Android authorship attribution through string analysis

by

Vaibhavi Kalgutkar

Bachelor of Computer Engineering
University of Mumbai, 2013

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Natalia Stakhanova, PhD, Faculty of Computer Science
	Paul Cook, PhD, Faculty of Computer Science
Examining Board:	Scott Bateman, PhD, Faculty of Computer Science, Chair
	Wei Song, PhD, Faculty of Computer Science
External Examiner:	Martin Wielemaker, PhD, Faculty of Business Administration

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

September, 2017

Abstract

With the rising popularity of Android mobile devices, the amount of malicious applications targeting the Android platform has been increasing tremendously. To mitigate the risk of malicious apps, there is a need for an automated system to detect these applications. Current detection techniques rely on the signatures of existing malware samples and hence may not be able to detect new malware samples. Instead of generating signatures for malware samples itself, in this thesis, we propose the development of a lightweight system that can generate signatures of malware writers by leveraging the strings components present in their Android binaries. Using these author signatures, we can effectively detect a wide range of existing as well as any new malware samples generated by particular authors. We evaluated the proposed system over datasets of 1559 benign, 262 malicious and 96 obfuscated Android applications. The proposed system was able to identify the authors of benign, malicious, and obfuscated Android applications with an accuracy of 98%, 96%, and 71% respectively.

Dedication

To my parents Satish and Suhasini, and my beloved late grandmother Manjula for their unconditional love, numerous sacrifices, and blessings.

To my sisters Manasi and Surabhi, and my husband Rushabh for supporting me through the ups and downs of life.

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Dr. Natalia Stakhanova and Dr. Paul Cook, for their guidance, support, encouragement, and most of all patience throughout the entire process. I have been extremely lucky to have supervisors who cared so much about my work, and who responded to my questions and queries so promptly.

I am grateful to my thesis committee members, Dr. Scott Bateman, Dr. Wei Song, Dr. Martin Wielemaker, and Dr. Patricia Evans, who were more than generous with their expertise and precious time.

A special thanks to my fellow lab mates, Hugo Gonzalez and Andi Fitriah, for their selfless helpfulness. I would also like to thank the members of the faculty for their support.

Finally, I must express my profound gratitude to my parents, my sisters, and my husband who endured this long process with me. This accomplishment would not have been possible without them. Thank you.

Table of Contents

List of Tables

List of Figures

List of Symbols, Nomenclature or Abbreviations

APK	Android application package
ART	Android Runtime
Tf-idf	Term frequency-inverse document frequency
SVM	Support vector machine

Chapter 1

Introduction

Mobile devices have made a great contribution in the field of information sharing. The mobile device market is expanding rapidly every year. In 2016, the total number of worldwide mobile phone users was about 4.6 billion, which is forecast to reach 5 billion by 2019 [21]. Smartphones with various operating systems are available in the market. However, Android mobile devices have dominated the mobile device market. In 2016, nearly 88% of the mobile device market was occupied by Android [59]. The Android market expanded rapidly with nearly 10% growth rate in total market share within the last three years 2013-2016 [31]. Figure 1.1 shows the worldwide smartphone OS market share from 2013 to 2016.

Mobile application markets are one of the main platforms for distribution of legitimate, as well as malware applications. Malware is an abbreviation used for malicious software. The Android operating system is an open source

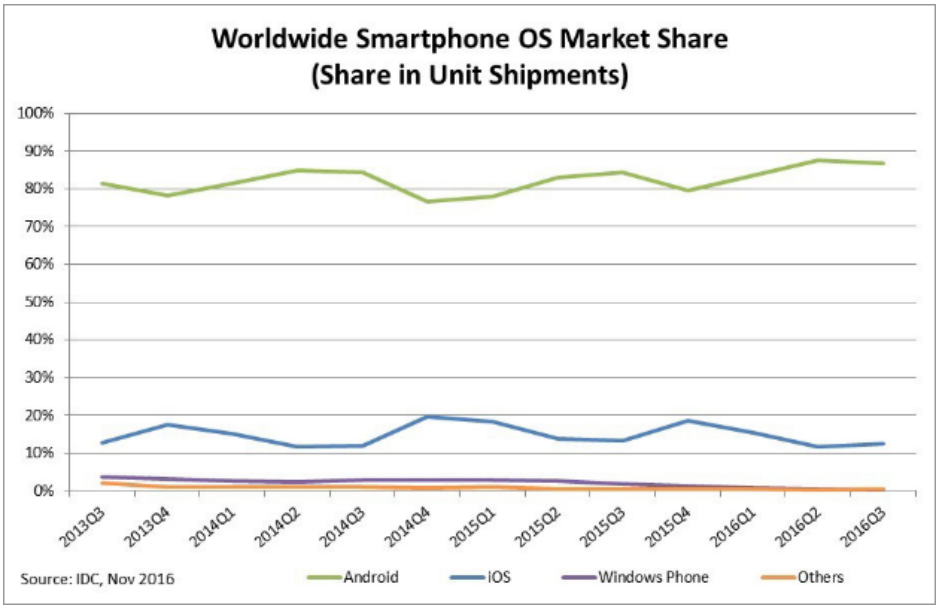

Figure 1.1: Worldwide smartphone OS market share [31]

mobile platform. In addition, Google Play Store, the official Android application distribution store, has very few restrictions on registering and distributing the applications. Due to this, the number of applications uploaded and distributed through Google Play Store has increased dramatically since 2011 [5]. Figure 1.2 shows the increase in the number of available applications on Google Play Store from 2009 to 2017.

The openness of the Android platform and lack of security checks in the application distribution process have resulted in an increasing number of malicious applications targeting the Android platform. According to AV-TEST 2015-16 report, an estimated 99% of all existing mobile malware samples target Android devices [32]. Figure 1.3 shows the drastic increase in the number

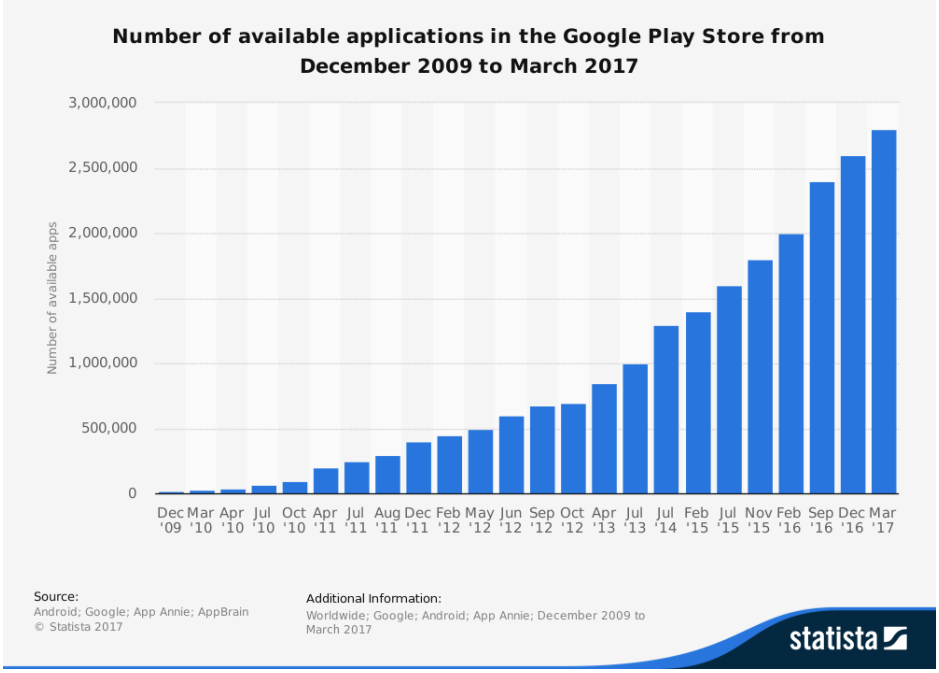

Figure 1.2: Number of applications on Google Play Store [5]

of malware samples targeting the Android platform from January 2013 to September 2016.

With the increasing number of malicious applications, Android users are becoming more susceptible to malware, and hence there is a need for an automated system to detect such malicious apps. On the industry side, the state-of-the-art malware detection systems employ signatures. A malware signature or a malware fingerprint is used to detect and uniquely identify a particular malware. These signatures can be used to design a detection system that examines an unknown sample and if an unknown sample matches the fingerprint of an existing malware, it is flagged as malicious. However,

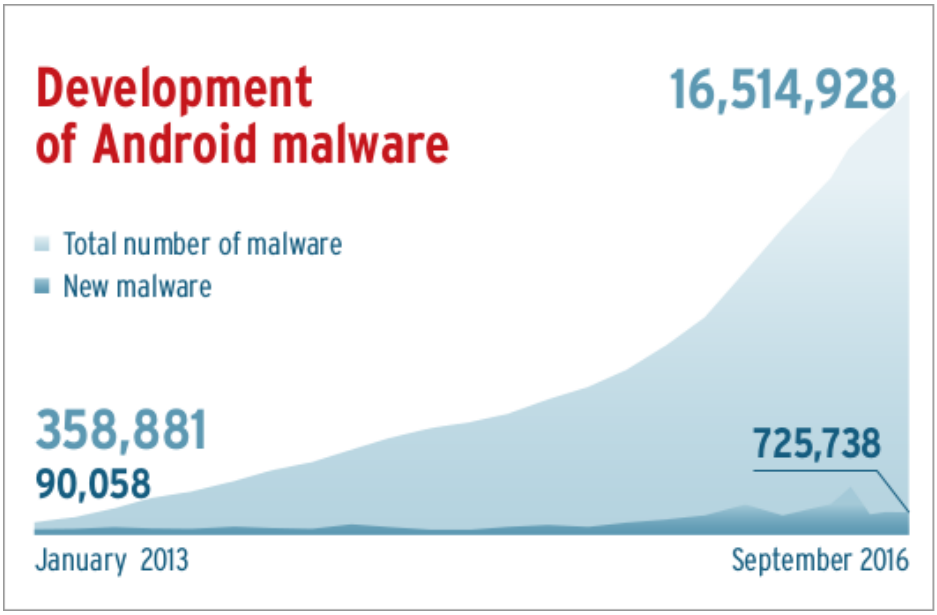

Figure 1.3: Android Malware Development [32]

systems employing signatures are unable to spot zero-day attacks. A zero-day attack exploits the unknown, undisclosed vulnerability of the software. The signature of malware samples exploiting such unknown vulnerabilities are not available. According to G DATA Security, malware writers are producing 8400 new Android malware applications per day in 2017 [42]. Thus, it is extremely important to design a system which can effectively detect new malware samples as well. Also, a wide range of proposed malware detection techniques are based on either the analysis of complex features extracted from the files present in Android executables (e.g., DroidSafe [28]) or dynamic features derived from an app's behaviour during runtime (e.g., RiskRanker [29]). Extraction of such complex features is a tedious task and requires a consider-

able amount of time and system resources. With the increasing amount and complexity of malware, it is beneficial to employ methods that do not rely on analysis of complex features and are time efficient. The task of identifying malicious apps is becoming more challenging as obfuscation in the mobile domain is gaining popularity in both legitimate and malware applications. Obfuscation is a transformation process that produces functionally identical code that is difficult to analyse understand or reverse engineer. Android developers can employ various obfuscation tools to generate obfuscated Android binaries.

In this work, we offer an alternative solution. Instead of detecting malware based on signatures of malware samples, we propose the development of a lightweight system to generate the signatures of malware writers which in turn will be useful to detect all malware samples generated by particular malware authors. Our hypothesis is that every developer has a unique programming style which is reflected through the various components of programs developed by them. By analysing such program components, we can possibly generate the author's signature (also termed fingerprint or profile) that can uniquely identify applications developed by that author. One of the simplest components that can represent an author's writing style is strings. Various string components such as variables, class names, method names, string literals reflect the writing preferences of the developer. We present an approach to generate Android author profiles by analysing different string components present in their sample applications. Every unknown Android

application is then compared with existing author profiles. If the sample corresponds to a malware author profile, it can be labelled malicious. This task of identifying the author of an unknown application is widely known as *authorship attribution*. By this method, we can cover a number of malware samples generated by malware authors. We examine four different kinds of strings namely *unreferenced*, *DEX*, *application* and *all* strings for generating author profiles. *Unreferenced* and *DEX* strings are present in the *DEX* file of the Android binary. *Application* specific strings are extracted from the *strings.xml* file, whereas, *all* strings combine all of these strings together. We further propose to generate Android author profiles even in the presence of obfuscation. With the help of this system we will be able to detect a wide range of malware applications developed by malware authors effectively. We present an efficient, lightweight, obfuscation resilient Android authorship attribution system for generating profiles of Android authors by leveraging the text strings present in an Android application binary. These generated profiles in turn will be useful to identify the author/developer of an unknown Android application.

We have pursued the following research questions in this thesis.

- Can we detect the author of an Android application (benign or malicious) by analysing the programming style of the author in terms of the strings that they use?

- Out of the different kinds of strings considered for our analysis, which

one is the most effective for the Android authorship attribution task?

- Does our approach perform well even in the presence of obfuscated Android applications?

To answer the above research questions, we have developed an Android attribution system based on the analysis of different kinds of strings such as *unreferenced*, *DEX*, *application* and *all* strings represented in the form of n-grams. We have employed a linear SVM classifier for the Android authorship attribution task. We have tested the approach over three different datasets, i.e., datasets of benign, malicious and obfuscated applications.

