Received July 31, 2021, accepted August 16, 2021, date of publication September 6, 2021, date of current version September 13, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3110408

Impact of Code Deobfuscation and Feature Interaction in Android Malware Detection

YUN-CHUNG CHEN[1], HONG-YEN CHEN[2], (Graduate Student Member, IEEE), TAKESHI TAKAHASHI[3], (Member, IEEE), BO SUN[3], AND TSUNG-NAN LIN[4,5], (Senior Member, IEEE)

[1]Graduate Institute of Electrical Engineering, National Taiwan University, Taipei 10617, Taiwan
[2]Graduate Institute of Communication Engineering, National Taiwan University, Taipei 10617, Taiwan
[3]National Institute of Information and Communications Technology, Koganei, Tokyo 184-8795, Japan
[4]Department of Electrical Engineering, National Taiwan University, Taipei 10617, Taiwan
[5]Graduate Institute of Communication Engineering (GICE), National Taiwan University, Taipei 10617, Taiwan

Corresponding author: Tsung-Nan Lin (tsungnan@ntu.edu.tw)

This work was supported in part by the Ministry of Science and Technology, Taiwan, under Grant MOST-110-2634-F-002-037 and Grant MOST-110-2218-E-002-018-MBK, and in part by the Ministry of Economic Affairs, Taiwan, through "Artificial Intelligence Oriented for Cyber Security Technology Collaboration Project (2/4)" of the Institute for Information Industry.

ABSTRACT With more than three million applications already in the Android marketplace, various malware detection systems based on machine learning have been proposed to prevent attacks from cybercriminals; most of these systems use static analyses to extract application features. However, many features generated by static analyses can be easily thwarted by obfuscation techniques. Therefore, several researchers have addressed this obfuscation problem with obfuscation-invariant features. However, to the best of our knowledge, no researcher has utilized deobfuscation techniques. To this end, we adopt a code deobfuscation technique with an Android malware detection system and investigate its effects. Experimental results indicate that code deobfuscation can successfully retrieve useful information concealed by obfuscation. Further, we propose interaction terms based on identified feature interactions. The proposed interaction terms aim to eliminate the interference caused by the size of the application and other features because many feature values are correlated to the size of the application. In addition, the experimental results indicate that these interaction terms have a high ranking in terms of feature importance values. Our proposed Android malware detection model achieves 99.55% accuracy and a 94.61% F1-score with the well-known Drebin dataset, which is better than the performance of previous works.

INDEX TERMS Android malware detection, classification, code deobfuscation, feature interaction, machine learning, static analysis, structural feature.

I. INTRODUCTION

Mobile security is gaining importance owing to the widespread use of mobile devices for performing many important tasks, such as health management, enterprise communications, money transactions, and banking. Android is currently the most popular smartphone operating system, with an 84.8% share of the global market in 2020; this percentage is predicted to increase over the forthcoming years [1]. The Android platform's large market share and its open-market policy make it the most valuable target for attackers. According to Kaspersky's malware report [2], Kaspersky mobile products and technologies detected 5,683,694 malicious installation packages, 156,710 new mobile banking Trojans, and 20,708 new mobile ransomware Trojans in 2020. Given that there are more than three million applications on the Android marketplace, there is an urgent need to develop an efficient and automated malware detection mechanism that can allow malware analysts to focus on suspicious applications.

Security analysts perform a program analysis for understanding the behavior and intent of an application. Existing program analysis methods can be categorized into static and dynamic analysis methods. Static analysis techniques have higher code coverage and lower computational cost compared to that of dynamic analysis techniques. Further, static analysis is typically preferred over dynamic analysis, because it is efficient and scalable, and therefore it is ideal for managing a great number of applications. However, static analysis techniques are more susceptible to code obfuscation.

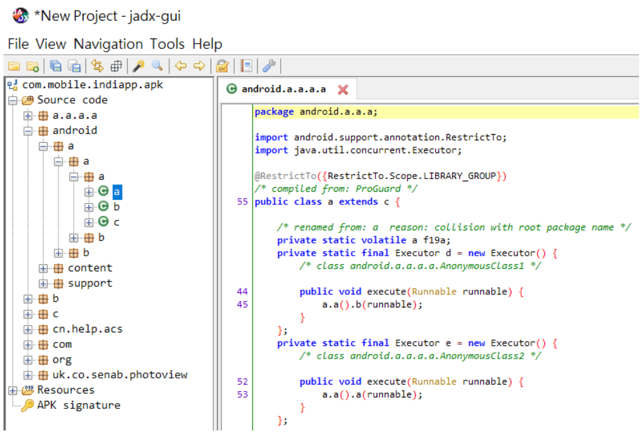
The associate editor coordinating the review of this manuscript and approving it for publication was Marco Martalo.

Previous studies ignored obfuscated codes and used obfuscation-invariant codes such as the Android OS application programming interface (API).

In this study, we directly address obfuscated codes such as third-party library APIs. We explore a code deobfuscation technique to address the challenge of code obfuscation in Android malware detection. Fig. 1 shows an obfuscated Android application package (APK). When an APK is obfuscated, its class, method and variable names are renamed to meaningless symbols, such as "`android.a.a.a`". We obtain many obfuscated and useless API calls when we extract API calls from an obfuscated APK. A code deobfuscation process is required to use these obfuscated API calls. To the best of our knowledge, this is the first paper that uses a code deobfuscation technique for Android malware detection. The primary contributions of this study are summarized as follows:

- We use a code deobfuscation tool to recover original API calls from obfuscated API calls. According to the experiment results obtained in this study, some recovered API calls act as important features of the malware detection model.
- We propose interaction terms based on identified feature interactions to eliminate interference caused by the application size. The experimental results suggest that these interaction terms have a high feature importance ranking.
- We propose certain novel features based on the application certificates and the disassembler's log file.

The remainder of this paper is organized as follows. Section II provides the relevant background for this study and Section III reviews the related studies. The proposed methods are described in Section IV. Section V presents the experiments and their results. Finally, we conclude this paper in Section VI.



```
com.mobile.indiapp.apk
  Source code
    a.a.a.a
    android
      a
        a
          a
            a
            b
            c
          b
        b
      content
      support
    b
    c
    cn.help.acs
    com
    org
    uk.co.senab.photoview
  Resources
  APK signature
```

```
package android.a.a.a;

import android.support.annotation.RestrictTo;
import java.util.concurrent.Executor;

@RestrictTo({RestrictTo.Scope.LIBRARY_GROUP})
/* compiled from: ProGuard */
public class a extends c {

    /* renamed from: a  reason: collision with root package name */
    private static volatile a f19a;
    private static final Executor d = new Executor() {
        /* class android.a.a.a.a.AnonymousClass1 */

        public void execute(Runnable runnable) {
            a.a().b(runnable);
        }
    };
    private static final Executor e = new Executor() {
        /* class android.a.a.a.a.AnonymousClass2 */

        public void execute(Runnable runnable) {
            a.a().a(runnable);
        }
    };
```

FIGURE 1. Depiction of an obfuscated APK.

II. BACKGROUND

A. STATIC ANALYSIS

Static analysis techniques use reverse engineering to extract the information of an application by scanning the entire APK without running the application. A static analysis technique extracts application information from configuration files, binary files, bytecode, source code, or other associated files. Then, a series of features are collected and embedded in numerical feature vectors to train machine learning models or statistical models. The static analysis techniques are widely used in Android malware detection because of their advantages such as high code coverage and high efficiency.

