IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 23, 2018, accepted November 16, 2018, date of publication November 29, 2018,
date of current version December 27, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2883975

Android Malware Permission-Based Multi-Class Classification Using Extremely Randomized Trees

FAHAD ALSWAINA AND KHALED ELLEITHY, (Senior Member, IEEE)
Computer Science and Engineering Department, University of Bridgeport, Bridgeport, CT 06604, USA
Corresponding author: Fahad Alswaina (falswain@my.bridgeport.edu)

ABSTRACT Due to recent developments in hardware and software technologies for mobile phones, people depend on their smartphones more than ever before. Today, people conduct a variety of business, health, and financial transactions on their mobile devices. This trend has caused an influx of mobile applications that require users' sensitive information. As these applications increase so too have the number of malicious applications that steal users' sensitive information. Through our research, we developed a reverse engineering framework (RevEng). Within RevEng, the applications' permissions were selected, and then fed into machine learning algorithms. Through our research, we created a reduced set of permissions by using extremely randomized trees that achieved high accuracy and a shorter execution time. Furthermore, we conducted two approaches based on the extracted information. Approach one used binary value representation of the permissions. Approach two used the features' importance; we represented each selected permission (in approach one) by its weighted value instead of the binary value. We conducted a comparison between the results of our two approaches and other related work. Our approaches achieved better results in both accuracy and time performance with a reduced number of permissions.

INDEX TERMS Malware application, reverse engineering, machine learning, static analysis, android permissions, android security.

I. INTRODUCTION

Due to recent developments in hardware and software technologies for mobile phones, people depend on their smartphones more than ever before. As of 2017, more than 407 million mobile devices were sold as reported by Gartner; devices that operate on Android represented 86% of the total market [1]. Although this popularity is beneficial to Google's operating system, Android, this popularity has encouraged malicious developers to target Android users. F-Secure, a cybersecurity corporation, has reported that more than 99% of total malware attacks on mobile devices have targeted Android devices [2]. These attacks include any software or a piece of code, called a payload, that performed harmful activities and therefore comprised the confidentiality, integrity, or availability of the victims' data or resources [3]–[6].

Alongside researchers in both academia and the industry, Google has devoted significant attention to security issues in Android's software stack's components, especially at the application level, such as in license and application verification, security vulnerability, and intrusion detection. Nevertheless, as smartphones advance and incorporate high-resolution cameras and online services such as banking and GPS, so too increases the number of malicious applications (or malware apps); users' data and resources are always at risk.

As defined by Google, there are 17 categories of malware, including spyware and backdoor attacks, which are categorized based on the malware's behavior [7]. A malware could secretly be embedded in a set of deceptive applications and can be identified by finding specific files, or similar app's characteristics (i.e. signature or requested permissions), on the set. This set containing the malware's files is identified as a family of the malware [8].

For instance, the *DroidDream*, also known as *RootCager*, was discovered in 2011 in the official Android market, GooglePlay. *DroidDream* family is a Trojan that collects the mobile device's ID/serial number and other related information by requesting administrator access on the device. This Trojan can be detected by locating the two code files *rageagainstthecage* and *exploid* in the family members [9]–[12]. This family is one example of advanced and sophisticated malware. In order to protect the system's resources, Android, like Linux, uses a security system that employs a forced access control mechanism.

A. ANDROID ACCESS CONTROL

Android uses a security system that employs a forced access control mechanism. It requires apps to request permissions prior to utilizing any of the system's resources [13]. All of the permissions must be declared inside an XML file called the *AndroidManifest* where essential information on the app and its components are located (i.e., package name and version number, activities and intents, content providers, broadcast receivers, services, and permissions). Prior to Android version 6, the user was required to give a full access to everything that an app requested at the time of the installation. This is risky because, besides the average user's lack of knowledge about the permissions requested, is that an app can deceive the user by requesting permissions unrelated to the app's main functionality. A malware app, then, can leverage that by accessing the device's resources to perform its malicious acts [14]–[16].

In Android, there are more than 300 permissions, each of which has a level of protection considered either normal or dangerous [17]. A designation of normal implies low risk to isolated resources. All permissions with normal level are automatically granted to the app by the system without the user's consent (i.e., *SET_WALLPAPER*). Permissions categorized as dangerous, however, have a higher risk on the user's data and the device (i.e., *ANSWER_PHONE_CALLS*). For this reason, dangerous permissions require the user's consent prior to installation in order for access to be granted to the application [17]. This paper examines the permissions that malware families request as a feature of our static analysis.

B. PROBLEM STATEMENT

Classifying malware families is an important approach for anti-virus companies (AVs). AVs, as well as other researchers, try to find new malware that does not correlate to previously found malware. Nevertheless, malicious developers try to find ways to bypass the AVs' detection by both closely studying the behavior of AVs and also by applying various techniques to get around their detection techniques, such as code obfuscation.

With this track of research, AVs will be able to match new malware faster by applying the same malware signatures that they detect and then adopting patches that they developed for previously identified malware. Moreover, this research will support malware researchers in their effort to study undiscovered malware.

C. CONTRIBUTION OF THE PAPER

This paper has proposed a novel framework, i.e., RevEng, that classifies 1,233 samples of malware. Our framework identifies an optimal, highly accurate set of permissions out of all of the permissions provided by an Android operating system. We employed the feature's ranking algorithm used in Extremely Randomized Trees. Our set of permissions is tested on six classifiers to assign malware to their malware families. RevEng achieved a high prediction accuracy rate, higher than that found by other related work. To evaluate our approach, we listed a detailed comparison with StomDroid's framework results [18]. In summary, the proposed contributions of this paper are as follows:

- *Reverse engineering tool.* We designed and implemented a RevEng that reverse-engineers malware data sets based on their families and extracts the permissions from apps.
- *Multi-class classification.* We targeted a multi-class classification problem to assign a detected malware sample to previously studied and dissected malware families.
- *Candidate subset.* The proposed approach was able to identify a minimal subset of features with higher accuracy and a minimum execution time as compared to other related work. The candidate subset is listed in Table. 6.

The remainder of the paper is organized as follows: Section II surveys the related work. We present our framework in Section III. Section IV shows the experimental setup and the implementation. Section V discusses the results. Finally, in Section VI, we conclude the paper.

II. RELATED WORK

Continuous advancements in machine learning contribute notably to security, especially in malware. There are two methods of classification based on the feature (attribute) of an observation: binary and multi-class. A binary classification is based on predicting an observation in one of two classes. With a multi-class classification, though, an observation is classified into one class, out of multiple classes (more than two). For instance, in malware detection using binary classification, a classifier categorizes an app as malware or benign. When multi-class classification is used, the classifier instead assigns the app into one of at least three classes (i.e., spyware, rootkit, ransomware, etc.). The data set used in a binary classification is a collection of both benign and malware applications, while in multi-class classifications, the data set contains malware families and their samples.