The contributions of this thesis are as follows:

- To the best of our knowledge, this is the first effort to design an Android authorship attribution system by leveraging different string components of apps.

- We have demonstrated the effectiveness of the proposed method to identify authors of benign, malicious as well as obfuscated Android applications.

- We have presented a comparative analysis of the performance of the different kinds of strings for the authorship attribution task.

- We have conducted our experiments over three different datasets. We have collected a total of 1559 benign application samples from eight different Android markets, 262 malicious application samples from the

koodous system, and 96 obfuscated Android applications from the GitHub repository. These datasets can facilitate further research in this area as designing the dataset is itself a big challenge for any attribution study.

Chapter 2

Related work

In recent years, the amount of research interest in the field of mobile security has been increasing. Many of the research studies have provided an overview of mobile malware characteristics, analysis and detection techniques [68][39][4][60]. A typical work flow of an Android malware detection system consists of two major steps. First the system extracts the features to generate a malware signature which can uniquely represent the malware and second the malware detection mechanism employing these signatures. Based on the type of features extracted, the existing Android malware analysis studies can be classified into two broad categories: static analysis and dynamic analysis. Static analysis examines the features extracted from the application binary. Dynamic analysis evaluates features derived from an application's runtime behaviour. In the case of dynamic analysis, a simulated environment may be needed to execute and monitor the behaviour of an ap-

plication. Whereas, in the case of static analysis, features are extracted by analysing different application components without executing the application itself.

Researchers have examined various components of Android binaries for static analysis e.g., files such as *classes.dex, AndroidManifest.xml, resources.arsc, source code files* and directories such as *assets, META-IN, and res* [23][67][33]. For example, a number of studies have analysed *permissions* in *AndroidManifest.xml* for the static analysis [58][34][11][9][33][67]. In DroidMat, along with permissions other features such as intents, component deployment and APIs from the Manifest were analysed using different machine learning algorithm such as k-means, k-nearest neighbors, and naive Bayes [64]. Similarly, DREBIN analysed intents, permissions, app components, APIs, and network addresses using support vector machines [10]. The study was able to detect malware with 94% accuracy. Apart from the analysis of the Android binary components, many of the studies have introduced more robust static analysis techniques based on the analysis of structural or semantic features. DroidSafe, a static Android analysis tool, examines the data flow of the application [28]. In CLAPP, authors examined Dalvik bytecode to extract static features such as the number of loop iterations and the body of the loop statements [25]. In contrast to static analysis techniques, studies based on dynamic analysis examine the results generated during the execution of the application [66]. TaintDroid, An Android malware analysis tool, monitors the interaction of the Android application with third-party applications [22].

Another dynamic analysis tool, Crowdroid, examines dynamic system calls of the Android application [14]. ProfileDroid, monitors and profiles activities at four different levels: static, user, OS and network [62]. Another tool RiskRanker employs techniques such as control flow graph analysis for detecting Android malware samples [29].

Though many of the studies have presented various static and dynamic analysis techniques, there are certain limitations of these techniques. Static analysis techniques are sensitive to various obfuscation techniques as obfuscation can alter the components of the Android binary such as source code and the control flow of the program. Static analysis does not execute the application, hence, it can not capture the details such as dynamic loading of malicious payload, network activities, malicious objects generated at runtime, etc. Also extraction, parsing and analysis of the static features can be a time consuming task. Hence it is often inefficient for large scale analysis. In the case of dynamic analysis, though these techniques can provide accurate assessments, a significant amount of time and system resources are needed to set up the simulated environment for every test case, to execute and monitor each application and finally to analyse the generated results. Hence, applying such techniques for a large scale analysis is a challenging task. Malware writers can misdirect dynamic analysis by using techniques such as tuning the system for virtual environment and active debuggers.

In contrast to the previous systems based on generating signatures of the mal-

ware sample itself, we propose to develop a system to detect the malicious Android applications by analysing the Android author profiles/signatures. The study of 'Authorship attribution' aims to solve such a problem. Authorship attribution is the task of identifying the author of an anonymous application. It provides a way to classify the applications by their respective authors by examining their writing style. In this chapter, we discuss the research advances in the field of authorship attribution in the literary and software domains, followed by the motivation behind our proposed approach.

2.1 Authorship attribution

2.1.1 Literary domain

Authorship attribution in the literary domain is a process to identify the probable author of a given anonymous or disputed text document. It has been widely studied since the 19th century. One of the earliest authorship attribution studies attempted to examine the authorship of Shakespeare's work [43]. The underlying assumption of the attribution is the existence of an inherent distinctive writing style, unique to the author. Automated authorship attribution in the literary domain offers various computational and statistical techniques to characterize the author of a document by employing a set of textual features that quantify an author's writing style. The features quantifying an author's writing style are known as style markers. The set of style markers is termed an author fingerprint or profile. The survey provides

a comprehensive overview of features, approaches and methodologies used to quantify the author's writing style in the literary domain [57]. Researchers have studied the variety of stylometric features for the authorship attribution task such as lexical features based on viewing text as a sequence of tokens (e.g., word length, sentence length, type token ratio, word frequency, spelling errors), character features based on viewing text as a sequence of characters (e.g., types of characters like digits, letters), syntactic features based on syntactic pattern used by authors (e.g., frequently used parts-of speech, sentence structure, syntactic errors), semantic features based on the semantics of the text (e.g., number of synonyms) and application specific features to represent an author's style in a particular domain (e.g., types of signatures in e-mail messages).

Over the years the authorship attribution field evolved substantially leveraging the recent advances in areas such as machine learning and natural language processing. Beyond the traditional focus on the analysis of literary work, authorship attribution techniques have practical applications in other fields such as biometric research [26], attribution of electronic texts (e.g., email messages, blog posts, web pages) [41], malware analysis [17] and software authorship attribution.

In the next section, we discuss the research advancements in the software authorship attribution domain.

2.1.2 Software authorship attribution

Software authorship attribution is a process to identify the author of a given software by examining the programmer's style. The main hypothesis of software authorship attribution is that every software developer has a unique style of developing software. Software authorship attribution is a much more difficult problem than traditional literary authorship attribution. In the traditional setting, authorship information can be extracted by performing deeper linguistic analysis of an authors' works, for example, vocabulary richness, tense of verbs and semantic analysis of sentences. In contrast, computer code has a more rigid format compared to literary work due to the structure and syntax requirement of any programming language. This makes extraction of the author's programming style difficult.

However, there are various software components that can still reflect the programmer's style. While developing any software, every developer has certain preferences, such as *for* loops over *while* loops, the programming language used, choice of data structures and algorithms, variables and keywords used, libraries and frameworks included, functions used, naming conventions, employed development tools, settings and so on. Analysis of software may unveil the digital identity of a programmer that is reflected through such software components. The software authorship attribution has a wide range of applications such as identifying the original author of plagiarized code, finding the author of a malware sample from a set of suspected attackers, copyright investigation or resolving authorship disputes over software.

In the next section, we discuss the origin of software authorship attribution.

2.1.2.1 Origin

The previous attempts in the software authorship attribution domain originated from the theory proposed by Halstead in the early 1970s [30]. The theory stated that simple counts of unique operators and operands and the total number of operators and operands are sufficient to reflect the implementation and structure of any algorithm. These four metrics proposed by Halstead became known as *Halstead's metrics* or *software science metrics*. These metrics represent the 'Internal Quality' of an algorithm and are unlikely to be the same among programs written by independent authors. This theory triggered a number of studies focusing on software similarity detection. Initially, the focus of these studies was to find similarities between the programming assignments submitted by students and thus, to detect the original and plagiarized documents. This research, known as plagiarism detection, laid the groundwork for software authorship attribution. Along the lines of plagiarism detection, the researchers started to investigate the relation between the authors and programs written by them. Similarly to plagiarism research, this topic started with the analysis of source code and then evolved into binary code analysis.

Each programmer has a unique programming style which can be used to characterize programs with respect to the author. This initiated the studies focusing on extracting style markers of the author from the programs.

Researchers started to examine programming style based on the analysis of various code components such as basic Halstead metrics analysis, variables, type of statements, functions, keywords, etc.

As the authorship attrition field evolved, researchers started to apply the attribution techniques in other domains such as in security to find cyber attackers. In the next section, we discuss the origin of software authorship attribution studies in the security domain.

2.1.2.2 Application in security domain

A new era of authorship attribution started with the application of the existing research to a new domain known as software forensics. Spafford and Weeber proposed to utilize features extracted from pieces of code and the remnants of software to identify a potential intruder [55]. They defined this technique as software forensics. This technique could be used to examine software in any form including source files, object files, executable code, shell scripts and so on. Later on, Krsul highlighted the effectiveness of authorship analysis to enhance real-time intrusion detection systems [38]. He proposed the use of programmer's style to detect abnormal system behaviour. Following this study, authorship attribution raised a lot of interest in the security domain.

Traditional software authorship attribution studies were based on the selection of the set of feature metrics to represent the author's style, followed by

the method to discriminate the authors. The researchers studied the range of features such as basic Halstead's metrics, number of spaces, tabs, indentation, use of upper or lower case for variable names, number of special macros, and so on. However, most of these proposed metrics were programming language dependent. Moreover, the selection of the metrics was a crucial task. This led to the development of the new techniques such as the n-gram analysis in the attribution field.

In the next section, we discuss the n-gram analysis techniques for software authorship attribution.

2.1.2.3 Authorship attribution through n-gram analysis

An n-gram is a sequence of n terms combined together. N-grams can be extracted as a sequence of bytes, characters or words. The performance of the n-grams in the software attribution field was first evaluated by Frantzeskou et al. [24]. Frantzeskou et al. presented a novel approach based on the most frequent byte level n-gram analysis for the source code authorship attribution. They developed a system to generate the author profiles by extracting the most frequent byte level n-grams from the source code samples. This promising method known as the source code author profiles approach (SCAP) was previously evaluated successfully for natural language authorship attribution by Keselj et al. [35]. The SCAP system was programming language independent and can be seamlessly applied for the attribution of source code written in any programming language. Software authorship attribution faces a lot of

challenges due to a lack of data samples. If enough code samples for a particular author are not available, it becomes difficult to extract style markers that can uniquely represent the author's programming style. As the SCAP approach considers the most frequently appearing byte level n-grams by combining the source code samples of the author, it performed surprisingly well even with the limited amount of code samples per author. Generally cyber criminals do not use comments in their code samples in order to hide their identity. However, the approach was effective even in the absence of comments. Due to various advantages, this study attracted the attention of the research community and underlined the effectiveness of n-grams to represent the author's programming style. Many other subsequent studies employed the n-gram analysis for the authorship attribution task.

Function words such as 'the', 'but', 'and', and 'or' have been used extensively in the literary authorship attribution domain as strong markers of an author's style [57]. Based on a similar hypothesis, Burrows and Tahaghoghi employed function words extracted from the source code, i.e., keywords and operators, for the attribution task [15]. They presented a system to generate the author fingerprint/profiles by leveraging n-grams of keywords and operators from the source code samples. The system based on n-gram analysis of the function words proved to be effective for the source code authorship attribution task.

Later, Kothari et al. evaluated the performance of character based n-grams and other stylistic and layout based features such as distribution of line size,

leading spaces, underscores per line, semicolons, words per line and commas per line for the attribution task [37]. They observed that character n-grams produce better author profiles than any of the other mentioned layout and stylistic features. This is because the character n-grams can capture programming style context as well as information, such as names of the variables and functions, producing strong style markers of the author.

In 2014, Burrows et al. compared the different source code authorship attribution techniques [16]. The study confirmed the effectiveness of n-gram analysis as presented in the SCAP method [24] for the source code attribution task.

Other than source code authorship attribution, the n-gram analysis technique proved to be effective for similar research problems as well. Layton et al. used n-grams to analyse phishing websites with an authorship perspective [40]. The phishing website is intended to steal sensitive, confidential information such as user names, passwords, or credit card details for malicious purposes from website users by pretending to be a legitimate site. Layton et al. extended the byte level n-gram analysis approach SCAP for the analysis of phishing websites. They extracted the most frequent n-grams from the phishing websites to generate clusters of these websites according to the phishing campaigns. These clusters were useful to study the size and scope of each phishing campaign. In another study, researchers evaluated the performance of byte n-grams for computer virus detection [53]. The popular plagiarism detection system MOSS uses n-grams to find the degree of simi-

larity between documents [1].

In the next section, we discuss the motivation behind our proposed approach.

2.2 Motivation

We propose to design an Android authorship attribution system by leveraging the string components in Android binaries. A string, which can be defined as a sequence of tokens, is one of the simplest components used for the authorship attribution task. A token can be viewed differently in different context. They are essentially instances of words in context. Software developers have certain preferences while developing software and they tend to use similar strings while developing software. The various string components in the program such as names of variables, procedures, functions, modules, and labels reflect the author's choice and thus can produce strong style markers of the author. As the field of authorship attribution evolved, researchers studied a variety of string-based features to quantify authors' styles. Starting from the basic Halstead's metric, string components such as keywords, operators, operands, variables and function names have been used extensively to analyse authors' programming styles. Most of the attribution studies have focused on the analysis of the strings extracted from the source code. However, the source code samples are not always easily available. Especially in the case of malware, it becomes extremely difficult to access the source code, as the malware samples are distributed in the form of binaries. However, the de-

compiled Android binary still has different string components that can reflect the Android author's style. Thus, for our analysis we have studied different kinds of strings extracted from Android binaries.