B. COMMON FEATURES IN STATIC ANALYSIS

Features represent the behavior of an app; different features represent the behavior of an app from different aspects. In static analysis, permissions and API calls are the most used features; the popularity of these two features is far greater than that of other features [3]. Some other features are also used in static analyses: required hardware and software features, registered intent filters, meta information, constant strings and opcode.

Several studies [4]–[6] have shown that API calls are the best features for static analysis. Permissions carry redundant information with API calls because of the relationship between the permission requested and the corresponding API call usage. Experimental results have shown that API calls outperform other features when used alone, and combining API calls with other features yields only a small improvement.

C. DYNAMIC ANALYSIS

Dynamic analysis techniques use information flow tracking to discover malicious behavior by acquiring information from running processes such as system calls and network traffic. Dynamic analysis techniques have certain advantages compared to static analysis; for example, acquiring the information of dynamically loaded code and network traffic. Dynamically loaded code is code loaded from a remote system or a path within the device. Many Android developers use this technique to update their application via dynamic class loading [7]; therefore, the presence of dynamic code loading may not directly indicate malicious behavior.

The disadvantage of dynamic analysis is that code execution in a restricted environment can influence the behavior of the malware. The malware may not perform the malicious event or load the code dynamically because the command and control server of the malware is no longer available. Further, there are methods that can detect sandbox environments and stop their malicious behavior to escape detection by dynamic analysis.

D. OBFUSCATION AND DEOBFUSCATION

Software obfuscation is a technique that creates source code or machine code that is difficult for humans to understand. Android developers obfuscate their code for several reasons, such as to protect their intellectual property and prevent tampering. In addition, certain code obfuscations occur as a side effect of shrinking and optimizing the code of an application. Code shrinking can remove unused classes, fields, methods,

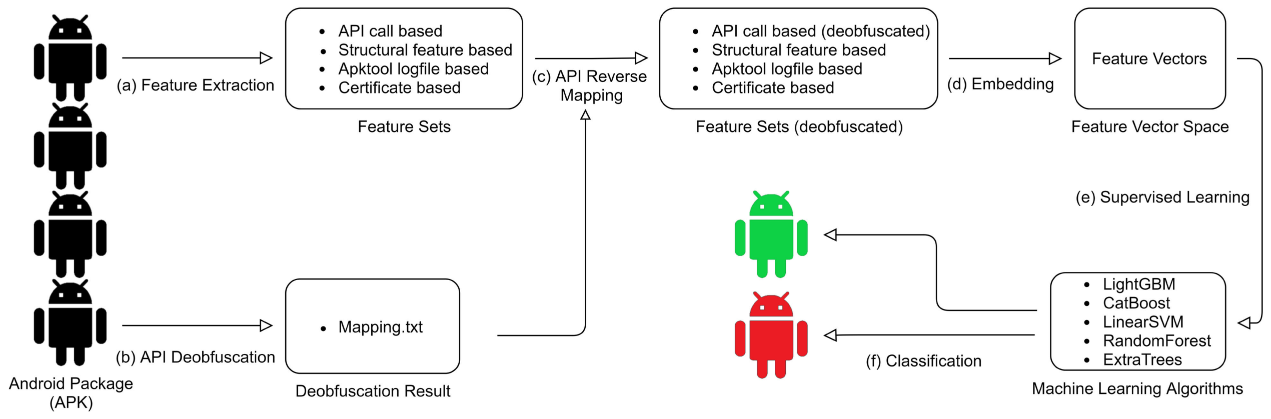


FIGURE 2. System framework divided into five major phases: (a) feature extraction. (b) API deobfuscation. (c) API reverse mapping. (d) feature embedding. (e,f) supervised learning.

and attributes from an application and shorten the name of packages, classes, methods, and fields in the application. Optimization inspects and rewrites the code to minimize the size of the Dalvik Executable (DEX) files [8] of the app. Certain basic obfuscation techniques exist, such as name obfuscation, name overloading, debug data obfuscation, annotation obfuscation, string encryption and DEX file encryption [9].

A large-scale investigation of obfuscation [9] showed that the main packages of 83.1% applications are obfuscated, and 100% of apps are obfuscated when considering whether an app contains any package that has obfuscation features. The percentage of apps with obfuscated code is high, because many applications include obfuscated third-party libraries.

Many automated deobfuscation systems have been proposed for Android applications to facilitate reverse engineering analysis. Yoo *et al.* [10] proposed a string deobfuscation scheme to acquire decrypted strings through a dynamic analysis approach. Kan *et al.* [11] proposed a system to recover the original control flow graph from the obfuscated control flow generated by Obfuscator-LLVM [12]. A control flow graph is the graphical representation of the control flow, and it is a directed graph where each node represents a basic block that is a linear sequence of program statements; each edge represents the flow of control between the basic blocks.

DeGuard [13] is a tool that attempts to reverse the layout obfuscation performed by ProGuard, which is a free software that shrinks, optimizes, and obfuscates Java code. Layout obfuscation modifies the layout structure of the program by renaming identifiers, such as class, package, and method names. Renaming these program elements with completely meaningless symbols increases the cost of reverse engineering. Layout obfuscation is the most well-studied and widely used technique for code obfuscation; nearly all Java obfuscators contain this technique. DeGuard was selected to solve the layout obfuscation problem in this study because it reveals over 90% of third-party libraries concealed by ProGuard, which is considered an efficient performance. Further, DeGuard maintains a web service that can be used.

Not all program elements in an application can be obfuscated because certain entities are external, such as methods that are part of the Android OS API and classes referenced in static files. Adopting these renamed API calls as features can degrade the performance of the detection model because they are like noise. Thus, when using API calls as features, obfuscated API calls should be either ignored or deobfuscated.

III. RELATED WORK

To the best of our knowledge, existing studies have not used deobfuscation techniques or the proposed feature interactions. Drebin [14] is a lightweight method proposed by Arp *et al.* to detect malicious applications. Drebin considers the most important characteristics of Android applications: hardware components, requested permissions, app components, filtered intents from the manifest file and restricted API calls, used permissions, suspicious API calls and network addresses from the disassembled code. These characteristics are embedded in a vector space using one-hot encoding. Finally, Drebin uses support vector machines (SVMs) to determine whether a given sample is malicious or benign. Arp *et al.* investigated a dataset collected from 2010 to 2012; the authors have released the dataset, and it is now referred to as the Drebin dataset. Drebin was evaluated using hold-out validation (67% training and 33% testing), and it achieved a detection rate of 93.90% with a 1% false positive rate.

DroidSieve [15] is a lightweight method proposed by Suarez-Tangil *et al.* to detect obfuscated Android malware. The authors categorized the features of an application into two high-level classes: syntactic (code) and resource-centric features. Syntactic features are derived from the source code and metadata of the application, and resource-centric features are derived from the assets of the applications. These features are used to train an extra tree model to predict the label of a given application; the accuracy reaches 99.82%. Among the features proposed by DroidSieve, the most important feature is derived from the application's certificate. Inspired by that study, we designed certain features extracted from the certificate.

Ban *et al.* [5] proposed a method using a linear SVM and four types of features: permission, category, description, and API features. They explored the potential of meta data in Android malware detection; their experimental results indicated that API calls are the best features for Android malware detection.

