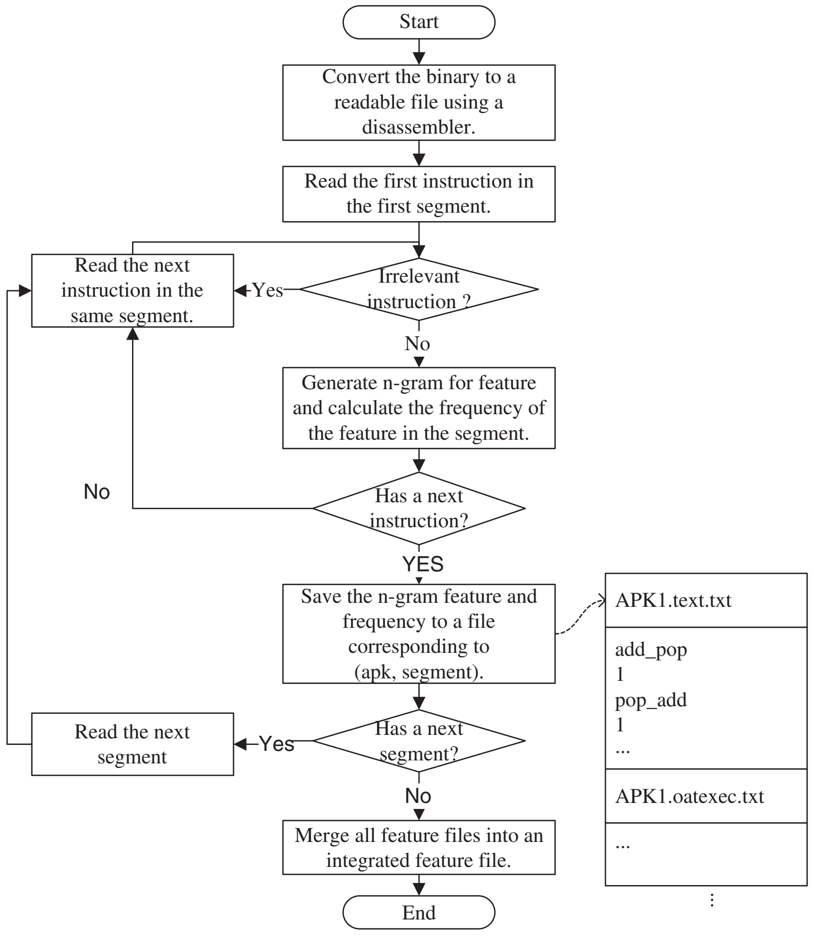


Figure 3: Relative file size, feature dimension, and extraction time for each data type used for feature extraction. The ratio is one if all data, i.e., raw bytes, hexadecimal, and instruction are used



Start
Convert the binary to a readable file using a disassembler.
Read the first instruction in the first segment.
Irrelevant instruction ?
Yes
Read the next instruction in the same segment.
No
Generate n-gram for feature and calculate the frequency of the feature in the segment.
Has a next instruction?
No
YES
Save the n-gram feature and frequency to a file corresponding to (apk, segment).
APK1.text.txt
add_pop
1
pop_add
1
...
APK1.oatexec.txt
...
Has a next segment?
Yes
Read the next segment
No
Merge all feature files into an integrated feature file.
End

Figure 4: Process for extracting features from an application

As n-gram-based analysis extracts all possible n-grams from a training dataset [26,27] and therefore provides larger feature dimensions than n-permutation, n-gram is used to combine the instructions. As an example of the different results obtained by the two approaches, the instruction sequence "pushf, add, add, pushf" produces three two-grams ("pushf, add," "add, add," and "add, pushf") but only two two-permutations ("pushf, add" and "add, add"). Tab. 1 shows the number of feature dimensions produced by each n-gram and n-permutation for an un-obfuscated dataset, respectively.

Table 1: Feature dimensions produced by n-gram and n-permutation by segments used from an un-obfuscated dataset sample

Method	All loadable code segments	.text only
n-gram	198,363	66,623
n-permutation	91,455	35,863

Fig. 4 shows how each instruction is combined using n-gram. As the n-gram-based mechanism generates a very large dataset, the feature extraction algorithm runs several times to reduce the footprint and extraction time. This step is repeated until there are no remaining loadable code segments. According to [29], the ELF in the Android ART has more segments compared than the original ELF. The AOT contains *.oatdata*, the segment containing the Of Ahead Time (OAT) headers and the embedded original DEX, and *.oatexec*, the segment that containing the generated native code for the compiled functions. Each disassembled file contains *.oatdata* in the *.rodata* section and *.oatexec* combined with *.oatlastword* in the *.text* section.

In feature extraction stage, irrelevant instructions are removed in the extraction process [4] since some features generated from the instructions are too general to determine malware code. For example, RDRAND is an instruction for reading random numbers that are widely used and therefore irrelevant for malware analysis. As shown in Fig. 4, upon finding an irrelevant instruction, the system will skip to the next one. This stage, called pre-feature selection process, is essential because it decreases the total feature dimension and speeds up the training performance of the machine learning model.

