Code clone detection in obfuscated Android apps

A thesis submitted to the
College of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Ardalan Foroughipour

Permission to Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Disclaimer

Reference in this thesis to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the University of Saskatchewan. The views and opinions of the author expressed herein do not state or reflect those of the University of Saskatchewan, and shall not be used for advertising or product endorsement purposes.

Requests for permission to copy or to make other uses of materials in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building, 110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan S7N 5C9 Canada

OR

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9 Canada

Abstract

The Android operating system has long become one of the main global smartphone operating systems. Both developers and malware authors often reuse code to expedite the process of creating new apps and malware samples. Code cloning is the most common way of reusing code in the process of developing Android apps. Finding code clones through the analysis of Android binary code is a challenging task that becomes more sophisticated when instances of code reuse are non-contiguous, reordered, or intertwined with other code. We introduce an approach for detecting cloned methods as well as small and non-contiguous code clones in obfuscated Android applications by simulating the execution of Android apps and then analyzing the subsequent execution traces. We first validate our approach's ability on finding different types of code clones on 20 injected clones. Next we validate the resistance of our approach against obfuscation by comparing its results on a set of 1085 apps before and after code obfuscation. We obtain 78%-87% similarity between the finding from non-obfuscated applications and four sets of obfuscated applications. We also investigated the presence of code clones among 1603 Android applications. We were able to find 44,776 code clones where 34% of code clones were seen from different applications and the rest are among different versions of an application. We also performed a comparative analysis between the clones found by our approach and the clones detected by Nicad on the source code of applications. Finally, we show a practical application of our approach for detecting variants of Android banking malware. Among 60,057 code clone clusters that are found among a dataset of banking malware, 92.9% of them were unique to one malware family or benign applications.

Acknowledgements

Foremost, I would like to express my deepest gratitude to my adviser, Prof. Natalia Stakhanova, for her never-ending support, encouragement and patience. I consider myself lucky to have worked under her supervision, as she truly cares for her students, and her guidance has carried me through all stages of my graduate studies.

I would also like to thank Dr. Fazaneh Abazari and Dr. Bahman Sistany, for all their support, guidance, and contributions in this project.

I would also to thank all my thesis committee members for for their encouragement and insightful comments.

Finally, I would like to express my gratitude to my family, which without their support, I would not have made it.

Contents

List of Tables

List of Figures

List of Abbreviations

AOSP	Android Open Source Project
API	Application Programming Interface
ART	Android-Runtime
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CFO	Control Flow Obfuscation
DalvikVM	Dalvik Virtual Machine
DEX	Dalvik Executable
HAL	Hardware Abstraction Level
I/O	Input and Output
PDG	Program Dependence Graph

1 Introduction

Software applications are no longer built from scratch, rather, they are assembled from open source and third-party code components. A recent survey of 2,292 IT professionals found that 80 - 90% of a single application now consists of third-party code components [58]. Moreover, to expedite the software development process, programmers tend to copy pieces of source code directly from another source code with no or few modifications. In the literature this is often called *code cloning* or *code reuse* [72].

Unfortunately, this phenomenon can lead to an increase in software maintenance costs as it can cause software bugs to spread as defective codes are copied, and introduce new bugs to when a developer does not modify different samples of reused code. Furthermore, this can lead to software vulnerabilities if a vulnerable piece of code is cloned [72]. Due to these facts code clone detection has been widely studied as a means of detection of bugs and vulnerabilities in software products [37, 36, 28].

While code reuse is typically associated with legitimate software development, malware writers are no exception. The increasingly professional and profit-driven underground cybercrime communities provide ample opportunities for adversaries to collaborate. Like traditional software development, adversaries work in teams, reuse existing code and components, use third-party libraries and tools that effectively mask malicious program behaviour from analysis, and expedite malware development. Indeed, every new malware instance can be created at a faster pace by modifying the existing malware samples with a few tweaks or by reusing existing code components. Establishing the presence of third-party code at the level of malware code may facilitate efficient triage of unknown samples, accurate detection and understanding of malicious functionality.

This has motivated many software security researchers to employ code clone and code similarity detection approaches for malware detection in binary programs [29, 35]. Android applications are no exception and many researchers have studied detection of malicious apps through code clone detection approaches [14, 5, 53]. A number of existing studies focusing on Android platform investigated a simple form of malicious code reuse known as piggybacking, a popular vehicle for malware propagation through Android markets by hijacking benign apps and grafting malicious functionality in them [19, 73, 71, 38]. For example, official GooglePlay market was reported to host 2% of piggybacked malware, while on several unofficial markets this number was reaching 50% [32]. Yet, researchers have noted that piggybacking phenomenon is largely more simplistic than it was commonly believed, with malicious functionality often added as a standalone code, appearing in resource files where it is obvious for detection [46]. However, more sophisticated code reuse that integrates reusable components intertwining both legitimate and malicious code instances remains

challenging for detection.

From a software security point of view, effective code clone detection remains challenging as the instances of reused are broken into non-contiguous pieces (i.e., code with statements that do not occur as contiguous text in the program), reordered, and intertwined with other reused code or original code written by a developer through code compilation, optimization, and obfuscation. Code obfuscation has became a commonly used approach for both legitimate and malicious software to disguise their appearance, intent, and to protect themselves from reverse engineering and automatic program analysis techniques. Although code obfuscation is mostly used as a form of software protection in benign applications, it also plays a key role in creating transient and evasive malware invisible for detection. Effective code clone detection in presence of code obfuscation has already been studied in case of x86 binaries [26, 35, 24], however, it has remained unexplored for Android applications. To the best of our knowledge, this is the first work that focuses on obfuscation-resistant code clone detection for Android applications.