Malware detection using machine learning algorithms (MLA) analyzes the malware to perform feature collection. In general, there are two main types of analyses. Dynamic (or behavioral) analysis studies the malware while the application is in the execution state. This analysis is effective in monitoring an application's activities in a controlled environment (sandbox) [19] and in understanding all communications within the device (i.e., communication between the app's components or IPC) and outside the device (i.e., networks traffic). Static analysis focuses on examining and collecting malware app attributes (i.e., callback sequence [20] and application programming interface (API) calls) while the application is in a static state. This analysis presents an advantage in its execution speed and low cost to detect the malware [21], since it does not require the app to be executed.

Several researchers have addressed general security risks on mobile devices, such as privacy, vulnerability, and

information leakage [22]–[28]. Other related works have been published on detecting or classifying malware apps using the following analysis techniques: static [29]–[31], dynamic [32], [33], or a combination of both techniques [18], [34]–[36].

In Drebin [29], the authors proposed a light framework capable of running on mobile devices; Drebin analyzed apps statically and gathered many features, such as the application's requested permissions and API calls. Analyzed apps were then classified, using machine learning, into benign class or malicious class. RNPDroid [37] is a framework for risk mitigation based on analyzing the application's permissions. The authors identified four risk factors: high, medium, low, or no risk. Based on the factors, the app is binarily classified as malicious or benign using statistical analyses such as ANOVA and T-test. The framework was tested on the M0Droid [38] data set with 400 samples. In [39], the authors proposed an MLDP model to rank permissions requested by the malicious app. This model used association rules to select a set of features and then used SVM for binary classification. Their SVM was trained with 5,000 different types of malware from 178 malware families and 5,000 types of benign software. Although the authors used malware families in their classification, the actual focus was not on multi-class classification. The effectiveness of MLDP was compared to 135 set of permissions. MalDozer [40] is a detection and multi-class classification framework based on API calls made by the application. Their framework studied the behavior of the application from its API calls' sequence and pattern. MalDozer used Deep Learning to classify malware applications into families.

Droid-Sec [36] used MLA and a deep learning classifier; it collected 200 features using static as well as dynamic analyses. The framework in [41] studied the dependency between the user's inputs (triggers) and the number of sensitive operations generated from apps' critical function calls. The framework used static and dynamic analyses to classify the app binarily. Their approach was tested on 482 malware and 708 benign apps. StormDroid [18] collected four sets of features: permissions, API calls, sequence, and the activities of the app; the framework used real-time stream processing and applied binary classification using machine learning to classify the app as benign or malicious.

The previous works gathered as many features as possible to achieve high accuracies, neglecting the overall performance overhead by increasing the number of features. This is especially important because some of the features collected might not have had a direct relationship to the malware being studied. As mentioned earlier, the number of applications available has increased significantly; therefore, there is a need to find a proper and minimum set of features to help researchers in detecting and classifying malware apps.

III. FRAMEWORK

RevEng consists of four main components that include the *Dataset*, *Family*, *App*, and *Analysis* components.

TABLE 1. List of abbreviations used in the article.

Abbreviation	Meaning
RevEng	Reverse Engineering Framework
AV	Anti-virus
API	Application Programming Interface
SF	Selected Features
SF_{Cand}	Selected Features Candidate $\subseteq$ SF
M_{Bcand}	Binary Matrix of SFcand
M_{Wcand}	Weighted Matrix of SFcand
AAPT	Android Asset Packaging Tool
SDK	Software Development Kit
p	Permission
w	Feature's Weight
ExF	Extracted Features Values (0's and 1's for binary)
MLA	Machine Learning Algorithm (Classifier)
SVM	Support Vector Machine Algorithm (Classifier)
ID3	Decision Tree Algorithm - ID3 (Classifier)
RF	Random Forest Algorithm (Classifier)
NN	Neural Network Algorithm (Classifier)
KN	K-nearest Neighbors Algorithm (Classifier)
ET	Extremely Randomized Trees Algorithm (Classifier)
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative

Each component parses and collects information on the data set. The *Dataset*, *Family*, and *App* components are included in the preprocessing stage, whereas the *Analysis* component is used in the processing stage. To explain our framework, we used the term *extracted features* to indicate the result of collecting the selected features from the application; this is not to be confused with feature extraction terminology.

In the following section, we identify the functionality of each component in our RevEng framework and their interactions in order to classify malware apps and predict malware families.

The following is a general flow of the framework. More details are added in the following section. The list of abbreviations used in the manuscript is provided in Table. 1.

1) The *Dataset* is needed to parse and maintain information about the malware families in the data set. The component takes the data set and assigns each family to a *Family* component to be processed. At the end of the preprocessing stage, the *Dataset* processes the results of each *Family* component, constructs, and prepare the input matrices (M_{Bcand} and M_{Wcand}) for the *Analysis* component in the processing stage.
2) The *Family* component processes one malware family and builds a list of all apps in the family. The component maintains and removes any duplicates of an application by calculating the hash value. Each member of the malware family is assigned to an App component to be processed. In the end, the *Family* component processes the result obtained from each *App* component and passes them back to the *Dataset*.
3) The *App* component represents a malware app. It reverse-engineers the malware application, extracts the features and passes them back to the *Family* component.

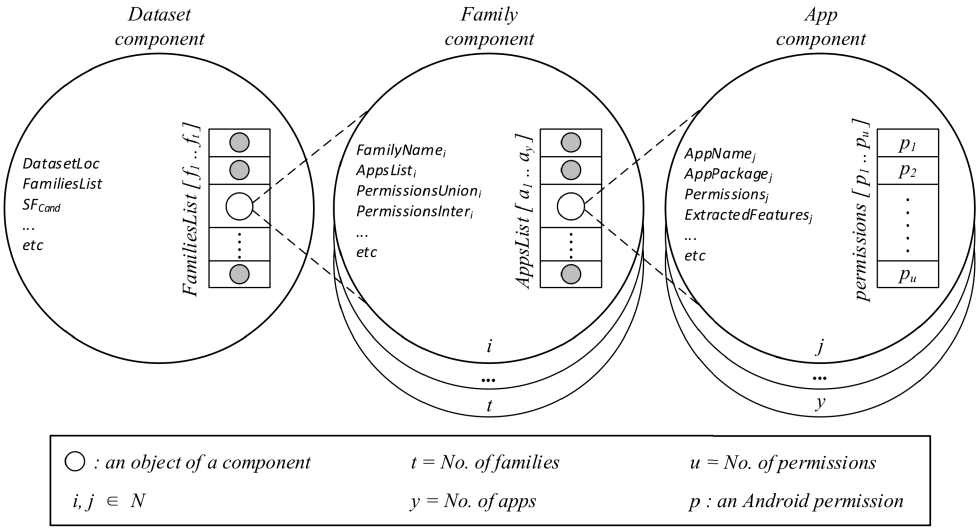

FIGURE 1. The contents of components and the relation between them.