SAFEDroid [16] mentioned that many studies proposed for Android malware detection are based on static analysis; these methods also primarily use API calls and permissions as their features. Apart from these traditional features, SAFEDroid uses novel structural features such as the number of methods/classes/files/folders, size of the APK archive, number of goto statements, and number of permissions used. Their experimental results indicated that these novel structural features are effective against new malware in their experiments. Inspired by SAFEDroid, we designed novel interaction terms based on these structural features.

Alotaibi [17] proposed the MalResLSTM framework that is based on deep residual long short-term memory(LSTM); the framework uses information in the manifest file, API calls, and network addresses as features. These features are embedded in a vector space using one-hot encoding. The authors evaluated their framework using the Drebin dataset and achieved a detection accuracy of 99.32% and an F1 score of 92%.

Anuar *et al.* [18] proposed a method using 213 opcode sequences that represents the corresponding chain of malicious activities. The authors evaluated their method on a dataset consisting of 500 APKs from AMD [19] and 500 APKs from Google Play and achieved an accuracy of 95.4% when using SVM.

AdMat [20] uses a convolutional neural network and it selects 219 popular Android methods to generate adjacency matrices for approximating API call-graphs. The authors evaluated their method on 4460 benign apps and 5560 malicious apps from the Drebin dataset, and they achieved an F1-score of 92%.

Rahman and Saha [21] used a two-stage stacking method to assemble certain machine learning algorithms such as extremely randomized tree, random forest, multi-layer perceptron, and stochastic gradient descent classifiers. The authors evaluated their framework using the Drebin dataset and achieved a detection accuracy of 97.00%.

IV. METHODOLOGY

An Android malware detection framework was developed based on static analysis and machine learning algorithms. Fig. 2 shows the proposed system framework.

The system framework is divided into five major phases: feature extraction, API deobfuscation, API reverse mapping, feature embedding, and supervised learning phase.

A. FEATURE EXTRACTION

Reverse engineering is required to determine the manifest file and Smali code from the collected APK. We used Apktool (version 2.4.1) [22], which disassembles the Dalvik executable bytecode (DEX) of the app into Smali code. In Android systems, the source code is compiled in the DEX file format to ensure compatibility with the Dalvik virtual machine and Android Runtime; Smali is the assembly language for the DEX format. After disassembling is performed by Apktool, we obtain information about the application, including Smali files, AndroidManifest.xml, the certificate file, and a file generated by Apktool called apktool.yml.

The features extracted in this study are classified into four sets: API call based, Apktool logfile based, certificate based and feature interaction based. The definitions of the important features are listed in Table 1. The descriptions of each feature set are provided below:

S_1: API call based

API calls are the most effective and most used features because API calls can represent the behavior of an application. A feature set of API calls was formed by extracting all function calls found in Smali files. The API calls can be categorized into Android OS APIs, programmer-defined functions and third-party libraries. Among these three types of API calls, only Android OS APIs cannot be obfuscated. Programmer-defined functions and third-party libraries are often obfuscated by code obfuscators [9] such as ProGuard [23]. Malware detection models do not benefit from these obfuscated codes, and therefore, we introduce a deobfuscation technique. If all API calls are considered, millions of unique API calls will need to be added to the dataset created in this study. Therefore, we selected the most common 100,000 API calls as the API call feature set. We represent the original API calls extracted by S_1-*original*; we use S_1-*deobfuscated* to represent the API calls after the deobfuscation process.

S_2: Apktool logfile based

Apktool.yml is a logfile generated by Apktool. This file contains information that can be used to rebuild the APK later. The file number in the *doNotCompress* category and that in the *unknownFiles* category are calculated as features. The categories in apktool.yml are defined as follows:

- doNotCompress: Used to record the names of non-compressed files in an APK.
- unknownFiles: Used to record the names/locations of nonstandard files in an APK.

S_3: Certificate based

DroidSieve [15] designed a feature based on the difference between the signing time of the app and the generation time of the corresponding certificate. Further, they designed features based on the time zone and string length of the common name for each certificate. In the proposed approach, we attain the time difference between the signing time of each file in the application and the generation time of the corresponding certificate. Then, we extract this information using the jarsigner [24] command line tool. Once we acquire a list of time differences, we calculate

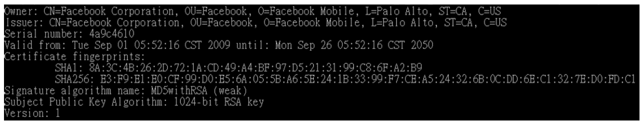
TABLE 1. Four feature sets used in the proposed framework and the definition of certain important features. For the API call-based category, we used the most common 100,000 API calls. For the feature interaction-based category, we list only some of the important features in this study.

Feature sets	Features	Descriptions
API call based	Most common 100,000 API calls	Number of appearances of an API call in an APK.
Apktool logfile based	doNotCompress_num	Number of noncompressed files in an APK.
	unknownFiles_num	Number of nonstandard files in an APK.
Certificate based	cert_entropy	Entropy of relative distinguished names(RDNs) in the certificate.
	cert_diff_max	Maximum value of the list that contains the time differences between the signing time of a file and the generation time of the corresponding certificate for each file in the app.
	cert_diff_min	Minimum value of the list described above.
	cert_diff_mean	Mean value of the list described above.
	cert_diff_std	Standard deviation of the list described above.
	verified_ratio	Ratio of files whose signature was verified.
	in_manifest_ratio	Ratio of files whose entry is listed in the manifest file.
Feature interaction based	total_apis_num//unique_apis_num	Number of API calls used in an APK divided by the number of unique API calls used in the APK.
	total_smalis_num//unique_apis_num	Number of Smali files in an APK divided by the number of unique API calls used in the APK.
	total_smalis_size//unique_apis_num	Total size of all Smali files in an APK divided by the number of unique API calls used in the APK.
	service_num//unique_apis_num	Number of services declared in the manifest file divided by the number of unique API calls used in the APK.
	intent-filter_num//unique_apis_num	Number of intent filters declared in the manifest file divided by the number of unique API calls used in the APK.
	activity_num//unique_apis_num	Number of activities declared in the manifest file divided by the number of unique API calls used in the APK.
	data_num//unique_apis_num	Number of <data> elements in the manifest file divided by the number of unique API calls used in the APK.
	invoke-interface//unique_apis_num	Number of invoke interface methods used in an APK divided by the number of unique API calls used in the APK.
	invoke-super//unique_apis_num	Number of invoke super methods used in an APK divided by the number of unique API calls used in the APK.

the minimum, maximum, mean and standard deviation of the list as features. Further, we use two features of jarsigner: whether the signature is verified and whether the file entry is listed in the manifest. These two features are used to calculate the percentage of files with verified signatures and the percentage of files listed in the manifest. Finally, we calculate the information entropy of the relative distinguished names (RDNs) [25] in the certificate as a feature. The RDN is an attribute-value pair within a distinguished and unique name given to an X.500 directory object. Fig. 3 shows the certificate information of a Facebook application. In this example, we collect the values of all attribute-value pairs(*Facebook Corporation*, *Facebook*, *Facebook Mobile*, *Palo Alto*, *CA*, *US*) to calculate the information entropy of the RDNs. These values are then used to calculate the information entropy using:

$$information\ \ entropy = -\sum p \log_2 p \tag{1}$$

where p denotes the frequency of each distinct value collected from the RDNs.



```
Owner: CN=Facebook Corporation, OU=Facebook, O=Facebook Mobile, L=Palo Alto, ST=CA, C=US
Issuer: CN=Facebook Corporation, OU=Facebook, O=Facebook Mobile, L=Palo Alto, ST=CA, C=US
Serial number: 4a9c4610
Valid from: Tue Sep 01 05:52:16 CST 2009 until: Mon Sep 26 05:52:16 CST 2050
Certificate fingerprints:
         SHA1: 8A:3C:4B:26:2D:72:1A:CD:49:A4:BF:97:D5:21:31:99:C8:6F:A2:B9
         SHA256: E3:F9:E1:E0:CF:99:D0:E5:6A:05:5B:A6:5E:24:1B:33:99:F7:CE:A5:24:32:6B:0C:DD:6E:C1:32:7E:D0:FD:C1
Signature algorithm name: MD5withRSA (weak)
Subject Public Key Algorithm: 1024-bit RSA key
Version: 1
```

FIGURE 3. Certificate information of the facebook application.