In this work, we present a effective and obfuscation-resistant byte-code level code clone detection for Android applications which is capable of finding cloned methods as well as finer-grained code clones. Our approach analyzes Smali code, a human readable format of Dalvik byte-code. The majority of previous studies leverage static analysis and propose leveraging the similarity of code statements or structural features of a program such as control-flow graph for detection of code clones. Yet, the wide use of obfuscation techniques renders most of these existing approaches ineffective.

To mitigate the effect of code obfuscation, we propose a hybrid approach which partially executes Smali code in order to achieve obfuscation resistance without the need to fully execute an Android application. We employ Smali code simulation on the methods in an Android apps that can act as an entry point to it. Thus, our approach analyzes only functional code that would normally be invoked during an actual execution of an app to both increase accuracy and performance. Our approach simulates the execution of Smali code to extract possible app execution traces. To analyze the traces, we employ multiple execution trace simplification strategies and program backward slicing to extract semantically significant parts of an trace execution. This approach allows us to slice code into smaller code segments which are then analyzed for potential similarity between apps. This approach allows us to recognize semantically identical and syntactically similar code clones often referred to as Type-3 in addition to some cases of syntactically different but semantically equivalent code clones, often referred to as Type-4 code clones even in presence of code obfuscation.

We validate our approach first by investigating the effectiveness of our approach on different types of code clones on 20 injected code clone examples and next measuring the obfuscation resistance of our approach on 1085 open-source Android applications, by comparing the results of our approach in presence and absence of obfuscation. We achieve 84%, 81%, 87%, 78% similarity on four obfuscated sets of Android apps compared to their non-obfuscated counterpart. We further perform code clone clusters detection among 1603 non-obfuscated applications to assess the ability of our approach in finding code clones. We detected 44,776 group of methods that were identical, or their similarity was greater than 80%.

We also performed a comparative analysis with another method-level clone detection tool, called Nicad [18]. When comparing our results with Nicad we observed that on code clones with high degree of similarity both approaches agreed on the detected clones. Moreover, our approach was also able to find many more code clones not detected by Nicad.

Finally, we show a practical application of our approach for detecting variants of Android banking malware. Our approach can find code clones with high similarity among applications' methods corresponding to the same malware family. Similar code clones derived for each banking malware family can be further used for detecting other samples of that family.

The following is a summary of our contributions:

- We introduce an approach for detection of non-contiguous code clones that can be employed for both at method level and code slice level in obfuscated Android apps through analysis of their execution traces. To the best of our knowledge, this is the first work that offers detection of code clones in obfuscated Android apps.

- As an alternative to purely static or dynamic analysis approaches for analyzing Smali code, we have developed an Smali execution simulator to extract possible execution traces of an Android app. This can help other Android security researchers to perform a hybrid analysis on Android applications.

- We have created multiple sets of obfuscated Android applications that can be further used by the security community in an effort to facilitate research in this area.

2 Background

This chapter provides the needed background on the Android OS architecture, Android applications, code obfuscation and code clone detection.

2.1 Android Architecture

Android is a mobile operating system maintained by Google under the Android Open Source Project (AOSP). Androids architecture consists of the Linux kernel, the hardware abstraction level (HAL), the native libraries, the Android-Runtime environment, the Android application framework layer, and the applications [3]. The Android operating system architecture is shown in Figure 2.1.

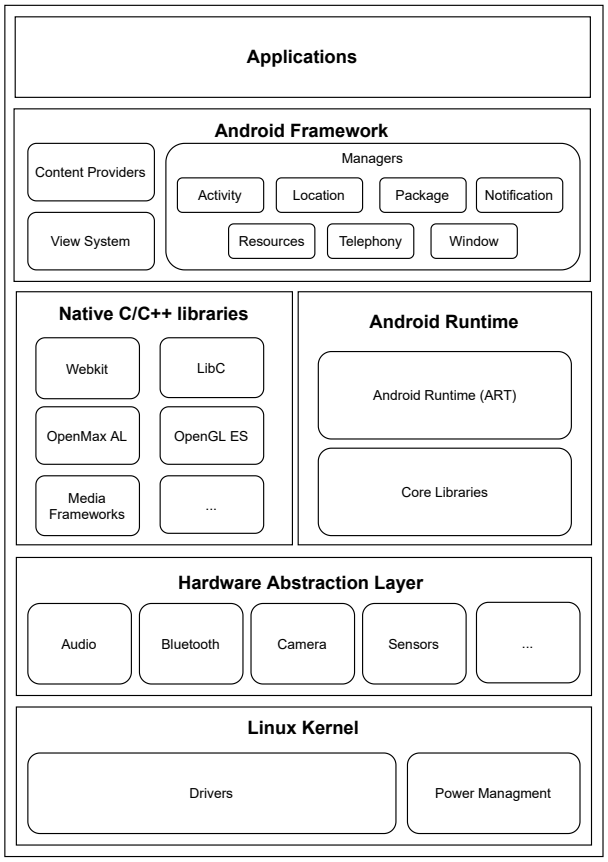

Figure 2.1: Android architecture stack

In Android, applications are executed by the Android-Runtime each in their own sandbox. Before Android

version 5.0, Android executed its applications using the Dalvik Virtual Machine (DalvikVM). DalvikVM is a register-based virtual machine specifically optimized for running on devices with limited computational resources. DalvikVM executes Dalvik byte-code, a simple and Java compatible byte-code proprietary to Google which allows Android application to take advantage of available common Java libraries. From Android 5.0, DalvikVM was replaced by Android-Runtime (ART). ART similar to its predecessor, can execute Dalvik byte-code, however, it introduces "Ahead-of-Time" compilation which generates a compiled executable from Dalvik byte-code. DalvikVM and ART both provide APIs for applications to execute native machine code and access native libraries' functionality. Android applications can take advantage of this feature to interact with Android native libraries, work with hardware such as Camera and Bluetooth or achieve higher performance for some critical tasks [1].