Once the features of each segment have been extracted, each feature file for every segment is merged into an integrated feature file for each application. This file is composed of columns corresponding to the respective feature names in which the frequency of use of each instruction by that feature is listed. The feature fusion process is implemented by examining each feature and its corresponding value on the APK file. Once the features have been fused, the missing values should be filled in. Although the treatment of missing values is a common problem in machine learning, its impact on classification can be significant for constructing a fine-training and accurate model [30].

In the proposed system, it is possible to use only *.text* segments. As *.text* contains all of the main processes of the application, it is the best segment for malware detection. Therefore, by using only the *.text* segment the feature dimension and processing time can be significantly decreased in cost of a slight loss in accuracy.

3.2.3 Feature Selection

The second primary component of the proposed method is feature selection. According to [31], there are two feature selection methods, namely, filter- and WSE-based methods, as discussed in Section 2. To test as many different algorithms as possible, we chose filter-based selection that is not limited to a specific algorithm unlike WSE. The subset selection procedure in filter-based selection is independent of the learning algorithm and is generally performed during a pre-processing step, which obviously leads to a faster learning pipeline.

There are many feature selection algorithms available [32–35]. Following extensive testing and analysis, we ultimately selected the six with the highest accuracy: Pearson correlation [33], Mutual information [33], Kendall correlation [34], Spearmen correlation [32], Chi squared [33], and Fisher scored [35]. Each of these performs a unique selection algorithm based on the class label to define whether an application is malware or benign and produces different numbers of features for the same dataset. Our approach is to generate a final feature set ($\mathbf{F}_{Final}$) by intersecting the selected feature sets and then chooses only common features.

Conventional feature selection methods apply a union operation to the features produced by multiple feature extraction algorithms to define a feature set [14]. Although it is clear that this approach has advantages in avoiding missing critical features, more features do not necessarily guarantee—and can sometimes reduce—accuracy. Furthermore, the use of many different feature selection algorithms simultaneously can generate too large feature sets. As a compromise, we adopted a different approach to select critical features through a fine-grained feature set based on intersection as follows:

$$\mathbf{F}_{Final} := \mathbf{F}_{Fisher} \cap \mathbf{F}_{ChiSquared} \cap \mathbf{F}_{Kendall} \cap \mathbf{F}_{Spearman} \cap \mathbf{F}_{Pearson} \cap \mathbf{F}_{Mutual},$$

where $\mathbf{F}_X$ is the feature set obtained by algorithm X and '∩' is the intersection operator.

3.2.4 Training and Testing Procedures

After obtaining a fine-grained feature set, machine learning is applied to determine whether the application is malware or benign. To develop a model to train the Decision Tree algorithm, which can quickly handle large feature sets, is applied. Easily obfuscated features can be regarded as less important, and features that are invariant under obfuscation can be more important.

In the testing process, a built-in scoring model [36] is used to evaluate testing sets based on a calculated scoring factor. The trained model can then apply this scoring factor to dataset features to decide whether an application is malware or benign.

4 Performance Evaluation

4.1 The Environment

To assess the performance of the proposed method, we carried out intensive comparative simulation using un-obfuscated, obfuscated, and native datasets. To measure overall performance in a practical scenario, we also conducted a simulation using a mixed dataset obtained by combining the datasets. The individual datasets were themselves obtained by combining multiple datasets from various sources such as Drebin [18], Marvin [5], and Praguard [17]. More specifically speaking, benign data were solely obtained from Marvin dataset while native but non-obfuscated data were obtained from Marvin and Drebin datasets. Praguard dataset contains obfuscated data obtained by obfuscating the MalGenome and Cotagio minidump datasets, so obfuscated data were obtained only from Praguard dataset. The specifications of each dataset are listed in Tab. 2.

Table 2: Datasets used for performance evaluation

Dataset	# of malware	# of benign apps	Total
Un-obfuscated	5,560	4,249	9,809
Obfuscated	8,982	8,498	17,480
Native	4,329	2,387	6,716
Mixed	28,724 (Native:3.6k, Obf:9k, Un-obf:16k)	92,544 (Native:7.5k, Un-obf:85k)	121,268

To test the effectiveness of our approach in detecting malware applications, we compared it with five of the latest approaches—Adagio, MUDFLOW, RevealDroid, Droid-Native, and DroidSieve among latest researches [4,5,17,18,37,38]. Tab. 3 lists the capabilities and years of introduction for each competitor.

Table 3: Malware types detectable by existing algorithms

Name	Year	Obfuscation malware	Native malware
Adagio	2013	-	-
MUDFLOW	2015	-	-
RevealDroid	2016	Supported	Partially supported
DroidNative	2016	-	Supported
DroidSieve	2017	Supported	Partially supported

The comparative evaluation was based on results reported in the literature from 2013 to 2017 under four scenario cases: traditional un-obfuscated, obfuscated, native, and mixed datasets. As each system has its own purposes and limitations, we took the capabilities in Tab. 3 into account in the comparison.