Using these extracted strings, we further propose to generate author profiles by selecting the most frequent word n-grams. N-gram analysis has been used successfully in a variety of research domains including software similarity detection, malware detection, and authorship attribution in the literary as well as the software domain. Due to the advantages of the n-grams technique, we propose to develop an Android authorship attribution system based on the analysis of string n-grams from Android application binaries.

Chapter 3

Proposed framework

In this chapter, we discuss the Android application background, describe the types of strings used for our analysis, and present the system architecture.

3.1 Android application background

In this section, we discuss the structure of Android application package file (APK) along with a description of various components of the APK.

3.1.1 Android application structure

Android is an open source operating system which runs on top of a modified Linux kernel. An Android application is distributed and installed on various Android devices as an archive file termed an Android application package file (*APK*). The structure of an APK file is illustrated in Figure 3.1.

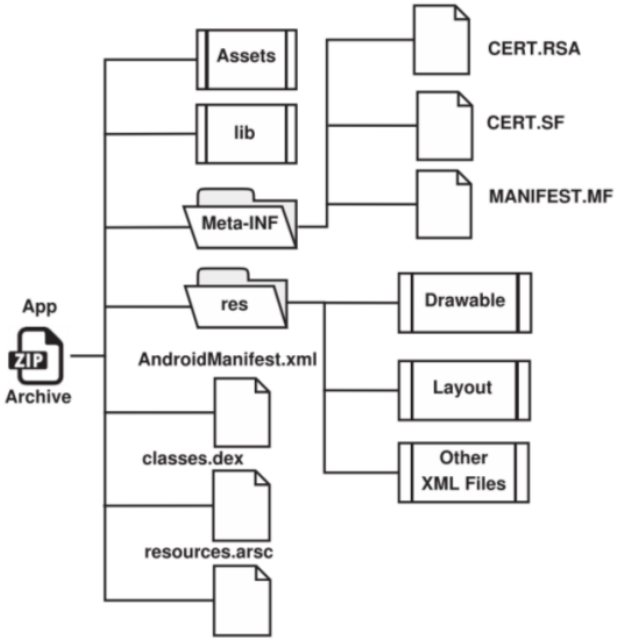

Figure 3.1: The structure of an APK [46]

An APK contains a variety of files and directories as follows:

- *AndroidManifest.xml*: An essential file in every Android application. It provides vital application details such as the unique application identifier, permissions required by the application, application version, referenced libraries, and description of several application components such as activities, services, broadcast receivers and content providers.

- *resources.arsc*: A file containing pre-compiled resources.

- *classes.dex*: This is an Android Runtime *ART* (which is a predecessor of Dalvik machine) executable file also known as a *DEX* file. It contains

compiled Android classes in *.dex format* for Android Runtime. Most of the APKs have a single DEX file. However, a larger APK is divided into multiple *classes.dex* files due to the size limitation of DEX files [47].

- *META-INF*: The directory containing several files responsible for ensuring the integrity and security of the application. It includes *MANIFEST.MF* — the manifest file which stores metadata of the application, *CERT.RSA* — the digital certificate of the application, and *CERT.SF* — the file containing list of resources and SHA-1 digest.

- *res*: The directory containing resources not compiled into *resources.arsc*.

- *assets*: The optional directory containing external resource files.

- *lib*: The optional directory containing compiled libraries in sub directories sorted as per specific processor. For example, *lib/armeabi* directory contains compiled code for all ARM based processors.

For the string analysis, we examine the *classes.dex* file. In the next section, we discuss the structure of an Android DEX file.

3.1.2 DEX file structure

Among the files and directories present in the APK, *classes.dex* is the most crucial file of the APK as it contains all of the application logic and work flow. The Android application code is written in Java. During the application compilation process, all the Java source code files are first compiled as

.class files containing Java bytecode. However, the Android Runtime which is responsible for executing the Android application recognizes only Dalvik bytecode. Thus, it requires files in *dex* format. Hence, Java *.class* files are further compiled into *classes.dex* files by *dx tool*.

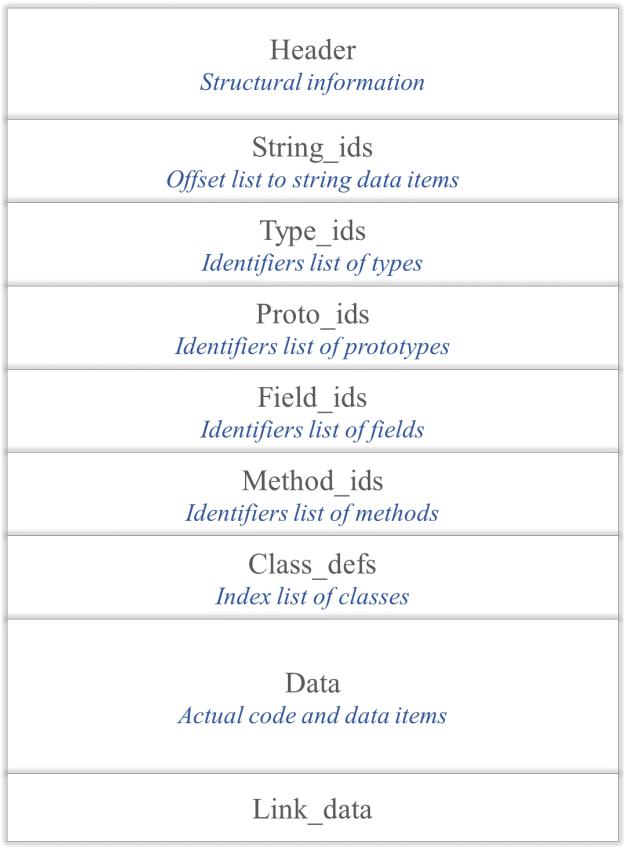

Figure 3.2: The structure of a DEX file

The structure of the DEX file is illustrated in Figure 3.2. The DEX file is partitioned into a number of sections. The DEX file consists of a header section containing the header information followed by several identifier lists

such as string, type descriptors, prototype, field, method, and class definitions. These identifier lists define all the functional content used by the DEX file such as variables, arrays, primitive data types, methods, and classes. After the list of identifiers, the DEX file has a data section containing the actual implementation data supported by the identifiers defined in the previous sections. Every element defined in the implementation list sections maintains the offset pointing to the corresponding entries in the data section. For example, the string identifier list section maintains offsets of all kinds of strings used in the DEX file. These strings are used for naming various program entities such as class, field, or local variable names or as the constant objects in the source code (e.g., string literals). The string offset points to the location in the data section where the string has been used. Other identifiers sections such as type ids, prototype ids, field ids, method ids, and class defs, also maintain the list of offsets pointing to the data section where the identifiers have been actually used. However, other than maintaining references to the data section, these sections also contain references to the string identifier list. For example, a class named *AccountActivity* will have a reference to the class data in the data section as well as a reference to the actual string *AccountActivity* defined in the string identifier section. This layout structure of the DEX file is defined in the formal Android development specification [48].
In the next section, we discuss the different types of strings extracted from the APK for our analysis.

3.1.3 Types of strings

The several types of identifiers present in the DEX file ultimately refer to a string. We propose to analyse these string components found in the DEX file. The various string components such as classes, methods, and variables used in the data section are linked to different identifier sections depending upon their type. Though Android application development should follow the specifications and guidelines stated by the official Android Development community, malware writers often violate these specifications to include malicious code inside the Android application. One of the approaches is to *hide* the malicious code by placing it in the data section of the DEX file while avoiding it being referenced by elements of the identifiers sections such as class ids or method ids [6][7]. The advantage of such *code hiding* is that it makes the reverse engineering of the application difficult. Reverse engineering is the process of examining the piece of software by analysing its components in order to understand the design, work flow of the software or to replicate the software functionality. Many Android reverse engineering tools analyse the DEX file of an APK. The identifier lists in the DEX files are typically used to invoke the various methods and classes and are useful for analysing the workflow of the application. However, the *code hiding* technique prevents the reverse engineering of an application, and thus it becomes difficult for anti-malware scans to detect such malware.

Due to such code hiding techniques, strings used in such hidden code do not have references to identifier sections other than the string ids list. Thus,

we differentiate the strings present in the data section of the DEX file as *'referenced strings'* and *'unreferenced strings'*. The referenced strings have a reference to the identifier list other than the string offset list, whereas the unreferenced strings are referred to only by the string identifier list. The referenced strings represent classes, methods, or different data types in the application depending on the type of the identifier list referring to them. Thus, these strings form a part of the functional, executable code of the application. On the other hand, the unreferenced strings are only referenced by the string offset list, and thus consist solely of non-executable code. These strings can contain hidden, interesting or malicious code details. For example, malware writers often use hard coded URLs, email addresses, and malicious code samples in the form of strings in malware applications. Thus, analysis of such hidden unreferenced strings can reveal malicious activity embedded in Android applications. We therefore analyse unreferenced strings found within the DEX file of the Android application. We also analyse the impact of all the *DEX strings components*, i.e., by combining both unreferenced and referenced strings for the authorship attribution task.

Other than the DEX strings present in the DEX file, an APK contains another source of string components, i.e., the resource file *'strings.xml'*. The file *strings.xml* is an XML resource file. This file is located under the *res/-values/* directory in the APK. It provides a list of all the strings that can be referenced within the application source code or from other resource files of the application [52]. Android developers may define some of the strings that

have been used throughout the application. These strings are hard coded in this file. Following is the example of *strings.xml* file format as specified by the official Android Development community [50].

```
<resources>
    <string name="test_text">Test</string>
    <string name="clear_text">Clear</string>
    <string name="wifi_connection">The active connection is
    wifi.</string>
    <string name="mobile_connection">The active connection is
    mobile.</string>
    <string name="no_wifi_or_mobile">No wireless or mobile
    connection.</string>
</resources>
```

Android developers use the attribute *'name'* of the XML element *'string'* to define the string used throughout the application. These author-defined application strings can be a strong style marker denoting the author's writing style and pattern. Thus, we examine the author style based on these *'application strings'* as well.

Below is a summary of the different kinds of strings used for our analysis:

- ***Unreferenced strings***: The strings present in the DEX file having reference to only the string identifier section.
- ***Referenced strings***: The strings present in the DEX file having ref-

erences to the string identifier section and any of the other identifier sections.

- ***DEX strings***: All the kinds of strings present in the DEX file, i.e., the set of referenced and unreferenced strings.
- ***Application strings***: The strings extracted from the XML resource file *strings.xml*.
- ***All strings***: All the above specified strings.

We evaluate the performance of these strings for the task of identifying the author of Android applications. In the next section, we discuss the architecture of our proposed system.

3.2 System architecture

The architecture of the proposed system is depicted in Figure 3.3. All the available Android binaries of Android authors are first collected for the analysis. To analyse the role of strings for the task of Android authorship attribution, the proposed system consists of several steps: the extraction of different kinds of strings from the Android APK, followed by feature generation based on string n-grams, feature vector generation for creating author profiles. Every APK belonging to each author in the dataset undergoes these steps. The generated author profiles are then used to classify any unknown

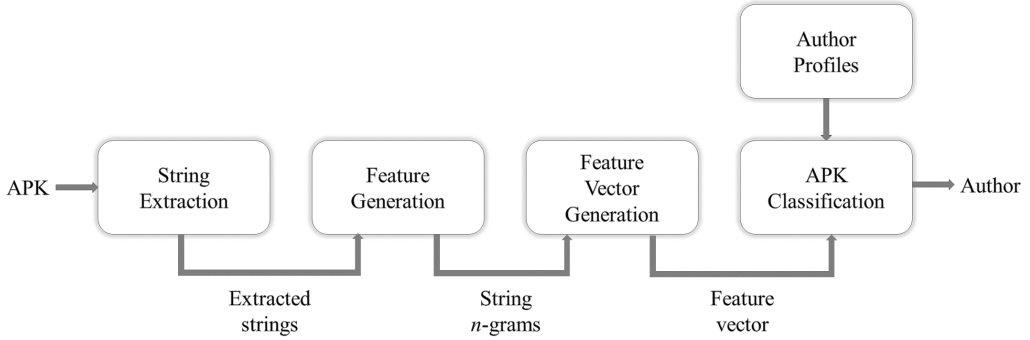

Figure 3.3: The proposed system architecture

APK to one of the Android authors in the dataset. We discuss each step in detail in the following sections.

3.2.1 String Extraction

The string extraction is the initial preprocessing step focusing on the extraction of essential string components from the APK file. These strings are used for generating the author's stylistic features. These features are further used for representing the author's programming style and serve as the basis for classification of Android applications with respect to their authors.

In the strings extraction step, the attribution system first unpacks every APK under the analysis. Depending on the type of the strings considered for the analysis, the strings extraction procedure differs as follows.