Furthermore, in Android an application is not a stand alone executable and works on top of the Android application framework. Android application framework defines four key components that applications use as their building blocks. These components consist of Activities, Services, BroadcastReceivers, and ContentProviders. Activities represent a user-interface that is shown in the device's screen with which a user can interact with. Activities can be launched by the Android system when an application is started, or they can be started by other components. Services are responsible for executing background tasks that do not need a user-interface. Services have to be started by other components such as activities. BroadcastReceivers are also important components that allow an Android application to be informed when a system-wide event occurs by listening to messages that are broadcasted for all apps. Many different events can create a broadcast such as when the screen has turned off or the battery is low, However, applications themselves can also initiate a broadcast. For example when an application wishes to notify other apps that it has finished downloading some data and it is now available for other apps to use. ContentProviders provide an interface for applications to share data between each other. Through content providers can query or modify data if the content provider allows it [2].

2.2 Android Applications

Android applications are packaged and distributed in the format of a .apk file. Each .apk file, when decompressed, includes one or multiple .dex files that contain the logic of an application in Dalvik byte-code. Dalvik byte-code is a high-level and object-oriented instruction set and thus, can be easily understood, modified, and even in some cases, decompiled back to the original Java code. Due to the security implications, code optimization and obfuscation are commonly adopted to make the analysis of Android programs more challenging. Dalvik byte-code can be converted to a human-readable format called Smali. Smali can be thought of as the equivalent of Assembly code for Dalvik byte-code.

2.3 Code Obfuscation

Code obfuscation is a technique to hinder the process of analyzing a program by making it more difficult and resource consuming to understand by a reverse engineer and program analysis tools. Obfuscation transformations are similar to compiler optimizations since they both perform semantic preserving transformations on a given code. Code obfuscation is commonly used as a method of software protection, however, it can also be used for hiding malicious code. Code obfuscation techniques can vary in their complexity, effectiveness, resistance against deobfuscation, and computational overhead [17].

Code Obfuscation traditionally has been widely used in both benign and malicious x86 binaries to hinder the process of reverse engineering. With the increasing popularity of mobile devices and the Android operating system in particular, obfuscation has become a common technique among Android applications. In Android applications, obfuscation can be both employed on the Dalvik byte-code of the application and any of the native codes used by an application. One of the reasons for the increasing usage of obfuscation in Android applications is the simplicity and high-level nature of Dalvik byte-code, which makes reverse engineering and repackaging of non-obfuscated Android applications very easy. Even for obfuscated Android applications, they can be relatively easily be analyzed and even deobfuscated by a reverse engineer provided given enough time.

2.4 Code Clone Detection

In software development, code clone detection is used to reduce the maintenance of code and facilitate detection of bugs [59]. From a security perspective, code clone detection is beneficial in vulnerability and malware detection [72]. In the literature, code clones have been categorized into four types [59]

- Type-1: Identical code pieces that are textually equivalent except for white spaces, comments, or layout characteristics.
- Type-2: Syntactically identical code pieces that can also differ in identifier names or literal values in addition to type-1 differences.
- Type-3: Syntactically similar code pieces with minor differences at statement level. The code pieces may differ by addition, removal, or modification of some statements in addition to type-1 and type-2 differences.
- Type-4: Syntactically different but semantically identical code clones.

In this work, we focus mainly on type-3 and also type-4 code clones that have been created by the application of code obfuscation to any type-3 code clones, a very challenging type of code clone detection. We demonstrate that our approach is capable of detecting any type-1, 2, and 3 code clones present in source code that through compilation and obfuscation often (but not always) transform to type-4 clones.

3 Literature Review

This chapter provides a succinct summary of previous researches done on the areas of code obfuscation, code clone detection, and application of code clone detection in software security.

3.1 Code Obfuscation

Code obfuscation can be defined as a series of code transformations that alters a code to another semantically equivalent code that is significantly more complex to analyze. Code obfuscation is widely used as a method of protecting a code base that needs to be shipped [17]. Code obfuscation can be used to hinder the analysis of code for both protecting intellectual properties of developers from reverse engineers and stopping malicious code from being detected by other security software such as anti-viruses. Code obfuscation has been tackled from many different perspectives in the literature. In this section, we discuss research done on code obfuscation regarding Java and Android applications.

Collberg et al. [17] presented one of the first and most significant works in this field. They studied different obfuscation transformations applicable for Java byte-code, classified them, and discussed the possible approaches that can be used for deobfuscation. They categorized code obfuscation into four groups: layout obfuscations, control obfuscation, data obfuscation, and preventive obfuscation. Layout obfuscations are transformations that remove the information left from source code in the compiled byte-code or binary code. Examples of such transformations are removing debug information or renaming identifiers. These transformations do not modify the content of a code and thus do not pose a great challenge for program analysis, however, they are one-way transformations since after removing original identifier names or debug information, the original code can not be recovered. Control obfuscations attempt to alter the control flow of a program. Control flow obfuscations encompass a variety of techniques such as reordering code, method inlining and outlining, insertions of dead or irrelevant code, etc. One key component of many control flow obfuscations is the use of opaque predicates, expressions whose value is known to the obfuscator but are difficult for the deobfuscator to infer their value. Data transformations are transformations that change the storage, encoding, aggregation, and ordering of the data. One example of data obfuscation is string encryption. Lastly, preventive transformations are defined as transformations that do not obscure a program but rather, they cause common tools and deobfuscators to not work properly. An example of these obfuscation techniques is modifying Java byte-code so that it cannot be properly decompiled to source code.

Many other studies also have been done regarding code obfuscation for Java and Android programs as

a means of software protection. Part of the previous studies have focused on studying current obfuscation techniques or even proposing new ones. *Memon et al.* [50] proposed two obfuscation techniques for Java programs which include un-letting the completion of statements and removing variable and method names. *Kumar et al.* [45] investigated Java control-flow obfuscations(CFO) which alter the control flow path of a program without changing its semantics. They concluded that there is a gap between theory and practice regarding CFO.