We first assessed the ability of our feature selection algorithm to decrease the number of feature dimensions. As the feature dimension directly affects detection time, it is very important to remove all redundant and unimportant features from the initial feature set. However, to avoid performance degradation in the selection process, the removal of relevant features should be avoided; accordingly, the malware detection performance of the proposed method was subsequently compared with those of the competitors.

4.2 *Comparison of Feature Dimension by Segment Type for Feature Extraction and Selection*

Tab. 4 shows the number of feature dimensions for each dataset before and after performing feature selection using all loadable segments. For reference, we also show the number of feature dimensions obtained using text segments only. We can see that the feature sets are reduced significantly through feature selection. Although each feature set produces slightly different results, the reduction ratio approaches 92% for all dataset types. The obfuscated dataset produces the smallest reduction ratio. Obfuscation techniques generate more complicated dex code than the original one, resulting in larger n-grams and, therefore, less reduction.

Table 4: Comparison of the size of feature dimensions in loadable segments and text-only segments for each dataset

Dataset	.text only	All loadable segments	All loadable segments w/feature selection (ratio %)
Un-obfuscated	61,880	143,177	11,834 (8.3)
Obfuscated	73,104	211,737	19,240 (9.1)
Native	34,147	112,141	8,001 (7.1)
Mixed	93,194	231,238	20,418 (8.8)

The effectiveness of the proposed algorithm in terms of feature dimension reduction was further assessed by comparing its results on a mixed dataset with those of Pearson, Mutual, Kendal, Spearman, Chi square, and Fisher as shown in Fig. 5. Although all of the algorithms were able to reduce the number of original feature dimensions by more than half, we can see that the proposed algorithm achieved the greatest reduction.

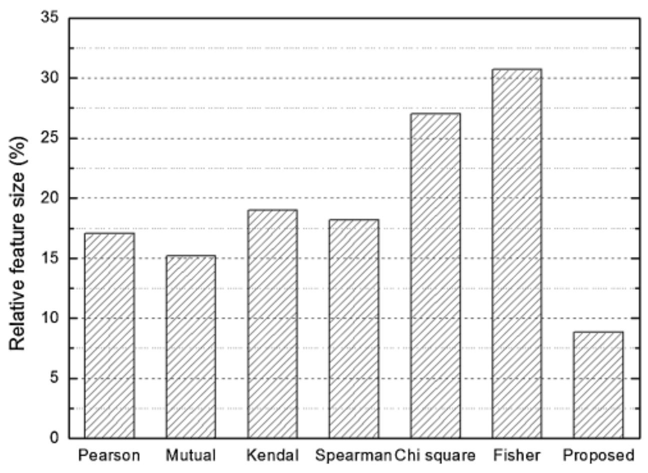
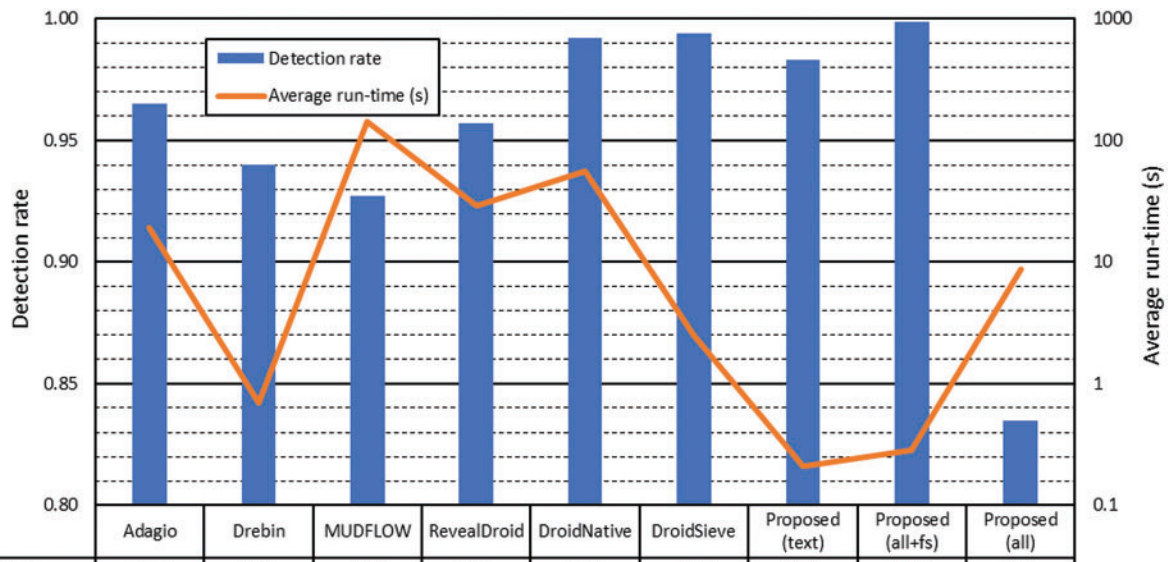


