PROPOSED FRAMEWORK FOR IMPROVING PERFORMANCE OF FAMILIAL CLASSIFICATION IN ANDROID MALWARE

Fahad Abdulaziz O. Alswaina

Under the Supervision of Dr. Khaled Elleithy

DISSERTATION

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING

THE SCHOOL OF ENGINEERING

UNIVERSITY OF BRIDGEPORT

CONNECTICUT

July 2020

PROPOSED FRAMEWORK FOR IMPROVING PERFORMANCE OF FAMILIAL CLASSIFICATION IN ANDROID MALWARE

Fahad Abdulaziz O. Alswaina

Under the Supervision of Dr. Khaled Elleithy

Approvals

Committee Members

Name	Signature	Date
Dr. Khaled Elleithy	[illegible]	10/26/2020
Dr. Ausif Mahmood	[illegible]	10-26-2020
Dr. Miad Faezipour	[illegible]	10/26/2020
Dr. Sarosh Patel	[illegible]	10/26/2020
Dr. Samir Hamada	[illegible]	10/26/2020

Ph.D. Program Coordinator

Dr. Khaled Elleithy	[illegible]	10/26/2020

Chairman, Computer Science and Engineering Department

Dr. Ausif Mahmood	[illegible]	10-26-2020

Dean, School of Engineering

Dr. Khaled Elleithy	[illegible]	10/26/2020

PROPOSED FRAMEWORK FOR IMPROVING PERFORMANCE OF FAMILIAL CLASSIFICATION IN ANDROID MALWARE

PROPOSED FRAMEWORK FOR IMPROVING PERFORMANCE OF FAMILIAL CLASSIFICATION IN ANDROID MALWARE

ABSTRACT

Because of the 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 increased, which may compromise users' sensitive information. Between all smartphone, Android receives major attention from security practitioners and researchers due to the large number of malicious applications. For the past twelve years, Android malicious applications have been clustered into groups for better identification. Characterizing the malware families can improve the detection process and understand the malware patterns. However, in the research community, detecting new malware families is a challenge. In this research, a framework is proposed to improve the performance of familial classification in Android malware. The framework is named a Reverse Engineering Framework (RevEng). Within RevEng, applications' permissions were selected and then fed into machine learning algorithms. Through our research, we created a reduced set of permissions using Extremely Randomized Trees algorithm that achieved high accuracy and a shorter execution time. Furthermore, we conducted two approaches based on the extracted information. The first approach used a binary value representation of the permissions. The second approach used the features' importance. We represented each selected permission in latter approach by its weight value instead of its binary value in the former approach. We conducted a comparison between the results of our two approaches and other relevant works. Our approaches achieved better results in both accuracy and time performance with a reduced number of permissions.

ACKNOWLEDGEMENTS

My thanks are wholly devoted to God who has helped me all the way to complete this work successfully. I owe a debt of gratitude to Prof. Elleithy for his guidance, patience, motivation, and immense knowledge during this research. To my parents for their prayers and patience. To my good wife for her encouragement and support throughout the academic journey; you and our little man, Abdulaziz, are my best friends and blessings in my life. Furthermore, to my teacher, Prof. Ausif Mahmood for being there for mentoring. Without them, this research would not have been possible.

I would also thank my respectful committee members: Prof. Ausif Mahmood, Dr. Miad Faezipour, Dr. Sarosh Patel, and Dr. Samir Hamada for their great feedbacks and constructive criticism. I am honored that my work has been supervised by Prof. Khaled Elleithy.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Due to the 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 recorded 86% of the total market [1]. Although this popularity is beneficial to Google's operating system, Android, this popularity has propelled malicious developers to target Android users. F-Secure, a known corporation on cybersecurity, has reported that more than 99% of total malware attacks on mobile devices have targeted Android devices [2]. The attacks include software with malicious code, called a payload, that performs harmful activities and compromises the confidentiality, integrity, or availability of the victims' data or resources [3]–[5].

Google, as well as a large group of researchers in academia and the industry, has devoted significant attention to security issues in Android's software stack's components, especially at the application level, such as in licensing and application verification, security vulnerability, and intrusion detection. Nevertheless, with smartphones' ever-increasing advanced features, such as high-resolution cameras and sensors, as well as online services such as banking and GPS, so too increase 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 like spyware and backdoors categorized based on the malware's behavior [6].

1.2 Research Problem and Scope

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 by applying various techniques to get around their detection techniques, such as code obfuscation.

AVs have been using the signature-based technique to detect any malware. Signatures (i.e. the hash value of the file) of a malware are a single or a group of blueprint data that is generated and associated with the malware. Every company has its own signature for the same malware. However, since malicious developers always attempt to bypass the detection and hide their code, they change the content and create variants of the malware. Any changes made to the malware will have a direct effect on the hash of the file and will impact the other signatures created for the malware.

With today's continues increasing number of applications, there is a need for a proper identification of malware features that would characterize both malware and its variants. Those features are used as inputs to machine learning algorithms for classifications. Such algorithms or classifiers are trained to create a model that would identify the finger-like of the malware and the malware's variants.

Although extracting features helps in studying the malware, the size of the features set affects the overall complexity of the learning process. In this case, researchers should focus on identifying the smallest set of features that give the highest and most accurate classification result.

1.3 Motivation behind the Research

Cybersecurity is an essential defense line to preserve the privacy of users. Users can be targeted such as a governmental employee in a military department or a head of a financial company. The information leakage has a tremendous impact on their own life or on the place they work for. Beside targeted users, an ordinary individual can be randomly hunted and the leakage of of data would impact them and their surroundings as well. Malicious developers utilize the information for their own evil benefits or can be used by other spying agencies.

Especially with the applications when the malicious developers get control of a device and are granted the trust from the user, they will last for long period and less likely to be removed unless their control was detected. By keeping the malware installed, information will be streamed without prior knowledge of the user.

With the advancement in data science and machine learning, classifiers can be trained on all sort of malware to come up with models. Models are used afterward for evaluation. When models are ready, they are deployed for testing and start classifying the application. The machine learning field, nowadays, is supported by many research efforts, academic, and industrial organizations for different problem domain: classification or regression problems.

With this track of research, AVs will be able to match any variant of the malware quickly to their families by applying the same detected malware signatures. Thus, AVs can easily adopt patches they developed for the previously identified malware. Moreover, this research will support malware researchers to complement their efforts to study undiscovered malware families.

1.4 Contributions of the Proposed Research

This research has proposed a novel framework, i.e., RevEng, that classifies 1,233 samples of malware. Our framework identifies an optimal set of permissions, a set that gives high accuracy, out of all the permissions provided by an Android operating system. We used the feature's ranking algorithm used in Extremely Randomized Trees. The set of permissions is tested on six classifiers to assign malware into its family. RevEng also achieves a higher prediction accuracy rate than other related work. To evaluate our approach, we list a detailed comparison with StomDroid's framework results [7]. The list of abbreviations used in the dissertation is provided in Table A.1. In summary, the proposed contributions in this research are summarized as follows:

- We introduce a novel taxonomy that categorizes all the related work in familial classification in terms of the type of analyses, features, and techniques that has been used.
- We design and implement RevEng that reverse-engineers malware datasets based on their families and extracts the permissions from apps.
- We are targeting a multi-class classification problem to assign a detected malware sample to previously studied and dissected malware families.
- The proposed approach can identify a minimal subset of features with higher accuracy and a minimum execution time compared to other related work.

CHAPTER 2: LITERATURE REVIEW

The chapter discusses the Android operating system, application, and Android malware with their attack and activation techniques. Also, we discuss in detail the related work in three dimensions: type of analysis, techniques used, and features. We identified the datasets are used and show the limitation and challenges in the literature. At the end we conclude with the future directions of the research. This chapter is published in [8].

2.1. Android and Malware

In this section, we discuss Android operating system and application. We address the main components inside them and define some technologies and fundamentals. Then, we discuss Android malware and the attacks they use to harm the user.

2.1.1 Android Operating System

Android is one of Google products that is designed for smartphones and mostly written in Java language. Android uses a Linux kernel to communicate with the hardware. The platform architecture [9] consists of: system apps, Java API, C/C++ libraries, HAL, and Linux kernel. The updated overall architecture of the Android in [10] is shown in Figure 2.1.

System Apps. System apps (also called core apps) are applications that come pre-installed on the Android system for email communication, SMS messaging, or calling service. Third-party applications are native apps that do not come preinstalled on the system and can be downloaded from the Android application store (Google Play) or other unofficial markets. Google Play is a place that indexes all Google trusted applications.

Java API Framework. A set of Application Programming Interface (API) written in Java to communicate with system components and services. Applications utilize APIs such as view system (i.e. button and text boxes), the resource manager (i.e. graphics and layouts), notification manager (i.e. alerts on the status bar), activity manager (i.e. managing back stack for activities), and content provider (i.e. share data).

Native C/C++ Libraries. Many of the kernel core components are built from libraries that are written on C and C++ languages. Libraries such as Webkit, Libc, and OpenGL ES are some of those libraries.

Hardware Abstraction Layer (HAL). HAL provides the hardware capabilities to Java API Framework.

Linux Kernel. Linux kernel is modified and updated especially for Android. The kernel does not include GNU C compiler, GNU libraries, or X server like in known Linux kernel. Some of the included features to this kernel are Low-memory Killer, Wakelocks, Anonymous Shared Memory, Paranoid Network, and Binder.

2.1.2. Android Application

Android applications are written on Java language and compiled to special bytecode called Dalvik bytecode. Dalvik bytecode is interpreted using Dalvik Virtual Machine (DVM). DVM is a register-based VM that uses CPU registers to store the data in the instruction.

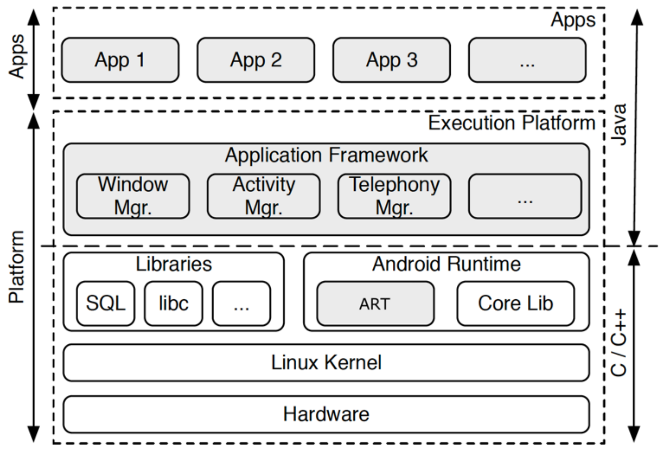


Figure 2.1 Android platform architecture.

In 2017, Google announced Kotlin to be another official Android development language. Android contains four main components that form the building blocks of the application [11]: Activities, Services, Broadcast receiver, and Content providers. *Activity* is a Java class (a single screen) and entry point that the user interacts with. For example, in a phone app, contacts screen is an instance of an activity that shows a list of contacts. *Services* are background processes that process long-running jobs. An example of a service is running some updates for the application. *Broadcast receiver* is a component that responds to system announcements or delivers broadcasts to another or within the same app. An example of this component is when the user notified that the battery is low. Finally, *Content provider* manages data stored in a database, i.e., SQLite, or in the file system. It allows other apps to query such data if they have the permissions. For instance, the content provider responses to the user click on the contacts list and show the list in the phone app. Moreover, it is important to mention an important message event called Intent. *Intent* is a message object that is used to perform some operations such as starting an activity or a service or delivering a broadcast message to broadcast receivers. The intent object contains a set of information such as component name, action to be performed, data type, category type, extras, and flag.

Android applications, either system or third-party app, communicate with the Android platform via defined Application Programming Interfaces (APIs). Android framework provides a list of APIs that a developer can call to extend the functionality of the hardware without direct use of lower layers of the architecture. Such functionalities are managing user interface (UI) elements, accessing shared data storage, and passing messages between application components. As in Linux, the Android app is assigned a unique user id (UID) and group id (GID). Each app runs in a separate process to identify and isolate each app's resources from each other. Using UID, Android creates kernel-level application sandbox to enforce kernel security.

Android application is compressed in an archive format file, like any other known formats such as ZIP and JAR, called Android Application Package (APK). APK contains seven files: asset, lib, meta-info, res, androidmanifest.xml, classes.dex, and resources.arsc. In this section, we limit our discussion on two main files: the manifest file (Androidmanifest.xml) and the code file (classes.dex).

Android manifest. The manifest file is an XML format file that provides beforehand a set of information about the app and declaration of the app components. Information such as the app's package name and version number, permissions required by the application, app entry points, and registered intents.

Dalvik executable (DEx). The file *classes.dex* contains a set of files (bytecodes). Those files are a special type of bytecode called Dalvik Bytecode that are compiled from normal Java classes. In Figure 2.2, we show the steps of converting Java classes and the generation of a DEx file [12].

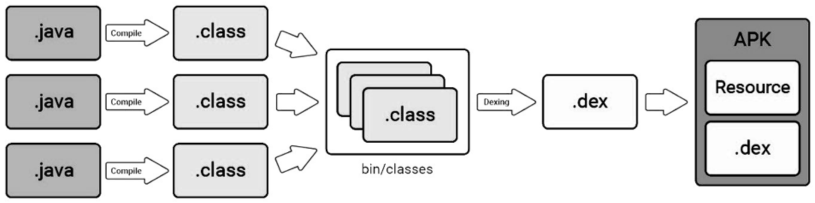


Figure 2.2 Dexing Java classes into classes.dex.