Regarding obfuscation for Android applications, *Kovacheva* [44] is one of the first works that studied obfuscation in Android applications. They showed that many of the applications available at official Android markets could be easily reversed. So they proposed obfuscator implementations which were able to defeat static analysis tools available to them. *Protsenko et al.* [54] proposed PANDORA, an Android byte-code obfuscator, which contained 7 data obfuscation techniques for obfuscating object-oriented programs. *Freiling et al.* [31] empirically evaluated the 7 different obfuscation techniques from PANDORA on 240 Android applications and evaluated using 8 software complexity metrics. *Balachandran et al.* [10] proposed 3 new control-flow obfuscation techniques for Android applications that were harder to be resolved by static analysis compared to previously proposed control-flow transformations. Their proposed obfuscation techniques were based on 1) packed-switch constructs, 2) try-catch constructs, and 3) combination of last two techniques. Recently, *Aonzo et al.* [7] proposed Obfuscapk, an obfuscation tool for Android which supports a variety of obfuscation techniques such as layout obfuscations, control-flow obfuscations, and string-encryption obfuscation. Their approach directly works at Smali code level and first disassembles an .apk file, applies transformations on its Smali code, and finally repackages the app.

Many other research have also be done on investigating the usage and effects of obfuscation among real world applications. *Linares-Vásquez et al.* [48] analyzed 24,379 free Android applications regarding the impact of presence of third-party libraries and code obfuscation on measuring code reuse in Android applications. *Maiorca et al.* [49] studied the effect obfuscation on the performance of anti-malware solutions. They concluded that first, manipulating strings in a benign Android applications, can cause anti-malware solutions to mark it as malware and second, anti-malware solutions even though, preform well under simple obfuscation techniques, they do no perform well under heavily obfuscated applications. *Wermke et al.* [63] performed a large scale investigation in the usage of obfuscation in 1.7 million applications from Google Play store. They found 24.92% of the applications were obfuscation by the developer. They also conducted a survey from 308 developers regarding use of obfuscation in their projects. They concluded that while developers are aware the their applications are at risk of plagiarism, the do not fear this issue much and even some of them had difficulty in applying obfuscation to their apps. *Dong et al.* [25] performed an study on usage of three different obfuscation techniques, identifier renaming, string encryption and use of Java reflection, on a dataset collected from Google Play and other third party markets. They showed that identifier renaming is widely used in apps found in Chinese markets, while more sophisticated obfuscation techniques such as string encryption and use of reflection are commonly used in Android malware.

Another group of studies have focused on detection of existence of obfuscation and also the obfuscation tool employed to perform the obfuscation on Android apps. *Wang et al.* [61] was the first to tackle the problem of obfuscation identification. They proposed a machine learning approach to detect what obfuscator has been used by extracting the obfuscation characteristics of a given .dex file. *Bacci et al.* [9] proposed a machine learning approach that aims to identify the obfuscation techniques (if any) that have been used in obfuscating an Android application. *Mirzaei et al.* [51] proposed an approach for detecting three popular obfuscation techniques in Android application, identifier renaming, string encryption, and control flow obfuscation. They used online learning techniques to make their approach more suitable for resource limited systems. *Mohammadinadooshan et al.* [52], proposed a generative classification approach for string encryption based on Naive Bayes which does not rely on positive samples for training which allowed them to achieve higher generalizability to different obfuscation schemes. For quantifying the quality of obfuscation in Android apps, *Cho et al.* [16] proposed an machine-learning based approach to evaluate the level of API hiding among Android apps.

Looking at adversarial approaches, some other researches have focused on the analysis of obfuscated Android applications and even deobfuscating them, however most of the previous research focused on one or a subset of obfuscation techniques (e.g. identifier renaming, string encryption and control flow). For deobfuscating identifier renaming, *Bichsel et al.* [12] proposed an approach based on a probabilistic model trained on thousands of non-obfuscated applications. *Baumann et al.* [11] proposed comparing obfuscated code of given application to dataset of non-obfuscated applications and replacing the modified names with their original names. In case of string encryption deobfuscation. *Yoo et al.* [68] proposed to dynamically execute Android application on a modified Dalvik VM which captures the executed Dalvik instructions and their runtime metadata which is then used to find the original value of obfuscated strings. *De Vos et al.*[22] proposed ASTANA, that reverts obfuscated string literals to their original value by executing the string deobfuscation logic inserted by the obfuscation tool. Regarding control flow deobfuscation, *You et al.* [69] investigated the effectiveness of ReDex optimizer, a Dalvik byte-code optimizer written by Facebook, for deobfuscating control-flow obfuscations applied by Obfuscapk. These studies on analyzing obfuscated Android applications and deobfuscation focus on particular obfuscations and do not generalize well for different obfuscation techniques.

3.2 Code Clone Detection

Code clone detection is a well-studied problem that has been examined both at the source code level, usually for program comprehension or plagiarism detection, and at binary code level for vulnerability analysis and malware detection [35].

Most of the works on source code clone detection employ text-based, token-based or tree-based approaches to find code clones. *Roy et al.* [18] proposed a lightweight text-based tool called Nicad. They start by parsing

given source files, then breaking them into functions or blocks of code, normalizing them, and finally using the longest common sequence (LCS) algorithm to detect code clones. *Kamiya et al.* [41] proposed CCFinder which employs lexical token sequences to detect code clones for codes written in C/C++, Java, and COBOL. *Sajnani et al.* [56] proposed SourcererCC, a scalable token-based code clone detection that is effective in finding code clone from large inter-project repositories. *Duric et al.* [60] proposed a token-based approach for finding similar source codes that have resulted from copied and then modified code. they start by tokenizing an input code, then removing some of the tokenized codes which have not been caused by copying code, and finally, using RKR-GST or Winnoing algorithm for similarity comparison. *Yang et al.* [66] proposed a token-based approach for finding code clones using token-sequences which also considered Java features such as lambda expressions, anonymous classes, and generics in its analysis. DECKARD [39] characterizes code using Abstract Syntax Tree (AST) subtrees with a vector of fixed-length which are then clustered using Locality Sensitive Hashing (LSH).