Figure 5: Ratio of feature dimension following feature selection to original feature dimension for various algorithms applied to the mixed dataset

4.3 Un-Obfuscated Malware Detection

To evaluate the malware detection performance of our algorithm, we first compared its performance with those of the competitors on the un-obfuscated dataset. The measured detection rate and average run-time for each application are shown in Fig. 6. Drebin had the shortest detection time. As Drebin analyzes manifest files in a static manner, it performs very well in terms of run-time but poorly in terms of detection rate. Thus, Drebin had a detection rate that was lower than that of the proposed algorithm by around 5% and, surprisingly, required approximately twice the processing time.

MUDFLOW, which generates features using the source and sink of data flow in a static manner, was the slowest algorithm. As MUDFLOW analyzes manifest in addition to bytecode, it takes a long time to build a CFG while consuming large amount of memory and therefore scales poorly, making it impractical.



	Adagio	Drebin	MUDFLOW	RevealDroid	DroidNative	DroidSieve	Proposed (text)	Proposed (all+fs)	Proposed (all)
Detection rate	0.97	0.94	0.93	0.96	0.99	0.99	0.98	1.00	0.84
Average run-time (s)	19.1	0.7	143.121	28.8	56.4	2.5	0.21	0.287	8.8

Figure 6: Detection rate and average run-time per sample by malware detection algorithm for un-obfuscated malware dataset. No source code available for Drebin and DroidSieve, so their results were estimated using [3,18]

RevealDroid and Adagio produced very similar results in terms of both detection rate and run-time. RevealDroid also statically utilizes manifest data and bytecode, while Adagio generates a CFG from bytecode without the use of manifest data, and therefore processes faster than RevealDroid. However, Adagio's application of a call-indirection transformation [21] to a full application makes it slower than the proposed feature extractor. Our algorithm also performed faster than RevealDroid owing to its optimized pre-feature selection without consideration of the manifest file. The results in Fig. 6 confirm that the proposed approach outperforms RevealDroid and Adagio in terms of both detection and run-time.

DroidNative employs ACFG and SWOD to obtain a high detection rate. As this results in the generation of two different graphs, a longer run-time is needed to detect all applications. The proposed method outperformed Droid-Native with 1% edge in accuracy and a much faster detection rate.

DroidSieve had the highest detection rate and the lowest run-time among all tested algorithms. DroidSieve extracts all possible features from the manifest, bytecode, and metadata in the native code. Although it handles a large number of features for detection, each process is carried out in a parallel manner, allowing for the very rapid detection of malware. Nevertheless, the proposed algorithm outperformed DroidSieve with a slightly better detection rate and a tenfold faster processing speed.

Overall, the proposed method outperformed the competitors in terms of both detection rate and run-time. Whereas the existing approaches analyze multiple data types, i.e., manifest, bytecode, library, etc., the proposed system simply uses an ART to generate an AOT file, which, as discussed previously, contains all features for bytecode and native code, to boost performance in terms of detection rate and average run-time.

The results in Fig. 6 demonstrate that the proposed approach can obtain very high detection rates and low run-times using only text segments. We further tested our approach on all segments in the absence of feature selection. The irrelevant feature dimension increases with the overall size, reducing the effectiveness of malware detection. On the other hand, including more segments in the feature selection process improves performance in terms of detection rate, although there is a slight increase in run-time. The last two rows of Fig. 6 show, respectively, the detection rates achieved by the proposed method on all segments with and without feature detection. It is seen that applying feature extraction increases detection rate by 16.4%, confirming the ability of the method to select tightly relevant features to enhance the detection rate. It is also seen that the use of feature selection significantly reduces the run-time. As the use of all segments without feature selection resulted in such a poor performance, this case was excluded from further evaluation. Except this case, our approach also shows 0.5% FPR, which is quite small. Such results confirm that our approach is very promising to achieve high detection performance for un-obfuscated malwares.