S_4: Feature interaction based

Some structural features have been proposed in the literature; for example, the number of requested permissions, API calls, goto statements, lines of manifest file, and elements in the manifest file [16], [26]. During the data analysis, we found that the number of API calls used in an APK is highly correlated with the size of the APK; this implies that these structural features are affected by the size of the APK. For example, although an application that declares many services or requests many permissions may seem malicious, larger applications generally declare more services and request more permissions. The feature **uses-permission_num** counts the number of permissions requested by the application. Fig. 4 shows the scatter plot and linear regression lines of **uses-permission_num** versus **unique_apis_num** in the Drebin dataset. The slopes of the two linear regression lines are considerably different, which indicates that malicious applications tend to request more permissions than benign apps under the same APK scale. Inspired by this observation, we propose certain interaction-based features to eliminate the interference caused by the size of the APK. We use the number of unique API calls used in an APK to represent the scale of the APK because the raw size of the APK can be affected by the size of the assets of the application, such as images. Fig. 4 indicated that there is a positive correlation between the variables, and the slopes of the two linear regression lines are considerably different. We designed a new interaction term calculated by dividing the number of requested permissions by the number of unique API calls used; this term includes information about the rationality of the number of requested permissions. If the value of this term for a sample is higher than that of the others, this sample is considered to request excess permissions based on its APK size, and it is deemed more likely to be a malicious sample.

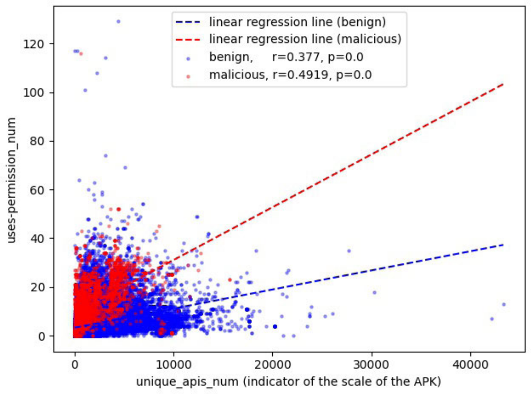


FIGURE 4. Scatter plot and linear regression lines of uses-permission_num versus unique_apis_num, where r represents the Pearson's correlation coefficient and p denotes the p-value. Dataset: Drebin [14].

The proposed interaction-based features can be viewed as decorrelated features. We use S_4-*original* to represent the original structural features; S_4-*decorrelated* represent the decorrelated features.

B. API DEOBFUSCATION AND API REVERSE MAPPING

We use DeGuard [13] to address the obfuscated API calls because it provides an online service that helps map obfuscated API calls to the original name of the API calls. After an APK is uploaded to the DeGuard online service, DeGuard performs code deobfuscation and returns a file called mapping.txt, which contains the prediction of the original API calls of an obfuscated APK. We perform a reverse mapping process on the API calls for each APK in this phase based on mapping.txt.

Fig. 6 shows a part of the Smali code of an obfuscated APK, and Fig. 7 is part of the mapping.txt returned by DeGuard. In Smali code, there is an object called `org/a/d/f` that has a method called `b`. The API call `org/a/d/f/b` is obfuscated by renaming. However, with the mapping.txt file, we know that the object `org/a/d/f` is probably `org/java_websocket/util/Handshakedata`, and method `b` is probably `getFieldValue`. In this case, the API call `org/a/d/f/b` is mapped to `org/java_websocket/util/ Handshakedata-/getFieldValue`. Using this API reverse mapping process, we can retrieve the original names of the obfuscated API calls.

The total number of unique original API calls is approximately 3 million; the total number of unique deobfuscated API calls is more than 6 million. This is attributed to the fact that different API calls can be renamed to the same obfuscated API call, such as `android.a.a.a`. This process can help retrieve certain information loss caused by code obfuscation. The prediction accuracy for Deguard is over 80%; it is 90% if only third-party libraries are considered.

We estimate the time of deobfuscating an APK by running the deobfuscation process on 100 random samples for 100 rounds. Fig. 5 shows the results; it takes 3,993 s to deobfuscate 100 APKs on average, which implies that it takes approximately 60 days to deobfuscate the entire Drebin dataset.

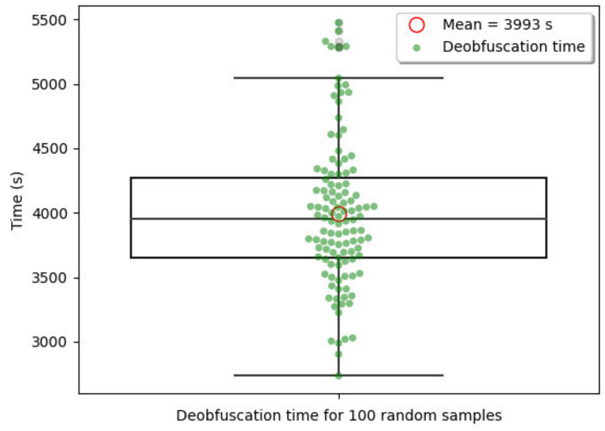
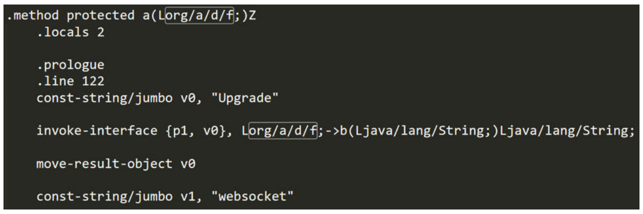
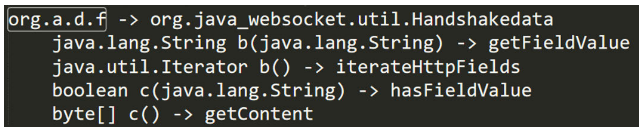


FIGURE 5. Deobfuscation time for 100 random samples.



```
.method protected a(Lorg/a/d/f;)Z
    .locals 2

    .prologue
    .line 122
    const-string/jumbo v0, "Upgrade"

    invoke-interface {p1, v0}, Lorg/a/d/f;->b(Ljava/lang/String;)Ljava/lang/String;

    move-result-object v0

    const-string/jumbo v1, "websocket"
```

FIGURE 6. Snippet of Smali code of an obfuscated APK.



```
org.a.d.f -> org.java_websocket.util.Handshakedata
    java.lang.String b(java.lang.String) -> getFieldValue
    java.util.Iterator b() -> iterateHttpFields
    boolean c(java.lang.String) -> hasFieldValue
    byte[] c() -> getContent
```

FIGURE 7. Snippet of mapping.txt that contains the prediction of original names according to Fig. 6.