Another group of studies have focused on using graph-based approaches or slicing based approaches to improve code clone detection for non-contiguous code clones. The study by *Komondoor et al.* [43] was one of the first attempts to represent C code with PDGs and employing program slicing to find isomorphic sub-graphs in order to detect duplicated code. *Higo et al.* [34] proposed Scorpio, a PDG based code clone detection tool to detect non-contiguous code clones. They introduce multiple heuristics in their approach to reduce the time needed for code clone detection and improve detection of continues code clones.

Other studied have also investigated the use of learning-based approaches for code clone detection. *Sheneamer et al* [57] employed various types of classification models for detecting cloned method pairs using features extracted from AST and PDG of methods. *White et al.* [64] propose applying deep learning on both the lexical and syntactical information extracted from code for detecting code clone fragments. *Li et al.* [47] proposed CCLearner, a method-level token-based code clone detection tool that leverages deep learning to improve code clone detection effectiveness compared to previous approaches. *Saini et al.* [55] proposed using deep learning on various software metrics extracted from source files for detecting clones that are usually missed by other approaches. They also leveraged size of methods and its set of method invocations and field accesses as heuristics to improve scalability and performance.

Some works have also been done on semantic code clone detection. *Elva et al.* [27] proposed JSCTracker which considers two methods semantically similar if they would show similar Input/Output behaviour. They employed auto generated test cases to execute the potentially similar methods to run those methods on JVM and then capture the behaviour of method. Recently, *Alomari et al.* [6] proposed srcClone, a tool that used program slicing to find semantic code clone which is more scalable compared to previous works that use code slicing.

Regarding code clone detection in Android, to best of our knowledge, there has been only one study done by *Akram et al.* [4]. They proposed DroidCC, a code clone detection tool for Android apps which first uses dex2jar tool to convert an applications .dex file to a Jar file which is then decompiled to Java source code.

Next, the Java code is normalized by replacing names and literal values with a default value and Finally, the hash value of sliding windows on the normalized code is used to detect code clones. This approach fails to detect code clones for Android applications that cannot be decompiled to source code. Moreover, the effects of obfuscation may be present in the source code which is problematic for accurate code clone detection.

Given the fact that source code program is not always available for analysis, several researchers have ventured to detect code clone in binary code or byte-code at function or block level. *Keivanloo et al.* [42] proposed SeByte, a token-based clone detection tool for Java byte-code. They use a heuristic to detect the similarity between byte-code contents and rank the extracted code clones. *Farhadi et al.* [29] proposed BinClone, a binary code clone detection tool that detects cloned binary code pieces by first normalizing the instructions and then using a sliding-window to extract pieces of code called regions which are then stored in a database for further code clone detection. They also employed their code clone detection approach to investigate code clone in binary malware. *Farhadi et al.* [30] then proposed ScalClone, an extension on BinClone which improved scalability and recall rate in searching code clones. *Eschweiler et al.* [28] proposed extracting numerical (e.g., number of instructions, size of local variables) and structural (e.g., CFG of the function and basic blocks) features for code clone detection. *Ding et al.* [23] represented assembly codes as control flow graph and find code clones by finding similar sub-graphs between a given function and a repository. *Ding et al.* [24] then proposed employing a modified version of Word2Vec algorithm which is commonly used in NLP domain to improve the static representation of assembly functions. *Yu et al.* [70] proposed a Java byte-code code clone detection tool. They used sequences of instructions and method calls as features to find code clones across methods and code blocks. These approaches perform poorly under obfuscation since the extracted features can change drastically in obfuscated code. *Yang et al.* [67] proposed Asteria, a deep learning-based approach that learns semantic representation of binary functions by first decompiling the binary to source code and then extracting the source codes AST find semantically similar functions.

Another group of researchers have focused on finding semantically similar code in binaries which should not be affected by obfuscation. *Hu et al.* [35] proposed converting binaries to an intermediate representation and emulating selected codes with random values to extract semantic signatures. One major drawback of this approach is its limited code coverage since it relies on random values. To solve this problem, *Egele et al.* [26], proposed blanket execution, a dynamic analysis approach that captures side effects of binary functions executed with random values to find functionally similar functions. To improve code coverage, they also proposed repeated execution of binary functions from the first un-executed instruction until every instruction has been executed at least once. Even though this work solves the problem of code coverage, it introduces another flaw by capturing features from invalid executions which heavily distort the results. This type of dynamic analysis is a challenge in Android because Android applications are developed in an event-oriented manner, meaning the Android-Runtime will invoke methods in response to events such as when an Android device is disconnected from the internet. Moreover, Android applications all rely on user

interaction with the GUI which is a major obstacle for automated dynamic analysis.

3.3 Code clone detection application in security

There have been much work done on studying the application of code clone detection in Android applications for security purposes. Many of these work employ code clone detection for app clone detection based on the assumption that similarity between applications is a sign of piggybakced applications and thus can be used for detecting malware. Some other works have also studied use of code clone detection approaches directly applied on malware samples and marking other similar application as malware.