4) The *Analysis* component is where the framework applies MLA to generate, train, and validate classification models. Then uses the data from the *Dataset* to predict the malware families.

A. FRAMEWORK COMPONENTS

Dataset: This component contains general information about the data set such as *FamiliesList* (a list of families in the data set), SF_{Cand} (a candidate subset of selected features SF), M_{Bcand} (a two-dimensional binary matrix result from applying SF_{Cand}), M_{Wcand} (a two dimensional weighted matrix result from applying the weight of each features in SF_{Cand}), and *NoOfThreads* (number of threads set for framework efficiency; the default is 4).

Family: This component contains detailed information on a malware family. Parameters such as *FamilyName* (name of the family), *AppList* (a list of apps in the malware family), *PermissionsUnion* (a set of all permissions declared in the malware family), and *PermissionsInter* (the intersection set of all permissions declared in the malware family) are collected by this component.

App: This component is responsible for reverse engineering a malware app. It extracts information such as *AppName* (the application file's name in the data set), *AppPackage* (the application's package name), *Permissions* (a list of permissions declared in the malware app), and *ExtractedFeatures* (a binary array result from applying feature selection in SF_{Cand}).

Analysis: This component consists of several machine learning algorithms or classifiers (MLAs). Each MLA creates a model with pre-set hyperparameters. These hyperparameters are elected and tuned based on trial and error to produce optimal results in our experiments. The MLAs' models are trained, validated, and tested on the M_{Bcand} and M_{Wcand} that are produced by the *Dataset* component as inputs to each model in order to classify each input into its predicted malware family. In this component, we take advantage of the Scikit Learn libraries [42] to implement machine learning algorithms. Fig. 1 demonstrates the *Dataset*, *Family*, and *App* components' data structure with pseudo-code.

B. FEATURES

The features used are the app's permissions as requested by the malware apps (samples). The focus is on finding the optimal set of permissions, a set that gives high accuracy, out of all of the permissions provided by an Android operating system. To accomplish this, one of the ensemble classifiers, called Extremely Randomized Trees (ET), [43] was utilized. ET, like Random Forest (RF) [44], is based on building a large collection (forest) of decision trees (DT). Each DT uses the whole set to build the tree and, for each split, finds the optimal cut-point based on information gain. RF develops each tree by selecting a random set of data and a random set of features. The target class of the observation predicted is based on the majority vote. For ET, the algorithms add more randomness to RF such that on each split in a tree, instead of selecting the optimal cut-point, ET selects a feature at random. In addition, ET ranks the importance of each feature using Gini importance [45].

Features Reduction: The SF here is the permissions feature used in StormDroid [18]. In order to extract the important features, we run an ET algorithm on the SF. As a result, each feature in SF is assigned an importance value between zero and one, based on the information that the attribute provides in ET's DT. All features with zero importance have been excluded, since such features either do not help classify malware into families or have some dependency between features. By collecting all features greater than zero, we have a Candidate Selected Features set (SF_{Cand}), which is a reduced set of features as shown in Fig. 2. The ultimate SF_{Cand} contains 42 out of 59 permissions. The SF_{Cand} chosen, with

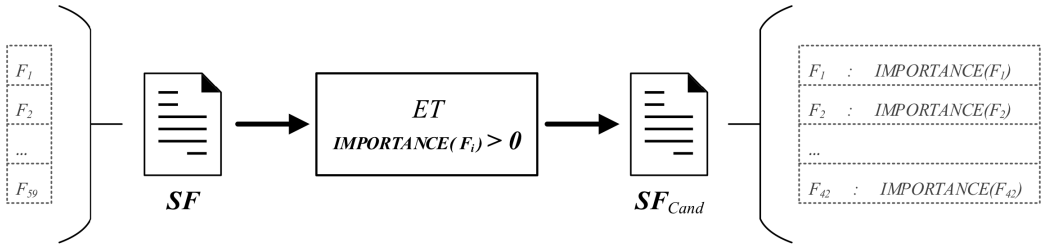

FIGURE 2. Extraction of the important features from *SF* and generating SF_{cand}.

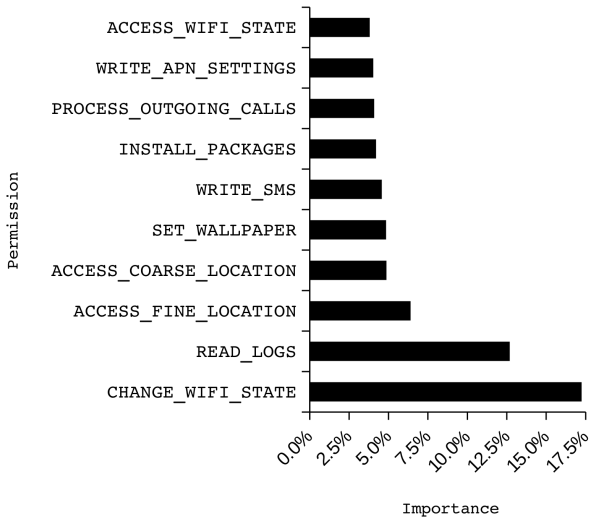

FIGURE 3. Top 10 permissions based on their importance.

their importance, are included in Appendix Table. 6. The top 10 permissions with high importance are shown in Fig. 3.

Our analysis of the data set shows that certain permissions are requested by many malware families. For example, *INTERNET* (which permits opening a network socket) is requested by more than 82% of the malware families; *READ_PHONE_STATE* (which permits a reading of the device's phone number, a status of ongoing calls, and phone accounts in the device) is requested by more than 60.5% of the malware families; and *ACCESS_NETWORK_STATE* (which permits querying into the status of the network, such as if the device is connected to a network) is requested by more than 42.5% of the malware families. These permissions are also the top three permissions in both [12] and [18]. For this reason, these permissions are not critical in order to identify and classify one malware family from another. Therefore, the ET classifier assigns a very low importance to such features, as shown in Table. 6.