C. FEATURE EMBEDDING

The extracted features need to be embedded into a vector space to train a machine learning model using these features. In the literature, certain studies have used one-hot encoding to embed API calls into a feature vector [14], [17], and others used the number of appearances per API call instead of the one-hot representation [27]. In the proposed approach, we use the latter, which is also called term-frequency encoding. The number of appearances of an API call in an APK can preserve additional information. All four feature sets are joined to define the entire feature space:

$$S_{all} = S_1 \cup S_2 \cup S_3 \cup S_4 \tag{2}$$

D. SUPERVISED LEARNING

In this phase, we use supervised learning algorithms to classify Android applications as either benign or malicious. For supervised learning, a model is trained on a labeled dataset. For model validation, we use 10-fold cross-validation to derive a more accurate estimate of model prediction performance. Further, we use a stratified sampling method: the ratio of malicious samples to benign samples in the whole dataset is preserved in the training set and testing set. We use several algorithms including LightGBM [28], CatBoost, LineraSVM, RandomForest, and ExtraTrees to evaluate the features more comprehensively.

E. DATASETS

The Drebin dataset [14] is an Android application dataset that includes 5560 APKs that belong to 179 different malware families and 123,453 benign APKs. These samples were collected between August 2010 and October 2012. This dataset is popular in Android malware detection research because it is open source, has many samples, and contains malware family labels.

The AndroZoo dataset [29] is an Android app dataset that is regularly updated and used to obtain certain Android application samples from recent years. There are many categories in the AndroZoo dataset depending on the market from which each sample is collected. We selected the VirusShare category to provide malicious samples and the Google Play category for benign samples. Further, we only used samples that were collected in 2018 or 2019.

The datasets used in this study are presented in Table 2. From the Derbin dataset, we use all samples that can be successfully processed by Apktool and DeGuard. From the AndroZoo dataset, the samples are randomly selected to ensure that the percentage of malicious samples in both datasets is the same. The same percentage of malicious samples helps eliminate the effects of data imbalance.

TABLE 2. Dataset description, including the number of malicious and benign samples in the Drebin and Androzoo datasets. We equalize the percentage of malicious samples in both datasets to eliminate any data imbalance effects.

Dataset	Malicious	Benign	Percentage of malicious
Drebin	5469	122869	4.26%
Androzoo	1785	40116	4.26%

V. EXPERIMENTS AND RESULTS

For the experiment, we use a machine with the following configuration: Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz, 64-bit PC with 512 GB RAM. The operating system is Linux ubuntu1804 4.15.0-55-generic. Python3 is used as a programming language with certain packages such as lightgbm, catboost, scikit-learn, pandas and numpy. We use accuracy (ACC), precision (PRC), recall, F1-score (F1), and area under the curve (AUC) as the evaluation metrics. We performed experiments to answer the following research questions:

*RQ*1: What is the performance of the proposed system with original API calls?

*RQ*2: What is the usefulness of the deobfuscation technique?

*RQ*3: Are interaction-based features better than existing structural features?

We need to explain the feature importance [30] used in the study before we can answer these questions. The feature importance ranking of each feature is obtained using machine learning algorithms. We selected LightGBM, which performed the best among the algorithms, and adopted the feature importance ranking derived from this algorithm. LightGBM is a gradient boosting framework that uses tree-based learning algorithms. LightGBM grows decision trees leafwise. When adding a new tree node, LightGBM selects the splitting point that has the largest information gain. After training the LightGBM model, it derives the feature importance ranking for each feature by calculating the number of times that feature is selected to be the splitting point. The importance of a feature is decided based on the frequency of its selection. Since we use 10-fold cross-validation for model validation, the final feature importance rankings are calculated from the average of ten training results.

A. RQ1: PERFORMANCE OF THE PROPOSED SYSTEM WITH ORIGINAL API CALLS

First, we evaluate the detection performance of the proposed system using S_1-*original*, which has been shown to be the best feature in many other studies [4]–[6]. Ensemble learning is the most used classifier for Android malware detection using static analysis. Random forest is the most used classifier in ensemble learning [3]. Table 3 shows that LightGBM outperforms all other classifiers with an AUC of 99.71% and LinearSVM achieves the worst performance with an AUC of 96.05% on the Drebin dataset. Accuracies are high because of the data imbalance. For the imbalanced dataset, F1 and AUC are better evaluation metrics; AUC is better than F1-score because the F1 changes because of the variation in the threshold of the classifier used for determining whether a sample is positive. Thus, we use AUC as our primary evaluation metric.

We obtain a similar result for the same experiment using the Androzoo dataset. LightGBM has the highest AUC of 99.27%, and LinearSVM has the lowest AUC of 95.23% for the Androzoo dataset.

[1]The low F1 score can be attributed to the fact that random forest uses a threshold that yields high precision (98.83%) and low recall (27.81%). These results cause F1 to be low based on the formula used to calculate F1. Further, we determine a threshold that yields an F1 score similar to that of other algorithms with the same random forest model. We must determine a proper threshold in this case, and thus, we use AUC as our primary metric.

TABLE 3. Performance and computational time using feature set S_1*-original*. Dataset: Drebin.

Algorithms	ACC(%)	F1(%)	AUC(%)	Time(s)
CatBoost	99.45	93.37	99.44	3,060
LineraSVM	98.51	83.01	96.05	7,769
RandomForest	97.05	47.43 [1]	97.68	4,893
ExtraTrees	99.37	92.27	98.89	5,514
LightGBM	99.54	94.52	99.71	1,959
Average	98.78%	82.12%	98.35%	4,639

B. RQ2: WHAT IS THE USEFULNESS OF THE DEOBFUSCATION TECHNIQUE?

We compare the detection performance of the models using feature sets S_1*-original* and S_1*-deobfuscated* to evaluate the impact of the code deobfuscation technique on Android malware detection. Table 6 shows that the average AUC increases from 98.35% to 98.53% when we use deobfuscated API calls instead of original API calls in the experiment with the Drebin dataset. We analyze the feature importance rankings of the deobfuscated API calls. Among the top 100 important features, 3 API calls were retrieved by the deobfuscation process, which indicates that these API calls are obfuscated in the original API calls, and they do not contribute to the malware detection process. The most important feature retrieved by the deobfuscation process is `com.apperhand.common.configurations.Bookmark->getStatus` has a feature importance ranking of 23. Table 5 lists the deobfuscated API calls retrieved by DeGuard among the top-100 important feature rankings.

In the same experiment with the Androzoo dataset, the average AUC marginally decreased from 98.22% to 98.16% when we used deobfuscated API calls instead of the original API calls. Although the AUC did not increase, the deobfuscation process retrieved some important API calls. The most important API call retrieved by the deobfuscation process is `gnu.bytecode.CodeAttr->emitInvokeVirtual`, which has a feature importance ranking of 4. This API is used to compile a virtual method call, and it identifies 17.74% of benign applications. However, none of the malicious applications use this API call.