Android Access Control

To protect the system's resources, Android, like in Linux, uses an forced access control mechanism to remove the malware and render it harmless. Android requires application to request permissions prior to utilizing the resource [13]. In the application level, permissions must be declared inside the *AndroidManifest* where essential information on the app and its components are located. Prior to Android version 6, the user was required to grant access to all what app requests at the time of the installation. The risk of this, besides the user's weak knowledge of the requested permissions and what they mean, is that an app can deceive the user by requesting permissions unrelated to the app's main functionality. Malware app can leverage some permissions to gain access to the resources and 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 the 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 the installation in order for access to be granted application [17]. This research examines the permissions that malware families request as features for our static analysis.

2.1.3. Android Malware

In this section, we discuss the most recognized type of malware attacks in the literature such as: repackaged, update attack, and drive-by download as listed in [18]. Furthermore, we discuss the way that malicious payload is executed. Finally, we conclude the section by discussing malware families and characteristics.

A malware could secretly be embedded in a set of deceptive applications and can be identified by detecting the malware’s files or similar malware 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 [19].

Malware families. A family of malware is a group of malware that shares common characteristics and behavior. Adopting an attack or malicious behavior by inserting a payload (or more than one payload) requires using the same package names used for the attack. By frequent use of package names (or other common characteristics), this becomes one identity (signature) of a group of malware (family). For example, AnserverBot family, a popular malware family, uses com.sec.android.provider.drm the package name in the code. Another example is that malware in DroidKungFu family contain a package named com.google.ssearch [18]. The family is identified by a unique name. Some of malware families are show in Table 4.1. Another family is *DroidDream*, also known as *RootCager*, was discovered in 2011 in the official Android market, Google-Play. *DroidDream* family is a Trojan that collects the mobile device's I.D. or serial number and other related information by requesting administrator access control on the device. This Trojan can be detected by locating the two code files *rageagainstthecage* and *exploid* in the family members [18], [20]–[22]. *DroidDream* is one of the advanced and sophisticated types of malware. Other common malware families are listed in [23].

Attack techniques. One of the most common techniques is to *piggyback* a known app with a malicious payload. This technique is known as repackaging as the malicious disassemble an app, insert the malicious code (payload), and repack the app. Examples of such malware families are ADRD, AnserverBot, and BgServ [18]. An alternative way of the same technique is update attack. This is in order to repackage the application when performing updates. A victim installs the modified app, without the payload, to avoid detection. When it is time for the update, a payload will be installed with the new version. Families such as BaseBridge, DroidKungFuUpdate, and Plankton are some examples of families adopting this technique.

Another technique is called *drive-by download*. In this technique, the victim installs an app that advertises another app that is either standalone or repackaged malware. In addition, instead of advertising, the download request can happen without user notification. This could happen when the user grants certain permissions to the app to download when the user first installs the main application. GGTracker, Jifake, Spitmo, and ZitMo are some of the families using this type of attack.

Obfuscation techniques. Obfuscation is a way to make code unclear. Malware (or commercial apps) use this technique to hide their actual code. Some of the obfuscation techniques are: *Renaming method and variable.* When it comes to naming a method and variable, the name should reflect the behavior of the method or variable. In the case of malware and when malware is identified by some methods or variables names as signatures, changing the names helps the malicious developer to bypass the AV.

- *String Encryption.* One technique that AV uses to quickly scan an application is to look for a link, a form of URL or IP address, to a remote server. The server is also called command and control (C&C) from which the application receives commands and run them on the victim's device. For the malware to hide this link, it needs to be encrypted. The encrypted link will be decrypted at the usage moment. The obfuscated strings are hard to reverse engineer and then hard to read. In general, encryption can make the application heavy and very slow to execute. But the technique has been used to encrypt part of the code to avoid detection.
- *Control Flow Obfuscation.* Even when the code is obfuscated, experience developer and malware analyst can draw an understanding of the behavior of the obfuscated code from matching the control flow of the program. To make the code unpredictable, developers obfuscate the control statements as well.
- *Dummy code insertion.* Last technique is to insert codes that do not relate to the actual behavior of the malware. This code is called dummy since it is not used, but to mislead the scanner from reading and understanding the code.

Finally, there are tools used for obfuscation. Common Java obfuscators are ProGuard [24] and DexGuard [25], which are widely used.

Activation techniques. This technique associated with Android events. BOOT_COMPLETED event, for example, is triggered when the device finishes the booting process. Malware uses this event to be notified when the device is up and running to activate the malicious process. Other events such as SMS_RECEIVED that is triggered when an SMS is received is utilized by zSone family. Another example is a ACTION_MAIN event that is triggered when an app's icon that is clicked is adopted by a DroidDream family.

There are many papers contributed to detecting such techniques such as [26]–[30]. For example, Tian et al. [30] designed a repackaged detection technique. Their technique based on partitioning the code into two levels, class-levels dependency graph (regions), and method-level call graphs. They utilize machine-learning to recognize internal behavior using three types of features: permissions, sensitive API calls, and user interaction.

How anti-virus works. Malware signatures, as they have been manually analyzed or detected, are saved in an AV database to be compared against files under scanning. When a match is found in the file, the file (or app) is considered malicious, and it will be quarantined.

2.2. Android Malware Related Work

In this section, we review the survey papers on Android malware. Most of the surveys focus on malware detection, including [31]–[39]. The most recent survey has reviewed papers on malware detection while focusing on their approaches; they discussed the advantages and disadvantages of each detection approaches and methods [32].

The following survey has proposed a taxonomy to categorize Android malware detection techniques; they highlighted the trends and the challenges [34]. The following two survey papers have provided an outline of the methodologies used in classifying malware based on work surveyed [33], [38]. The authors in [35] have focused on the state-of-the-art papers in identifying malware behaviors based on a diverse set of features; they highlighted the effective features in detecting malware. Yan and Yan have surveyed the related work in dynamic malware detection; they focused on the performance evaluation criteria on malware detection [36].

Souri and Hosseini have conducted a systematic survey on the state-of-the-art papers in utilizing data mining techniques in malware detection; they categorize the techniques into signature-based and behavioral-based. Furthermore, they discuss the importance of data mining techniques in malware detection [37]. Riasat et al. have provided a comprehensive survey on the tools and methods used on malware detection; they highlighted the various types of tools used in the research field [39]. Arshad et al. categorize the antimalware and penetration techniques proposed by state-of-the-art research to protect the Android system; they highlighted their limitation and benefits [31].

The previous surveys on malware detection have focused on malware detection. In this survey, our focus is on malware familial classification, detection, and analysis, which will introduce a baseline for future work in this domain.

To conduct our review, we followed an exploratory research approach. We investigated more than a thousand papers published in journals and conferences. To filter out the selected papers, we considered keywords. The following respectable scientific databases are explored: IEEE Xplore [40], ACM Digital Library [41], MDPI [42], ScienceDirect [43], Hindawi [44], Springer [45], and arXiv [46], and we also used reputable literature search engines such as Microsoft Academic [47], Semantic Scholar [48], and Google Scholar [49]. Keyword criteria for selecting a literature contain main and optional keywords. Main keywords are Android malware and malware family. Optional keywords are malware detection, familial classification, malware family identification, and malware family categorization. We have classified the related work according to their type of analysis, techniques, and features.

Our complete taxonomy is shown in Figure 2.3. and Table 2.1. The taxonomy categorizes all the related work in familial classification in terms of the type of analyses, features, and techniques that has been used.

2.3 Analysis

In this section, we discuss the type of analysis followed by the state-of-the-art. They are static, dynamic, and hybrid analysis.

2.3.1. Static Analysis

Static analysis is applied while the app is in a static state. It basically collects information about the app such as the app's name, size, permissions, code, and programing pattern. Some of the information requires reverse engineering the app from machine code to a readable format to analyze the code. The advantage of performing such analysis is that it is fastest and cheapest since it doesn't require executing the application nor does it require monitoring activities. A drawback of the analysis is that many malware launch their attack at runtime. In addition, other malware use an obfuscation technique or encrypted methods which cannot be read or decrypted unless the app is executed. A set of papers [50]–[77] used static analysis. Details on the static features used by the papers were discussed in Section 2.5, Features.

2.3.2. Dynamic Analysis

This type of analysis (also known as behavioral analysis) performed during the execution of an app. It monitors the inside and outside action, connections, calls, and clicks that happen while the app is being executed. Such analysis has the advantage of detecting wide-range and sophisticated malware. Malware families that are bound to an event that were mentioned earlier can only be detected while the app is running. The disadvantage of such analysis is that it is time-consuming. In addition, it requires a priori knowledge of the malware technique to monitor. Several papers have applied dynamic analysis such as [78]–[83]. Details on the dynamic features used by the papers were discussed in Section 2.5, Features.

2.3.3. Hybrid Analysis

Hybrid analysis is a combination of both static and dynamic analysis. Although hybrid analysis has the advantage of covering both analyses, it has a major drawback. Such analysis is a time-consuming process considering the huge number of malware samples to be detected and analyzed. Papers such as [84]–[88] have used hybrid analysis and the details on the features used were discussed in Section 2.5, Features.

2.4. Techniques

In this section, we discuss the techniques used by the state of the art to address the familial malware problem. There are two main techniques used: model-based and analysis-based.

2.4.1. Model-Based

In a model-based technique, a model is created to classify malware into families. There are four main categories of techniques used, which are machine learning, similarity analysis and image processing, and evasion.

Machine learning. The literature use machine learning to classify malware samples into families.

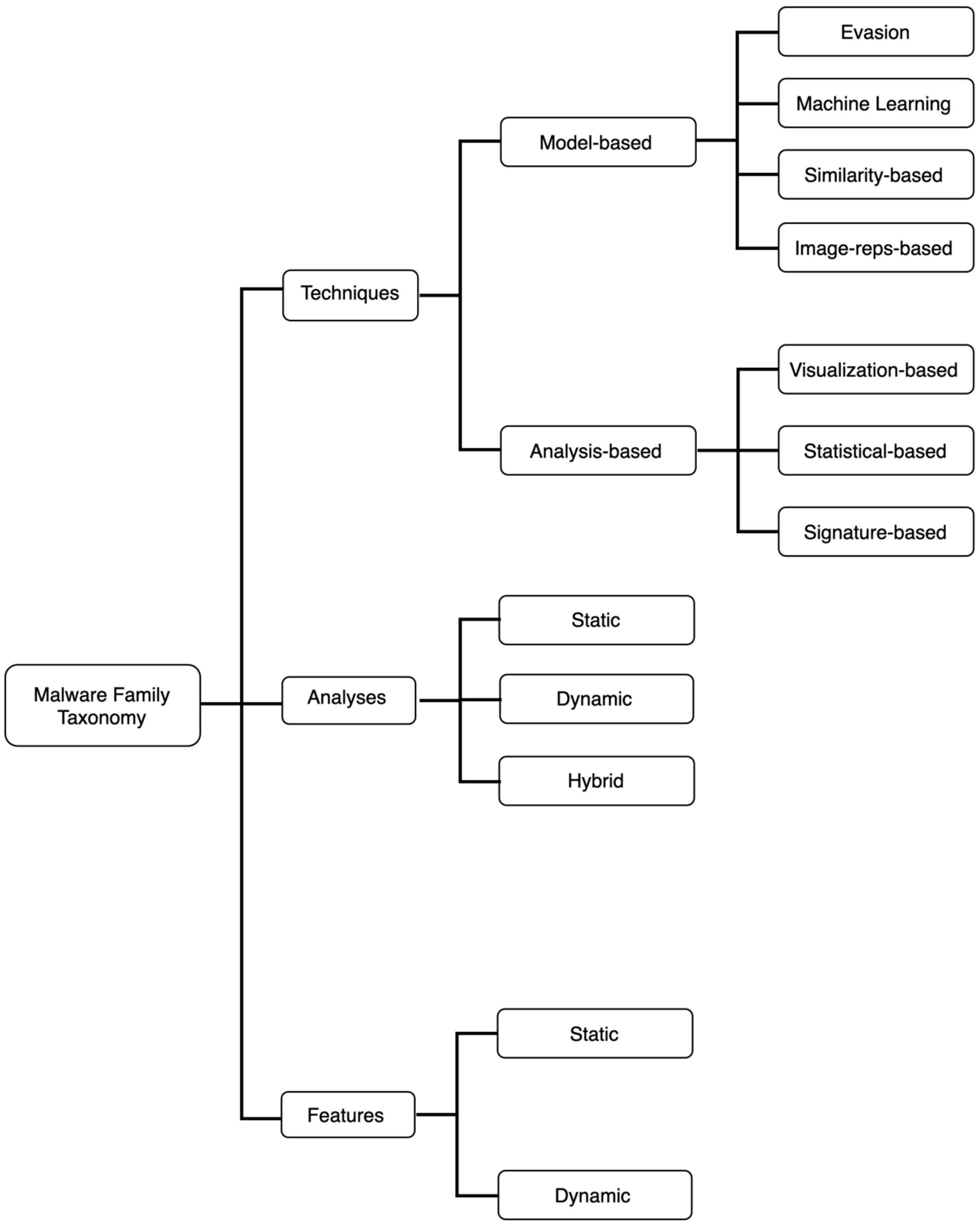


Figure 2.3 Android malware families' taxonomy.