C. DATA PREPROCESSING

Upon beginning the execution of the framework, the *Dataset* component is initialized (*Dataset.Init*) with the data set. Once the component is ready, RevEng starts loading and parsing the data set by executing *Dataset.Load*. In order to start creating the families' objects, RevEng forks a number of threads (*NoOfThreads*) assigned in the initialization during the execution of *Dataset.Run* as illustrated in Fig. 4.

Multi-threading utilizes the processor and increases the reverse engineering process of the applications as illustrated in Fig. 5. All objects of the *Family* component–in this case, malware families–are inserted in a list (i.e., Q). Each thread processes one object as a task, (i.e., t_i). Each task initializes a family (*Family.Init*), loads a family's contents, and starts parsing a family's application (*Family.Parse*) as shown in Fig. 4. *Family.Parse* initializes the *App* component (*App.init*) and parses the component (*App.Parse*).

The *App.Parse* method, in turn, extracts from the application information such as the package name and all permissions in the manifest file, and then checks the existence of each permission in *SF* in the app's list of permissions. To extract the package name and the declared permissions in the app's manifest file, we used the Android Asset Packaging Tool (AAPT), which is part of the Android Software Development Kit (SDK). AAPT is a utility with powerful features that decompiles the package's permissions listed in the Application manifest XML file; it can also extract the resources' table. The items' indices in ExF_A (extracted features) and *SF* (selected feature) are in the same order. If an app A has a feature $p \mid p \in SF$ in index i, then $ExF_A(i) = 1$, otherwise $ExF(i) = 0$, and so on.

$$ExF_A(i) = \begin{cases} 1 & \text{if } p \in A \wedge p \in SF; \\ 0 & \text{otherwise} \end{cases}$$

Each App's *ExF* is cascaded back to the app's family and then to the *Dataset* components as shown in Fig. 6.

$$SF = \left(p_1 \,..\, p_j \,..\, p_n\right)$$
$$ExF_A = \left(ExF(1) \;..\; ExF(j) \,..\, ExF(n)\right)$$

The *Dataset* joins all the *ExFs* in M_{Bcand} for analysis as illustrated in Fig. 6. The size of M_{Bcand} is $m \times n$, where $m = 1,233$ (total number of samples) and $n = 42$ (number of permissions in SF_{Cand}) as shown previously in Fig. 2.

$$w_{ij} = \rho_{ij} \times \; importance(SF_{Cand}[j])$$
$$i = 1, 2, ...|samples|, \quad j = 1, 2, ...|SF_{Cand}| \qquad (1)$$

In order to generate the Weighted Candidate Matrix M_{Wcand}, each element is calculated as in (1). Each ρ_{ij} value

```
Input: datasetLocation as DL,
       NoOfThreads as NT
Output: matrix as M

Dataset:
 Init(DL, NT):
   <Initialize local variables>

 Load():
   <Parse the data set & locate the families>

 Run():
  foreach familyi in dataset.families:
    IF familyi.Init(FL, SL=4) THEN
       start threadi from ThreadPool(NT)
         threadi exec familyi.Parse()
```

```
Input: familyLocation as FL,
       sizeLimit as SL
Output: familyName, appList,
        appPermissions, appExF

Family:
 Init(FL, SL)
   <Initialize local variables>
  Build AppList
  IF |apps| < SL THEN
    Return 0
  ELSE:
   Return 1

 Parse():
   foreach appi in family.apps:
      appi.Init(AL)
      appi.Parse()
```

```
Input: appLocation as AL, SF
Output: permissions as PRs,
        extractFeatures as ExF,
        appPackage as AP

App:
 Init(AL, SF):
    <Initialize local variables>

 Parse():
   Extract the AP
   Retrieve PRs

   // building ExF
   foreach itemi in SF:
      IF item in app.PRs THEN
         ExFi = 1
      ELSE
         ExFi = 0
```

FIGURE 4. Pseudo-code of the *Dataset*, *Family*, and *App* components, which show the main parts of the code.

in M_{Bcand} is multiplied by the permission's importance as generated by ET for the permission's index j. The Y matrix contains the malware families (classes: c_i) of each malware

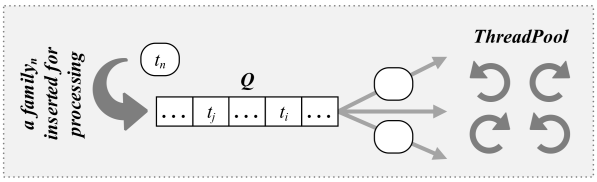

FIGURE 5. Multi-threading processes of the list of tasks in queue.

TABLE 2. The data set used in RevEng. A list of malware families with their samples.

Malware Family	No of Samples	Malware Family	No of Samples
GingerMaster	4	jSMSHider	16
HippoSMS	4	ADRD	22
FakePlayer	6	YZHC	22
GPSSMSSpy	6	DroidKungFu2	30
Asroot	8	DroidKungFu1	34
BeanBot	8	DroidDreamLight	46
Bgserv	9	GoldDream	47
Gone60	9	KMin	52
RogueSPPush	9	Pjapps	58
SndApps	10	Geinimi	69
Plankton	11	DroidKungFu4	96
zHash	11	BaseBridge	122
Zsone	12	AnserverBot	187
DroidDream	16	DroidKungFu3	309
Total	**1,233**		

sample at row i in both M_{Bcand} and M_{Wcand}. M_{Bcand} and Y matrices are shown below:

$$\underbrace{\left(\begin{array}{ccccc} p_{11} & \cdots & p_{1j} & \cdots & p_{1n} \\ \vdots & \cdots & \vdots & \cdots & \vdots \\ p_{i1} & \cdots & p_{ij} & \cdots & p_{in} \\ \vdots & \cdots & \vdots & \cdots & \vdots \\ p_{m1} & \cdots & p_{mj} & \cdots & p_{mn} \end{array}\right.}_{\text{MBcand}} \underbrace{\left.\begin{array}{c} c_1 \\ \vdots \\ c_i \\ \vdots \\ c_m \end{array}\right)}_{\text{Y}}$$

The overall framework is shown in Fig. 7.

IV. EXPERIMENTAL SETUP

A. DATA SET

We relied on the data set provided by [18]. This data set contained 49 malware families with a total of 1,260 applications. Each family differed in size between 1 and 300 applications. In this research, families that contained less than 4 applications have been excluded to maintain accurate results. Table. 2 lists the malware families and their samples used in our experiments, for a total of 1,233 applications in 28 families.