We evaluate the relevance of S_1*-deobfuscated* and S_1*-original* to class labels using an information theory-based measure. First, we calculate the correlation coefficient of the class label and each feature using:

$$corr(X, Y) = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X\sigma_Y}, \tag{3}$$

where E denotes the expected value operator; μ_X and μ_Y represent the expected values of the feature and class label, respectively; and σ_X and σ_Y represent the standard deviations of the feature and class label, respectively. We calculate the correlation between class labels and feature sets by calculating the softmax-weighted average correlation coefficients of the features in each feature set. We use the softmax-weighted average because the features we are interested in have a high correlation with the class labels. A softmax-weighted average provides a feature with a higher correlation a higher weight. The sum of all softmax weights is equal to one. The softmax weight of each correlation coefficient $\sigma(z)_i$ is calculated using:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad for\ i = 1, \ldots, K,$$
$$z = (z_1, \ldots, z_K) \in \mathbb{R}^K, \tag{4}$$

where z denotes the correlation coefficient vector and $\sigma(z)$ represents the softmax vector. The softmax-weighted average correlation coefficient is calculated using:

$$\sum_{j=1}^{K} z_j * \sigma(z)_j, \quad z = (z_1, \ldots, z_K) \in \mathbb{R}^K. \tag{5}$$

The softmax-weighted average correlation coefficients of S_1*-original* and S_1*-deobfuscated* are summarized in Table 4. The result of the information theory-based measure shows that the proposed feature set has a higher correlation with the class label; this is consistent with our experimental result.

TABLE 4. Relevance of feature sets S_1*-original* and S_1*-deobfuscated* to class labels. Dataset: Drebin.

	$S_1 - original$	$S_1 - deobfuscated$
Relevance	0.01976	0.02013

TABLE 5. Deobfuscated API calls generated by Deguard among the top 100 important features and their ranking of feature importance. Dataset: Drebin.

Rank	API calls
10	jp.co.lilhermit.android.core.Native->runcmd
18	com.google.devtools.simple.runtime.components .android.DashboardFragment->GotFile
49	com.apperhand.device.ui.asm.Content.CType->init
59	com.google.asm->close
62	com.apperhand.common.configurations.protocol.BaseResponse ->getAbTest

C. RQ3: ARE INTERACTION-BASED FEATURES BETTER THAN EXISTING STRUCTURAL FEATURES?

Existing structural features in the literature can be categorized into manifest-based, code-based, and file-based features. Examples of manifest-based features include the count of xml elements in a manifest file [26], number of requested permissions [15] and number of lines in the manifest file [16]. Examples of code-based features include the number of goto statements, annotations, classes, and methods [16]. Examples of file-based features include application size [26], and the count of files and directories [16]. We construct the proposed novel interaction-based features based on these existing structural features and the feature interaction described in section IV.

We use both interaction-based features and existing structural features and compare the feature importance between them to evaluate the effectiveness of the

TABLE 6. Performance and computational time using feature set S_1-*deobfuscated*. Dataset: Drebin.

Algorithms	ACC(%)	F1(%)	AUC(%)	Time(s)
CatBoost	99.46%	93.55%	99.51%	2,550
LineraSVM	98.49%	83.13%	96.55%	5,091
RandomForest	96.91%	43.41%	97.84%	4,163
ExtraTrees	99.33%	91.72%	99.02%	5,248
LightGBM	99.53%	94.45%	99.71%	2,140
Average	98.74%	81.25%	98.53%	3,838

proposed interaction-based features. The top 20 important features are shown in Fig. 9 for the experiment using the Drebin dataset. The most important feature is **uses-permission_num//unique_apis_num**, which is an interaction-based feature. Since **uses-permission_num** is ranked No. 11, we know that the interaction-based feature is more useful than the existing structural feature. This improvement is indicated by many other structural features. Table 7 lists the feature importance ranking of several structural features and the corresponding interaction-based features. As indicated by Table 7, **service_num** is ranked No. 311, and the corresponding interaction-based feature is ranked No. 10, which is a large improvement. Fig. 8 shows the scatter plot and linear regression lines for **service_num** versus **unique_apis_num** in the Drebin dataset. As shown in Fig. 8, if malicious and benign apps have the same APK scale, the number of services of the malicious app is double that of the benign app. It is intuitive that malicious apps often use services to perform malicious activities because the services lack a visual user interface and they use a long-running background operation. It is difficult to differentiate between malicious and benign apps if we simply look at the number of services used by an application. However, if we look at the interaction-based feature that considers the feature interaction between the number of services and the APK scale, the difference is clear.

The proposed interaction-based features can be seen as decorrelated features. Let feature set F-*existing* represent the union of feature sets S_1-*original* and S_4-*original*, and let F-*proposed* represent the union of feature sets S_1-*deobfuscated*, S_2, S_3 and S_4-*decorrelated*. We calculate the redundancy of two feature sets F-*existing* and F-*proposed* to identify the minimum redundancy between these decorrelated features. The redundancy of a feature set is calculated using:

$$\begin{aligned} Redundancy(F) &= E[corr(F_i, F_j)], \quad i \neq j \\ &= \frac{2}{K(K-1)} \sum_{i=1}^{K} \sum_{j=i+1}^{K} corr(F_i, F_j), \end{aligned} \tag{6}$$

where E, $corr()$, F_i, and K denote the expected value operator, correlation coefficient operator, i-th feature of feature set F, and size of the feature set F, respectively. The redundancies of feature sets F-*existing* and F-*proposed* are summarized in Table 8; they are 0.11228 and 0.10313, respectively. The result indicates that the decorrelation process can reduce redundancy and create a better feature set; this is consistent with our experimental result.

Figure 9 shows that many of the proposed certificate-based features have high feature importance scores. Compared to benign apps, malicious ones have a larger time difference between the signing time of each file in the application and the generation time of the corresponding certificate, on average. For features related to the time difference, our observation is similar to that of DroidSieve [15]. They consider the signing time of an APK, and we look further at the signing time of every file in an APK. Other novel certificate-based features such as **cert_entropy** (ranked No. 3) and **in_manifest_ratio** (ranked No. 4) are also useful. Compared to benign apps, malicious apps on average have lower values of **cert_entropy** and **in_manifest_ratio**. The average values of these two features for benign apps are 1.67 and 92.17%, respectively, whereas those for malicious apps are 1.34 and 85.53%, respectively. A lower value of **cert_entropy** indicates that certain RDNs in the certificate are duplicated. This duplication may be caused by an autogenerated certificate or a developer that does not pay attention to the certificate's information. A lower value of **in_manifest_ratio** indicates that there are more files wherein file entries are not listed in the manifest file. The observation of these two features imply that malware developers do not pay considerable attention to the certificate, and they are attempting to hide certain files by concealing the file entries.

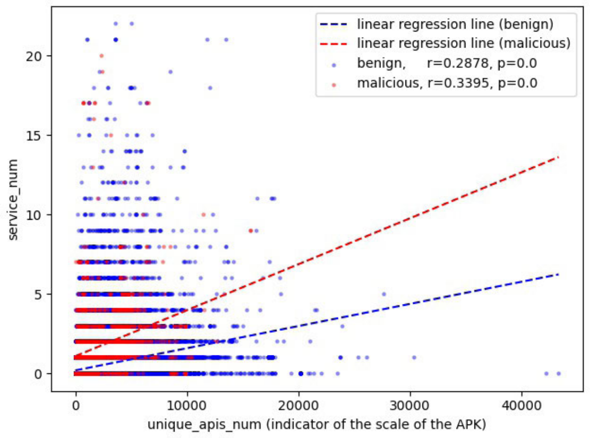


FIGURE 8. Scatter plot and linear regression lines of service_num versus unique_apis_num, where r represents the Pearson's correlation coefficient and p represents the p-value. Dataset: Drebin [14].