Table 2.1 Taxonomy table of the literature.

Index	Year	Reference	Analysis	Features	Technique
1	2020	Fang et al. [50]	Static	Static	Image-reps-based
2	2019	Qiu et al. [51]	Static	Static	Similarity-based and Machine Learning
3	2019	Zhang et al. [52]	Static	Static	Signature-based and Machine Learning
4	2019	Zhiwu et al. [53]	Static	Static	Visualization-based and Machine Learning
5	2019	Mirzaei et al. [54]	Static	Static	Visualization-based
6	2019	Vega et al. [55]	Static	Static	Visualization-based
7	2019	Vega et al. [56]	Static	Static	Visualization-based
8	2019	Jiang et al. [57]	Static	Static	Machine Learning
9	2019	Fasano et al. [58]	Static	Static	Machine Learning
10	2019	Blanc et al. [84]	Static	Static	Machine Learning
11	2019	Xie et al. [60]	Static	Static	Statistical-based and Machine Learning
12	2019	Turker et al. [61]	Static	Static	Statistical-based and Machine Learning
13	2018	Atzeni et al. [84]	Hybrid	Dynamic and Static	Signature-based
14	2018	Kim et al. [85]	Hybrid	Dynamic and Static	Visualization-based and Machine Learning
15	2018	Fan et al. [62]	Static	Static	Visualization-based and Machine Learning
16	2018	Sun et al. [78]	Dynamic	Dynamic	Visualization-based
17	2018	Martin et al. [79]	Dynamic	Dynamic	Machine Learning and Statistical-based
18	2018	Aktas et al. [86]	Hybrid	Dynamic and Static	Machine Learning
19	2018	Garcia et al. [63]	Static	Static	Machine Learning

Index	Year	Reference	Analysis	Features	Technique
20	2018	Calleja et al. [64]	Static	Static	Evasion and Machine Learning
21	2018	Alswaina et al. [65]	Static	Static	Machine Learning
22	2017	Massarelli et al. [80]	Dynamic	Dynamic	Signature-based and Machine Learning
23	2017	Zhou et al. [66]	Static	Static	Visualization-based and Similarity-based
24	2017	Chakraborty et al. [87]	Hybrid	Dynamic and Static	Machine Learning
25	2017	Sedano et al. [67]	Static	Static	Statistical-based
26	2016	Battista et al. [71]	Static	Static	Signature-based
27	2016	Hsiao et al. [81]	Dynamic	Dynamic	Visualization-based
28	2016	Gonzale et al. [68]	Static	Static	Visualization-based
29	2016	Fan et al. [69]	Static	Static	Visualization-based and Machine Learning
30	2016	Kang et al. [70]	Static	Static	Similarity-based
31	2016	Malik et al. [82]	Dynamic	Dynamic	Statistical-based
32	2016	Sedano et al. [72]	Static	Static	Statistical-based
33	2016	Feng et al. [88]	Hybrid	Dynamic and Static	Visualization-based, Machine Learning, and Signature-base
34	2015	Aresu et al. [83]	Dynamic	Dynamic	Signature-based and Similarity-based
35	2015	Lee et al. [73]	Static	Static	Signature-based and Similarity-based
36	2015	Li et al. [74]	Static	Static	Visualization-based and Machine Learning
37	2015	Garcia et al. [89]	Static	Static	Machine Learning
38	2014	Deshotels et al. [75]	Static	Static	Visualization-based and Similarity-based
39	2014	Suarez et al. [76]	Static	Static	Statistical-based and Machine Learning

Index	Year	Reference	Analysis	Features	Technique
40	2013	Kang et al. [77]	Static	Static	Statistical-based and Machine Learning

In [53], the authors classify the malware using Deep Learning (DL) techniques. In [79], the authors classify malware into families using classical machine learning such as Support Vector Machine (SVM) and DL algorithms such as CNN and RNN. In [76], the authors use a Nearest Neighbor classifier (NN) to classify malware into families. In [57], the authors preprocess the data and extract the sensitive opcode sequence. For the minor families, they use the oversampling technique to overcome this issue. To represent the semantic features of the sensitive opcode sequence, they use text mining (i.e., Doc2Vec algorithm [90]). Finally, they train their model using nine machine learning algorithms such as SVM and Randomforest. In [80], the authors feed the fingerprint to an SVM algorithm to classify malware into families. In [74], the authors construct the feature vector and feed it to several machine learning algorithms such as Randomforest. In [60], the authors used SVM to classify the samples into families. In [77], the authors feed the features to several machine learning classifiers such as Decision Tree and Association rules. In [64], the authors build a framework to train the classifier algorithm with a set of samples to drive the heuristic search using a Genetic algorithm. In [62], [69], the authors use frequency graphs (FreGraph) as their features to be fed into several machine learning algorithms such as SVM, Decision Tree, and Randomforest to classify the malware into families. In [59], the authors feed the Android-oriented matrices to several machine learning algorithms such as SVM, KNN, and Decision Tree. In [63], the authors apply machine learning algorithms to extract complex features and used them to classify malware into families. In [61], the authors use three machine learning techniques: standard classifier such as SVM, ensemble classifier, and Neural Network to classify malware into families. In [65], Alswaina et al. use two models to perform familial classification. The authors use the binary representation of the features and weighted importance. Then, they use six machine learning algorithms to predict malware families. In [86], the authors apply three filters to filter the features. The dynamic and static features are combined and fed to machine learning algorithms, such as Randomforest and KNN for classification. In [51], the authors apply Linear SVM, DT, and DL algorithms. Fene et al. [88] utilize the SVM algorithm.

In [87], the authors use supervised algorithms such as Randomforest. Moreover, the authors use unsupervised learning such as K-means and mean-shift due to unbalanced samples in each family. They also propose ensemble clustering and classification techniques, which integrate the results generated from the supervised and unsupervised algorithms. In [85], the authors optimize the weight of features using community detection algorithms. They further classify the malware into families using machine learning. In [52], the authors use the fingerprints to classify malware into families using online passive-aggressive (PA) classifiers. Further details of PA can be found in [91]. In [58], the authors extract features from the apps and create code metrics. Then, they binary classify (coarse-grain) the samples. The malware is further classified into families (fine-grain).

Evading detection. In this technique, the goal is to evade detection or elude classifiers into misclassification. In [64], the authors build a framework to alter the malware to perform an attack and misclassify the results.

Similarity analysis. Literature computes the distance between any malware and the family.

In [83], the authors use the token-subsequence algorithm to extract and generate signatures from each family based on network traffic analysis. In [70], the authors represent opcode as a vector of binary and frequency to compute the similarity between the malware and families. In [66], the authors evaluate their approach by performing similarity analysis. In [75], the authors perform two tests. The first test is used to binary classify malware. In the second test, they apply the agglomerative clustering algorithm to cluster the apps into families. To evaluate their model, they compute the distance between the malware and the clusters' centroids to validate which family the sample belongs to. In [73], the authors cluster the families based on the most frequent key terms used by each family. Then, they use the dictionary search method for classification. In [51], the authors use TF-IDF to represent the frequency of the features.

Image representation. Some literature classifies the malware to malware families based on image representation. In [50], the authors convert the DEX file into an image and plain text. Then, they extract the color and the texture feature from the image. For the three features: color, texture, and text, they feed them into the feature Fusion algorithm to classify malware into families.

2.4.2. Analysis-Based

In the analysis-based technique, an analysis is carried to analyze and construct features to observe families' characteristics. There are three sub techniques under this approach, which are signature-based, statistical analysis, and visualization analysis.

Signature-based. They construct a signature for each family to identify the families. In [83], the authors use a multi-step clustering approach: First, they apply coarse-grained clustering and then apply fine-grained clustering. In [52], the authors construct the fingerprint of the malware families using n-grams analysis and features hashing. In [80], the authors generate a fingerprint for each family. In [73], the authors construct a signature of each malware family based on the collected features. Feng et al. [88] propose an approximate signature matching algorithm to generate signature for malware families.

Statistical analysis. They applied statistical tools to analyze and identify the family's characteristics and the important features. In [76], the authors use statistical analysis and text mining to extract the features. In [79], the authors use Markov chain to represent the features. In [60], the authors eliminate unimportant features using the frequency-based approach. In [77], the authors compute the bytecode frequency. In [61], the authors apply a feature ranking algorithm to identify the most important features.

Visualization analysis. They visualize the characteristics of families using graph mining and PCA. In [53], the authors extract DFG and CFG. Then, they encode the graphs into a matrix. In [85], the authors represent the features using a network graph. In [66], the authors collect the sensitive API calls and then construct graphs based on sensitive API calls. Then, they characterize malware families based on the subgraph isomorphism. In [74], the authors construct a short and long APIs dependency path to perform context and constant analysis. In [75], the authors disassemble the app into Smali files. Then, they create class dependency graph (CDG) to group the classes into modules to identify which module contains malicious code. In [62], [69], the authors use community detection, subgraph matching, and subgraph clustering to generate the FreGraph. Feng et al. [88] utilize an inter-component call graph (ICCG) to represent the communication in the app to construct the features.

2.5. Features

In this section, we discuss the types of features used by works of literature to classify malware into families. They are classified into static and dynamic features.

2.5.1. Static Features

Static features are any features that can be recognized or utilized without the execution of the application. Some examples of static features are package name, application size, permissions, and list of APIs.

A set of papers [55], [56], [65], [67], [68] uses features that are related to malware installation such as repackage and update, payload activation such as on booting and receiving calls, and privilege escalation attack such as asroot and exploid families [18]. Moreover, in [55], [56], [67], they include other features related to financial charges such as SMS and phone calls. Vega et al. in [55], [56], also include features related to personal information stealing such as phone number.

In [51], [54], [57], [60]–[64], [66], [69], [72], [74], [75], [88], [89], a subset of sensitive or suspicious API calls are utilized in their feature set. Permissions used in the app are included as features in [51], [60], [61], [65], [72]. Moreover, in [57], sensitive opcode sequence, actions, and strings are utilized in their features. Garcia et al. [63], [89] added native code-based to their set of features.

Fasano et al. [58] and Blanc et al. [59] use a set of metrics generated from Smali files to measure the quality code of the app to be used as features. However, in [57], [70], [71], [73], [77], code-based analysis such as Java bytecode, bytecode frequency, opcode, or opcode sequence are used as features.

Other papers such as [53], [76] use data-flow graph (DFG) and control-flow graph (CFG) as features. In [50], the authors extract the texture, color, and text features from the DEX file. Zhang et al. [52] use features extracted from DEX as n-gram and hash code.

Finally, some works of literature have applied a set of static features in addition to dynamic features. In [87], the authors use 190 static features such as permissions. In [86], the authors use static features such as the number of services and receivers. In [85], the static features such as permissions, filename, and activity name are utilized. In the paper [84], a set of static features from the Android manifest in addition to an APK file that is generated from Androguard tool [92], a Python code to reverse engineer Android files.

2.5.2. Dynamic Features

Features that require execution of the application are considered dynamic. For example, network traffic, send/receive SMS, resource consumption, system logs, and I\O operations.

In [82], the author traced the system calls during the execution of the application. Aresu et al. [83] utilize network traffic (HTTP) in their classification. Martin et al. [79] depend on the features that are generated by a DroidBox [93] tool, an Android sandbox for dynamic analysis, which is represented as operations and function of time. In [81], the authors record the API calls that are performed during application execution. In [80], resources' consumption is utilized as features for their classification. In [78], the authors use sensitive and permission-related API calls.

Finally, a group of literature works has applied a set of dynamic features in addition to static features. In [87], the authors use around 2048 dynamic features logs such as file I/O, network usages, and cryptographic usage. In [86], the authors use dynamic features that are generated using a DroidBox tool [93] such as the number of open/closed connections and the number of sent/received network packets. In [85], the dynamic features such as API call sequence are utilized. In [84], a set of dynamic features uses DroidBox [93] and CuckooDroid [94]. Feng et al. [88] use suspicious API call behaviors such as sendSMS API and data leakage.

2.6. Discussion

In this section, we highlight the datasets that have been used, the limitation of literature, the general challenges related to malware families, and we also report future directions.

2.6.1. Experimental Datasets

There are many datasets used in the literature that contain a collection of Android malware grouped into families such as: Android Malware Genome Project (Malgenome) [18], Drebin [95], the AMD [96] Project, and AndroZoo [97]. Some papers collected the malware samples from the Android market such as Anzhi, or a repository such as VirusTotal [98] and VirusShare [99].

The datasets differ in the number of samples and number of families. For example, AMD [96] contains 4354 malware samples grouped in 42 families. While Drebin [95] has 5560 samples grouped in 179 families, other datasets such as AndroZoo [97] contain many more samples and families, where the number of samples is 10.7 million grouped into more than 3000 families. In Figure 2.4, we show the number of publications that uses each dataset found in the literature. Furthermore, Table 2.2 shows detailed information where the publications are included. The repository category includes sites like VirusTotal, VirusShare, and Koodous, for which there is no fixed set to be used as benchmarks. Collection category refers to either an unknown collection performed by the author or sites such as HelDroid, FalDroid, and the Anzhi app market. As we see from Figure 2.4, the most used datasets are Drebin [95] and Genome [18]. More details on the commonly used datasets are reported in Table 2.3.