B. IMPLEMENTATION

The programming language Python was used in all of our implementations. Python is supported by the research community in various fields, and it has rich libraries. Scikit-Learn is one of the communities that has implemented Machine Learning Algorithms [42]. In our *Analysis* component, we used the following classifiers: Support Vector Machine (SVM), Decision Tree (ID3), Random Forest (RF),

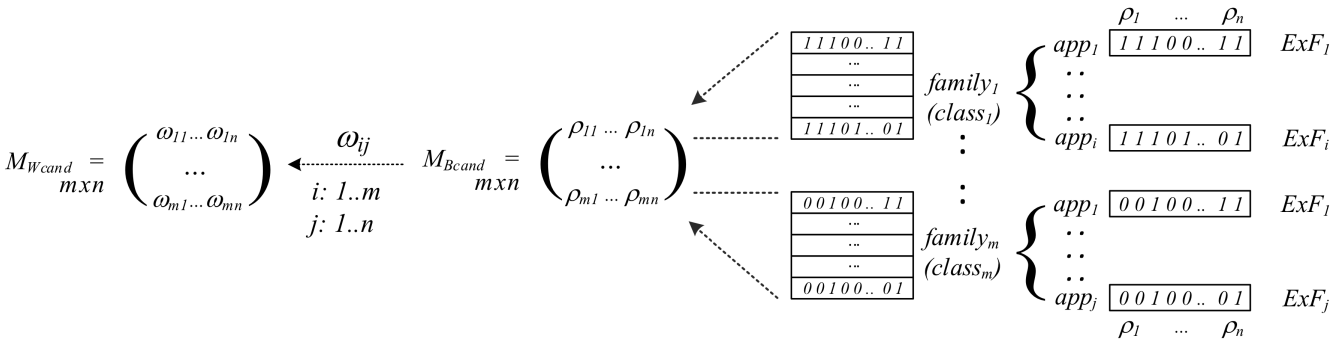

FIGURE 6. Preparation of M_{Bcand} and M_{Wcand} matrices for processing.

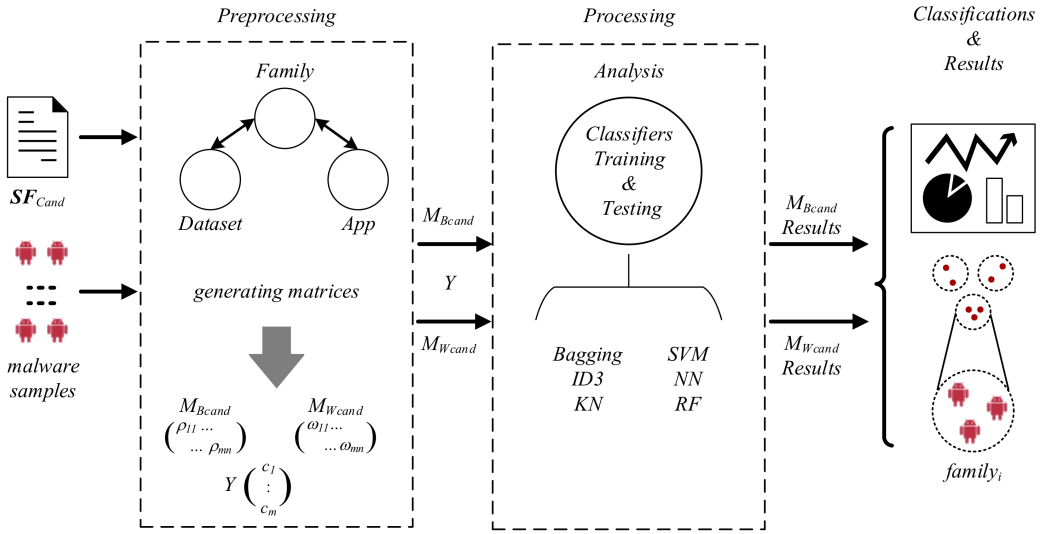

FIGURE 7. RevEng framework.

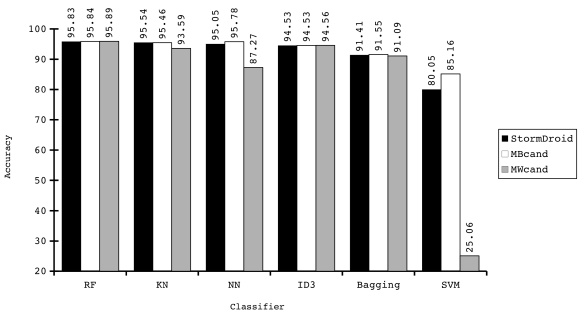

FIGURE 8. Comparison between StormDroid and our approach based on classifiers' accuracies.

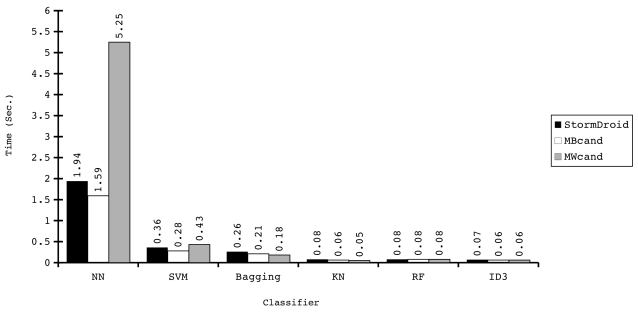

FIGURE 9. Comparison between StormDroid and our approach based on time performance.

Neural Network (NN), K-Nearest Neighbor (KN), and Bagging, as implemented by [42].

$$\begin{aligned} Accuracy &= (TP+TN)/(TP+TN+FP+FN) \\ Let\ S &= a\ malware\ sample\ and \\ C &= a\ malware\ family\ or\ class,\ then \\ TP(True-Positive) &\Rightarrow \\ prediction &: S \in C, actual-classification : S \in C \\ FP(False-Positive) &\Rightarrow \\ prediction &: S \in Cactual-classification : S \notin C \\ TN(True-Negative) &\Rightarrow \\ prediction &: S \notin C, actual-classification : S \notin C \\ FN(False-Negative) &\Rightarrow \\ prediction &: S \notin C, actual-classification : S \in C \end{aligned} \quad (2)$$