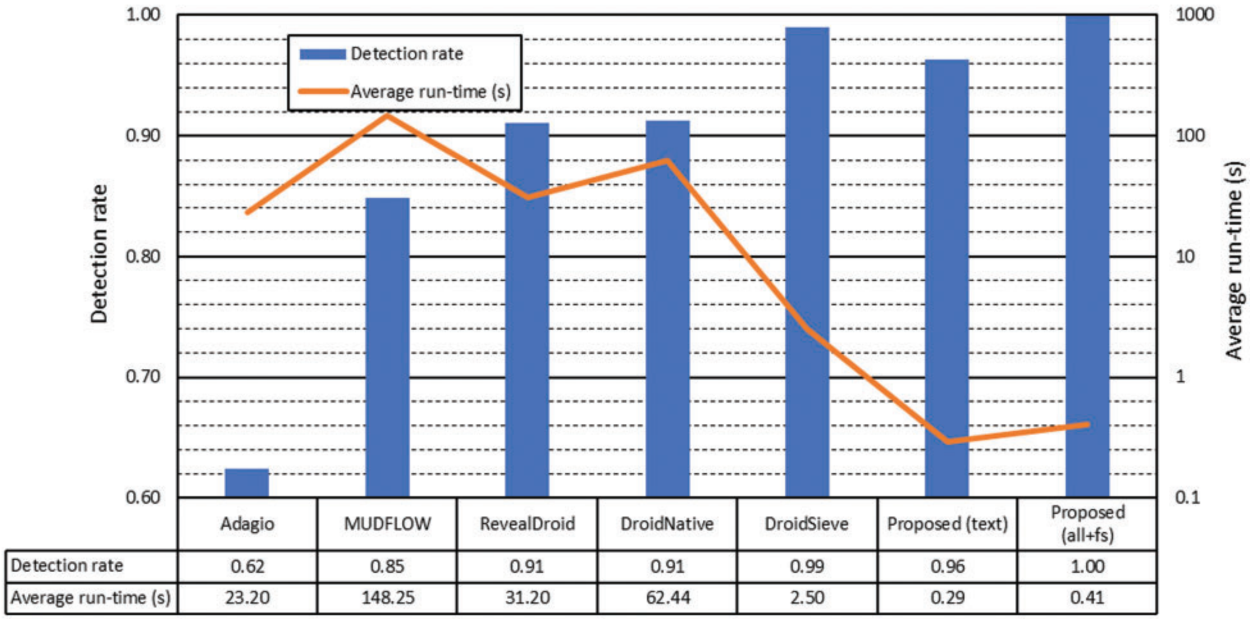
4.4 Obfuscated Malware Detection

To compare the proposed and existing methods in detecting malware using obfuscation, another performance evaluation was carried out on an obfuscated dataset as shown in Fig. 7. It is seen that Adagio, MUDFLOW, and DroidNative perform poorly in terms of detection rate for obfuscated dataset. These algorithms are not designed to detect obfuscated malware; in particular, DroidNative can suffer from control flow flattening caused by excessive control flow when ART is used for a very large APK file. The incidence of misdetection can also increase because obfuscation causes excessive control flow in generating ACFG and SWOD.

Although RevealDroid is one of a few existing systems designed to detect obfuscated malware, its results in Fig. 7 correspond to only a moderate detection rate. DroidSieve, by contrast, is specifically designed to detect obfuscated malware. As mentioned earlier, the algorithm extracts all possible features from all possible file types, including resources, to develop a more effective set of features for the detection of obfuscated code. It is, therefore, unsurprisingly that DroidSieve performed best in terms of this metric; however, our approach had a detection rate nearly as high as that of DroidSieve and higher than those of other four algorithms. In addition, it again had the shortest run-time with 0 FPR among the competitors. The longer run-time of DroidSieve than that of the proposed method is likely a result of its increased feature generation. Such results prove that our approach provides a very confident solution for detecting obfuscated malwares.