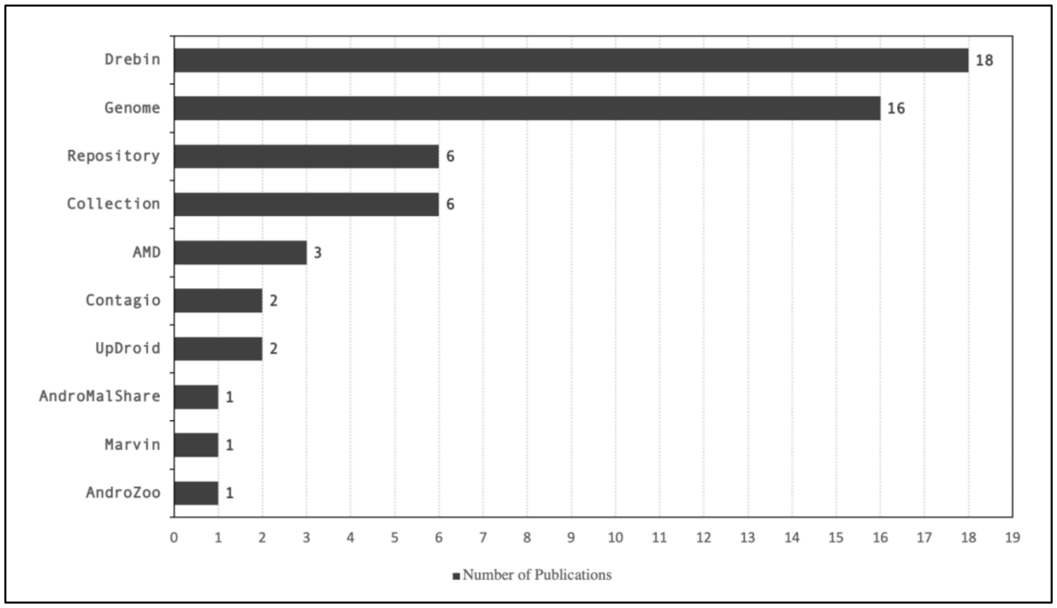


Figure 2.4 Dataset reported by publications, where the x-axis represents the number of publications.

Table 2.2 Dataset, number of publications, and publication details.

Dataset	No. of Pubs.	Publications
Drebin [95]	18	[51]–[53], [57]–[59], [61]–[64], [71], [72], [78]–[80], [83], [87], [88]
Genome [18]	16	[52], [55], [56], [63], [65]–[68], [70], [74]–[77], [81], [83], [88]
Collection	6	[58], [60], [62], [73], [82], [85]
Repository	6	[53], [60], [63], [69], [84], [87]
AMD [96]	3	[50], [51], [61]
UpDroid [86]	2	[61], [86]
Contagio [100]	2	[53], [83]
AndroZoo [101]	1	[54]
Marvin [102]	1	[53]
AndroMalShare [103]	1	[66]

Table 2.3 Commonly used datasets and their details.

Dataset	Total Samples	Number of Families
Drebin [95]	5,560	179
Genome [18]	1,260	49
AMD [96]	4,354	42
AndroZoo [101]	10.7M	3K+

2.6.2. Limitations

As we surveyed forty research papers, we summarize the limitations to the following: First, most of the literature uses small datasets such as a few numbers of families or a few malware samples for studying families. Moreover, they use outdated or discontinues datasets such as Contagiodump (Contagio) [100] and Malgenome Project (Genome) [18]. In addition, several papers build their experiments on manually collected data without testing their model on benchmarked data. Several papers lack the disclosure of the list of features applied to reproduce the work.

2.6.3. Challenges

Family naming. One of the challenges that we observe is that there are no naming schemes (conventions) for the malware family. Naming a family is varied depending on the AV company. Families such as BaseBridge (or adSMS), Smssend (or fakeplayer), and DroidDream (or DORDRAE) are some of the families that have multiple names. One of the reasons is that one company names a family based on different share characteristics than other companies. Characteristics such as installation methods, activation, or the name of their malicious file name are discussed in [18], [104], [105]. Attempts have been made by [101], [106], [107] to establish naming standards. Sebastian et al. [108] address the issue of inconsistent labeling (naming) of malware family and contribute the AVclass tool, an auto-labeling, as an effort to unify labeling. In addition, Euphony is a system proposed by [101] to unify different AV companies.

Imbalance Dataset. Some of the malware families contain hundreds of samples, while others contain as little as one sample such as DroidKungFuUpdate family in the Genome [18]. Other families with one sample in Genome dataset [18] are listed in Table 2.4. The rest of families in the dataset are shown in Table 4.1. This cause identifies the characteristics of a family as challenging. In case of standalone malware (not repackaged), the identification is almost impossible.

Table 2.4 Genome dataset's families with one sample.

Family
SMSReplicator
Walkinwat
Endofday
GGTracker
GamblerSMS
Lovetrap
Zitmo
CoinPirate
DogWars
NickyBot
DroidCoupon
DroidDeluxe
Spitmo
DroidKungFuUpdate
FakeNetflix
Jifake

2.6.4. Directions

Advanced machine learning. Malware families should be deeply analyzed and identified. Deep learning technology has been adapted to address various research problems including voice recognition, image processing, and text analysis. One of the advanced techniques of Deep Learning is reinforcement learning, which can be utilized to better understand the families' characteristics. Reinforcement learning has shown very promising results, especially in dynamic analysis. Another technique that should be adopted is transferred learning, which can be utilized to address the lack of samples in families.

Big data handling. Since the amount of malware is increasing exponentially, as it was reported by GDATA that almost 9K of new malware programs are reported daily [109], a scalable solution should be considered. For example, the AndroZoo [97] dataset has millions of samples that can be handled using big data technologies. One of the most important tools are Hadoop [110] and Spark [111]. They can handle a huge amount of malware data with fast processing.

Crowdsourcing. Beside Big data technologies, a group of malware family analyzers can be utilized to better identify and characterize the families. For example, a source can use a subset of features, while other sources investigate other feature sets. A malware repository VirusTotal [98] and VirusShare [99] are some examples.

Automated detection. The huge number of generated malware necessitate the call for automated analysis and classification of malware family rather than performing such tasks manually [85], [112].

CHAPTER 3: RESEARCH PLAN

The research developed a Reverse Engineering framework (RevEng). Within RevEng, applications' permissions were selected and then fed into machine learning algorithms (MLA). Through our research, we created a reduced set of permissions using Extremely Randomized Trees that achieved high accuracy and a shorter execution time. Furthermore, we conducted two approaches based on the extracted information. The first approach used a binary value representation of the permissions. The second approach used the features' importance. We represented each selected permission in Approach One by its weight value instead of its binary value. We conducted a comparison between the results of our two approaches and other relevant works. Our approaches achieved better results in both accuracy and time performance with a reduced number of permissions.

3.1 Proposed Framework

RevEng consists of four main components that include the Dataset, Family, App, and Analysis components. Each component parses and collects information on the dataset. 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 RevEng framework and their interactions to classify malware app and predict malware family. The following is a general flow of the framework. More details are added in the following section.

1) The Dataset is needed to parse and maintain information about the malware families in the dataset. The component takes the dataset 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 and constructs 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 keeps a list of all malware apps in the family. The component maintains and removes any duplicate of an application using the hash value of the malware. 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.

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

3.2 Framework Components

Dataset: This component contains general information about the dataset such as FamiliesList (a list of families in the dataset), 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, 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 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 dataset), 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 M_{Bcand} and M_{Wcand} that are produced by the Dataset component as inputs to each model to classify each input into its predicted malware family. In this component, we take advantage of Scikit-Learn libraries [113] to implement machine learning algorithms. Figure 3.1 demonstrates the *Dataset*, *Family*, and *App* components' data structure with pseudo-code.

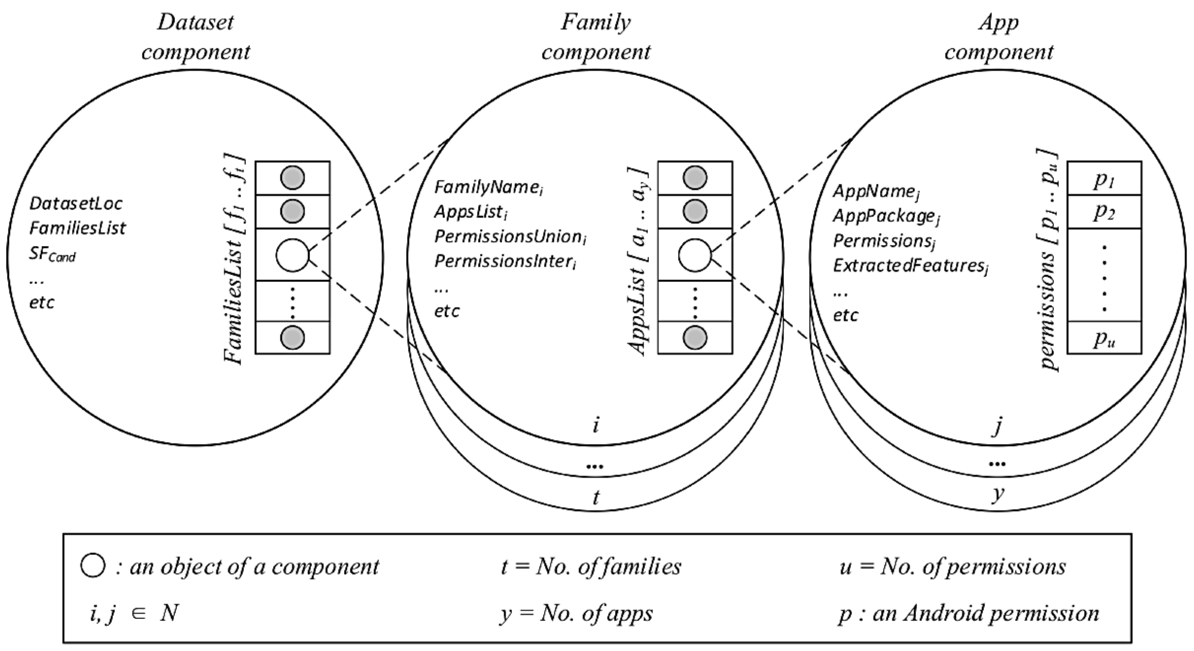


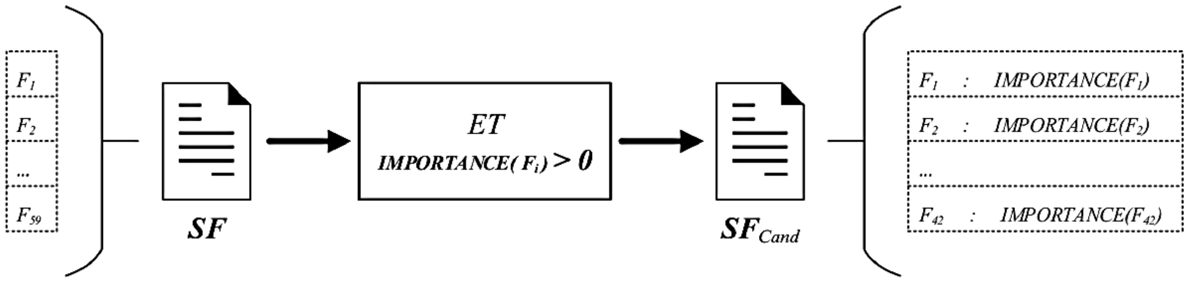
Figure 3.1 Demonstrates the contents of components and the relation between them.

3.3 Application's Features

The features used in this research is an 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 the permissions provided by an Android operating system. To accomplish this, one of the ensemble classifiers, called Extremely Randomized Trees (ET), [114] was utilized. ET, like Random Forest (RF) [115], 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 [116].

Features Reduction: The SF here is the permissions feature used in StormDroid [7]. 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 add significance to uniquely classify a malware family (noise) or have some dependency between features. By collecting all features greater than zero, we have a Candidate Selected Features set (SF_{Cand}), a reduced set of features as shown in Figure 3.2. The ultimate SF_{Cand} contains 42 out of 59 permissions. The SF_{Cand} chosen, with their importance, are included in Table 5.6. The top 10 permissions with high importance are shown in Figure 3.3.

Our analysis of the dataset shows that a group of permissions is 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; *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 [18] and [17]. For this reason, these permissions are not critical 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 5.6.



F_1
F_2
...
F_{59}
SF
ET
$IMPORTANCE(F_i) > 0$
SF_{Cand}
F_1 : $IMPORTANCE(F_1)$
F_2 : $IMPORTANCE(F_2)$
...
F_{42} : $IMPORTANCE(F_{42})$

Figure 3.2 Extraction of the important features from SF and generating SF_{Cand}.