	BaseBridge	RogueSPPush	zHash	Zsone	ADRD	DroidKungFu4	Plankton	DroidDreamLight	SndApps	DroidKungFu2	Asroot	DroidDream	KMin	HippoSMS	YZHC	jSMSHider	Pjapps	AnserverBot	GingerMaster	GPSSMSSpy	Gone60	DroidKungFu3	Geinimi	GoldDream	FakePlayer	BeanBot	Bgserv	DroidKungFu1	
BaseBridge	**112**	2	-	-	2	-	-	2	-	-	2	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	92.8%
RogueSPPush	-	**9**	-	-	-	-	1	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100%
zHash	-	-	-	**11**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100%
Zsone	-	-	-	-	**12**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100%
ADRD	-	-	-	-	-	**22**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100%
DroidKungFu4	-	-	-	-	-	-	**94**	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	97.9%
Plankton	-	1	-	-	-	-	-	**10**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	90.9%
DroidDreamLight	2	-	-	-	-	-	1	**41**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	89.1%
SndApps	-	-	-	-	-	-	-	-	**10**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100%
DroidKungFu2	-	1	-	-	-	1	-	-	-	**25**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	83.3%
Asroot	2	-	-	-	-	-	-	-	-	-	**5**	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	62.5%
DroidDream	1	-	-	-	-	-	-	-	-	8	1	**6**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	37.5%
KMin	-	-	-	-	-	-	-	-	-	-	-	-	**52**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100%
HippoSMS	-	-	-	-	-	-	-	-	-	-	-	-	-	**4**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	100%
YZHC	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**22**	-	-	-	-	-	-	-	-	-	-	-	-	-	100%
jSMSHider	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**16**	-	-	-	-	-	-	-	-	-	-	-	-	100%
Pjapps	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	**55**	-	-	-	-	-	-	-	-	-	-	-	94.8%
AnserverBot	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**187**	-	-	-	-	-	-	-	-	-	-	100%
GingerMaster	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**4**	-	-	-	-	-	-	-	-	-	100%
GPSSMSSpy	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**6**	-	-	-	-	-	-	-	-	100%
Gone60	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**8**	-	-	-	-	-	-	-	88.9%
DroidKungFu3	-	-	-	-	-	1	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	**307**	-	-	-	-	-	-	99.4%
Geinimi	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**69**	-	-	-	-	-	100%
GoldDream	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**47**	-	-	-	-	100%
FakePlayer	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**6**	-	-	-	100%
BeanBot	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**8**	-	-	100%
Bgserv	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**9**	-	100%
DroidKungFu1	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**33**	97.1%

PREDICTED CLASS (FAMILY)

ACTUAL CLASS (FAMILY)

FIGURE 10. Confusion Matrix of One RF Execution.

TABLE 3. Detailed performance (prediction accuracy and time) for each classifier.

	M_{Bcand}				M_{Wcand}			
Classifier	Worst	**Avg.**	Best	Time	Worst	**Avg.**	Best	Time
SVM	85.16	**85.16**	85.16	0.28	25.06	**25.06**	25.06	0.44
NN	95.78	**95.78**	95.78	1.57	87.27	**87.27**	87.27	5.17
ID3	94	**94.52**	94.73	0.06	94.08	**94.42**	94.89	0.06
KN	95.46	**95.46**	95.46	0.06	93.59	**93.59**	93.59	0.05
Bagging	90.59	**91.56**	92.21	0.21	90.11	**91.09**	92.05	0.18
RF	94.73	**95.81**	96.27	0.08	95.38	**95.99**	96.43	0.08

TABLE 4. Classifiers' average accuracies and time performance(s) for 100 experiments.

	M_{Bcand}		M_{Wcand}		**StormDroid [18]**	
Classifier	Accuracy	Time	Accuracy	Time	Accuracy	Time
SVM	85.16	0.28	25.06	0.44	80.05	0.36
NN	95.78	1.57	87.27	5.17	95.05	1.92
ID3	94.52	0.06	94.42	0.06	94.52	0.07
KN	95.46	0.06	93.59	0.05	95.54	0.08
Bagging	91.56	0.21	91.09	0.18	91.65	0.26
RF	**95.81**	**0.08**	**95.99**	**0.08**	**95.97**	**0.08**

TABLE 5. Comparison between classifiers in terms of the best accuracy and best time performance.

	M_{Bcand}		M_{Wcand}		**StormDroid [18]**	
Best	Accuracy	Time	Accuracy	Time	Accuracy	Time
Accuracy	95.81 (RF)	0.08	95.99 (RF)	0.08	95.97 (RF)	0.08
Time	94.52 (ID3)	0.06	93.59 (KN)	0.05	94.52 (ID3)	0.07

C. EVALUATION

Cross-Validation: Since the number of malware families is very low, as is the number of malware samples, we used the cross-validation (or stratified k-fold) technique to split and alternate between the training and testing sets. We set up the number of folds (k=4) such that on each iteration, the classifier used 75% of a family's samples for training and 25% for testing. In the processing stage, the *Analysis* component is fed with M_{Bcand} and M_{Wcand}. Each classifier trains, validates, and tests the model on the two inputs using the aforementioned setup. As a result of the analysis, we calculated each classifier's accuracy (2) and the execution time in seconds.

V. RESULTS AND DISCUSSION

We conducted 100 experiments using M_{Bcand} and M_{Wcand} on each classifier. The experiment measured two factors: the classifier's prediction accuracy and the time performance. For all total experiments on each classifier, we calculated the worst, the best, and the average accuracy and the average execution time. Table. 3 shows the details of the experiments in two main columns: the first and second column represent the results of our approach with M_{Bcand} and M_{Wcand}, respectively.

The results show that using M_{Bcand}, RF, KN, and NN achieve high accuracy (average $\approx$ 95.68% and standard deviation $\approx$ 0.19%) in comparison with other classifiers (such as SVM, ID3, and Bagging). From the best selected classifiers, we can see that RF achieves the highest prediction, on average, of 95.81%. In terms of time performance, KN and RF complete their analyses in 0.06 seconds and 0.08 seconds, respectively, while NN achieves the lowest performance. SVM has the highest misclassification rate using M_{Bcand}.

For M_{Wcand}, the results have higher variations than the previous approach. The top three classifiers are RF, KN, and ID3 (average $\approx$ 94.66% and standard deviation $\approx$ 0.21%). The RF classifier also achieves the highest accuracy of 95.99%. SVM produces the lowest accuracy score using this feature. In terms of time performance, we can see that RF completed the experiments in 0.08 seconds on average. KN completed faster than the previous approach with an execution time of 0.05 seconds.

Comparing our two approaches, M_{Bcand} and M_{Wcand}, we can see that RF achieves the highest accuracy with a rate of 95.99% using M_{Wcand}, which was slightly higher than when using M_{Bcand}, by 0.18%. RF's took 0.08 seconds using both approaches.

We applied StormDroid's feature (59 permissions) [18] as shown in Table. 4 and Fig. 8. We found that the RF classifier produced the highest accuracy of 95.97% versus the other classifiers. RF also completed in 0.08 seconds.

In Table. 5, we summarized our comparison based on two categories: the classifiers' highest accuracies and the classifiers' best time performances. Of all three approaches, RF achieved the highest accuracy on M_{Wcand} with a rate of 95.99% in 0.08 seconds. For the best execution time, we found that KN was the best on M_{Wcand} with 0.05 seconds and an accuracy of 93.59%. Regarding time performance, we can see that ID3 performed faster using M_{Bcand}, although the classifier had the exact same accuracy as in the related work [18].