For the Apktool logfile-based features, **doNotCompress_num** is a useful feature ranked No. 8 in feature importance. Compared to benign apps, malicious apps have a lower value of **doNotCompress_num** on average. The average value of this feature for benign apps is 5.24, and the average value for malicious apps is 4.72.

For the same experiment using the Androzoo dataset, we obtain a similar result and the same conclusion. Fig. 10 shows the feature importance ranking for this

TABLE 7. Feature importance ranking of structural features with and without considering feature interaction. Dataset: Drebin.

Interaction Features	Rank	Original Features	Rank
uses-permission_num//unique_apis_num	1	uses-permission_num	11
invoke-super//unique_apis_num	7	invoke-super	74
service_num//unique_apis_num	10	service_num	311
try_catch//unique_apis_num	13	try_catch	143
category_num//unique_apis_num	14	category_num	159
invoke-interface//unique_apis_num	19	invoke-interface	49
invoke-direct//unique_apis_num	22	invoke-direct	183
activity_num//unique_apis_num	24	activity_num	193
URLs//total_apis_num	25	URLs	79
framework_methods//unique_apis_num	27	framework_methods	294
implements//unique_apis_num	29	implements	164
total_smalis_num//unique_apis_num	30	total_smalis_num	182

TABLE 8. Redundancy of feature sets *F-existing* and *F-proposed*. Dataset: Drebin.

	F-*existing*	F-*proposed*
Redundancy	0.11228	0.10313

experiment using the Androzoo dataset. The average values of **cert_entropy**, **in_manifest_ratio**, and **doNotCompress_num** for benign apps are 0.73, 72.98%, and 16.72, respectively, and the average values for malicious apps are 0.67, 66.97%, and 8.94, respectively.

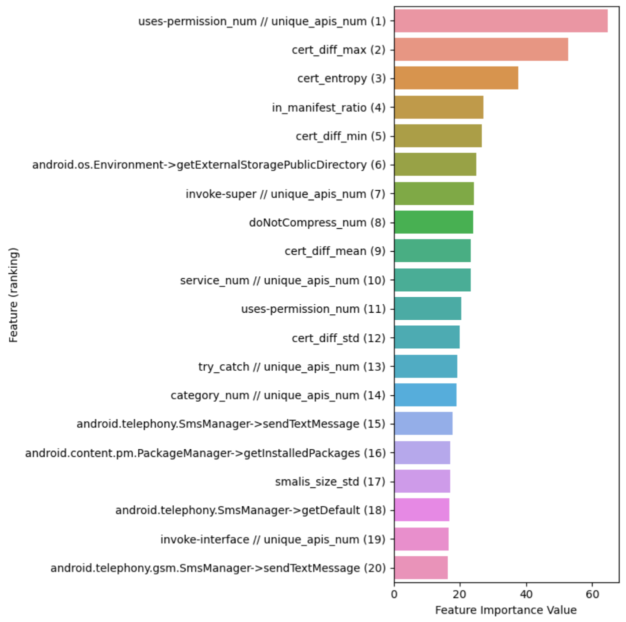


FIGURE 9. LightGBM importance values. Dataset: Drebin. Feature sets: S_4-*original*, S_4-*decorrelated*, and S_1-*original*.

D. PERFORMANCE EVALUATION AND COMPARISON

We evaluate the relevance of feature sets S_1-*deobfuscated* and S_1-*original* to class labels; the results are summarized in Table 4. Here, we evaluate the relevance of feature sets F-*existing* and F-*proposed* to the relevant class labels. The relevance values are calculated by Equation (5) and summarized in Table 9.

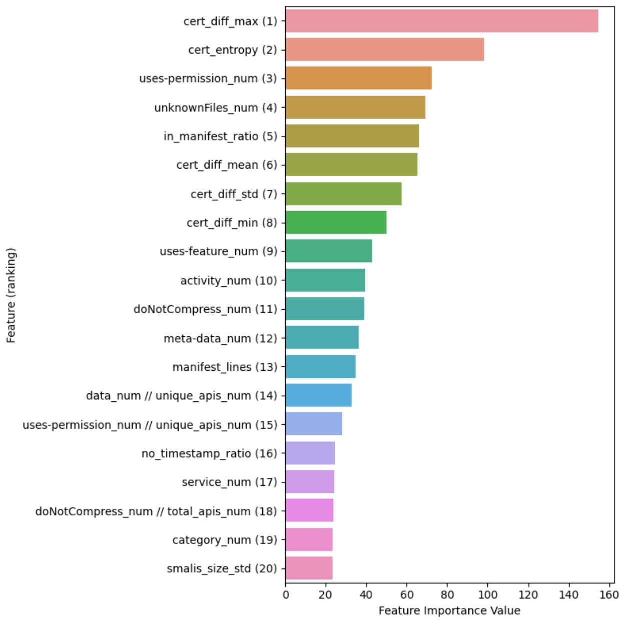


FIGURE 10. LightGBM importance values. Dataset: Androzoo. Feature sets: S_4-*original*, S_4-*decorrelated*, and S_1-*original*.

TABLE 9. Relevance of feature sets *F-existing* and *F-proposed* to class labels. Dataset: Drebin.

	F-*existing*	F-*proposed*
Relevance	0.01977	0.02014

We use the Drebin dataset [14] to evaluate our method and compare the performance with previous Android malware detection systems. None of the previous works apply the deobfuscation technique or propose interaction features or decorrelated features. Further, most of the previous works use a one-hot encoding method for feature embedding, whereas we use term frequency encoding. Further, we compare our method with previous works that use the same Drebin dataset and the same validation strategy. The experimental results are listed in Table 10. The proposed framework achieves the best performance according to the comparison result obtained using the common metric accuracy presented in Table 10.

TABLE 10. Comparison on Drebin dataset. Abbreviations: ALGO - Algorithm, LGBM - Light gradient boosting machine, LSTM - Long short-term memory, SVM - Support vector machine, TF - Term frequency.

Method	ALGO	Encoding	N-Fold	ACC(%)
D. Arp [14]	SVM	One-hot	3-Fold	93.90
Rahman et al. [21]	Stacking	One-hot	5-Fold	97.00
Alotaibi, A. [17]	LSTM	One-hot	3-Fold	99.32
Proposed method	LGBM	TF	3-Fold	**99.52**
			5-Fold	**99.51**

VI. CONCLUSION AND FUTURE WORK

We introduced novel features based on Apktool and certificates, some of which were found to be important features in the malware detection model. These novel features were resilient against common obfuscation techniques such as name obfuscation, name overloading, debug data obfuscation, annotation obfuscation, string encryption, and DEX file encryption [9].

We proposed interaction-based features based on existing structural features. We used a code deobfuscation tool to recover the original naming of obfuscated API calls. The results indicated that certain recovered API calls can be considered important features. Thus, there are potential benefits of adopting deobfuscation techniques in Android malware detection. We plan to investigate other deobfuscation techniques such as string deobfuscation and control flow deobfuscation in the future.

REFERENCES

[1] IDC Corporate. *Device Market Trends: Smartphone Market Share*. Accessed: Mar. 11, 2021. [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os

[2] (2020). *Mobile Malware Evolution 2019*. Accessed: Mar. 11, 2021. [Online]. Available: https://securelist.com/mobile-malware-evolution-2020/101029/ and https://securelist.com/mobile-malware-evolution-2019/96280/

[3] Y. Pan, X. Ge, C. Fang, and Y. Fan, "A systematic literature review of Android malware detection using static analysis," *IEEE Access*, vol. 8, pp. 116363–116379, 2020.