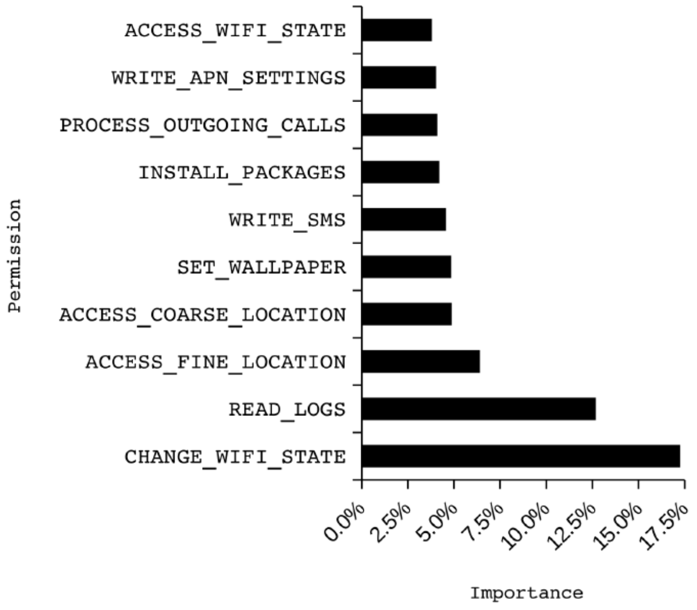


Figure 3.3 Top 10 permissions based on their importance.

3.4 Data Preprocessing

Upon beginning the execution of the framework, the Dataset component is initialized (Dataset.Init) with the dataset. Once the component is ready, RevEng starts loading and parsing the dataset by executing Dataset.Load. To start creating the families' objects, RevEng forks several threads (NoOfThreads) assigned in the initialization during the execution of Dataset.Run as illustrated in Figure 3.4.

```
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 $family_i$ in dataset.families:
      IF $family_i$.Init(FL, SL=4) THEN
        start $thread_i$ from ThreadPool(NT)
          $thread_i$ exec $family_i$.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 $app_i$ in family.apps:
      $app_i$.Init(AL)
      $app_i$.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 $item_i$ in SF:
      IF item in app.PRs THEN
        $ExF_i = 1$
      ELSE
        $ExF_i = 0$
```

Figure 3.4 Pseudo-code of the *Dataset*, *Family*, and *App* components shows the main parts of the code.

Multi-threading utilizes the processor and increases the reverse engineering process of the applications as illustrated in Figure 3.5. 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 Figure 3.4. Family.Parse initializes the App component (App.init) and parses it (App.Parse).

The App.Parse method, in turn, extracts information from the application such as the package name, all permissions in the manifest file, and 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

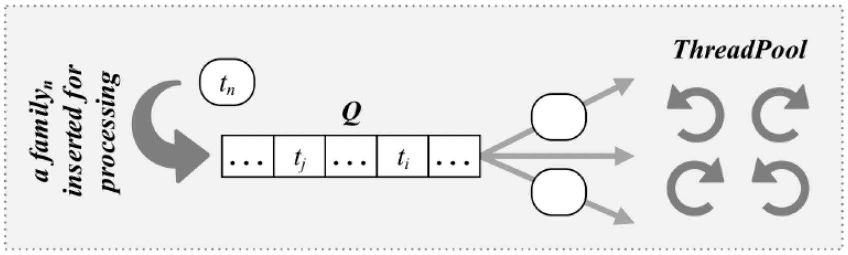


Figure 3.5 Multi-threading processes of the list of tasks in queue.

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, and 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 $\rho \mid \rho \in SF$ in index i, then ExFA(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 Figure 3.6.

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

The Dataset joins all the ExFs in M_{Bcand} for analysis as illustrated in Figure 3.6. 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 Figure 3.2.

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

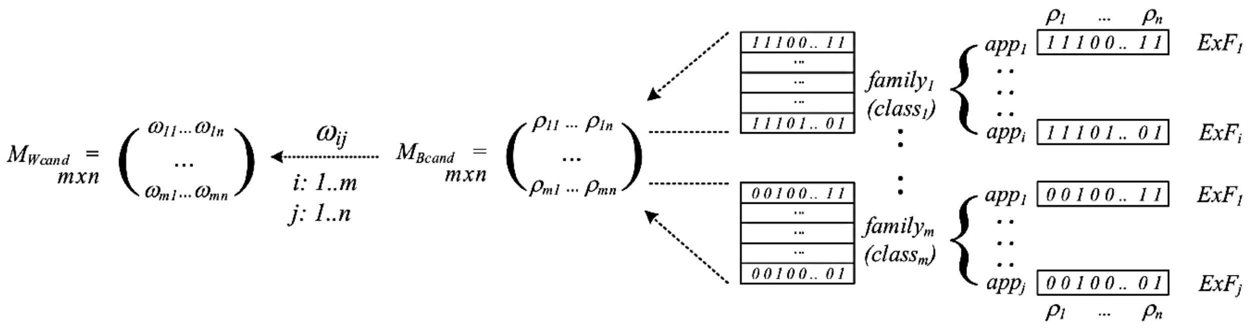


Figure 3.6 Preparation of M_{Bcand} and M_{Wcan} matrices for processing.

To generate the Weighted Candidate Matrix M_{Wcand}, each element is calculated as in (1). Each ρ_{ij} value in M_{Bcand} is multiplied by the permissions' importance as generated by ET for the permission's index j. The Y matrix contains the malware families (classes: c_i) of each malware sample at row i in both M_{Bcand} and M_{Wcand}. M_{Bcand} and Y matrices are shown below:

$$\left(\begin{array}{ccccc|c} p_{11} & \cdots & p_{1j} & \cdots & p_{1n} & c_1 \\ :: & \cdots & :: & \cdots & :: & : \\ p_{i1} & \cdots & p_{ij} & \cdots & p_{in} & c_i \\ :: & \cdots & :: & \cdots & :: & : \\ p_{m1} & \cdots & p_{mj} & \cdots & p_{mn} & c_m \end{array}\right)$$
$$\underbrace{\qquad\qquad\qquad\qquad}_{M_{Bcand}}\ \underbrace{\quad}_{Y}$$

The overall framework is shown in Figure 3.7.

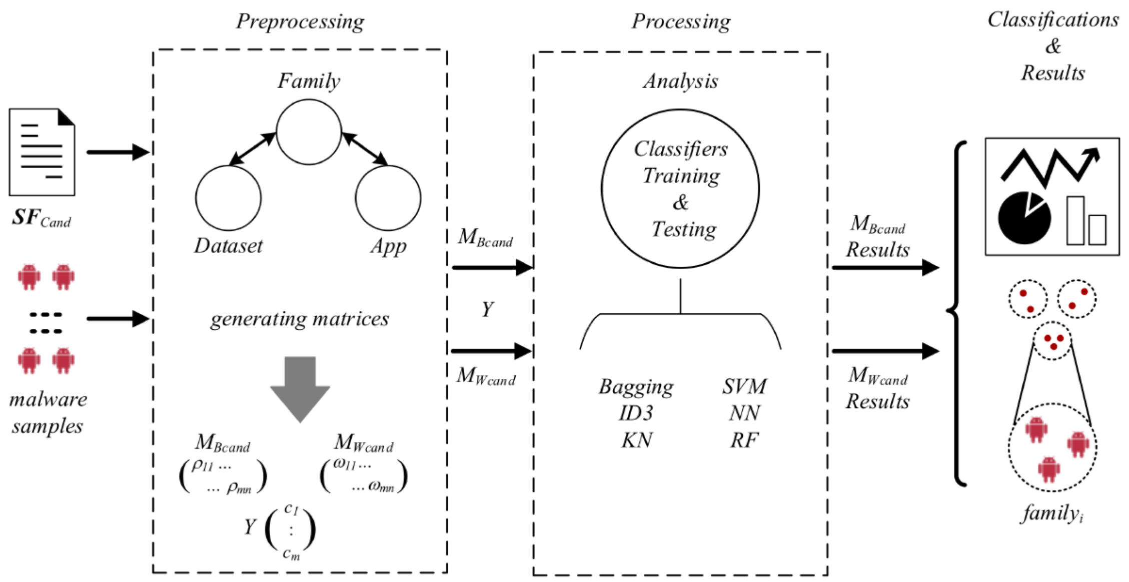

Preprocessing
Processing
Classifications & Results
Family
Dataset
App
SF_{Cand}
malware samples
generating matrices
M_{Bcand}
M_{Wcand}
$\begin{pmatrix} \rho_{11} \dots \\ \dots \rho_{mn} \end{pmatrix}$
$\begin{pmatrix} \omega_{11} \dots \\ \dots \omega_{mn} \end{pmatrix}$
$Y \begin{pmatrix} c_1 \\ : \\ c_m \end{pmatrix}$
M_{Bcand}
Y
M_{Wcand}
Analysis
Classifiers Training & Testing
Bagging
ID3
KN
SVM
NN
RF
M_{Bcand} Results
M_{Wcand} Results
$family_i$

Figure 3.7 RevEng framework.

CHAPTER 4: EXPERIMENTAL SETUP

In this chapter, we show some details on the Genome dataset that we use in our research. Then will show the implementation and the evaluation of the approaches.

4.1 Dataset

We relied on the Genome dataset that was provided by [17]. This dataset 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 4.1, lists the malware families and their samples used in our experiments, for a total of 1,233 applications in 28 families.

4.2 Implementation

The programming language Python was used in all 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 [113].

The Analysis component contains the following classifiers: Support Vector Machine (SVM), Decision Tree (ID3), Random Forest (RF), Neural Network (NN), K-Nearest Neighbor (KN), and Bagging, as implemented by [113].

Table 4.1 List of malware families with their samples in the dataset Genome used in RevEng.

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	DroidDearmLight	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**		

4.3 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 M_{Bcand} and M_{Wcand} to be processed. 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.

Training-testing split: To address the imbalance data issue, we split the data into two sets, training, and testing set. We further apply random oversampling technique to the training set. We evaluate the model on the testing set.

$$Accuracy = (TP + TN)/(TP + TN + FP + FN) \tag{2}$$

$$\begin{aligned}
&Let\ \ S = a\,malware\,sample\ \ and\,C = a\,malware\,family\,or\,class,\ then\\
&\qquad TP \Rightarrow prediction : S \in C\ \wedge\ actual\text{-}classification : S \in C\\
&\qquad FP \Rightarrow prediction : S \in C\ \wedge\ actual\text{-}classification : S \notin C\\
&\qquad TN \Rightarrow prediction : S \notin C\ \wedge\ actual\text{-}classification : S \notin C\\
&\qquad FN \Rightarrow prediction : S \notin C\ \wedge\ actual\text{-}classification : S \in C
\end{aligned}$$

CHAPTER 5: RESULTS AND DISCUSSIONS

In this chapter we show the results of the classification in the two approaches. We conducted two experiments: with and without oversampling the data.

5.1 Experiment 1: Familial Classification without Oversampling

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 average accuracy, and the average execution time. Table 5.1 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.

Table 5.1 Detailed performance (prediction accuracy and time) for each classifier (without oversampling).

	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

The results show that using M_{Bcand}, RF, KN, and NN achieve high accuracy (average ≈ 95.68%) and standard deviation ≈ 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 the M_{Wcand}, the results have higher variations than the previous approach. The top three classifiers are RF, KN, and ID3 (average ≈ 94.66% and standard deviation ≈ 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 set of features (59 permissions) [7] as shown in Table 5.2 and Figure 5.1. We found that the RF classifier produced the highest accuracy of 95.97% versus the other classifiers. RF also completed in 0.08 seconds.

Table 5.2 Average of the classifiers' accuracies and time performance for our 100 experiments (without oversampling).

	$\mathbf{M_{Bcand}}$		$\mathbf{M_{Wcand}}$		**StormDroid**	
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**

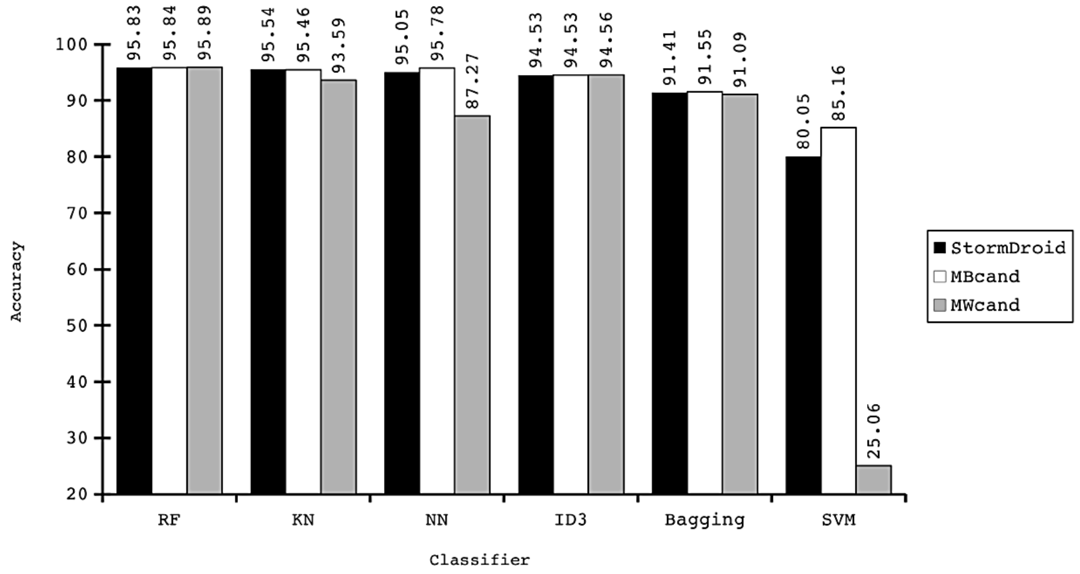


Figure 5.1 Comparison between StormDroid and our approach based on classifiers' accuracies (without oversampling).