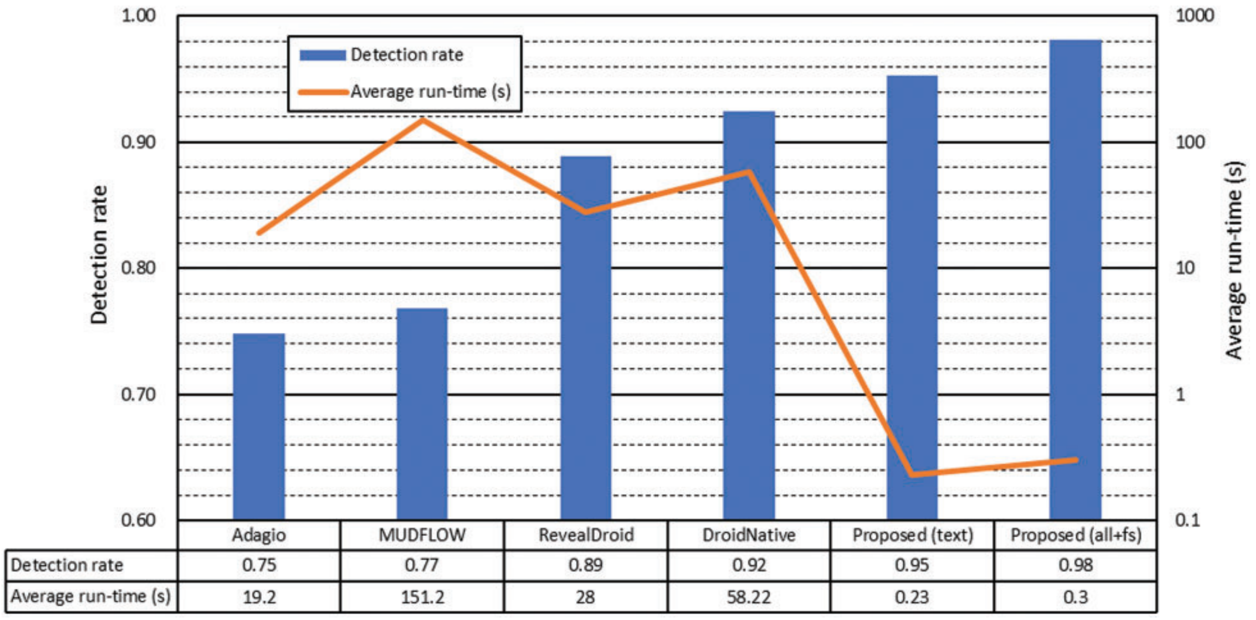
4.5 Native Malware Detection

The algorithms were then compared in terms of their ability to detect malware residing in a native library. As shown in Fig. 8, DroidNative, which uses a combined ACFG and SWOD to detect malware through comparison of the combined flows, achieves the highest native detection rate among the existing approaches but is still outperformed by the proposed method by about 7%. Furthermore, DroidNative increases its processing time for each instance when it generates overall application control flow, making it very difficult to satisfy practical requirements.



	Adagio	MUDFLOW	RevealDroid	DroidNative	DroidSieve	Proposed (text)	Proposed (all+fs)
Detection rate	0.62	0.85	0.91	0.91	0.99	0.96	1.00
Average run-time (s)	23.20	148.25	31.20	62.44	2.50	0.29	0.41

Figure 7: Detection rate and average run-time per sample by malware detection algorithm for obfuscated malware dataset. No source code available for DroidSieve, so its result was estimated using [3]



	Adagio	MUDFLOW	RevealDroid	DroidNative	Proposed (text)	Proposed (all+fs)
Detection rate	0.75	0.77	0.89	0.92	0.95	0.98
Average run-time (s)	19.2	151.2	28	58.22	0.23	0.3

Figure 8: Detection rate and average run-time per sample by malware detection algorithm for native malware dataset

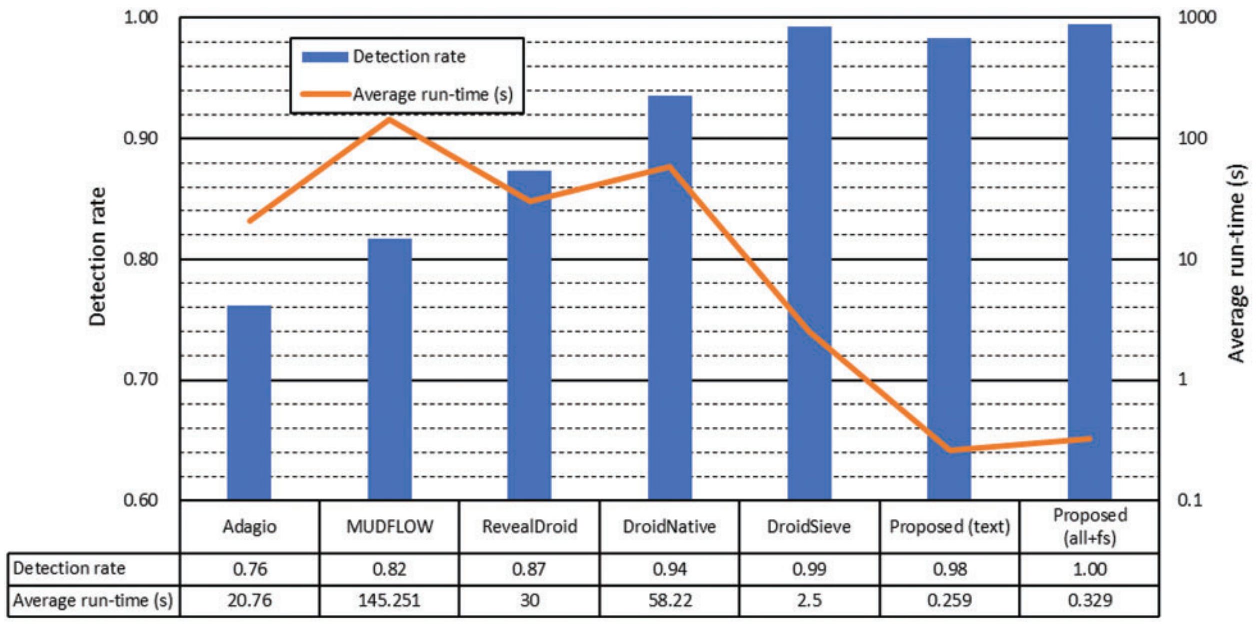
RevealDroid also focuses on native malware detection. However, as it focuses only on meta-data it looks only at partial native code information and can fail to achieve a high detection rate when the majority of the malicious code is located in the native library. On the other hand, as a result of its simplicity its run-time is reduced by nearly half compared to that of DroidNative.