From our previous discussion, we concluded that M_{Wcand} achieved 0.02% better accuracy than StormDroid [18] with an exactly equal execution time. The accuracy of the RF classifier using all the three approaches is similar, in general. However, minimizing the number of features from 59 to 42 (0.28% of features) means a reduction in the dimensionality.

In conclusion, our framework improved the accuracy. Moreover, when using M_{Wcand} with KN, we achieved a shorter execution time than the related work [18] with a 37.5% improvement as shown in Fig. 9. A sample confusion matrix of our RF is presented in Appendix Fig. 10.

VI. CONCLUSION AND FUTURE WORK

Malware detection and analysis have been a problem for many years. With the escalation in the number of applications, especially on mobile devices, researchers have studied malware in-depth using various tools, such as machine learning. In this paper, we adopted machine learning to analyze and identify malware features such as the permissions requested by malware. Our focus in this paper was to find a small subset

TABLE 6. List of SF_{Cand} with their importance ($\omega > 0$).

Permission	Weight	Permission	Weight
CHANGE_WIFI_STATE	17.242%	RESTART_PACKAGES	0.852%
READ_LOGS	12.695%	CHANGE_NETWORK_STATE	0.786%
ACCESS_FINE_LOCATION	6.395%	RECEIVE_MMS	0.624%
ACCESS_COARSE_LOCATION	4.865%	BLUETOOTH	0.575%
SET_WALLPAPER	4.846%	DELETE_PACKAGES	0.565%
WRITE_SMS	4.580%	DISABLE_KEYGUARD	0.555%
INSTALL_PACKAGES	4.204%	WRITE_SETTINGS	0.528%
PROCESS_OUTGOING_CALLS	4.089%	CALL_PHONE	0.513%
WRITE_APN_SETTINGS	4.035%	RECEIVE_WAP_PUSH	0.410%
ACCESS_WIFI_STATE	3.807%	INTERNET	0.395%
RECEIVE_SMS	2.919%	READ_EXTERNAL_STORAGE	0.381%
SEND_SMS	2.854%	CLEAR_APP_CACHE	0.123%
ACCESS_NETWORK_STATE	2.794%	WRITE_SYNC_SETTINGS	0.087%
READ_SMS	2.498%	BLUETOOTH_ADMIN	0.086%
RECEIVE_BOOT_COMPLETED	2.265%	READ_SYNC_SETTINGS	0.076%
READ_PHONE_STATE	1.901%	ACCESS_MOCK_LOCATION	0.070%
READ_CONTACTS	1.871%	RECORD_AUDIO	0.023%
VIBRATE	1.733%	SYSTEM_ALERT_WINDOW	0.019%
MODIFY_PHONE_STATE	1.691%	N / A	-
MODIFY_AUDIO_SETTINGS	1.673%	N / A	-
WAKE_LOCK	1.624%	N / A	-
GET_ACCOUNTS	1.574%	N / A	-
BROADCAST_STICKY	1.168%	N / A	-

of permissions that could be used to classify applications into their proper malware families. We utilized Extremely Randomized Trees to further reduce the number of features from 59 to 42 (by 0.28%). In our two approaches, we represented the selected features as binary values, M_{Bcand}, and as weighted values, M_{Wcand}. We evaluated our approaches based on the accuracy and time performance of six classifiers, and we achieved both a higher accuracy by 0.02% (RF, 95.99%) and a shorter time performance by 37.5% with KN over StormDroid [18]. In future work, malware sensitive API calls should be investigatedto identify a subset that will further improve our framework's ability to predict malware families. We also recommend using deep neural network (DNN) classifier in the future. Because DNN performs better with large data sets, AndroZoo [46] could be a good candidate for our future experiments.

APPENDIX

See Table 6.

REFERENCES

[1] *Gartner Says Worldwide Sales of Smartphones Recorded First Ever Decline During the Fourth Quarter of 2017*. Accessed: Apr. 1, 2018. [Online]. Available: https://www.gartner.com/newsroom/id/3859963

[2] *Another Reason 99% of Mobile Malware Targets Androids—Safe and Savvy Blog by F-Secure*. Accessed: Apr. 10, 2018. [Online]. Available: https://safeandsavvy.f-secure.com/2017/02/15/another-reason-99-percent-of-mobile-malware-targets-androids/

[3] *NCP—Checklist McAfee Antivirus 8.8 STIG*. Accessed: Apr. 10, 2018. [Online]. Available: https://nvd.nist.gov/ncp/checklist/479

[4] D. Moon, H. Im, J. D. Lee, and J. H. Park, "MLDS: Multi-layer defense system for preventing advanced persistent threats," *Symmetry*, vol. 6, no. 4, pp. 997–1010, 2014.

[5] B. Gupta, D. P. Agrawal, and S. Yamaguchi, *Handbook of Research on Modern Cryptographic Solutions for Computer and Cyber Security*. Hershey, PA, USA: IGI Global, 2016.

[6] T. Akhtar, B. B. Gupta, and S. Yamaguchi, "Malware propagation effects on SCADA system and smart power grid," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, Jan. 2018, pp. 1–6.

[7] *Android Security 2017 Year in Review*. Accessed: Apr. 10, 2018. [Online]. Available: https://goo.gl/hiCgHQ

[8] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh, "Automatic generation of string signatures for malware detection," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*. Springer, 2009, pp. 101–120.

[9] C. Nachenberg, "A window into mobile device security," *Symantec Secur. Response*, pp. 4–9, 2011.

[10] K. Dunham, S. Hartman, M. Quintans, J. A. Morales, and T. Strazzere, *Android Malware and Analysis*. Boca Raton, FL, USA: CRC Press, 2014.

[11] H. Pieterse and M. S. Olivier, "Android botnets on the rise: Trends and characteristics," in *Proc. Inf. Secur. South Africa (ISSA)*, Aug. 2012, pp. 1–5.

[12] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2012, pp. 95–109.

[13] Y. Peng, M. Zhang, J. Zheng, and Z. Qian, "Research on Android access control based on isolation mechanism," in *Proc. 13th Web Inf. Syst. Appl. Conf.*, Sep. 2016, pp. 231–235.

[14] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Automatically securing permission-based software by reducing the attack surface: An application to Android," in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2012, pp. 274–277.

[15] S. Rastogi, K. Bhushan, and B. B. Gupta, "Measuring Android app repackaging prevalence based on the permissions of app," *Procedia Technol.*, vol. 24, pp. 1436–1444, 2016.