In Table 5.3, we summarized our comparison based on two categories: 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%. In the time performance, we could see that ID3 using M_{Bcand} performed faster, although the classifier had the exact same accuracy as in the related work [7]. The chart for Table 5.3 is show in Figure 5.2.

Table 5.3 Comparison between classifiers in terms of the best accuracy and best time performance (without oversampling).

	M_{Bcand}		M_{Wcand}		StormDroid	
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

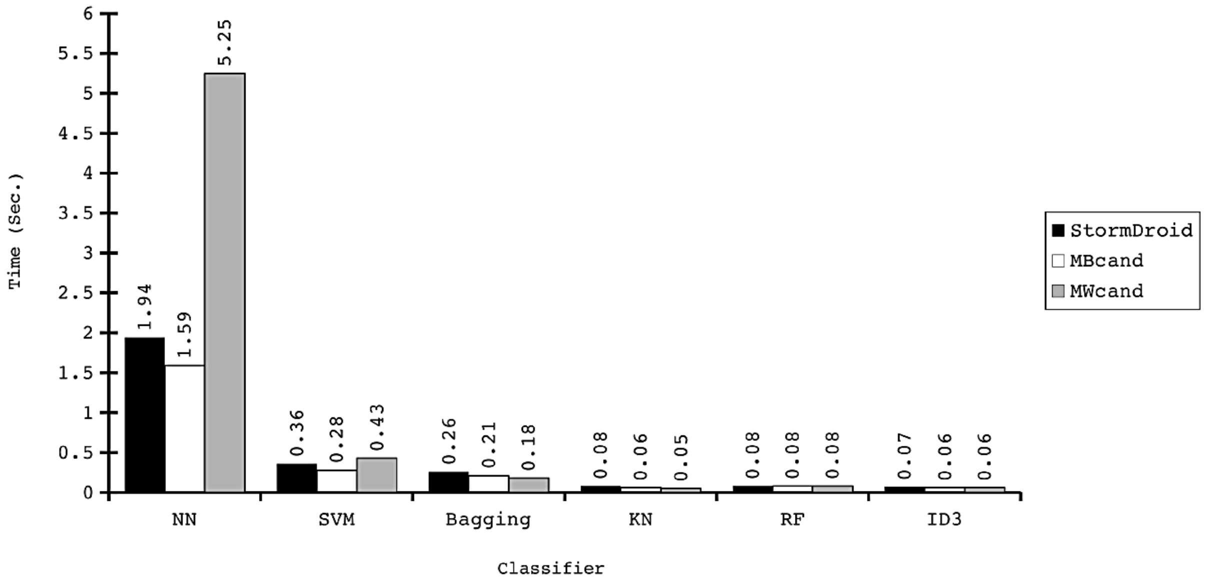


Figure 5.2 Comparison between StormDroid and our approach based on time performance (without oversampling).

From our previous discussion, we concluded that M_{Wcand} achieved 0.02% better accuracy than StormDroid [7] with exactly equal execution time. The accuracy of 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, we improved the accuracy. Moreover, when using M_{Wcand} with KN, we achieved less execution time than the related work [7] with 37.5% improvement as shown in Figure 5.2. A sample of RF's confusion matrix on one iteration is presented in Figure 5.1.

5.2 Experiment 2: Familial Classification with Oversampling

To address the imbalanced data issue, we have deployed random sampling on the training set. Random sampling works as follows: it duplicates the data samples in the minority classes such as GingerMaster family (with 4 samples) to be equal in size to the majority class such as DroidKungFu3 family (with 309 samples); Thus, random sampling algorithm will oversample GingerMaster class from 4 to 309 samples. We recomputed the accuracy and time reported in Section 5.1 as in Table 5.4 and Figure 5.3. The comparison is shown in Table 5.5 and Figure 5.4.

Table 5.4 Average of the classifiers' accuracies and time performance for our 100 experiments (with oversampling).

	M_{Bcand}		**M_{Wcand}**		**StormDroid**	
Classifier	Accuracy	Time	Accuracy	Time	Accuracy	Time
SVM	96.68	0.76	86.54	0.78	96.36	0.94
NN	96.48	3.89	88.95	12.95	96.35	4.59
ID3	95.88	0.1	95.84	0.12	95.85	0.12
KN	96.03	0.24	94.28	0.24	95.97	0.29
Bagging	93.25	0.72	92.89	0.66	93.22	0.88
RF	**97.35**	**2.24**	**97.39**	**2.26**	**97.26**	**2.29**

Table 5.5 Comparison between classifiers in terms of the best accuracy and best time performance (with oversampling).

	M_{Bcand}		**M_{Wcand}**		**StormDroid**	
Best	Accuracy	Time	Accuracy	Time	Accuracy	Time
Accuracy	97.35 (RF)	2.24	**97.39 (RF)**	**2.26**	97.26 (RF)	2.29
Time	**95.88 (ID3)**	**0.1**	95.84 (ID3)	0.12	95.85 (ID3)	0.12

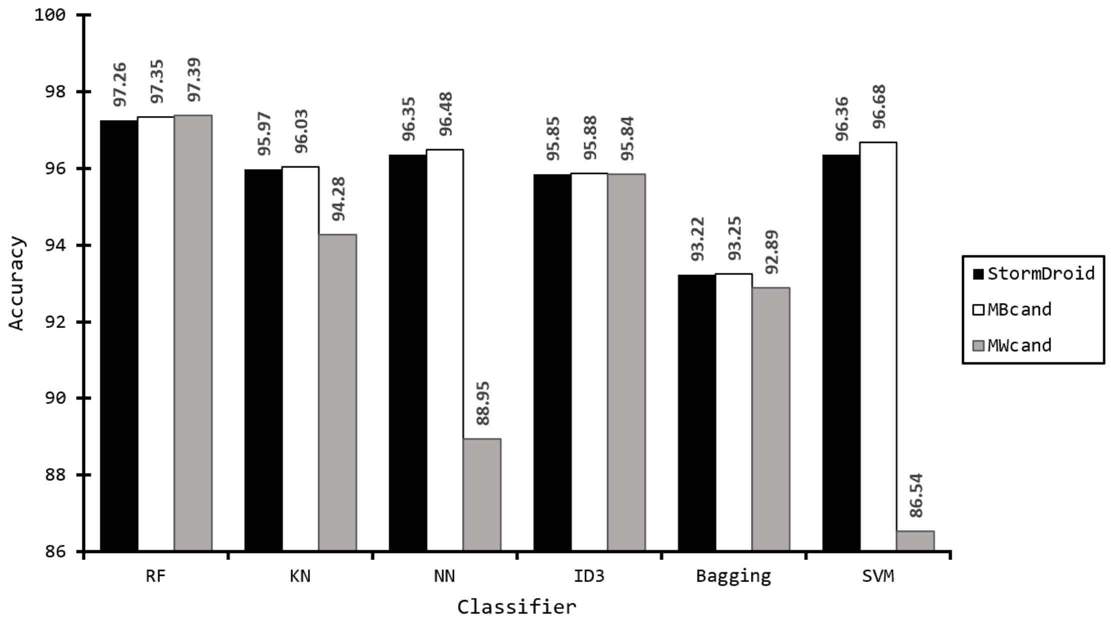


Figure 5.3 Comparison between StormDroid and our approach based on classifiers' accuracies (with oversampling).

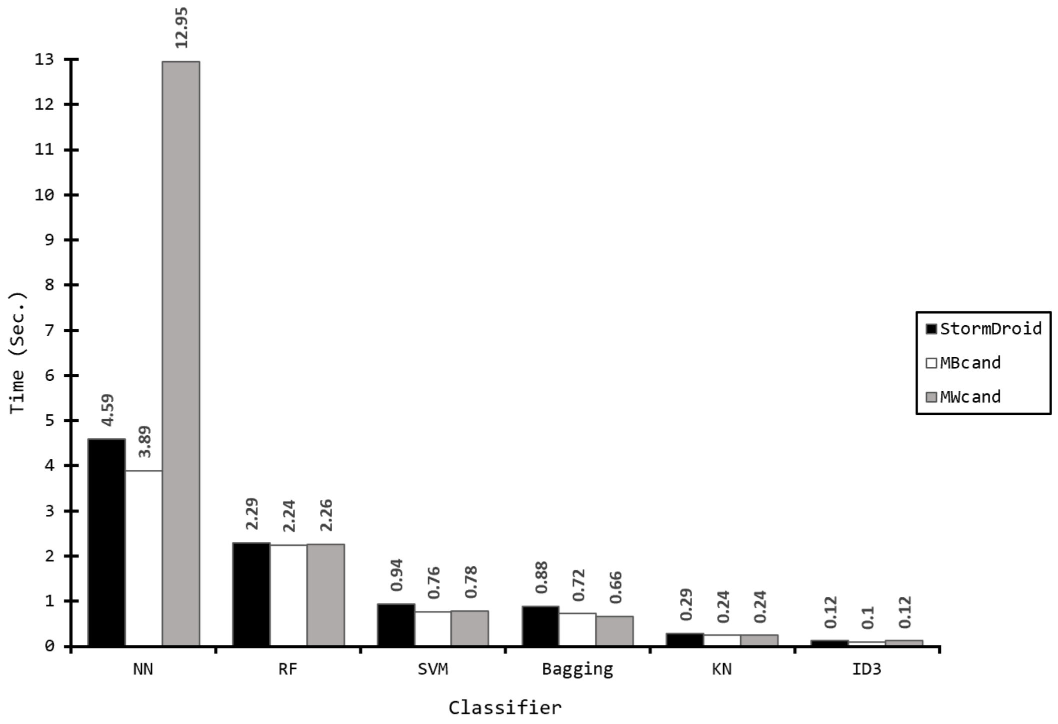


Figure 5.4 Comparison between StormDroid and our approach based on time performance (with oversampling).

In conclusion, by oversampling the data, we gain an increase of 2% in the accuracy than before. However, we can see that the average execution time has increased 30 times than the previous results. The longer execution time was due to the significant increase of the size of the training set because of the oversampling.

By comparing our result with the related work [7], we achieved the same trend as the first experiment. Moreover, M_{Wcand} achieved 0.13% better accuracy and around 5 sec faster.

Table 5.6 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	-

	BaseBridge	RogueSPPush	zHash	Zsone	ADRD	DroidKungFu4	Plankton	DroidDreamLig	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	1	-	-	1	1	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	*92.8%*
RogueSPPush	-	**9**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*100%*
zHash	-	-	**11**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*100%*
Zsone	-	-	-	**12**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*100%*
ADRD	-	-	-	-	**22**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*100%*
DroidKungFu4	-	-	-	-	-	**94**	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	*97.9%*
Plankton	-	1	-	-	-	-	**10**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*90.9%*
DroidDreamLight	2	-	-	-	-	-	1	**41**	-	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*89.1%*
SndApps	-	-	-	-	-	-	-	-	**10**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	*100%*
DroidKungFu2	-	1	-	-	-	1	-	-	-	**25**	-	1	-	-	-	-	-	-	-	-	-	2	-	-	-	-	-	-	*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	1	-	-	-	-	-	-	-	-	-	1	-	**55**	-	-	-	-	-	-	-	-	-	-	-	*94.8%*
AnserverBot	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**187**	-	-	-	-	-	-	-	-	-	-	*100%*
GingerMaster	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**4**	-	-	-	-	-	-	-	-	-	*100%*
GPSSMSSpy	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	**6**	-	-	-	-	-	-	-	-	*100%*
Gone60	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	**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 5.5 Confusion matrix of one RF execution.

CHAPTER 6: CONCLUSIONS

Though out the research, we investigate a total of forty research papers on Android malware familial detection, classification, and categorization from various scientific databases. We classified the literature according to their type of analysis, type of features, and the techniques applied. We further report the datasets that have been used and include details about each of them. Moreover, we discussed the limitations of the literature approaches, challenges faced by the researchers, and future trends for the research community. Our findings show that most of the limitations circulate around the availability and the size of benchmarked datasets. In addition, some common challenges are the lack of samples and standardization of family naming.

Finally, we adopted machine learning to analyze and identify malware features such as the permissions requested by malware. Our focus in this research was to find a small subset of permissions that classified sets of 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 value, M_{Bcand}, and as weighted value, 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 shorter time performance by 37.5% with KN than StormDroid [7]. As for the future work, investment in advanced artificial intelligence techniques such as reinforcement learning and big data technologies should be considered.

REFERENCES

[1] T. McCall and R. van der Meulen, “Gartner Says Worldwide Sales of Smartphones Recorded First Ever Decline During the Fourth Quarter of 2017,” Egham, UK, Feb. 22, 2018.

[2] S. Proske, “Another Reason 99% of Mobile Malware Targets Androids - F-Secure Blog.” https://blog.f-secure.com/another-reason-99-percent-of-mobile-malware-targets-androids/ (accessed Jul. 16, 2020).