To analyse ***DEX strings***, i.e., all the kinds of strings present in the DEX file of the application (both referenced and unreferenced strings), the system first extracts all *classes.dex* files from the unpacked APK. As the string ids

section in the DEX file maintains the offset for all kinds of strings used in the data section of the DEX file, the system extracts the *DEX* strings by traversing the string ids list and collecting all the strings referenced by the list. However, to analyse only ***Unreferenced strings*** present in data section of the DEX file, the system further discards the *referenced* strings from the set of *DEX* strings. Unlike *unreferenced* strings which are referenced only from the string identification section, *referenced strings* have references from the string identification section as well as from any of the identifier sections other than the string identifier section. Hence, to discard the *referenced* strings, the system traverses the other identifier lists, i.e., type, prototype, field, method, and class and then removes strings that are referenced by these lists. For the extraction of ***Application strings***, the system follows a different approach. The system extracts *strings.xml* from the unpacked APK. The system then parses this *strings.xml* file to extract the strings defined by the attribute *'name'* of the XML *'string'* elements. In the case of ***All strings*** analysis, all kinds of strings contribute to the final string data. Thus, the system extracts both type of strings, i.e., *DEX* strings from the DEX file as well as *Application* strings from *strings.xml* file. All the extracted strings during the string extraction step then form the basis for the string n-grams analysis.

In the next section, we discuss the string n-grams feature generation process.

3.2.2 Feature generation

Once the system extracts the required strings from the APK, the next step is to generate features from these strings for generating author profiles. The system employs word level string n-grams derived from the extracted strings as features for the author classification task.

An n-gram is a sequence of n terms. Word level n-grams can be viewed as a sequence of n words combined together from a given sentence or text. For example, consider a sentence, *A computer is an electronic device.* From the given sentence, word level 3-grams can be extracted by combining 3 words together as follows:

1. *A computer is*
2. *computer is an*
3. *is an electronic*
4. *an electronic device.*

Here, we have considered white-space to separate two words. N-grams can also be extracted as a sequence of characters from a given string. However, such character level n-grams can contain characters belonging to different words. Whereas, in the case of n-grams at the word level, the semantics of the original words from a given string is preserved as word level n-grams represent sequence of whole words. Thus, for our n-gram analysis, we propose to generate string features at the word level, i.e., word n-grams.

During the feature generation step, the system generates the list of extracted strings in lexicographical order. The system outputs each extracted string on a separate line. The system generates word level n-grams by combining the words from a string on every line. This is done in order to generate line bounded n-grams. The advantage of line bounded n-gram is that it does not contain words from different lines. Strings on different lines are not necessarily associated. Combining the words belonging to different lines would have resulted in the generation of n-grams containing words that are not a part of a single string, therefore by generating n-grams exhibiting meaningless semantics. Such n-grams would have generated noisy author profiles which could affect the performance of the system.

While generating line bounded word n-grams, the system ignores a line if it represents a string with less than n words. However, this can cause a loss of the information. Thus, the system augments each extracted string

Table 3.1: String n-grams feature examples

1-grams	2-grams	3-grams	4-grams
mReset=	<LB>mReset= mReset=<LB>	<LB>mReset=<LB>	
Cookie value must not be null	<LB>Cookie Cookie value value must must not not be be null null<LB>	<LB>Cookie value Cookie value must value must not must not be not be null be null<LB>	<LB>Cookie value must Cookie value must not value must not be must not be null not be null<LB>

with a unique line boundary token at the start and end of the string before extracting n-grams. The addition of line boundary tokens ensures that every line contains at least 3 words including the line boundary tokens. Thus, more features are available to represent an author's style. The line boundary tokens also give information about the context of the first and the last word. This can provide crucial information for example, a string beginning with words like *Error* or *Warning* can mean something different than a string containing such words elsewhere. Line boundary tokens are not added for generating *uni*-grams (1-grams) as *uni*-grams represent only a single word. Table 3.1 demonstrates examples of how the system generates n-gram features from a string. It shows the n-gram features generated from two strings — *mReset=* and *Cookie value must not be null*. From the table it is clear that there are no line boundary tokens (<LB>) added for the 1-gram features. Also, in the case of a string containing only a single word (e.g., *mReset=*), there is no 4-gram feature available as even with line boundary tokens the given string contains only 3 words.

The generated word level n-gram features are further used to form feature vectors. In the next section, we discuss the feature vector generation process.

3.2.3 Feature vector generation

The extracted n-grams in the feature generation stage form the basis of the feature vector generation process. Many machine learning algorithms used for classification require numerical representation of features for the

analysis. Thus, after the extraction of string n-grams, the system produces the numerical representation of generated n-grams features in the form of *vector*. A vector is defined as a list of objects. A numerical feature vector is a series of numbers representing different features.

In the feature vector generation process, the system calculates frequencies of extracted word n-grams to represent each APK under analysis. The system generates a *frequency vector* for every APK. Every element in the *feature vector* represents the number of times a particular n-gram occurs in a given APK. For example, consider the following strings extracted from an APK:

- Cookie map must not be null
- Cookie name must not be empty

These extracted strings produce a total of 12 word level 3-gram features as follows:

- Word 3-grams extracted from first string: <LB> Cookie value, Cookie value must, value must not, must not be, not be null, be null <LB>
- Word 3-grams extracted from second string: <LB> Cookie name, Cookie name must, name must not, must not be, not be empty, be empty <LB>

Thus, in the above case, the APK has 11 unique word level 3-gram features, i.e., [<LB> Cookie value, Cookie value must, value must not, must not be, not be null, be null <LB>, <LB> Cookie name, Cookie name must, name

must not, not be empty, be empty <LB>] which can be represented in the form of *frequency vector* as:

$$v = [1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1]$$

Here, each element in the vector represents the frequency of the corresponding n-gram feature. The system generates feature vectors for every Android application of each author. These vectors are then used to generate Android author profiles using a *supervised learning* process. In the next section, we discuss the supervised classification process used to identify the author of an Android application.

3.2.4 APK classification

After the feature vector generation step, the system employs these vectors to train the supervised classifier.

In a supervised learning process, the system analyses the set of labelled input data, i.e, samples with known output label to produce an inferred function which can be used further for predicting the label for unknown, unseen samples. A supervised classification algorithm first *learns* the characteristics of given input samples belonging to different categories. After the learning process, the algorithm classifies the unseen, unknown sample into one of the existing categories.

Our proposed Android attribution system employs feature vectors generated

in the previous stage as input data for the supervised classifier. The system analyses the set of feature vectors belonging to each Android author whose authorship is known. The system trains the supervised machine learning classifier in order to produce the fingerprints/profiles/signatures of all the authors in the dataset. Using these *inferred* author profiles, this trained classifier model predicts the authorship of any unknown Android application.

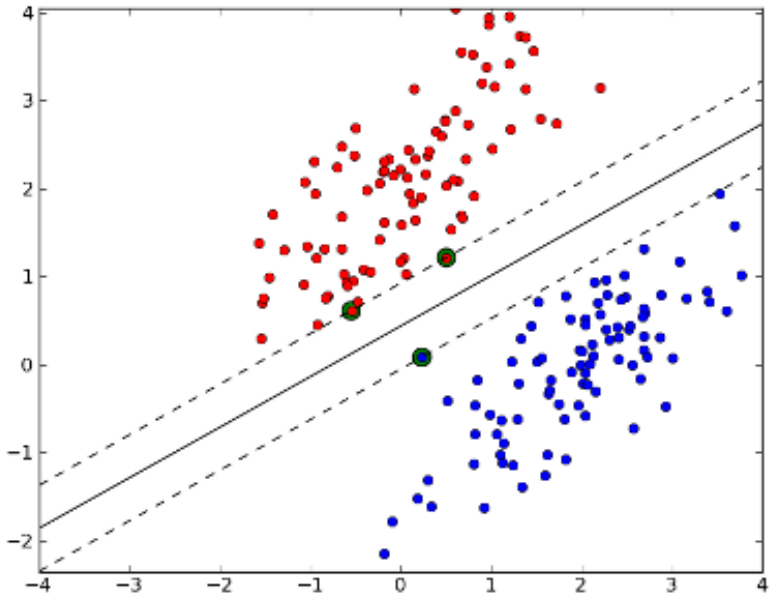

Figure 3.4: The separating hyperplane — SVM

We selected support vector machine *SVM* as the supervised classifier for our system. The support vector machine classifier is one of the robust classi-

fiers [3]. It has successfully been applied to solve real world problems such as pattern recognition, face detection, handwritten digit recognition, and text categorization [13]. SVM works on a principle of building a set of hyper planes separating samples belonging to different classes [20]. The goal of SVM is to generate the separating hyperplanes having maximum distance from the nearest training data sample in a feature space. Figure 3.4 shows the maximum margin hyperplane separating samples of two classes in a 2 dimensional space. The samples present on the maximum distance margins are known as *support vectors*. SVM performs well even in high dimensional feature space. SVM classifier has been successfully used for Android malware detection and malware family classification system based on strings present in Android applications, similar to our proposed approach [36]. This system analysed the string components of APKs to detect a potential malware and its family. Thus, due to the advantages of SVM, we employed SVM classifier for our Android authorship attribution system.

Chapter 4

Experimental setup

In this chapter, we discuss the experimental setup for the system. It includes details about how the dataset is built, description of various tools and methods used, implementation details of the system and strategies used for the evaluation of the system performance.

4.1 Dataset

Our system employs a supervised learning algorithm for the attribution task. The availability of the labelled dataset is one of the main requirements of any supervised learning algorithm. Thus, for the evaluation of our system, we need to have a labelled Android author dataset, i.e., a set of Android developers with their Android application samples. However, the availability of the dataset is a major challenge for any authorship attribution study.

There are no standard open benchmark datasets available for Android authorship attribution research. Though many open source Android malware projects such as Android Malware Genome project [65], AndroMalShare [12], Contagio mobile mini-dump [44] host Android samples, the samples are not grouped by their authors. Thus, we need to manually collect the Android samples belonging to different authors. We propose to evaluate our system on three different datasets, i.e., benign authors dataset, malware authors dataset and dataset of obfuscated applications. We discuss each dataset in detail in following sections.

4.1.1 Benign application dataset

To build the dataset, we must have Android applications grouped by their respective developers. According to the specification stated by the official Android community, each Android APK developer needs to digitally sign the APK with a certificate, in order to install the APK on any Android device [49]. This certificate contains the information that uniquely identifies the developer of the APK, such as metadata along with the public key. The developer possesses the corresponding private key for the public-key certificate. The Android system recognizes the developer through this certificate. When the developer *signs* the APK, the public-key certificate is attached to the original APK. The certificate acts as the identity of the developer of the APK. Thus, we used the certificates attached to the Android application to collect the samples belonging to the same author. APKs having the same

certificates are considered to be developed by same author/origin. Some Android developers can generate different certificates to sign different APKs. However, such cases are out of scope of our analysis.

To build the dataset of legitimate authors, we collected more than ten thousand Android APKs from eight different Android markets (Google play store, Appland, Anzhi, Aptoide, MoboMarket, Nduoa, Tencent, Xiaomi). We grouped these APKs by their certificates. Thus, APKs belonging to the same author/developer were grouped together. Some certificates are publicly available and anyone can use them for signing the APK. Keeping this in mind, we discarded the APKs signed by public certificates. Android studio has a provision to sign an APK with debug certificates generated by the Android SDK tools for debugging the application. However, these debug certificates are insecure and should not be used for application distribution purposes [49]. Thus, we also discarded APKs signed by debug certificates. Also, some of the APKs have been published in multiple markets. We removed such duplicate APKs. Finally, for our dataset, we considered only authors having more than 20 applications. The final dataset contains a total of 40 authors with 1559 Android applications.

4.1.2 Malware application dataset

As the risk of Android malware is increasing day by day, our proposed Android attribution system must effectively detect samples produced by Android malware authors as well. As the authorship information of many

malware samples is not known, the task of collecting the malware samples grouped by authors is difficult. However, there are Android malware research communities that provide a platform to share malicious Android APKs. *'Koodous'* is an open source, collaborative, web based platform that allows researchers to share and analyse Android malware samples [19]. The system has various features that facilitate Android malware analysis such as effective navigation through the entire repository, a set of APIs and Python modules for APK analysis, and the provision of customizable rules (written in *Yara* format) for defining specific tasks. Due to the various advantages of *Koodous* system, we built the malware authors dataset using this system. We used the *search API* provided by the system to search for malicious APKs having the same certificate. We used the same concept of using certificates embedded in the APK to group the Android malware samples by their author. We collected a total of 262 malicious APKs from 10 different authors with each author having at least 10 unique malicious APKs.

4.1.3 Obfuscated application dataset

Obfuscation is a process to make the program difficult to analyse and understand for both humans and decompilation tools without changing its functionality [63]. Obfuscated Android apps are difficult to reverse engineer. Modern obfuscation tools provide many functionalities such as encryption of strings, code obfuscation, control flow obfuscation, and optimization by removing redundant code. Due to a wide range of advantages, Android ob-

fuscation is gaining popularity among both legitimate as well as malware writers. As obfuscation affects the underlying APK, it is important to test the system performance over the dataset of obfuscated applications.

However, collecting obfuscated Android APKs is a challenging task. One of the ways to collect such APKs is first to identify the unobfuscated APKs in our existing benign and malware application datasets. Using different Android obfuscation tools on these unobfuscated APKs, we can then generate a dataset of obfuscated applications. There are research studies focusing on the detection of the type of obfuscation used in Android binaries [61][8]. However, none of the proposed methods is 100% accurate and therefore, we can not rely on these techniques to identify and collect obfuscated APKs.