MUDFLOW and Adagio, which were not designed for detecting native malware, achieved detection rates of approximately 76% on the native malware dataset. Some malware applications implement malware functions in the native library as well as in the bytecode; it appears that Adagio and MUDFLOW can partially detect native malware applications of this type.

Current approaches such as RevealDroid extract features in two phases, one for extracting code parts and another for extracting non-code parts, i.e., the manifest file and other resources. Both feature sets are then combined into one. This approach enables simple and efficient implementation in exchange for an increase in run-time. Our proposed system, by contrast, uses ART to provide unified feature extraction regardless of whether a malicious function is implemented using native code or bytecode. Furthermore, it uses a pre-defined feature selection process during feature extraction. This results in the removal of irrelevant features before the dataset is passed over to the feature selection procedure, which gives our approach both high detection rate and low run-time.

4.6 Mixed Malware Detection

Fig. 9 shows the performance comparison results obtained by applying the respective algorithms to a mixed dataset comprising non-obfuscated, obfuscated, and native applications. As this reflects a more realistic scenario, the results are very important in the assessment of the practical performance of each approach. Our algorithm was designed to focus on enhancing performance on both native code and code obfuscation, and therefore the proposed method outperformed the competitors in terms of both run-time and detection rate in a mixed malware dataset. Our approach shows slightly higher detection rate while it maintains quite low FPR (0.4%) compared with the DroidSieve which shows the best performance among all competing algorithms. From these results, we can say that it guarantees the high performance for various real environments.



	Adagio	MUDFLOW	RevealDroid	DroidNative	DroidSieve	Proposed (text)	Proposed (all+fs)
Detection rate	0.76	0.82	0.87	0.94	0.99	0.98	1.00
Average run-time (s)	20.76	145.251	30	58.22	2.5	0.259	0.329

Figure 9: Detection rate and average run-time per sample by malware detection algorithm for mixed malware dataset

5 Conclusion

In this paper, we presented a new approach for detecting malware residing in either bytecode or native code. Our solution is both scalable and efficient because it generates unified native code using the Android ART to build feature sets for applications. Also, it outperforms state-of-art of malware detection methods in terms of detection rate and processing performance, so it overcomes malware hindering techniques such as code obfuscation and native implementation.

The results of extensive performance evaluation revealed that, while choosing a classification algorithm is important, the method for extracting and selecting features is crucial to the improvement of malware detection accuracy. To address this requirement, we developed a unique feature extraction and selection approach to optimizing feature sets with higher detection rates and reduced detection times. Obfuscation and native malware remain a largely open problem in the Android malware detection landscape; under a mixed malware environment containing both obfuscated and native code, our approach can achieve a very high detection performance level with a 99.5% detection rate and a 0.4% FPR. As Android has become a global operating system that aids millions of people in their daily lives, we expect our solution will play an important role to avert Android malware attacks.

Funding Statement: This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2019R1F1A1062320) and the Information Technology Research Center (ITRC) Support Program supervised by the Institute for Information and Communications Technology Planning and Evaluation (IITP) (IITP-2021-2016-0-00313).

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

References

[1] S. O'Dea, "Global market share held by smartphone operating systems 2009–2018, by quarter," [Online]. Available: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems (November 25, 2020).

[2] V. Rastogi, Y. Chen and X. Jiang, "Droidchameleon: Evaluating android anti-malware against transformation attacks," in *Proc. ACM SIGSAC Symposium on Information, Computer and Communications Security*, New York, NY, USA, pp. 329–334, 2013.

[3] 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. ACM on Conf. on Data and Application Security and Privacy*, New York, NY, USA, pp. 309–320, 2017.

[4] 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.

[5] M. Lindorfer, M. Neugschwandtner and C. Platzer, "MARVIN: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *Proc. IEEE Annual Computer Software and Applications Conf.*, Taichung, pp. 422–433, 2015.

[6] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo and F. Massacci, "Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications," in *Proc. ACM Conf. on Data and Application Security and Privacy*, New York, NY, USA, pp. 37–48, 2015.

[7] L. Deshotels, V. Notani and A. Lakhotia, "Droidlegacy: Automated familial classification of android malware," in *Proc. ACM SIGPLAN on Program Protection and Reverse Engineering Workshop, PPREW'14*, New York, NY, USA, no. 3, 2014.

[8] X. Sun, Y. Zhongyang, Z. Xin, B. Mao and L. Xie, "Detecting code reuse in android applications using component-based control flow graph," in *Proc. Springer ICT Systems Security and Privacy Protection*, Berlin, Heidelberg, Germany, pp. 142–155, 2014.

[9] S. Seo, A. Gupta, A. M. Sallam, E. Bertino and K. Yim, "Detecting mobile malware threats to homeland security through static analysis," *Journal of Network and Computer Applications*, vol. 38, pp. 43–53, 2014.

[10] J. Huang, X. Zhang, L. Tan, P. Wang and B. Liang, "Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proc. ACM Int. Conf. on Software Engineering, ICSE*, New York, NY, USA, pp. 1036–1046, 2014.