[3] “NCP - Checklist McAfee Antivirus 8.8 STIG.” https://nvd.nist.gov/ncp/checklist/479 (accessed May 16, 2020).

[4] D. Moon, H. Im, J. Lee, and J. Park, “MLDS: Multi-Layer Defense System for Preventing Advanced Persistent Threats,” *Symmetry*, vol. 6, no. 4, pp. 997–1010, Dec. 2014, doi: 10.3390/sym6040997.

[5] G. Brij, A. P Dharma, and Y. Shingo, *Handbook of Research on Modern Cryptographic Solutions for Computer and Cyber Security*. IGI Global, 2016.

[6] “Android Security 2017 Year in Review,” *Google Online Security Blog*. https://security.googleblog.com/2018/03/android-security-2017-year-in-review.html (accessed Jul. 16, 2018).

[7] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, “StormDroid: A Streaminglized Machine Learning-Based System for Detecting Android Malware,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security - ASIA CCS ’16*, Xi’an, China, 2016, pp. 377–388, doi: 10.1145/2897845.2897860.

[8] F. Alswaina and K. Elleithy, “Android Malware Family Classification and Analysis: Current Status and Future Directions,” *Electronics*, vol. 9, no. 6, p. 942, Jun. 2020, doi: 10.3390/electronics9060942.

[9] “Platform Architecture,” *Android Developers*. https://developer.android.com/guide/platform (accessed May 16, 2020).

[10] S. Shiraishi, “SDK-Based Quality Assurance Framework for Third Party Apps of IVI Systems,” in *Proceedings of the First International Workshop on Software Development Lifecycle for Mobile (DeMobile13), Saint Petersburg, Russia*, 2013, vol. 19.

[11] “Application Fundamentals,” *Android Developers*. https://developer.android.com/guide/components/fundamentals (accessed May 14, 2020).

[12] A. Dangizyan, “[AAR to DEX] Loading and Running Code at Runtime in Android Application,” *Medium*, Jun. 27, 2019. https://medium.com/@artyomdangizyan/aar-to-dex-loading-and-running-code-at-runtime-in-android-application-69089a30c715 (accessed May 14, 2020).

[13] Y. Peng, M. Zhang, J. Zheng, and Z. Qian, “Research on Android Access Control Based on Isolation Mechanism,” in *2016 13th Web Information Systems and Applications Conference (WISA)*, Wuhan, China, Sep. 2016, pp. 231–235, doi: 10.1109/WISA.2016.53.

[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 *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, Essen, Germany, 2012, p. 274, doi: 10.1145/2351676.2351722.

[15] S. Rastogi, K. Bhushan, and B. B. Gupta, “Measuring Android App Repackaging Prevalence based on the Permissions of App,” *Procedia Technology*, vol. 24, pp. 1436–1444, 2016, doi: 10.1016/j.protcy.2016.05.172.

[16] S. Rastogi, K. Bhushan, and B. B. Gupta, “Android applications repackaging detection techniques for smartphone devices,” *Procedia Computer Science*, vol. 78, no. C, pp. 26–32, 2016.

[17] “Permissions overview,” *Android Developers*. https://developer.android.com/guide/topics/permissions/overview (accessed Oct. 30, 2018).

[18] Y. Zhou and X. Jiang, “Dissecting Android Malware: Characterization and Evolution,” in *2012 IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, May 2012, pp. 95–109, doi: 10.1109/SP.2012.16.

[19] K. Griffin, S. Schneider, X. Hu, and T. Chiueh, “Automatic Generation of String Signatures for Malware Detection,” in *Recent Advances in Intrusion Detection*, vol. 5758, E. Kirda, S. Jha, and D. Balzarotti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 101–120.

[20] C. Nachenberg, “A window into mobile device security–Examining the security approaches employed in Apple’s iOS and Google’s Android,” *Symantec Security Response*, 2011.

[21] K. Dunham, S. Hartman, M. Quintans, J. A. Morales, and T. Strazzere, *Android Malware and Analysis*. Auerbach Publications, 2014.

[22] H. Pieterse and M. S. Olivier, “Android botnets on the rise: Trends and characteristics,” in *2012 Information Security for South Africa*, Johannesburg, Gauteng, South Africa, Aug. 2012, pp. 1–5, doi: 10.1109/ISSA.2012.6320432.

[23] “Current Android Malware,” *forensic blog*, Oct. 02, 2016. https://forensics.spreitzenbarth.de/android-malware/ (accessed Jul. 14, 2020).

[24] “ProGuard,” *Guardsquare*, Feb. 02, 2015. https://www.guardsquare.com/en/products/proguard (accessed Jul. 14, 2020).

[25] “DexGuard,” *Guardsquare*, Jun. 30, 2015. https://www.guardsquare.com/en/products/dexguard (accessed Jul. 14, 2020).

[26] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, “Detecting repackaged smartphone applications in third-party android marketplaces,” in *Proceedings of the second ACM conference on Data and Application Security and Privacy - CODASKY ’12*, San Antonio, Texas, USA, 2012, p. 317, doi: 10.1145/2133601.2133640.

[27] W. Hu, J. Tao, X. Ma, W. Zhou, S. Zhao, and T. Han, “MIGDroid: Detecting APP-Repackaging Android malware via method invocation graph,” in *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*, China, Aug. 2014, pp. 1–7, doi: 10.1109/ICCCN.2014.6911805.

[28] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai, “Identifying android malicious repackaged applications by thread-grained system call sequences,” *Computers & Security*, vol. 39, pp. 340–350, Nov. 2013, doi: 10.1016/j.cose.2013.08.010.

[29] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, “Towards a scalable resource-driven approach for detecting repackaged Android applications,” in *Proceedings of the 30th Annual Computer Security Applications Conference on - ACSAC ’14*, New Orleans, Louisiana, 2014, pp. 56–65, doi: 10.1145/2664243.2664275.

[30] K. Tian, D. Yao, B. G. Ryder, G. Tan, and G. Peng, “Detection of Repackaged Android Malware with Code-Heterogeneity Features,” *IEEE Trans. Dependable and Secure Comput.*, vol. 17, no. 1, pp. 64–77, Jan. 2020, doi: 10.1109/TDSC.2017.2745575.

[31] S. Arshad, M. Ali, A. Khan, and M. Ahmed, “Android Malware Detection & Protection: A Survey,” *ijacsa*, vol. 7, no. 2, 2016, doi: 10.14569/IJACSA.2016.070262.

[32] O. Aslan and R. Samet, “A Comprehensive Review on Malware Detection Approaches,” *IEEE Access*, vol. 8, pp. 6249–6271, 2020, doi: 10.1109/ACCESS.2019.2963724.

[33] E. Gandotra, D. Bansal, and S. Sofat, “Malware Analysis and Classification: A Survey,” *JIS*, vol. 05, no. 02, pp. 56–64, 2014, doi: 10.4236/jis.2014.52006.

[34] T. Wu, X. Deng, J. Yan, and J. Zhang, “Analyses for specific defects in android applications: a survey,” *Front. Comput. Sci.*, vol. 13, no. 6, pp. 1210–1227, Dec. 2019, doi: 10.1007/s11704-018-7008-1.

[35] W. Wang *et al.*, “Constructing Features for Detecting Android Malicious Applications: Issues, Taxonomy and Directions,” *IEEE Access*, vol. 7, pp. 67602–67631, 2019, doi: 10.1109/ACCESS.2019.2918139.

[36] P. Yan and Z. Yan, “A survey on dynamic mobile malware detection,” *Software Qual J*, vol. 26, no. 3, pp. 891–919, Sep. 2018, doi: 10.1007/s11219-017-9368-4.

[37] A. Souri and R. Hosseini, “A state-of-the-art survey of malware detection approaches using data mining techniques,” *Hum. Cent. Comput. Inf. Sci.*, vol. 8, no. 1, p. 3, Dec. 2018, doi: 10.1186/s13673-018-0125-x.

[38] K. Shaerpour, A. Dehghantanha, and R. Mahmod, “Trends in Android Malware Detection,” *JDFSL*, 2013, doi: 10.15394/jdfsl.2013.1149.

[39] R. Riasat, M. Sakeena, C. Wang, A. H. Sadiq, and Y. Wang, “A Survey on Android Malware Detection Techniques,” *dtcse*, no. wcne, Jan. 2017, doi: 10.12783/dtcse/wcne2016/5088.

[40] “IEEE Xplore.” https://ieeexplore.ieee.org/Xplore/home.jsp (accessed Jul. 14, 2020).

[41] “ACM Digital Library.” https://dl.acm.org/ (accessed May 14, 2020).

[42] “MDPI.” https://www.mdpi.com/ (accessed Jul. 14, 2020).

[43] “ScienceDirect.” https://www.sciencedirect.com/ (accessed Jul. 14, 2020).

[44] “Hindawi,” *Hindawi*. https://www.hindawi.com/ (accessed Jul. 14, 2020).

[45] “Springer,” *www.springer.com*. https://www.springer.com/gp (accessed Jul. 14, 2020).

[46] “ArXiv.” https://arxiv.org/ (accessed Jul. 14, 2020).

[47] “Microsoft Academic.” https://academic.microsoft.com/home (accessed Jul. 14, 2020).

[48] “Semantic Scholar | AI-Powered Research Tool.” https://www.semanticscholar.org/ (accessed Jul. 14, 2020).

[49] “Google Scholar.” https://scholar.google.com/ (accessed Jul. 14, 2020).

[50] Y. Fang, Y. Gao, F. Jing, and L. Zhang, “Android Malware Familial Classification Based on DEX File Section Features,” *IEEE Access*, vol. 8, pp. 10614–10627, 2020.

[51] J. Qiu *et al.*, “A3CM: automatic capability annotation for android malware,” *IEEE Access*, vol. 7, pp. 147156–147168, 2019, doi: 10.1109/ACCESS.2019.2946392.

[52] L. Zhang, V. L. L. Thing, and Y. Cheng, “A scalable and extensible framework for android malware detection and family attribution,” *Computers & Security*, vol. 80, pp. 120–133, Jan. 2019, doi: 10.1016/j.cose.2018.10.001.

[53] Z. Xu, K. Ren, and F. Song, “Android Malware Family Classification and Characterization Using CFG and DFG,” in *2019 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, Guilin, China, Jul. 2019, pp. 49–56, doi: 10.1109/TASE.2019.00-20.

[54] O. Mirzaei, G. Suarez-Tangil, J. M. de Fuentes, J. Tapiador, and G. Stringhini, “AndrEnsemble: Leveraging API Ensembles to Characterize Android Malware Families,” in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Auckland New Zealand, Jul. 2019, pp. 307–314, doi: 10.1145/3321705.3329854.

[55] R. Vega Vega, H. Quintián, J. L. Calvo-Rolle, Á. Herrero, and E. Corchado, “Gaining deep knowledge of Android malware families through dimensionality reduction techniques,”

Logic Journal of the IGPL, vol. 27, no. 2, pp. 160–176, Mar. 2019, doi: 10.1093/jigpal/jzy030.

[56] R. Vega Vega, H. Quintián, C. Cambra, N. Basurto, Á. Herrero, and J. L. Calvo-Rolle, “Delving into Android Malware Families with a Novel Neural Projection Method,” *Complexity*, vol. 2019, pp. 1–10, Jun. 2019, doi: 10.1155/2019/6101697.

[57] J. Jiang *et al.*, “Android Malware Family Classification Based on Sensitive Opcode Sequence,” in *2019 IEEE Symposium on Computers and Communications (ISCC)*, Barcelona, Spain, Jun. 2019, pp. 1–7, doi: 10.1109/ISCC47284.2019.8969656.

[58] F. Fasano, F. Martinelli, F. Mercaldo, and A. Santone, “Cascade Learning for Mobile Malware Families Detection through Quality and Android Metrics,” in *2019 International Joint Conference on Neural Networks (IJCNN)*, Budapest, Hungary, Jul. 2019, pp. 1–10, doi: 10.1109/IJCNN.2019.8852268.

[59] W. Blanc, L. G. Hashem, K. O. Elish, and M. J. Hussain Almohri, “Identifying Android Malware Families Using Android-Oriented Metrics,” in *2019 IEEE International Conference on Big Data (Big Data)*, Los Angeles, CA, USA, Dec. 2019, pp. 4708–4713, doi: 10.1109/BigData47090.2019.9005669.

[60] N. Xie, X. Wang, W. Wang, and J. Liu, “Fingerprinting Android malware families,” *Front. Comput. Sci.*, vol. 13, no. 3, pp. 637–646, Jun. 2019, doi: 10.1007/s11704-017-6493-y.

[61] S. Turker and A. B. Can, “AndMFC: Android Malware Family Classification Framework,” in *2019 IEEE 30th International Symposium on Personal, Indoor and Mobile Radio*

Communications (PIMRC Workshops), Istanbul, Turkey, Sep. 2019, pp. 1–6, doi: 10.1109/PIMRCW.2019.8880840.

[62] M. Fan *et al.*, “Android Malware Familial Classification and Representative Sample Selection via Frequent Subgraph Analysis,” *IEEE Trans.Inform.Forensic Secur.*, vol. 13, no. 8, pp. 1890–1905, Aug. 2018, doi: 10.1109/TIFS.2018.2806891.