Another way to build an obfuscated dataset is first to collect the source code of the Android applications and then apply different obfuscation techniques. Generating obfuscated APKs from the source code is beneficial as most of the Android obfuscation tools work at the source code level. Due to security concerns and the possibility of code theft, many Android developers do not host their code publicly. Thus, collecting source code samples is not a trivial task. The open source community *'GitHub'* is one of the largest open source collaborative platforms.[1] GitHub provides the ability to host and share multiple projects through repositories. A repository is used to host a particular project. It can be viewed as a location to store all the project contents. The project can be retrieved by cloning/downloading the repository on to a lo-

[1]https://github.com/

cal machine. Many Android developers share their Android projects on the GitHub platform. Thus, we collected the source code samples from GitHub. We used API provided by GitHub to search for the Android repositories. We downloaded the Android repositories hosted by different Android authors. After cloning the source code of Android projects from various repositories, we employed different obfuscation tools to generate the dataset of obfuscated applications. However, as GitHub is a collaborative platform, a single repository can have multiple contributions; i.e., it can contain code developed by different authors. Considering this, for our dataset we selected repositories satisfying the following criteria:

- The repository must not be a forked repository. A forked repository is a copy of another repository. Developers often copy a project from another repository belonging to a different author to perform additional experiments or to add extra functionalities without changing the original repository content. Hence, we discarded such forked repositories as they can potentially contain code belonging to multiple authors and do not represent the coding style of a single author.

- The repository should not have more than one contribution as multiple contributions indicate that the project is developed by multiple authors.

- The repository should not contain code to include external repositories during the application compilation process. Android developers can add external projects belonging to some other author as project depen-

dencies. During the application build process such external projects are complied and added into the generated APK. Hence, each repository must be examined manually to ensure that it does not contain such external project dependencies.

We initially collected more than 255 Android projects from 28 authors. However, many of the projects could not be compiled due to various reasons such as being unable to locate external dependencies files, compilation errors due to incomplete code or improper project settings. We discarded authors having very few Android projects as enough samples are needed to represent the author's style. We were finally able to generate a total of 96 unobfuscated APKs belonging to 9 authors. We compared the system performance over this set of unobfuscated APKs with the set of obfuscated APKs generated using different obfuscation tools applied to the final 96 source code projects. Different obfuscation techniques are used to obscure the application. Obfuscation techniques can be classified into the following four major categories depending on the nature of transformation performed by them [18]:

1. Layout obfuscation: These obfuscation techniques alter the overall layout of the code by performing transformations such as removing comments, debug information and unused code, and renaming different identifiers, i.e., methods, variables, classes, and packages.

2. Data obfuscation: These techniques transform the code by modifying various code components such as splitting the variable declaration and

initialization, encoding the data, and altering conditional statements.

3. Control obfuscation: These techniques modify the flow of the program to make it more difficult to reverse engineer. It includes techniques such as the addition of dead code to alter the data flow and changing the order of execution of statements.

4. Preventive transformations: These techniques do not obfuscate the code itself, rather they make the decompilation of the code difficult by exploiting the weakness in current decompilation techniques. For example, *HoseMocha*, a Java program was designed to prevent decompilation of Java applications by *Mocha* decompiler [18]. It adds an extra instruction statement after a return statement in every method of the Java program. This does not affect the actual working of the Java application. However, Mocha decompiler fails to decompile such application.

Many of the Android obfuscation tools employ a combination of these obfuscation techniques to provide various functionalities such as name obfuscation, resource obfuscation, control flow obfuscation, code optimization and so on. Though there are many Android obfuscation tools available, most of them are commercial with high cost. For our research, we used free/evaluation version of these tools. We selected 3 Android obfuscation tools as follows:

1. *ProGuard*: ProGuard is one of the most popular Android obfuscation tools [56]. It is free and easy to use. It is integrated with Android

Studio [51]. Android Studio is the official IDE for developing Android applications. Android projects developed with Android Studio can be easily configured for obfuscation using ProGuard. Android Studio provides the default configuration settings for obfuscation using ProGuard; however, the developer can also modify these settings for customized obfuscation. ProGuard is considered as a Java class file obfuscator as it alters Java bytecodes. As such, it obfuscates Java class files of the Android application. These obfuscated Java class files are compiled further to produce DEX files. ProGuard performs three major steps for obfuscating the Android APK. Initially, it performs *code shrinking* by removing unused code such as unused methods, variables, and classes. Developers can also configure ProGuard to remove unused resources. It then performs Java bytecode optimization. Finally, it obfuscates the code by renaming variables, classes, methods, and fields to some short random meaningless words. ProGuard can also use a user-defined dictionary for renaming. Each of the steps, i.e., shrinking, optimization, and obfuscation, is optional as a developer can enable or disable it. The default ProGuard obfuscation configuration file without any optimization provided by Android Studio is sufficient for removing unused code and can effectively obfuscate Android applications. Thus, we employed this default configuration file to generate the obfuscated APKs using ProGuard.

2. *Allatori*: Another Android obfuscator used in this work is Allatori [2].

It can be seamlessly integrated with the Android Studio APK build process. Though Allatori is a commercial Android obfuscation tool, its free educational version with all the features of the commercial version is available. Similar to ProGuard, Allatori is a Java obfuscator which obfuscates Java class files. It also provides functionality to rename variables, classes, methods, fields, and packages. However, unlike ProGuard, it does not provide an option to use a user-defined dictionary for renaming, and employs its own internal functionality for obfuscating these string components. Apart from name obfuscation, Allatori provides features such as string encryption, control flow obfuscation, and optimization. String encryption enables encoding of string data used in the application. We used the default Allatori configuration in which all the above features are enabled.

3. *DashO*: DashO is a commercial Android obfuscation tool with a free evaluation version available [54]. Allatori and DashO both have similar configurations. DashO is a Java obfuscator which provides string encryption, control flow obfuscation, and optimization in addition to name obfuscation by renaming variables. We used the default configuration setting provided by DashO which has all of these mentioned features to generate the obfuscated APKs.

We used the above tools to obfuscate a total of 96 Android source code projects cloned from GitHub. Thus, each Android application project pro-

duced one unobfuscated APK and 3 versions of the obfuscated APKs using the above obfuscation tools. Table 4.1 summarises the datasets used for the analysis.

Table 4.1: Summary of datasets

Dataset	Authors	Apps	Description
Benign	40	1559	Benign applications collected from 8 different Android markets
Malicious	10	262	Malicious applications collected from *Koodous* system
Obfuscated	9	96	Collected from *GitHub* platform. Each project produced 1 unobfuscated and 3 obfuscated APKs using different obfuscation tools

In the next section, we discuss the implementation details of the system.

4.2 Implementation details

In this section, we discuss the various methods, tools and techniques used for the implementation of the proposed Android authorship attribution system. We evaluated the system performance over each dataset discussed in the previous section. The system extracts various string components from APKs in the given dataset. These extracted strings are used to generate author profiles and to train the supervised classifier. We implemented the proposed system in the Python language. We used various Python modules available

in the **scikit-learn** library. Scikit-learn is a Python-based, open source, efficient library for handling various data mining and analysis tasks [45]. It provides implementations of a wide range of machine learning algorithms. Scikit-learn provides functionality to generate feature vectors from string n-grams extracted from every APK under the analysis. We employed SVM (support vector machines), a supervised machine learning classification algorithm to train the classifier which is then used to predict the authorship of the Android applications. After the feature vector generation process, we used *scikit-learn* to incorporate the linear SVM classifier in our attribution system.

Another important tool used in our analysis is **DroidKin**. DroidKin is a lightweight tool used to measure the similarity between Android applications [27]. It analyses DEX files and other metadata such as resources, libraries, and certificates to calculate the similarity between different Android applications. Android authors can repackage existing applications to produce more APKs. Many of the malware writers use repackaging techniques to generate malware samples from existing legitimate applications. Such repackaged APKs have identical or almost similar DEX file content. Authors having such similar APKs can affect the performance of our system. This is because string components extracted from APKs having similar DEX file content will be almost identical. Thus, n-grams extracted from such strings can generate false representations of Author profiles. DroidKin examines the APKs of an author and can identify such similar APKs. Thus, we

evaluated the system performance by discarding such identical APKs from our dataset. DroidKin also provides **APK similarity** scores between APKs. We used this similarity score as a threshold value. We evaluated the system performance over the dataset generated after discarding all the APKs having equal or greater similarity score than a specified threshold. This analysis can give insights into the system performance over datasets having varied levels of APK similarity. We present the detailed analysis of this comparison in the next chapter.

4.3 Evaluation methodology

In this section, we discuss how the system performance is evaluated. We also discuss various measures used for evaluating the system such as *accuracy*, *precision*, *recall*, and *F1 measure*.

4.3.1 Cross validation

The goal of our proposed authorship attribution system is to classify an unknown APK to one of the authors from a set of candidate Android authors available in the given dataset. Any supervised classification task consists of two stages, i.e., a training stage and a testing stage. During the training stage, our system trains a supervised classifier using training data, i.e., a set of known APKs belonging to a set of known authors. This trained classifier can then map any unknown APK (i.e., APK not used for the training

purpose) to its predicted author from a set of known authors (i.e., the set of authors used for training a classifier). During the testing stage, the performance of the classifier is evaluated over test data samples. One potential approach is to divide the given dataset into two parts, one for training the classifier and another for testing. However, a fixed set of samples may not be effective for training the classifier. This is because if a classifier is trained over a fixed set of samples, it can produce an estimation that is biased towards these samples. Thus, we used *cross validation* for our evaluation.

In k-fold cross validation, the given dataset is divided into k equal parts/folds. Some of the parts are selected as a training set and others as a testing set. Typically the instances are randomly selected to be included in folds. However, this might lead to samples of only some of the classes to be selected for training the classifier which can generate biased results. The *stratified* cross validation technique provides a solution for this problem as it preserves a proportionate amount of samples belonging to every class in each fold. The classifier is trained over training set samples and its performance is evaluated over test samples. This procedure is repeated k times with different sets of training and testing samples. Finally, the results of k experiments are aggregated to get the final evaluation results. For our system, we adopted a 5-times 5-fold stratified cross validation strategy. During each experiment, we divided the dataset into 5 parts. We used one part for evaluating the system and the rest for training the classifier. We performed stratified cross validation experiments repeatedly for comparing performance

of different kinds of strings on a single dataset. To ensure repeatability of folds for each of such experiments, we used ransom seed. Figure 4.1 shows

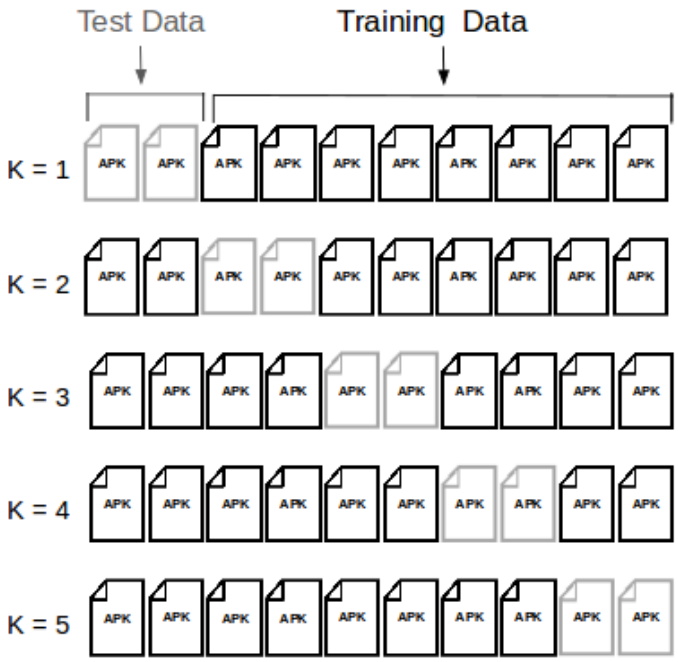

Figure 4.1: 5-fold cross validation

the k-fold cross validation with k=5. It consists of 5 experiments each with different sets of training and testing APKs. In 5-times 5-fold cross validation, these 5-fold cross validation experiments are repeated 5 times. In the next section, we discuss different measures used for evaluating the performance of the system.