[11] 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.

[12] Android. ART and Dalvik, 2017. [Online]. Available: https://source.android.com/devices/tech/dalvik.

[13] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu *et al.*, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proc. ACM SIGSAC Conf. on Computer & Communications Security*, New York, NY, USA, pp. 611–622, 2013.

[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. 600–610, 2019.

[15] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez and J. Blasco, "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families," *Expert Syst. Appl*, vol. 41, no. 4, pp. 1104–1117, 2014.

[16] 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, SecureComm*, Sydney, Australia, vol. 127, pp. 86–103, 2013.

[17] D. Maiorca, D. Ariu, I. Corona, M. Aresu and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on android malware," *Computers and Security*, vol. 51, pp. 16–31, 2015.

[18] D. Arp, M. Spreitzenbarth, M. Huebner, H. Gascon and K. Rieck, "Drebin: Efficient and explainable detection of android malware in your pocket," in *Proc. Annual Network and Distributed System Security Symposium, NDSS*, San Diego, California, 2014.

[19] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt *et al.*, "Mining apps for abnormal usage of sensitive data," in *Proc. IEEE/ACM Int. Conf. on Software Engineering*, Florence, Italy, pp. 426–436, 2015.

[20] 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, pp. 1–29, 2018.

[21] H. Gascon, F. Yamaguchi, D. Arp and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proc. ACM Workshop on Artificial Intelligence and Security, AISec*, ACM, New York, NY, USA, pp. 45–54, 2013.

[22] J. Kennedy and R. C. Eberhart, *Swarm Intelligence*, Boston, MA, USA: Springer, pp. 187–219, 2001.

[23] A. Perrier, "Feature importance in random forests," 2015. [Online]. Available: http://alexperrier.github.io/jekyll/update/2015/08/27/feature-importance-random-forests-gini-accuracy.html.

[24] A. Lakhotia, A. Walenstein, C. Miles and A. Singh, "VILO: A rapid learning nearest-neighbor classifier for malware triage," *Journal in Computer Virology*, vol. 9, no. 3, pp. 109–123, 2013.

[25] 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.

[26] M. M. Masud, L. Khan and B. M. Thuraisingham, "A scalable multi-level feature ex-traction technique to detect malicious executables," *ISF*, vol. 10, no. 1, pp. 33–45, 2008.

[27] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov and G. Giacinto, "Novel feature extraction, selection and fusion for effective malware family classification," in *Proc. ACM Conf. on Data and Application Security and Privacy*, New York, NY, USA, pp. 183–194, 2016.

[28] B. Schwarz, S. Debray and G. Andrews, “Disassembly of executable code revisited,” in *Proc. IEEE Working Conf. on Reverse Engineering*, Richmond, VA, USA, pp. 45–54, 2002.

[29] Nairobi-embedded. “ELF sections & sections and linux VMA mappings,” 2017. [Online]. Available: http://nairobi-embedded.org/040_elf_sec_seg_vma_mappings.html.

[30] E. Acuna and C. Rodriguez, “The treatment of missing values and its effect on classifier accuracy,” in *Springer Classification, Clustering, and Data Mining Applications*, Berlin, Heidelberg, pp. 639–647, 2004.

[31] P. Somol, B. Baesens, P. Pudil and J. Vanthienen, “Filter- versus wrapper-based feature selection for credit scoring,” *International Journal of Intelligent Systems*, vol. 20, no. 10, pp. 985–999, 2015.

[32] H. Lim, J. Lee and D. Kim, “Optimization approach for feature selection in multi-label classification,” *Pattern Recognition Letters*, vol. 89, pp. 25–30, 2017.

[33] C. R. Rao, “Karl Pearson chi-square test the dawn of statistical inference,” *Springer Goodness-of-fit Tests and Model Validity*, pp. 9–24, 2002.

[34] A. Stepanov, “On the kendall correlation coefficient,” *ArXiv*, 2015.

[35] Q. Gu, Z. Li and J. Han, “Generalized fisher score for feature selection,” in *Proc Conf. on Uncertainty in Artificial Intelligence*, UAI, Barcelona, Spain, pp. 266–273, 2011.

[36] F. Á., Anilú, J. A. Carrasco-Ochoa, G. Sánchez-Díaz and J. F. Martínez-Trinidad, “Decision tree based classifiers for large datasets,” *Computacióny Sistemas*, vol. 17, no. 1, pp. 95–102, 2013.

[37] Z. H. Qaisar, S. H. Almotiri, M. A. Al Ghamdi, A. A. Nagra and G. Ali, “A scalable and efficient multi-agent architecture for malware protection in data sharing over mobile cloud,” *IEEE Access*, vol. 9, pp. 76248–76259, 2021.

[38] L. Xiao, Y. Li, X. Huang and X. Du, “Cloud-based malware detection game for mobile devices with offloading,” *IEEE Transactions on Mobile Computing*, vol. 16, no. 10, pp. 2742–2750, 2017.