Crussell et al. [20] proposed DNAdroid, a tool for detecting Android application clones by assigning a similarity degree to two applications. To achieve this first they extract data dependency graph of all methods, excluding commonly used libraries based on their hashes and removing graphs less than 10 nodes, and using subgraph isomorphism to compare the graphs of methods with each other to measure method similarity, and finally calculating the similarity of applications base on the ratio of number similar methods to all methods of an application. Next year, *Crussell et al.* [21] proposed AnDarwin, which also uses dependency graphs which are then broken into multiple sub set of nodes which they refer to as semantic blocks. These semantic blocks are then converted into a vector which are then compared to each other using LSH algorithm in order to find and group similar vectors. Using their semantic vectors they also exclude common library code and lastly they use Jaccard Index to find similar applications. *Chen et al.* [15] proposed using 3D control flow graphs on Android applications byte-code for finding similar methods. Next the ration of similar methods to the number methods in an apps is used to measure app similarity. They also tried excluding applications show similarity because they have been developed by the same developer or share much library code based on the key to sign an application and the size of applications respectively. *Gonzalez et al.* [33] proposed DroidKin for Android application similarity detection based on first extracting features from the application code and also the applications meta information. For extracting features from code they employed opcode n-gram on the applications byte-code. Then they applied filtering to reduce the number of comparisons needed app clone detection. Lastly, they calculate the similarity of two applications based the intersection of their profiles. *Chen et al.* [14] investigated use of off the shelf code clone detection tools, namely Nicad on the Java code generated by decompiling Android .dex files, for malware detection. They analyzed more than 1000 malware applications from 19 malware families and demonstrated that they successfully could detect 95% of previously known malware families. *Alam et al.* proposed DroidClone [5], a code clone detection approach for malware detection in Android applications. Their work is one of the few works that analyzed both the Android application byte-code and also native code. They employed Control Flow Graphs (CFG) for creating a signatures at method and block level which are subsequently used for code clone detection. They evaluted their approach using 166 malware sample which were divided to a train and test set with the size of 136 and 30 samples. *Passeto et al.* [53] proposed StrAndroid, a method level code similarity detection

tool for Android applications. In their approach they start by analyzing a set of Android applications by first parsing the applications Smali methods, creating CFG for each method and extracting the CFG nodes known as basic blocks. In their work they ignore methods that are smaller than a minimum size or do not invoke any risky APIs. Next they perform backward slicing on interesting instructions of basic blocks to extract all data-dependent instructions which are then referred to as strands. Then these strands are compared for finding similar methods between apps. Finally they test their approach for malware classification on an Android ransom malware dataset.

Code clone detection in Android applications has been widely explored for detection of piggybacked and malware application, However, none of the proposed methods truly consider obfuscation as they only rely on static features that are extracted from code and neglect the semantics of an application.

4 Methodology

In this chapter, we shall explain our proposed method for analyzing obfuscated Android applications for finding code clones among them.

4.1 Overview

In this work, we take a path of code simulation to overcome the challenges of purely static or dynamic approaches. Approaches that rely on static analysis do not perform well with obfuscated code due to uncertainties of predicting the runtime code behaviour. Dynamic analysis techniques although less affected by obfuscation, given that code obfuscation is semantic preserving, require generation of a sufficient number of valid inputs to achieve high code coverage, the challenges that make the pure dynamic analysis not suitable for large code bases. This becomes more challenging for Android applications that rely heavily on user interactions with GUI.

To combine the best of both static and dynamic approaches, we proposed a hybrid technique that relies on simulating the execution of an app by partial execution of its Smali code to extract possible app execution traces enhanced with the runtime metadata for each instruction during simulation. These execution traces are then further analyzed to perform code clone detection. The benefits of analyzing the execution traces generated from Smali code simulation is twofold:

- First, each execution trace corresponds to one execution path which means all the instructions in an execution trace have been executed sequentially.
- Second, the added runtime metadata in each execution trace can help significantly in analyzing execution traces, particularly in case of obfuscated code.

Normally, extracting execution traces of a program is much challenging since it means a program needs to be executed multiple times with all valid inputs. To overcome this challenge we developed our Smali simulator which is a custom execution sandbox in the form of an Android application running in an emulated Android device, which given any Smali method and the inputs needed for its execution, it tries to execute the Smali code and capture its execution trace with its runtime metadata. This Smali simulator partially executes Smali instructions, meaning some instructions are executed and some are not. The insight behind this idea is that to execute instructions that are added by an obfuscator to achieve obfuscation resistance without needing to execute all the code of an Android application. One key challenge when analyzing Android applications

especially if obfuscated is that they can include much code that is not used by the application. This happens when applications include a library but only use a part of the library code or when an obfuscator injects dead code into an application to hinder the analysis process. To solve this problem we proposed to extract methods in an application that act as an entry point to it using an initial static analysis and then only analyzing those entry point methods. Figure 4.1 summarizes the flow of the proposed approach that follows four main steps:

1. Entry Point Extraction
2. Execution simulation
3. Execution trace analysis
4. Code clone detection

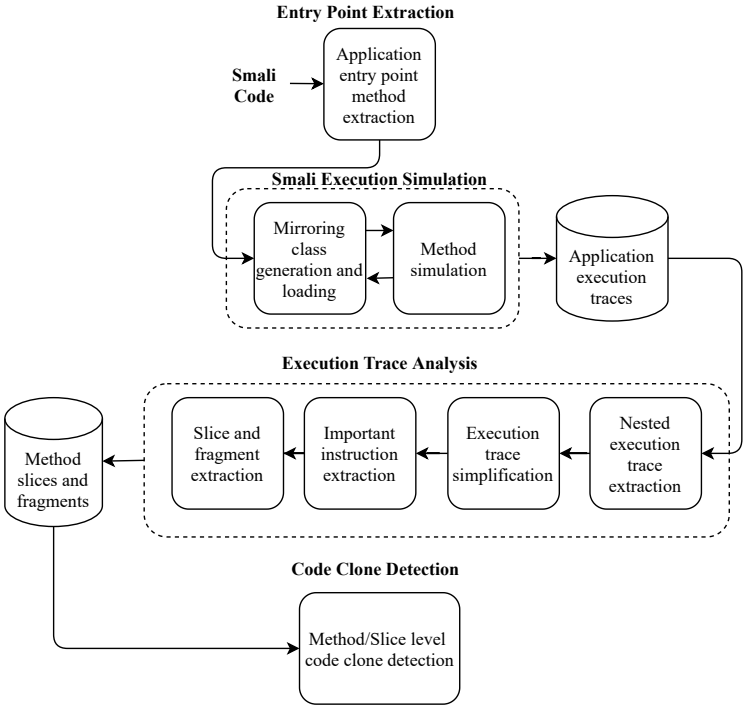

Figure 4.1: The flow of the proposed approach

4.2 Entry Point Extraction

Android applications unlike traditional Java programs do not contain a main method and are compromised of multiple entry points. Hence, the first step of our analysis is to extract the entry point methods to an Android application. Android applications consist of four key components as their building blocks. Each