4.4 Evaluation metrics

The goal of the classification model is to correctly classify an input sample to one of the output classes from a set of discrete output categories. Our Android authorship attribution system trains a classifier over training samples. The classifier then predicts the author of APKs in the test data. The best way to represent such output predictions of the classification model is to use a *confusion matrix*. A confusion matrix is an $N \times N$ contingency table, where N is the number of output labels. It shows the number of samples correctly and incorrectly classified by the model as compared to the actual target output values. For example, consider a binary classification model with output labels/categories as either *Positive* or *Negative*. The prediction

Table 4.2: Confusion Matrix for a binary clasification problem

	Target positive	**Target negative**
System positive	True positive (t_p)	False positive (f_p)
System negative	False negative (f_n)	True negative (t_n)

results of such a binary classification model can be represented in the form of a 2×2 *confusion matrix* as shown in table 4.2. In this table, target/actual output categories are represented by columns; whereas, rows represent the system predictions. *True positives* indicate the number of positive samples correctly labelled as positive by the system. Similarly, *true negatives* represent the number of correctly predicted negative samples. An ideal classification model would correctly assign samples to their actual categories and

thus, it will not have any *false negative* or *false positive* samples as these cells represent number of incorrectly predicted positive and negative samples respectively. Various measures can be calculated from a confusion matrix to evaluate the system performance as follows:

- ***Accuracy***: Accuracy is the most basic measure used for evaluating the classifier performance. Accuracy is calculated as the number of correctly predicted samples divided by the total number of samples. It represents the proportion of correct predictions made by the classifier.

$$Accuracy = \frac{t_p + t_n}{t_p + f_p + t_n + f_n}$$

  However, accuracy is not sufficient to evaluate the performance of the classifier as it can be misleading in the case of highly imbalanced data. In an imbalanced dataset, the number of samples representing a single class (*majority class*) is significantly high as compared to the other class samples (*minority class*). In such a case, the classification model can produce high accuracy by simply classifying every sample into the majority class. Nevertheless, such a model is not practically useful as it fails to correctly predict the minority class elements. Thus, we used additional measures, i.e., *precision* and *recall*, to evaluate the performance of the classifier.

- ***Precision***: Precision is calculated as the number of samples correctly predicted as positive divided by the total number of samples predicted

as positive in the dataset.

$$Precision = \frac{t_p}{t_p + f_p}$$

It demonstrates the exactness of the classifier as it represents the proportion of samples predicted as positive that are actually positive.

- ***Recall***: Recall is calculated as the number of samples correctly predicted as positive divided by the total number of actual positive samples in the dataset.

$$Recall = \frac{t_p}{t_p + f_n}$$

It demonstrates the comprehensiveness of the classifier as it represents the detection rate, i.e., the proportion of actual positive samples that are correctly predicted as positive.

- ***F_1 score***: F_1 score also known as F_1 measure combines precision and recall into a single measure. It is calculated as the harmonic mean of precision and recall.

$$F_1 measure = \frac{2 * Precision * Recall}{Precision + Recall}$$

The above measures are described for evaluating the binary classification model. However, our system employs a multiclass classification model as it predicts the author of a test APK from a set of discrete authors. To evalu-

ate the performance of such models, also known as multinomial classification models, the above measures can be aggregated over a set of output class labels in two ways, i.e., ***microaveraging*** or ***macroaveraging***.

In the case of **microaveraging**, a multinomial classification problem is converted to a binary classification problem by combining predictions over all the class labels into a single confusion matrix. Thus, an $N \times N$ confusion matrix, where N is the number of output labels, is converted into a 2×2 confusion matrix. This matrix is then used to calculate precision, recall and F_1.

In the case of **macroaveraging**, performance measures (precision, recall and F_1) are calculated for every class and the results are then averaged to produce macroaveraged measures. In the macroaveraging method, each class contributes equally to get the final average results. Whereas, in the case of the microaveraging method, the final average results are influenced by the majority class as all classes are combined together to produce the averaged result. Hence, we calculated **macroaveraged** precision, recall and F_1 scores for evaluating the system performance.

Chapter 5

Experimental results and discussion

In this chapter, we discuss various experiments carried out for evaluating system performance followed by the analysis of the experimental results.

5.1 Preliminary analysis

We conducted a variety of preliminary experiments after implementing the proposed Android authorship attribution system. In the preliminary analysis, we examined different parameters to improve the system performance.

- **Size of *n*-grams**: Initially, we analysed the system performance by changing the size of word n-grams. We observed the improvement in system performance with increase in size of n-grams from 1 to 3.

This is because of more contextual information present in higher order n-grams as compared to lower order n-grams. However, the system performance degrades with further increase in size of n-grams. Our system ignores the strings having less than n terms even after addition of line boundary tokens. Furthermore, many 4-grams or even higher order n-grams will be unique which leads to data sparsity (i.e., lack of enough data for accurate estimation).These factors may have played an important role in generating poor representations of author styles. Thus, we used 3-grams for further experiments.

- **Feature vectors**: We compared the performance of plain frequency count feature vectors with that of *tf-idf* (term frequency and inverse document frequency) count feature vectors . Frequency count vectors (discussed in the section 3.2.2) represent the number of times a feature (in our case a word level n-gram) occurs in a given APK. Whereas, in the case of tf-idf count feature vectors, the n-gram features appearing in many APKs get lower weight in a feature vector. Tf-idf vectors did not enhance the system performance; hence, we employed the frequency count feature vectors for the rest of the experiments.

- **Linear SVM classification strategies**: Our system handles a multinomial classification problem as it attributes an unseen APK sample to one of the candidate Android authors in the dataset. We employed the linear SVM classifier provided in the scikit-learn Python library as

a supervised classifier. Support vector machine classifiers inherently support binary classification. However, it can handle a multinomial classification problem by transforming it into a binary classification problem by two techniques - ***One vs one (OVO)*** and ***One vs rest (OVR)***. In the case of the One vs one classification technique, the classification algorithm generates $n * (n-1)/2$ binary classifiers for n output classes. Each classifier is trained to predict samples belonging to two different classes. For an unseen sample, the prediction results of all the classifiers are considered. The class having the highest number of predictions is selected as the final output class of the unseen sample. In the case of the One vs rest technique, the classification algorithm produces only n binary classifiers. Each classifier is trained to distinguish samples belonging to one particular class from samples of the rest of the classes. Every classifier produces the score of the class for which it has been trained. For any unseen sample, the class having the highest score is selected as the final output class.

Linear SVM can implement both the strategies (OVO or OVR) for handling multinomial classification problems. The One vs rest technique is preferred and is the default strategy of the linear SVM classifier [45]. We evaluated the system performance using both strategies. Though both strategies demonstrated similar performance, average training time for the one vs one technique was significantly higher than for the one vs rest technique. Thus, we employed the one vs rest

strategy for further experiments.

Based on the preliminary analysis, we employed 3-gram word level frequency feature vectors to train Linear SVM classification employing the one vs rest strategy for the remainder of the analysis. We carried out a variety of experiments to evaluate the system performance as listed below:

1. Evaluating the system over three different kinds of dataset, i.e., datasets of benign, malware and obfuscated applications.

2. Evaluating the system over datasets with different level of APK similarity.

3. Comparing the performance of different kinds of strings (i.e., *all*, *DEX*, *unreferenced* and *application* strings described in section 3.1.3)

We discuss these experimental results in the next sections.

5.2 Benign application dataset results

In this section, we discuss the performance of our system on the benign application dataset (section 4.1.1). This dataset contains a total of 40 Android authors with 1559 APKs. All experiments are performed 5 times each with 5-fold cross validation.

5.2.1 Comparison of different types of strings

We compared the performance of the Android attribution system over various kinds of strings. Android author profiles are generated using word level 3-grams extracted from these strings. These profiles are then used to predict the author of an unseen APK. These results are illustrated in table 5.1. It shows the different performance measures along with the training time, i.e., the time taken by the classifier to train and produce author profiles.

Table 5.1: Performance comparison of different types of strings over the benign application dataset

String Type	Accuracy	Macro Average Precision	Macro Average Recall	Macro Average F1	Average Training Time (Seconds)
All	98.19%	98.13%	97.49%	97.55%	606.78
DEX	98.17%	98.12%	97.48%	97.55%	625.30
Unreferenced	97.55%	97.42%	96.52%	96.64%	208.81
Application	94.40%	95.93%	92.80%	93.10%	5.53

The results for all kinds of strings and all DEX strings are quite similar. These two kinds of strings perform better than *unreferenced* strings in the DEX file or *application* strings present in the *strings.xml* file. It is quite apparent as more strings are preserved in the case of *All* and *DEX* strings. Thus, more features can be extracted for analysing authors' styles. Nevertheless, the training time of the classifier is significantly reduced in the case of *unreferenced* and is lowest for *application* strings. Around 36 million

strings are processed in the case of *All* and *DEX* strings which is reduced drastically in the case of *unreferenced* and *application* strings (to around 12 million and 380 thousand strings, respectively).

5.2.2 Effect of APK similarity

In this section, we discuss the performance of our system over the datasets with varying levels of APK similarity.

We used the DroidKin tool which compares two APKs and provides a similarity score between them (discussed in the section 4.2). We examined the similarity between APKs of every author in the dataset. We used the similarity score as a threshold value. All the APKs of an author having a similarity score equal or greater than the specified thresholds are discarded from the dataset. After removing such similar APKs, authors having less than 5 APKs are also discarded, as at least 5 APKs are needed to perform stratified 5-Fold cross validation experiments and have at least one app from each author in each fold. The performance of the system over datasets at varying similarity thresholds is illustrated in Figure 5.1. This figure shows the performance of all four kinds of strings by varying APK similarity level threshold. In every string performance chart, the *X axis* indicates the performance of the system on the scale of 0 to 100%, whereas, the *Y axis* indicates the similarity level threshold of APKs in the dataset.

The results are quite predictable. The system performance gradually degrades with the APK similarity level threshold. Removal of similar APKs

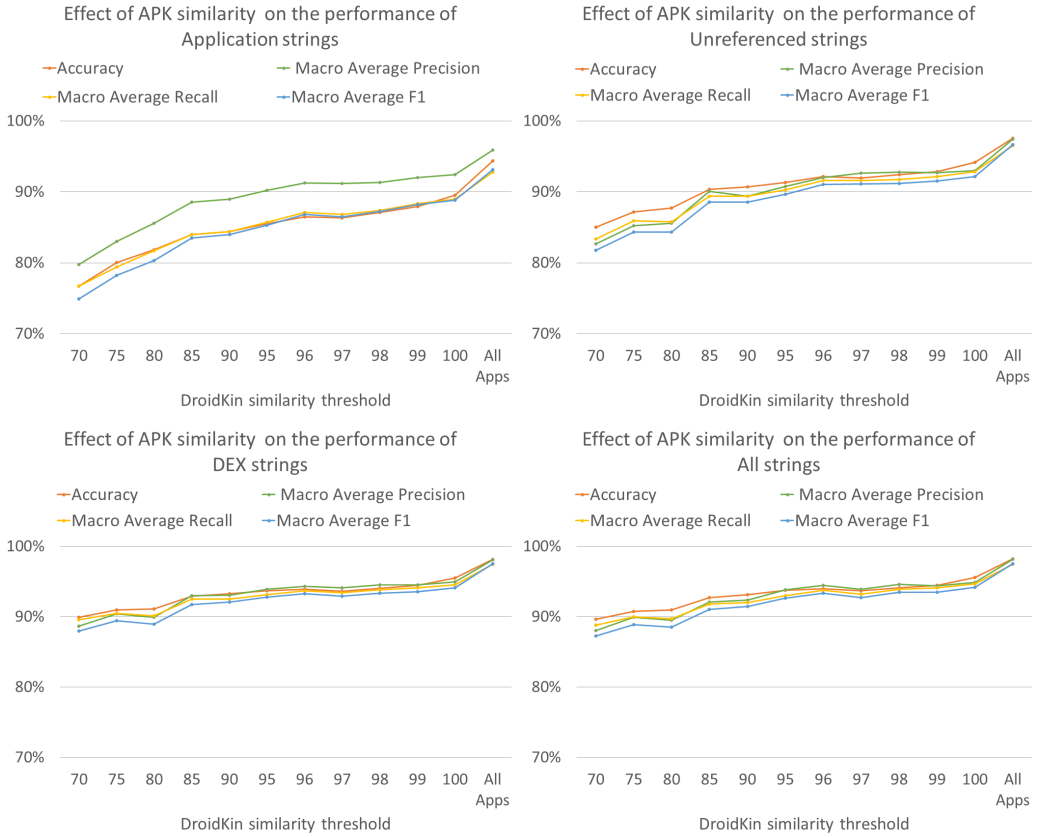

Figure 5.1: APK similarity threshold vs system performance on the benign dataset

from an author leads to a reduction of frequently appearing strings, which affects the frequency count of string n-gram features used to represent the author's style. Another contributing factor is decrease in the size of the dataset with every similarity threshold. Initially the dataset contains 40 authors with 1599 APKs, whereas, the dataset with a similarity threshold of 65 contains 20 authors with only 270 APKs. Fewer features are available to represent an author's style with a lower number of APKs per author. Thus,

the performance of the attribution system is lower on a smaller dataset. Nonetheless, our system performance is quite promising even with the small amount of data and lower number of APKs. For example, even at the similarity threshold of 65%, the system maintains an accuracy of around 88% for *All* and *DEX* strings; whereas *unreferenced* and *application* strings are able to predict the author of an unseen APK with an accuracy of 84% and 78%, respectively.

5.3 Malware application dataset results

In this section, we discuss the performance of our system on the malware application dataset (section 4.1.2). This dataset contains a total of 10 Android authors with 262 malicious APKs. We performed the same set of experiments as previously discussed for the benign application dataset.