[63] 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, doi: 10.1145/3162625.

[64] A. Calleja, A. Martín, H. D. Menéndez, J. Tapiador, and D. Clark, “Picking on the family: Disrupting android malware triage by forcing misclassification,” *Expert Systems with Applications,* vol. 95, pp. 113–126, Apr. 2018, doi: 10.1016/j.eswa.2017.11.032.

[65] F. Alswaina and K. Elleithy, “Android Malware Permission-Based Multi-Class Classification Using Extremely Randomized Trees,” *IEEE Access*, vol. 6, pp. 76217–76227, 2018, doi: 10.1109/ACCESS.2018.2883975.

[66] H. Zhou, W. Zhang, F. Wei, and Y. Chen, “Analysis of Android Malware Family Characteristic Based on Isomorphism of Sensitive API Call Graph,” in *2017 IEEE Second International Conference on Data Science in Cyberspace (DSC)*, Shenzhen, China, Jun. 2017, pp. 319–327, doi: 10.1109/DSC.2017.77.

[67] J. Sedano, S. González, C. Chira, Á. Herrero, E. Corchado, and J. R. Villar, “Key features for the characterization of Android malware families,” *Logic Jnl IGPL*, vol. 25, no. 1, pp. 54–66, Feb. 2017, doi: 10.1093/jigpal/jzw046.

[68] A. González, Á. Herrero, and E. Corchado, “Neural Visualization of Android Malware Families,” in *International Joint Conference SOCO’16-CISIS’16-ICEUTE’16*, vol. 527, M. Graña, J. M. López-Guede, O. Etxaniz, Á. Herrero, H. Quintián, and E. Corchado, Eds. Cham: Springer International Publishing, 2017, pp. 574–583.

[69] M. Fan *et al.*, “Frequent Subgraph Based Familial Classification of Android Malware,” in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, Ottawa, ON, Canada, Oct. 2016, pp. 24–35, doi: 10.1109/ISSRE.2016.14.

[70] B. Kang, S. Y. Yerima, K. Mclaughlin, and S. Sezer, “N-opcode analysis for android malware classification and categorization,” in *2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security)*, London, United Kingdom, Jun. 2016, pp. 1–7, doi: 10.1109/CyberSecPODS.2016.7502343.

[71] P. Battista, F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio, “Identification of Android Malware Families with Model Checking.,” in *ICISSP*, 2016, pp. 542–547.

[72] J. Sedano, C. Chira, S. González, Á. Herrero, E. Corchado, and J. R. Villar, “Characterization of Android Malware Families by a Reduced Set of Static Features,” in *International Joint Conference SOCO’16-CISIS’16-ICEUTE’16*, vol. 527, M. Graña, J. M. López-Guede, O. Etxaniz, Á. Herrero, H. Quintián, and E. Corchado, Eds. Cham: Springer International Publishing, 2017, pp. 607–617.

[73] J. Lee, S. Lee, and H. Lee, “Screening smartphone applications using malware family signatures,” *Computers & Security*, vol. 52, pp. 234–249, Jul. 2015, doi: 10.1016/j.cose.2015.02.003.

[74] Y. Li, T. Shen, X. Sun, X. Pan, and B. Mao, “Detection, Classification and Characterization of Android Malware Using API Data Dependency,” in *Security and Privacy in Communication Networks*, vol. 164, B. Thuraisingham, X. Wang, and V. Yegneswaran, Eds. Cham: Springer International Publishing, 2015, pp. 23–40.

[75] L. Deshotels, V. Notani, and A. Lakhotia, “DroidLegacy: Automated Familial Classification of Android Malware,” in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014 - PPREW’14*, San Diego, CA, USA, 2014, pp. 1–12, doi: 10.1145/2556464.2556467.

[76] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, “Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families,” *Expert Systems with Applications*, vol. 41, no. 4, pp. 1104–1117, Mar. 2014, doi: 10.1016/j.eswa.2013.07.106.

[77] B. Kang, B. Kang, J. Kim, and E. G. Im, “Android malware classification method: Dalvik bytecode frequency analysis,” in *Proceedings of the 2013 Research in Adaptive and Convergent Systems on - RACS ’13*, Montreal, Quebec, Canada, 2013, pp. 349–350, doi: 10.1145/2513228.2513295.

[78] Y. S. Sun, C.-C. Chen, S.-W. Hsiao, and M. C. Chen, “ANTSdroid: Automatic Malware Family Behaviour Generation and Analysis for Android Apps,” in *Information Security and Privacy*, vol. 10946, W. Susilo and G. Yang, Eds. Cham: Springer International Publishing, 2018, pp. 796–804.

[79] A. Martín, V. Rodríguez-Fernández, and D. Camacho, “CANDYMAN: Classifying Android malware families by modelling dynamic traces with Markov chains,” *Engineering Applications of Artificial Intelligence*, vol. 74, pp. 121–133, Sep. 2018, doi: 10.1016/j.engappai.2018.06.006.

[80] L. Massarelli, L. Aniello, C. Ciccotelli, L. Querzoni, D. Ucci, and R. Baldoni, “Android malware family classification based on resource consumption over time,” in *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*, Fajardo, Oct. 2017, pp. 31–38, doi: 10.1109/MALWARE.2017.8323954.

[81] S.-W. Hsiao, Y. S. Sun, and M. C. Chen, “Behavior grouping of Android malware family,” in *2016 IEEE International Conference on Communications (ICC)*, Kuala Lumpur, Malaysia, May 2016, pp. 1–6, doi: 10.1109/ICC.2016.7511424.

[82] S. Malik and K. Khatter, “System Call Analysis of Android Malware Families,” *Indian Journal of Science and Technology*, vol. 9, no. 21, Jun. 2016, doi: 10.17485/ijst/2016/v9i21/90273.

[83] M. Aresu, D. Ariu, M. Ahmadi, D. Maiorca, and G. Giacinto, “Clustering android malware families by http traffic,” in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, Fajardo, Oct. 2015, pp. 128–135, doi: 10.1109/MALWARE.2015.7413693.

[84] A. Atzeni, F. Diaz, A. Marcelli, A. Sanchez, G. Squillero, and A. Tonda, “Countering Android Malware: A Scalable Semi-Supervised Approach for Family-Signature Generation,” *IEEE Access*, vol. 6, pp. 59540–59556, 2018, doi: 10.1109/ACCESS.2018.2874502.

[85] H. M. Kim, H. M. Song, J. W. Seo, and H. K. Kim, “Andro-Simnet: Android Malware Family Classification using Social Network Analysis,” in *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, Belfast, Aug. 2018, pp. 1–8, doi: 10.1109/PST.2018.8514216.

[86] K. Aktas and S. Sen, “UpDroid: Updated Android Malware and Its Familial Classification,” in *Secure IT Systems*, vol. 11252, N. Gruschka, Ed. Cham: Springer International Publishing, 2018, pp. 352–368.

[87] T. Chakraborty, F. Pierazzi, and V. S. Subrahmanian, “EC2: Ensemble Clustering and Classification for Predicting Android Malware Families,” *IEEE Trans. Dependable and Secure Comput.*, vol. 17, no. 2, pp. 262–277, Mar. 2020, doi: 10.1109/TDSC.2017.2739145.

[88] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand, “Automated Synthesis of Semantic Malware Signatures using Maximum Satisfiability,” *arXiv:1608.06254 [cs]*, Jun. 2017, Accessed: Jul. 14, 2020. [Online]. Available: http://arxiv.org/abs/1608.06254.

[89] J. Garcia, M. Hammad, B. Pedrood, A. Bagheri-Khaligh, and S. Malek, “Obfuscation-Resilient, Efficient, and Accurate Detection and Family Identification of Android Malware,” p. 15.

[90] J. H. Lau and T. Baldwin, “An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation,” *arXiv:1607.05368 [cs]*, Jul. 2016, Accessed: Jul. 14, 2020. [Online]. Available: http://arxiv.org/abs/1607.05368.

[91] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer, “Online Passive-Aggressive Algorithms,” *Journal of Machine Learning Research*, vol. 7, no. Mar, pp. 551–

585, 2006, Accessed: Jul. 14, 2020. [Online]. Available: http://www.jmlr.org/papers/v7/crammer06a.

[92] A. Desnos, “Androguard-reverse engineering, malware and goodware analysis of android applications,” *URL code. google. com/p/androguard*, vol. 153, 2013.

[93] “Droidbox: An android application sandbox for dynamic analysis.” https://code.google.com/archive/p/droidbox/ (accessed Jul. 14, 2020).

[94] “Cuckoo Sandbox Book — Cuckoo Sandbox v2.0.5 Book.” https://cuckoo.sh/docs/ (accessed Apr. 14, 2018).

[95] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, “Drebin: Effective and Explainable Detection of Android Malware in Your Pocket,” 2014, doi: 10.14722/ndss.2014.23247.

[96] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, “Deep Ground Truth Analysis of Current Android Malware,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 10327, M. Polychronakis and M. Meier, Eds. Cham: Springer International Publishing, 2017, pp. 252–276.

[97] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, “AndroZoo: Collecting Millions of Android Apps for the Research Community,” in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 468–471.

[98] “VirusTotal.” https://www.virustotal.com/gui/home/upload (accessed Jul. 14, 2020).

[99] “VirusShare.com.” https://virusshare.com/ (accessed Jul. 14, 2020).

[100] “ContagioDump Blog.” https://contagiodump.blogspot.com/ (accessed Jul. 14, 2020).

[101] M. Hurier *et al.*, “Euphony: Harmonious Unification of Cacophonous Anti-Virus Vendor Labels for Android Malware,” in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, Buenos Aires, Argentina, May 2017, pp. 425–435, doi: 10.1109/MSR.2017.57.

[102] M. Lindorfer, M. Neugschwandtner, and C. Platzer, “MARVIN: Efficient and Comprehensive Mobile App Classification through Static and Dynamic Analysis,” in *2015 IEEE 39th Annual Computer Software and Applications Conference*, Taichung, Taiwan, Jul. 2015, pp. 422–433, doi: 10.1109/COMPSAC.2015.103.

[103] “AndroMalShare - SandDroid.” http://202.117.54.231:8080/ (accessed Jul. 18, 2020).

[104] levinec, “Malware names - Windows security.” https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/malware-naming (accessed Jul. 14, 2020).

[105] H. L. Thanh, “Analysis of Malware Families on Android Mobiles: Detection Characteristics Recognizable by Ordinary Phone Users and How to Fix It,” *JIS*, vol. 04, no. 04, pp. 213–224, 2013, doi: 10.4236/jis.2013.44024.

[106] “A New Virus Naming Convention (1991) - CARO - Computer Antivirus Research Organization.” http://www.caro.org/articles/naming.html (accessed May 14, 2020).

[107] “Virus Bulletin :: The Common Malware Enumeration Initiative.” https://www.virusbulletin.com/conference/vb2006/abstracts/common-malware-enumeration-initiative/ (accessed Jul. 14, 2020).

[108] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, “AVclass: A Tool for Massive Malware Labeling,” in *Research in Attacks, Intrusions, and Defenses*, vol. 9854, F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, Eds. Cham: Springer International Publishing, 2016, pp. 230–253.

[109] “G DATA Software AG. News.” https://www.gdatasoftware.com/news/2017/02/threat-situation-for-mobile-devices-worsens (accessed Nov. 13, 2018).

[110] C. Lam, *Hadoop in action*. Greenwich, Conn: Manning Publications, 2011.

[111] M. Armbrust *et al.*, “Spark SQL: Relational Data Processing in Spark,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD ’15*, Melbourne, Victoria, Australia, 2015, pp. 1383–1394, doi: 10.1145/2723372.2742797.

[112] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, “Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification,” in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, New York, NY, USA, 2016, pp. 183–194, doi: 10.1145/2857705.2857713.

[113] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in Python,” *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[114] P. Geurts, D. Ernst, and L. Wehenkel, “Extremely randomized trees,” *Mach Learn*, vol. 63, no. 1, pp. 3–42, Apr. 2006, doi: 10.1007/s10994-006-6226-1.

[115] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[116] “3.2.4.3.3. sklearn.ensemble.ExtraTreesClassifier — scikit-learn 0.23.1 documentation.” https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html (accessed Jul. 14, 2020).

APPENDIX A

Abbreviations

Table A.1 List of abbreviation used in the dissertation.

AV	Anti-Virus
ML/MLA	Machine Learning
API	Application Programming Interface FreGraph
SF	Selected Features
SF_{Cand}	Selected Features Candidate $\subset$ SF
M_{Bcand}	Binary Matrix of SF_{Cand}
M_{Wcand}	Weighted Matrix of SF_{Cand}
AAPT	Android Asset Packaging Tool
SDK	Software Development Kit
ρ	Permission
ω	Feature's Weight
ExF	Extracted Features Values (0's and 1's for binary)
SVM	Support Vector Machine
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 Tree Algorithm (classifier)
DL	Deep Learning
CNN, RNN	Convolutional, Recurrent Neural Networks NN
DFG	Data-Flow Graph
CFG	Control-Flow Graph
CDG	Class Dependency Graph
APK	Android Application Package
DEX	Dalvik Executable
PA	Passive-Aggressive Algorithm
UI	User Interface UID, GID