[16] S. Rastogi, K. Bhushan, and B. B. Gupta, "Android applications repackaging detection techniques for smartphone devices," *Procedia Comput. Sci.*, vol. 78, pp. 26–32, 2016.

[17] *Permissions Overview: Android Developers*. Accessed: Apr. 10, 2018. [Online]. Available: https://developer.android.com/guide/topics/permissions/index.html

[18] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "StormDroid: A streaminglized machine learning-based system for detecting Android malware," in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur.*, 2016, pp. 377–388.

[19] *Cuckoo Sandbox Book*. Accessed: Apr. 15, 2018. [Online]. Available: https://cuckoo.sh/docs

[20] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in Android applications," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng. (ICSE)*, vol. 1, May 2015, pp. 89–99.

[21] P. Faruki *et al.*, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 2, pp. 998–1022, 2nd Quart., 2015.
[22] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh, "Cells: A virtual mobile smartphone architecture," in *Proc. 23rd ACM Symp. Oper. Syst. Princ.*, 2011, pp. 173–187.
[23] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "MockDroid: Trading privacy for application functionality on smartphones," in *Proc. 12th Workshop Mobile Comput. Syst. Appl.*, 2011, pp. 49–54.
[24] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. 9th Int. Conf. Mobile Syst., Appl., Services*, 2011, pp. 239–252.
[25] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *Proc. USENIX Secur. Symp.*, vol. 31, 2011, p. 23.
[26] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting privacy leaks in iOS applications," in *Proc. NDSS*, 2011, pp. 177–183.
[27] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, p. 5, 2010.
[28] I. Forain *et al.*, "Endpoint security in networks: An openmp approach for increasing malware detection speed," *Symmetry*, vol. 9, no. 9, p. 172, 2017.
[29] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "DREBIN: Effective and explainable detection of Android malware in your pocket," in *NDSS*, 2014, pp. 23–26.
[30] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode sequences as representation of executables for data-mining-based unknown malware detection," *Inf. Sci.*, vol. 231, pp. 64–82, May 2013.
[31] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna, "A static, packer-agnostic filter to detect similar malware samples," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Springer, 2012, pp. 102–122.
[32] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets," in *Proc. NDSS*, Feb. 2012, vol. 25. no. 4, pp. 50–52.
[33] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, "Novel feature extraction, selection and fusion for effective malware family classification," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, 2016, pp. 183–194.
[34] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An Android application sandbox system for suspicious software detection," in *Proc. 5th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2010, pp. 55–62.
[35] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. 23rd Annu. Comput. Secur. Appl. Conf. (ACSAC)*, Dec. 2007, pp. 421–430.
[36] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: Deep learning in Android malware detection," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 371–372, 2014.
[37] K. Sharma and B. B. Gupta, "Mitigation and risk factor analysis of Android applications," *Comput. Elect. Eng.*, vol. 71, pp. 416–430, Oct. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045790618305494
[38] M. Damshenas, A. Dehghantanha, K.-K. R. Choo, and R. Mahmod, "M0droid: An Android behavioral-based malware detection model," *J. Inf. Privacy Secur.*, vol. 11, no. 3, pp. 141–157, 2015.
[39] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An, and H. Ye, "Significant permission identification for machine-learning-based Android malware detection," *IEEE Trans. Ind. Informat.*, vol. 14, no. 7, pp. 3216–3225, Jul. 2018.
[40] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: Automatic framework for Android malware detection using deep learning," *Digit. Invest.*, vol. 24, pp. S48–S59, Mar. 2018.
[41] K. O. Elish, D. D. Yao, B. G. Ryder, and X. Jiang, "A static assurance analysis of Android applications," Tech. Rep., 2013.
[42] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.
[43] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Mach. Learn.*, vol. 63, no. 1, pp. 3–42, 2006.
[44] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[45] *sklearn.tree.ExtraTreeClassifier*. Accessed: Apr. 18, 2018). [Online]. Available: http://scikit-learn.org/stable/modules/generated/sklearn.tree.ExtraTreeClassifier.html
[46] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proc. IEEE/ACM 13th Work. Conf. Mining Softw. Repositories (MSR)*, May 2016, pp. 468–471.

FAHAD ALSWAINA received the B.S. degree in computer science and information systems from King Saud University, Riyadh, Saudi Arabia, in 2005, and the M.Sc. degree from California Lutheran University, Thousand Oaks, CA, USA, in 2011. He is currently pursuing the Ph.D. degree in computer science and engineering with the University of Bridgeport (UB), Bridgeport, CT, USA.

From 2005 to 2006, he was a Software Engineer at King Abdulaziz City for Science and Technology, Riyadh. He has been a Ph.D. Teaching Assistant with the School of Engineering and is a member of the Wireless and Mobile Communications Laboratory, UB. His research interests include malware analysis, cybersecurity, data science, and artificial intelligence.

KHALED ELLEITHY received the B.Sc. degree in computer science and automatic control and the M.S. degree in computer networks from Alexandria University in 1983 and 1986, respectively, and the M.S. and Ph.D. degrees in computer science from the Center for Advanced Computer Studies, University of Louisiana at Lafayette, in 1988 and 1990, respectively. He is currently the Associate Vice President for graduate studies and research with the University of Bridgeport. He is also a Professor of computer science and engineering. He supervised hundreds of senior projects, M.S. theses, and Ph.D. dissertations. He developed and introduced many new undergraduate/graduate courses. He also developed new teaching/research laboratories in his area of expertise. He has authored over 350 research papers in national/international journals and conferences in his areas of expertise. His research interests include wireless sensor networks, mobile communications, network security, quantum computing, and formal approaches for design and verification. He has been a member of the ACM since 1990, a member of the ACM Special Interest Group on Computer Architecture since 1990, a member of the Honor Society of the Phi Kappa Phi University of South Western Louisiana Chapter since 1989, a member of the IEEE Circuits and Systems Society since 1988, a member of the IEEE Computer Society since 1988, and a Lifetime Member of the Egyptian Engineering Syndicate since 1983. He is a member of the technical program committees of many international conferences as recognition of his research qualifications. He is also a member of several technical and honorary societies. He is a Senior Member of the IEEE Computer Society. He was a recipient of the Distinguished Professor of the Year at the University of Bridgeport for academic year 2006–2007. He is an editor or co-editor for 12 books published by Springer.

His students received over 20 prestigious national/international awards from the IEEE, the ACM, and the ASEE. He was the Chair Person of the International Conference on Industrial Electronics, Technology, and Automation. He was the Co-Chair and the Co-Founder of the Annual International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering virtual conferences 2005–2014. He served as a guest editor for several international journals.