5.3.1 Comparison of different types of strings

We compared the performance of our system over various kinds of strings extracted form malicious applications. These results are shown in table 5.2. Similar to the performance of different kinds of strings over the benign application dataset, in the case of the malware dataset, the *All* strings approach performs better than all other kinds of strings due to the higher number of strings retained. However, the *unreferenced* and *application* strings provide satisfactory results even with a smaller amount of strings and lower training

Table 5.2: Performance comparison of different types of strings over the malware application dataset

String Type	Accuracy	Macro Average Precision	Macro Average Recall	Macro Average F1	Average Training Time (Seconds)
All	96.03%	97.00%	96.16%	95.99%	20.80
DEX	95.78%	96.80%	96.04%	95.78%	21.44
Unreferenced	94.76%	95.93%	94.92%	94.70%	8.09
Application	81.63%	85.21%	84.08%	82.12%	0.33

time. The system shows better results over the benign application dataset (discussed in the previous section) than the malware application dataset. This is because the malware application dataset is smaller in size than the benign application dataset. Thus, fewer samples are available for analysis of a given malware authors' styles. However, even with limited samples, our system produces good accuracy and demonstrates that it can effectively identify malware writers as well.

5.3.2 Effect of APK similarity

In this section, we discuss the effect of similarities between APKs on the malware application dataset.

To examine the effect of APK similarity in the malware dataset on system performance, we performed the same experiments as discussed for the benign application dataset (section 5.2.2). The DroidKin tool is used to remove

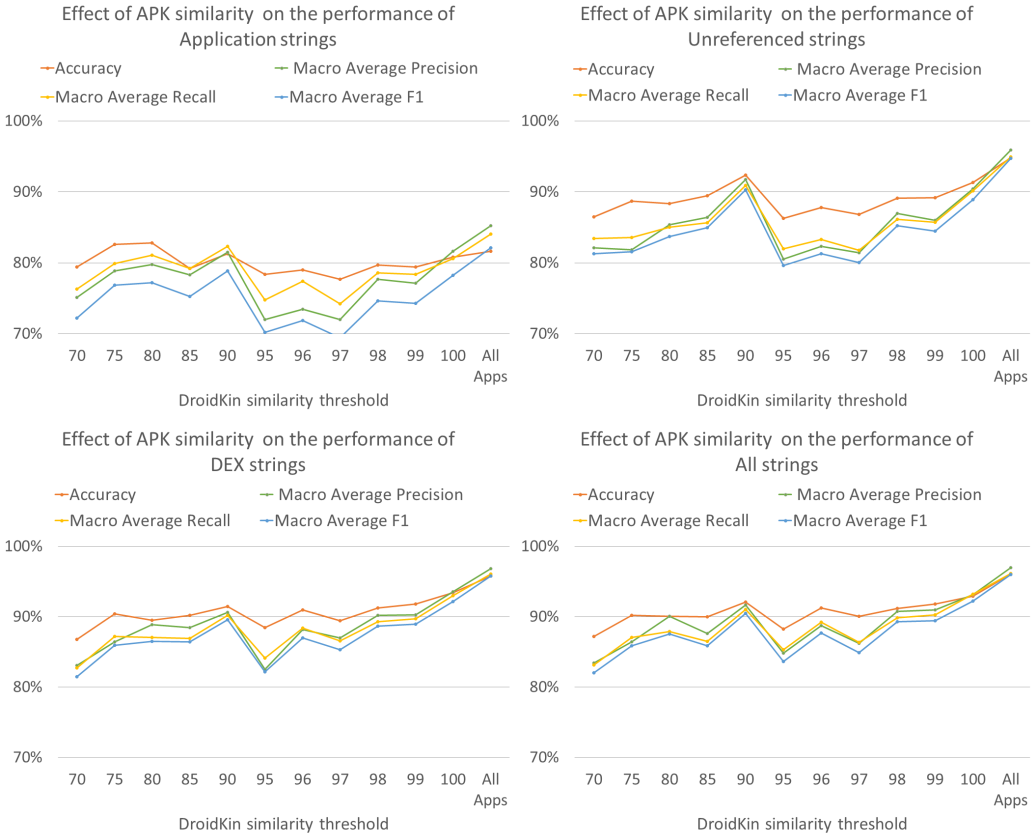

Figure 5.2: APK similarity threshold vs system performance on the malware dataset

similar APKs to generate the malware datasets at different levels of APK similarity. The results of these experiments are shown in Figure 5.2.

Results on the malware dataset follow similar trends to those on the benign application dataset; i.e., performance of the system degrades with the similarity threshold. However, the performance is quite volatile due to the small size of the dataset which is further reduced for the lower values of similarity threshold. For example, initially the dataset contains 10 authors with 262

APKs. Even at a similarity threshold of 100% (in this case only APKs having identical DEX files are removed), 81 APKs are removed from the original dataset leaving only 181 samples for the analysis. With every subsequent similarity threshold value, the number of APKs in the dataset is reduced, making the dataset smaller. Thus, we observe a few spikes in the performance charts shown in Figure 5.2. Nevertheless, even with a small number of samples, our system can still attribute malware authors. For example, at the 65% similarity threshold with only 55 APKs and 5 authors available in the dataset, the system was able to achieve accuracies of around 86% using *All* and *DEX* strings, around 84% for *unreferenced* strings and 77% for *application* strings.

5.4 Obfuscated application dataset results

In this section, we discuss the performance of our system over the obfuscated applications datasets (section 4.1.3). For this we collected a total of 96 source code Android projects belonging to 9 different authors from the GitHub repository. From these source code projects, we generated an unobfuscated dataset version along with 3 versions of obfuscated datasets using the ProGuard, Allatori and DashO Android obfuscation tools. We studied the system performance over these 4 datasets.

5.4.1 Comparison of different obfuscation tools over different kinds of strings

We compared the performance of our system over all 3 versions of the obfuscated datasets as well as the unobfuscated dataset. We also compared performance of all kinds of strings over these datasets. The results are presented in table 5.3.

Table 5.3: Performance comparison of different types of strings over the datasets obfuscated with different tools

Obfuscation Type	String Type	Accuracy	Macro Average Precision	Macro Average Recall	Macro Average F1	Average Training Time (Seconds)
No obfuscation	All	70.68%	64.31%	69.70%	64.79%	14.38
	DEX	70.57%	63.91%	69.30%	64.36%	14.26
	Unreferenced	69.68%	64.58%	69.07%	64.37%	4.95
	Application	62.20%	54.00%	57.96%	53.46%	0.02
ProGuard	All	76.96%	72.19%	75.78%	72.07%	3.68
	DEX	76.91%	71.87%	76.00%	71.94%	3.65
	Unreferenced	65.52%	59.15%	63.78%	59.19%	1.03
	Application	62.20%	54.00%	57.96%	53.46%	0.02
Allatori	All	70.00%	62.77%	68.93%	63.76%	12.97
	DEX	69.23%	61.86%	67.93%	62.69%	13.02
	Unreferenced	66.15%	59.61%	65.11%	59.84%	4.42
	Application	62.20%	54.00%	57.96%	53.46%	0.02
DashO	All	67.44%	59.96%	65.48%	60.30%	14.68
	DEX	67.42%	59.73%	65.15%	59.90%	14.66
	Unreferenced	64.26%	55.68%	62.63%	56.46%	5.08
	Application	62.20%	54.00%	57.96%	53.46%	0.02

As shown in the table, without any kind of obfuscation, our system is able to

predict the author of an Android binary with an accuracy of almost 71% for *All* and *DEX* strings, 70% for *unreferenced* strings and 62% for *Application* strings. The experiments performed until now highlighted the correlation between the size of the dataset and system performance. From the given dataset of only 96 APKs, the system was able to extract around 1.4 million *all* strings (this number was around 36 million for the benign dataset and 5.4 million for the malware dataset). Whereas, in the case of *unreferenced* strings, this number dropped to 400k, and for *application* strings it was only 2689 strings. The smaller number of strings is a likely reason why the system was not able achieve accuracy as high as the benign or malware application datasets. Another factor we must consider is that all the source code Android projects are downloaded from the GitHub repository, which is an open source collaborative platform. Even with all the verifications checks and precautions (discussed in the section 4.1.3), there is a possibility that a program is written by multiple authors or contains code components directly copied from some other source. We also ignored many of the source code projects for several reasons such as compilation errors, incomplete code and missing library files. Such excluded projects might contain important information about an author's writing style but were not considered in our analysis. All these factors could have led to the generation of noisy author profiles. Yet, the results are quite promising, and the system was able to predict authors with an accuracy of 71%.

In the case of the obfuscated dataset experiments, except for the *application*

strings, the system performance degrades slightly in the case of Allatori and moderately in the case of the DashO obfuscation tool. Obfuscation changes the names of various string components such as classes, methods, and fields to random strings. Along with the obfuscation, Allatori and DashO both have the provision of string encryption. Encryption encodes a given string to produce a new unreadable string. In both cases, the original string is converted to a random string. Such random strings are not informative for the authorship task and can produce noisy author profiles. Nevertheless, system performance is not affected significantly for the Allatori and Dasho obfuscated datasets, even after obfuscation and encryption of string components. The performance of the ProGuard obfuscation tool is quite interesting. In the case of *unreferenced* strings, the system performance is affected slightly as compared to the unobfuscated dataset. This is not surprising as ProGuard obfuscates various string components and this affects the classification performance. However, in the case of *All* and *DEX* strings, the ProGuard dataset results are better than for the unobfuscated dataset. The way in which the ProGuard obfuscation operates may be the reason of such an unusual performance. With the default obfuscation settings, ProGuard extensively removes various unused code components such as debug information and unused code. This is reflected in the number of strings available for analysis in the ProGuard obfuscated dataset. For example, the number of *All* or *DEX* strings extracted in the case of all other datasets except ProGuard (unobfuscated, Allatori, and DashO) is around 1.4 million. In the case of the ProGuard ob-

fuscated dataset, this number reduced drastically to only 377k strings. This might have led to removal of noisy data from the samples, and therefore focusing on only relevant features representing an author's style. Unlike *unreferenced* strings which mostly contain only user-defined code components, *All* and *DEX* strings contain library and third party classes as well. Hence, removal of redundant information could have eliminated strings that are not very informative for authorship analysis. Also, the size of the datasets is rather small and thus a few different classification predictions can influence accuracy.

As none of the obfuscation tools used for our analysis obfuscate *application* strings, the system has the same performance for this type of strings over all of the obfuscated and the unobfuscated datasets.

To study the performance of the system over the obfuscated datasets with varied levels of APK similarity, APKs having higher similarity score than the given similarity threshold must be removed from the dataset. Thus, the number of APKs in the dataset is reduced with the similarity threshold levels. As the number of APKs is already relatively small (96 APKs only), this would lead to a very small dataset, and the results over such a small dataset would not be very informative. Hence, we did not include this analysis in our thesis.

5.5 Discussion

Our proposed Android authorship attribution system is quite promising in many respects. Our results showed that we can predict the author of an unknown binary just by analysing string components at the binary level. The system proved to be very efficient as it can handle many android samples with a relatively low classification training time. Even though *All* and *DEX* strings perform better than *unreferenced* and *application* strings, both *unreferenced* and *application* strings can effectively identify the author of an APK with even lower training time. Our system works efficiently on any kind of dataset including benign, malicious and obfuscated applications. We should also note that the size of the dataset plays a crucial role in the attribution task. With a larger number of training samples, the system can generate stronger author profiles and can perform even better.

In order to mislead our proposed authorship attribution system, malware authors can disguise their identity by employing methods such as renaming the strings to some random variables, mimicking the coding style of a legitimate author or addition of redundant code components. However, an APK contains a number of strings that can reveal the identity of an author. For example, in the case of the benign dataset, our system analysed around 23 thousand strings per APK. Altering each of these strings will a challenging task. Thus, techniques used for hiding the author identity will require a considerable amount of time and efforts. We should also acknowledge the fact

that malware authors can use different developer certificates to sign their malware applications. As we have used the certificates to group the applications generated by a particular author, in such cases, our system can end up generating multiple profiles for a single author. However, such cases are out of the scope of our analysis.

Chapter 6

Conclusion

In this chapter, we discuss the contributions made by this thesis followed by the discussion of future work in this field.

6.1 Summary of contributions

In this thesis, we presented a lightweight, efficient system to identify the author of an Android binary by leveraging string components present in the application. The system generates author profiles by extracting word level n-grams from the strings. These profiles are then used to identify the author of an unseen Android binary. To the best of our knowledge, this is the first attempt to design an Android authorship attribution system leveraging string data within APKs.

The number of malware samples targeting the Android platform is increas-

ing rapidly [32]. Thus, an automated system that can effectively detect such malware samples is needed. Our thesis proposed a solution to detect such Android malware samples. As our system generates profiles of Android authors to predict the author of an unknown application, Android applications attributed to malware author profiles by our system can be flagged as malicious. Our results demonstrated that our system can effectively attribute benign as well as malware Android applications.

Besides our main contribution of developing the attribution system to identity benign as well as malware authors, we compared the performance of different kinds of string components such as *all*, *DEX*, *unreferenced* and *application* strings for the task of Android authorship attribution. In terms of system accuracy, *All* and *DEX* strings outperform *unreferenced* and *application* strings. However, *unreferenced* and *application* strings substantially reduce the overall time required for training the system, while maintaining fairly good accuracy.

We tested the Android authorship attribution system over obfuscated applications. We studied the effect of obfuscation tools such as ProGuard, Allatori and Dasho on our proposed system. The system performed quite well even in the presence of obfuscation.

We demonstrated the relation between the size of the datasets and the system performance. We also presented results over datasets with varied levels of APK similarity.