component is an entry point through which the Android system or a user can interact with an application. These components contains Activities, Services, BroadcastReceivers and ContentProviders.

For an Android application to perform some task, it needs to override specific methods from the defined Android components. Each of the components can be thought of as an entry point through which the system or a user can execute application logic. Furthermore, these components can define and register callback methods that should be invoked in response to a particular event. These methods will be invoked by the Android-Runtime during execution. We define app's *Entry Point Methods* as a set of methods in an Android application that need to be invoked by the Android framework during the execution of an app.

Separating the entry point methods from the rest of the application methods results in a selective, yet efficient simulation and analysis of Android apps. This is necessary as Android applications often include residual and unused classes and methods due to including all the code in an imported library even if only part of it is used by the application or dead code added by an obfuscator.

Android entry point methods can be categorized into 3 broad groups:

1. Methods defined by the Android application components which are overridden by the developer and subsequently invoked by the Android system. Examples of these methods include:
   `android.app.Activity->onCreate(Bundle)`
   `android.app.Service->onBind(Intent)`

2. Methods defined as event handlers which are registered into the system during execution of an application. For example: `android.view.View.OnClickListener->onClick(View)`
   `android.view.View.OnFocusChangeListener->onFocusChange(View,boolean)`

3. Methods declared as the click event handlers in layout files which are usually used by an Activity of an application. In an Android application, a layout file is an XML file that defines the hierarchy structure for a user interface and consists of View and ViewGroup objects. A View usually represents elements that a user can see and interact with. For example, the element "<Button>" can declare an "android:onClick" property which should be invoked when an onClick event happens. By Android convention, the method corresponding to the value of "android:onClick" property should be public, return void, and define on parameter with the type of View. The class containing this method is the class of Activity hosting the layout.

Figure 4.1 shows a sample code of an application that calculates the factorial of a given number. This application consists of one Activity which contains one EditView, one TextView, and two Button objects. The corresponding layout file of this Activity is also shown in figure 4.2. In this example, the *onCreate* method is a type-1 entry point method which is invoked by the Android framework when this Activity has to be shown on the device screen, the *onClick* method set as the *onClickListener* of *calcFactorialBtn* is a type-2 entry point method which is invoked by the Android framework when the corresponding button is clicked on the screen, and the *resetTextView"* method is a type-3 entry point method which is invoked by the Android

framework when the reset button is clicked. This method is not explicitly set as the *onClickListener* for the reset button, however, it is implicitly set as the on click event handler in the layout file corresponding to this activity.

Figure 4.1: Sample factorial calculator app

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        EditText editText = findViewById(R.id.editText);
        TextView textView = findViewById(R.id.textView);
        Button calcFactorialBtn = findViewById(R.id.calculateButton);
        calcFactorialBtn.setOnClickListener(new OnClickListener() {
            public void onClick(View view) {
                try {
                    int num = Integer.parseInt(editText.getText().toString());
                    textView.setText("Factorial: " + calculateFactorial(num));
                } catch (IllegalArgumentException e){
                    textView.setText("Invalid Number");
                }
            }
        });
    }
    private static long calculateFactorial(int n){
        if(n < 0) throw new IllegalArgumentException();
        if(n==0 || n==1) return 1;
        return n*calculateFactorial(n-1);
    }
    public void resetTextView(View view){
        ((TextView) findViewById(R.id.textView)).setText("");
    }
}
```

Figure 4.2: Layout of sample factorial calculator app

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <EditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:inputType="numberDecimal"
        android:hint="Enter number"
        android:layout_marginTop="20dp"/>
    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text=""
        android:textAlignment="center"
        android:layout_marginTop="10dp"
        />
    <Button
```

```
        android:id="@+id/calculateButton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Calculate Factorial"
        android:layout_margin="8dp"/>
    <Button
        android:id="@+id/resetButton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Reset"
        android:onClick="resetTextView"
        android:layout_margin="8dp" />
</LinearLayout>
```

Extraction of most type-1 entry points is trivial as they correspond to component that should be declared in the app's "AndroidManifest.xml" file. One exception case for type-1 entry point methods is the entry point methods to BroadcastReceivers that are registered dynamically. Locating the entry points of dynamically registered BroadcastReceivers and type-2 and type-3 entry points is more challenging since they represent callbacks that are dynamically registered by the code of an application. Given that all of these callbacks should inherit from specific classes or implement specific interfaces and their method names should remain unchanged, it is possible to extract these entry points with some degree of error.

We design a heuristic to extract these entry points through static analysis of the app's Smali code and the corresponding layout files. The first step for extracting entry points of an Android application is to define a list of methods in the Android-Runtime that will be invoked by the Android system if overridden and registered to the system by an application. This list was compiled from previous research works that have already extracted some of the entry points of an Android application. [8, 13, 62]. Next, we parse all the Smali files of an application and check if they are overriding any of the methods specified in our list. If the overridden method belongs to either a Service, Activity, or ContentProvider component, we also check if the component is registered in the AndroidManifest.xml file. This is done to exclude entry point methods declared in any of mentioned components that are not used by the application. For entry point methods belonging to BroadcastReceivers and type-2 entry points, we take a more conservative approach by including all methods that override methods specified in our list of Android API entry point methods.