[4] K. Dillon, "Feature-level malware obfuscation in deep learning," 2020, *arXiv:2002.05517*. [Online]. Available: http://arxiv.org/abs/2002.05517

[5] T. Ban, T. Takahashi, S. Guo, D. Inoue, and K. Nakao, "Integration of multi-modal features for Android malware detection using linear SVM," in *Proc. 11th Asia Joint Conf. Inf. Secur. (AsiaJCIS)*, Aug. 2016, pp. 141–146.

[6] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in Android," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.* Cham, Switzerland: Springer, 2013, pp. 86–103.

[7] A. I. Aysan and S. Sen, "'Do you want to install an update of this application?' A rigorous analysis of updated Android applications," in *Proc. IEEE 2nd Int. Conf. Cyber Secur. Cloud Comput.*, Nov. 2015, pp. 181–186.

[8] (2020). *Android Developers Android Studio User Guide: Shrink, Obfuscate, and Optimize Your App*. Accessed: Oct. 10, 2020. [Online]. Available: https://developer.android.com/studio/build/shrink-code

[9] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A large scale investigation of obfuscation use in Google play," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 222–235.

[10] W. Yoo, M. Ji, M. Kang, and J. H. Yi, "String deobfuscation scheme based on dynamic code extraction for mobile malwares," *IT Converg. Pract.*, vol. 4, no. 2, pp. 1–8, 2016.

[11] Z. Kan, H. Wang, L. Wu, Y. Guo, and D. X. Luo, "Automated deobfuscation of Android native binary code," 2019, *arXiv:1907.06828*. [Online]. Available: http://arxiv.org/abs/1907.06828

[12] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM—Software protection for the masses," in *Proc. IEEE/ACM 1st Int. Workshop Softw. Protection*, May 2015, pp. 3–9.

[13] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of Android applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 343–355.

[14] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, vol. 14, 2014, pp. 23–26.

[15] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "DroidSieve: Fast and accurate classification of obfuscated Android malware," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2017, pp. 309–320.

[16] S. Sen, A. I. Aysan, and J. A. Clark, "SAFEDroid: Using structural features for detecting Android malwares," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.* Cham, Switzerland: Springer, 2017, pp. 255–270.

[17] A. Alotaibi, "Identifying malicious software using deep residual long-short term memory," *IEEE Access*, vol. 7, pp. 163128–163137, 2019.

[18] N. A. Anuar, M. Z. Mas'ud, N. Bahaman, and N. A. M. Ariff, "Analysis of machine learning classifier in Android malware detection through opcode," in *Proc. IEEE Conf. Appl., Inf. Netw. Secur. (AINS)*, Nov. 2020, pp. 7–11.

[19] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current Android malware," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, 2017, pp. 252–276.

[20] L. N. Vu and S. Jung, "AdMat: A CNN-on-matrix approach to Android malware detection and classification," *IEEE Access*, vol. 9, pp. 39680–39694, 2021.

[21] S. S. M. M. Rahman and S. K. Saha, "Stackdroid: Evaluation of a multi-level approach for detecting the malware on Android using stacked generalization," in *Proc. Int. Conf. Recent Trends Image Process. Pattern Recognit.* Singapore: Springer, 2018, pp. 611–623.

[22] (2020). *Apktool: A Tool for Reverse Engineering Android Apk Files*. Accessed: Oct. 10, 2020. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[23] (2020). *ProGuard: Open Source Optimizer for Java and Kotlin*. Accessed: Oct. 10, 2020. [Online]. Available: https://www.guardsquare.com/en/products/proguard

[24] (2020). *Jarsigner: Signs and Verifies Java Archive (JAR) Files*. Accessed: Oct. 10, 2020. [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html

[25] (2020). *Relative Distinguished Name*. Accessed: Oct. 10, 2020. [Online]. Available: https://ldapwiki.com/wiki/Relative Distinguished Name

[26] A. Shabtai, Y. Fledel, and Y. Elovici, "Automated static code analysis for classifying Android applications using machine learning," in *Proc. Int. Conf. Comput. Intell. Secur.*, Dec. 2010, pp. 329–333.

[27] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of Android malware," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 3, pp. 1–29, Jan. 2018.

[28] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "LightGBM: A highly efficient gradient boosting decision tree," in *Advances in Neural Information Processing Systems*, vol. 30, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Red Hook, NY, USA: Curran Associates, 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[29] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proc. 13th Int. Conf. Mining Softw. Repositories (MSR)*, New York, NY, USA, 2016, pp. 468–471, doi: 10.1145/2901739.2903508.

[30] G. Louppe, L. Wehenkel, A. Sutera, and P. Geurts, "Understanding variable importances in forests of randomized trees," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 26, 2013, pp. 431–439.

YUN-CHUNG CHEN received the B.S. degree in computer science and information engineering from the National Taiwan University, Taipei, Taiwan, in 2019, where he is currently pursuing the Ph.D. degree with the Graduate Institute of Information Security. His research interests include cybersecurity, machine learning, and mobile security.

HONG-YEN CHEN (Graduate Student Member, IEEE) received the B.S. degree in electronics engineering from the National Taiwan Normal University, Taipei, Taiwan, in 2019. He is currently pursuing the Ph.D. degree with the Graduate Institute of Communication Engineering, National Taiwan University, Taipei. His research interests include cybersecurity, machine learning, and deep learning.

TAKESHI TAKAHASHI (Member, IEEE) received the Ph.D. degree in telecommunications from Waseda University, in 2005. He was with Tampere University of Technology as a Researcher, from 2002 to 2004, and Roland Berger Ltd. as a Business Consultant, from 2005 to 2009. Since 2009, he has been with the National Institute of Information and Communications Technology, where he is currently a Research Manager. He was a Visiting Research Scholar with the University of California, Santa Barbara, from 2019 to 2020. His research interests include cybersecurity and machine learning.

BO SUN received the B.E. degree in science from Jilin University, in 2007, the M.E. degree in engineering from Yokohama National University, in 2012, and the Ph.D. degree in engineering from Waseda University, in 2018. He was with Waseda University as a Research Associate, from 2016 to 2018, and a Researcher with the National Institute of Information and Communications Technology, from 2018 to 2020. He is currently an Assistant Professor with Saitama Institute of Technology, a Collaborative Researcher with the National Institute of Information and Communications Technology, and a Visiting Researcher with Waseda University. His research interests include web security, mobile security, and offensive security.

TSUNG-NAN LIN (Senior Member, IEEE) received the B.S. degree from the National Taiwan University, Taipei, Taiwan, in 1989, and the M.S. and Ph.D. degrees from Princeton University, Princeton, NJ, USA, in 1993 and 1996, respectively. He then joined EPSON Research and Development, Inc., San Jose, CA, USA, and EMC Corporation, Hopkinton, MA, USA. Since February 2002, he has been with the Department of Electrical Engineering and the Graduate Institute of Communication Engineering (GICE), National Taiwan University. He had been the Director of the Division of Network Management, Computer and Information Networking Center, National Taiwan University, and the Vice President and the Director General of the Cybersecurity Technology Institute, Institute for Information Industry. He is a member of the Phi Tau Phi Scholastic Honor Society.