Another important contribution of our work is the datasets used for our

analysis. In the authorship attribution domain, the lack of open benchmark datasets is a serious challenge faced by researchers. In our thesis, we build 3 different datasets to study benign, malware and obfuscated Android applications and these are the only currently existing datasets for Android authorship attribution. These datasets will be helpful for further research in Android authorship attribution.

6.2 Future work

There are many opportunities to extend our research as listed below:

- In this work, we employed a linear SVM classifier. The system performance can be evaluated further using a variety of other classifiers such as multinomial Naive Bayes, k-nearest neighbour and logistic regression.

- Different feature selection strategies such as *chi-square*, *information gain* and *pointwise mutual information* can be implemented to select the highly informative features and to reduce total number of features used for the analysis. Linear SVM with the One vs rest classification strategy generates feature coefficients for the classification task. These coefficients can also be used to select the highest ranked features.

- The string extraction strategy can be improved further by incorporating different natural language processing techniques such as stemming and

case folding. Case folding converts all the strings to either lower case or upper case, whereas stemming reduces strings to their root words. Such text normalization techniques enable more standard forms of strings to be available for analysis.

- Attempts should be made to add further Android authors and new APKs to the current datasets, especially for the malware and obfuscated datasets, which currently contain fewer samples than the benign application dataset. Also, in the case of the obfuscated dataset, the effect of many other Android obfuscation tools such as DexGuard, JODE and Jshrink could be studied by generating obfuscated applications using these tools.

- The current system is designed to solve a closed world problem, i.e., an unknown Android application can only be attributed to one of the authors from a set of known candidate authors. The system currently employs a supervised classifier for the attribution task. Supervised classifiers require a labelled training dataset. However, in the real world, the author of many Android applications is unknown. Thus, unsupervised classification techniques which do not need a labelled training dataset could be used. Various clustering algorithms such as k-means algorithm and support vector clustering could be applied to form clusters of Android authors' applications. The system could be enhanced further by transferring knowledge obtained in the supervised domain to

the unsupervised domain, i.e., by generating clusters of existing known authors in the dataset, and then forming clusters of unseen samples belonging to unseen Android authors.

Bibliography

[1] Alex Aiken et al., *Moss: A system for detecting software plagiarism*, University of California–Berkeley. http://theory.stanford.edu/ aiken/moss/ **9** (2005).

[2] Allatori, *Allatori java obfuscator*, http://www.allatori.com/, accessed July 13, 2017.

[3] Mohamed Aly, *Survey on multiclass classification methods*, 2005.

[4] Abdullah J Alzahrani, Natalia Stakhanova, Hugo Gonzalezand Ali, and A Ghorbani, *Characterizing evaluation practices of intrusion detection methods for smartphones*, Journal of Cyber Security and Mobility **3** (2014), no. 2, 89–132.

[5] AppBrain, *Number of available applications in the google play store from december 2009 to march 2017*, https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, accessed June 16, 2017.

[6] A. Apvrille, *Guns and smoke to defeat mobile malware*, In Hashdays Conference., 2012.

[7] ———, *Playing hide and seek with dalvik executables*, In Hacktivity, 2013.

[8] Axelle Apvrille and Ruchna Nigam, *Obfuscation in android malware, and how to fight back*, Virus Bulletin (2014), 1–10.

[9] A. Armando, R. Carbone, G. Costa, and A. Merlo, *Android permissions unleashed*, 2015 IEEE 28th Computer Security Foundations Symposium, July 2015, pp. 320–333.

[10] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens, *Drebin: Effective and explainable detection of android malware in your pocket.*, NDSS, 2014.

[11] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji, *A methodology for empirical analysis of permission-based security models and its application to android*, Proceedings of the 17th ACM Conference on Computer and Communications Security (New York, NY, USA), CCS '10, ACM, 2010, pp. 73–84.

[12] Xi'an Jiaotong University Botnet Research Team, *Andromalshare*, http://sanddroid.xjtu.edu.cn:8080/\#home, accessed July 13, 2017.

[13] Christopher JC Burges, *A tutorial on support vector machines for pattern recognition*, Data mining and knowledge discovery **2** (1998), no. 2, 121–167.

[14] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani, *Crowdroid: behavior-based malware detection system for android*, Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, ACM, 2011, pp. 15–26.

[15] Steven Burrows and Seyed MM Tahaghoghi, *Source code authorship attribution using n-grams*, Proceedings of the Twelth Australasian Document Computing Symposium, Melbourne, Australia, RMIT University, 2007, pp. 32–39.

[16] Steven Burrows, Alexandra L Uitdenbogerd, and Andrew Turpin, *Comparing techniques for authorship attribution of source code*, Software: Practice and Experience **44** (2014), no. 1, 1–32.

[17] Radhouane Chouchane, Natalia Stakhanova, Andrew Walenstein, and Arun Lakhotia, *Detecting machine-morphed malware variants via engine attribution*, Journal of Computer Virology and Hacking Techniques **9** (2013), no. 3, 137–157.

[18] Christian Collberg, Clark Thomborson, and Douglas Low, *A taxonomy of obfuscating transformations*, Tech. report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[19] Koodous community, *Koodous*, https://koodous.com/, accessed July 13, 2017.

[20] Corinna Cortes and Vladimir Vapnik, *Support-vector networks*, Machine learning **20** (1995), no. 3, 273–297.

[21] eMarketer and AP, *Number of mobile phone users worldwide from 2013 to 2019 (in billions)*, https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/, accessed June 16, 2017.

[22] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth, *Taintdroid: an information-flow tracking system for real-time privacy monitoring on smartphones*, ACM Transactions on Computer Systems (TOCS) **32** (2014), no. 2, 5.

[23] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken, *Apposcopy: Semantics-based detection of android malware through static analysis*, Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 576–587.

[24] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, Carole E Chaski, and Blake Stephen Howald, *Identifying authorship by byte-level n-grams: The source code author profile (scap) method*, International Journal of Digital Evidence **6** (2007), no. 1, 1–18.

[25] Yanick Fratantonio, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna, *Clapp: Characterizing loops in android applications*, Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, 2015, pp. 687–697.

[26] Marina L Gavrilova and Roman V Yampolskiy, *Applying biometric principles to avatar recognition*, Cyberworlds (CW), 2010 International Conference on, IEEE, 2010, pp. 179–186.

[27] Hugo Gonzalez, Natalia Stakhanova, and Ali A Ghorbani, *Droidkin: Lightweight detection of android apps similarity*, International Conference on Security and Privacy in Communication Systems, Springer, 2014, pp. 436–453.

[28] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard, *Information flow analysis of android applications in droidsafe.*, NDSS, 2015.

[29] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang, *Riskranker: scalable and accurate zero-day android malware detection*, Proceedings of the 10th international conference on Mobile systems, applications, and services, ACM, 2012, pp. 281–294.

[30] Maurice H Halstead, *Natural laws controlling algorithm structure?*, ACM Sigplan Notices **7** (1972), no. 2, 19–26.

[31] IDC, *Smartphone osmarket share, 2016 q3*, http://www.idc.com/promo/smartphone-market-share/os, accessed June 16, 2017.

[32] AV TEST The Independent IT-Security Institute, *The av-test security report 2015/2016*, Tech. report, 2016.

[33] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein, *Dr. android and mr. hide: fine-grained permissions in android applications*, Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices, ACM, 2012, pp. 3–14.

[34] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall, *A conundrum of permissions: Installing applications on an android smartphone*, pp. 68–79, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[35] Vlado Kešelj, Fuchun Peng, Nick Cercone, and Calvin Thomas, *N-gram-based author profiles for authorship attribution*, Proceedings of the conference pacific association for computational linguistics, PACLING, vol. 3, 2003, pp. 255–264.

[36] Stakhanova N Killam R, Cook P, *Android malware classification through analysis of string literals*, First Workshop on Text Analytics for Cybersecurity and Online Safety (TA-COS 2016), European Language Resources Association (ELRA), 2016.

[37] Jay Kothari, Maxim Shevertalov, Edward Stehle, and Spiros Mancoridis, *A probabilistic approach to source code authorship identification*, Information Technology, 2007. ITNG'07. Fourth International Conference on, IEEE, 2007, pp. 243–248.

[38] Ivan Krsul and Eugene H Spafford, *Authorship analysis: Identifying the author of a program*, Computers & Security **16** (1997), no. 3, 233–257.

[39] Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra, *A survey on security for mobile devices*, IEEE communications surveys & tutorials **15** (2013), no. 1, 446–471.

[40] Robert Layton, Paul Watters, and Richard Dazeley, *Automatically determining phishing campaigns using the uscap methodology*, eCrime Researchers Summit (eCrime), 2010, IEEE, 2010, pp. 1–8.

[41] Jiexun Li, Rong Zheng, and Hsinchun Chen, *From fingerprint to writeprint*, Communications of the ACM **49** (2006), no. 4, 76–82.

[42] Christian Lueg, *8,400 new android malware samples every day*, https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day, accessed June 16, 2017.

[43] Thomas Corwin Mendenhall, *The characteristic curves of composition*, Science **9** (1887), no. 214, 237–249.

[44] Mila Parkour, *contagio mobile malware mini dump*, http://contagiominidump.blogspot.ca/, accessed July 13, 2017.

[45] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al., *Scikit-learn: Machine learning in python*, Journal of Machine Learning Research **12** (2011), no. Oct, 2825–2830.

[46] Matteo Pomilia, *A study on obfuscation techniques for android malware*, (2016).

[47] The Android Open Source Project, *Configure apps with over 64k methods*, https://developer.android.com/studio/build/multidex.html, accessed July 13, 2017.

[48] ———, *Dalvik executable format*, https://source.android.com/devices/tech/dalvik/dex-format, accessed July 13, 2017.

[49] ———, *Publish your app*, https://developer.android.com/studio/publish/index.html, accessed July 13, 2017.

[50] ———, *Samples strings.xml*, https://developer.android.com/samples/BasicNetworking/res/values/strings.html, accessed July 13, 2017.

[51] ———, *Shrink your code and resources*, https://developer.android.com/studio/build/shrink-code.html, accessed July 13, 2017.

[52] ———, *String resources*, https://developer.android.com/guide/topics/resources/string-resource.html, accessed July 13, 2017.

[53] D Krishna Sandeep Reddy and Arun K Pujari, *N-gram analysis for computer virus detection*, Journal in Computer Virology **2** (2006), no. 3, 231–239.

[54] PreEmptive Solutions, *Dasho: Java & android obfuscator & runtime protection*, https://www.preemptive.com/products/dasho/overview, accessed July 13, 2017.

[55] Eugene H Spafford and Stephen A Weeber, *Software forensics: Can we track code to its authors?*, Computers & Security **12** (1993), no. 6, 585–595.

[56] Guard Square, *Proguard: The open source optimizer for java bytecode*, https://www.guardsquare.com/en/proguard, accessed July 13, 2017.

[57] Efstathios Stamatatos, *A survey of modern authorship attribution methods*, Journal of the Association for Information Science and Technology **60** (2009), no. 3, 538–556.

[58] Eric Struse, Julian Seifert, Sebastian Üllenbeck, Enrico Rukzio, and Christopher Wolf, *Permissionwatcher: Creating user awareness of application permissions in mobile systems*, Ambient Intelligence (2012), 65–80.

[59] Linda Sui, *Strategy analytics: Android captures record 88 percent share of global smartphone shipments in q3 2016*, https://www.strategyanalytics.com/strategy-analytics/news/strategy-

analytics-press-releases/strategy-analytics-press-release/2016/11/02/strategy-analytics-android-captures-record-88-percent-share-of-global-smartphone-shipments-in-q3-2016#.WUQA0u0tVz1, accessed June 16, 2017.

[60] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro, *The evolution of android malware and android analysis techniques*, ACM Computing Surveys (CSUR) **49** (2017), no. 4, 76.

[61] Yan Wang and Atanas Rountev, *Who changed you?: obfuscator identification for android*, Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, IEEE Press, 2017, pp. 154–164.

[62] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos, *Profiledroid: multi-layer profiling of android applications*, Proceedings of the 18th annual international conference on Mobile computing and networking, ACM, 2012, pp. 137–148.

[63] Wikipedia, *Obfuscation (software)*, https://en.wikipedia.org/wiki/Obfuscation_(software), accessed July 13, 2017.

[64] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu, *Droidmat: Android malware detection through manifest and*

api calls tracing, Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on, IEEE, 2012, pp. 62–69.

[65] Xuxian Jiang Yajin Zhou, *Android malware genome project*, http://www.malgenomeproject.org/, accessed July 13, 2017.

[66] Lok-Kwong Yan and Heng Yin, *Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis.*, USENIX security symposium, 2012, pp. 569–584.

[67] Liu Yang, Nader Boushehrinejadmoradi, Pallab Roy, Vinod Ganapathy, and Liviu Iftode, *Short paper: enhancing users' comprehension of android permissions*, Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices, ACM, 2012, pp. 21–26.

[68] Yajin Zhou and Xuxian Jiang, *Dissecting android malware: Characterization and evolution*, Security and Privacy (SP), 2012 IEEE Symposium on, IEEE, 2012, pp. 95–109.

Vita

Candidate's full name: Vaibhavi Satish Kalgutkar

University attended:
Master of Computer Science, University of New Brunswick, 2015-2017
Bachelor of Computer Engineering, University of Mumbai, 2009-2013

Publications: In progress