The type-3 entry point methods, which are the last of entry points, are the hardest type of entry points to extract. This is because extracting them needs a matching between layout declaring an onClick event handler for a view element to the Activity component that uses the layout. This matching becomes more challenging in cases that a Activity would contain sub-components called Fragments (a piece of an application's Activity) that would also register layouts at runtime. For extracting this group of entry points we also take a conservative approach, which is if the method defined as a onClick handler in a layout file, has a unique name in an entire application we also mark it as an entry point.

The algorithm of the entry point extraction is shown in Algorithm 1.

Algorithm 1 Extracting Android application entry points

```
1: function extractApplicationEntryPoints(appSmaliFiles,appManifestFile, appLayoutFiles)
2:     result ← []
3:     layoutOnclickListenerMethods ← extractOnClickHandlersInLayout(appLayoutFiles)
4:     for all methodName in layoutOnclickListenerMethods do
5:         methodSignature ← createOnClickListenerMethodSignature(methodName)
6:         if methodSignatureUniqueInAllSmaliFiles(methodSignature, applicationSmaliFiles) then
7:             smaliMethod ← getCorrespondingSmaliMethod(methodSignature, applicationSmaliFiles)
8:             result.add(smaliMethod)
9:     for all smaliClass in applicationSmaliFiles do
10:        if smaliClass.isAbstract then
11:            continue
12:        if smaliClass.extends("Landroid/app/Activity;") or smaliClsss.extends("Landroid/app/Service;") or smaliClass.extends("Landroid/content/ContentProvider;") then
13:            if !smaliClassDeclaredInManifets(smaliClass, appManifestFile) then
14:                continue
15:        for all smaliMethod in smaliClass do
16:            if isEntryPointMethod(smaliMethod) then
17:                result.add(smaliMethod)
18: function isEntryPointMethod(smaliMethod)
19:     if overridesComponentEntryPoint(smaliMethod) then
20:         return True
21:     else if overridesEventCallBackInterfaceMethods(smaliMethod) then
22:         return True
23:     return False
24: function extractOnClickHandlersInLayout(layoutFile)
25:     result ← []
26:     parsedLayout ← parseXml(layoutFile)
27:     for all element in parsedLayout do
28:         if "android:onClick" in element.attributes then
29:             result.add(element.attributes["android:onClick"].value)
30:     return result
```

4.3 Execution Simulation

Once all entry points of an app are obtained, they are simulated for extracting their possible execution traces. In order to simulate the execution a Smali method, we partially execute Smali instructions in an Android application running on an emulated Android device. Due to the dynamic features of the Android-Runtime such as reflection and method handles, the logic of all Smali instructions can be emulated at application layer which can be leveraged to create a custom execution environment for simulating Smali code. However, fully emulating the logic of an application is not possible in practice since this requires all correct values and objects needed for the execution of these instructions. Yet, to determine the similarity of two code pieces, it is often unnecessary to know the all precise values in an real execution and it is sufficient to determine values that are used by the applied obfuscation techniques in order for the obfuscation to be reversed.

This approach, in spirit, is similar to concolic execution since it combines real time execution of some parts of a code without executing all of the code. For capturing values that correspond to execution of instructions that we leave un-executed, we introduce a notion of *ambiguous value* that denotes any possible value of a type in the execution.

To extract possible execution traces of a method, our simulator takes the method's Smali code with its arguments and simulates the execution with the maximum threshold of 10 executions per method. In our current implemented prototype, we use ambiguous values for all the input values needed to execute a method. While this reduces the accuracy of simulation in some cases, we show empirically that this is sufficient to detect code clones. To realize our vision of a Smali simulator the twp main challenges have to be addressed:

1. creating and loading a class into the ART engine mirroring the types defined in the .smali files.

2. separating the methods that our simulator is allowed to execute from the methods we do not want to be invoked during the execution.

In the following, we discuss how we have tackled these challenges in our work.

4.3.1 Mirroring class generation and loading

One major challenge we face when simulating Smali code, is modelling user-defined types that are given to the simulator in the form of .smali files. This issue stem from the fact that Dalvik byte-code is a object oriented instruction set and the runtime behaviour of instructions heavily relies on types. This means user-defined classes need to be modelled in a way that can work intertwined with the existing Java or Android classes which dictates that the types declared in the .smali files need to be loaded into the ART engine the same way that would happen in the real execution of an application. To solve this issue, we have used the ByteBuddy framework [65], a dynamic Java class generator. With the help of ByteBuddy framework, we can dynamically create and inject a class to ART engine that mirrors the Smali class. The only difference would be that the methods and constructors inside the dynamically created types by ByteBuddy invoke a special method in our tool that simulates the execution of the method's original Smali instructions. With this approach, the Smali class is loaded to the ART engine which then can be treated as a normal Java class.

4.3.2 Selecting methods for partial execution

The logic of partial emulation is simple, i.e., any instruction that can return an invalid or unpredictable result when executed in a simulator should not be executed. For example,
`android.content.Context->getPackageName()` API if executed in our custom execution environment, would incorrectly return the name of the simulator and not the package name of an app under analysis. Similarly, method invocations that perform I/O operations may produce unexpected results.

To realize partial execution, we divided methods defined in the core library classes bundled with the Android-Runtime into *safe* and *unsafe* groups. A method that can be correctly executed in the context of the simulator or does not perform any I/O functionality is considered to be *safe*. Similarly, a safe class is a class that only includes only safe methods. Any other method defined in the Android-Runtime which is not considered as safe is considered to be unsafe and when an unsafe method is invoked its return value is modelled using an ambiguous value. Furthermore, all mutable objects passed to this unsafe method are also replaced with ambiguous values. Other instruction that operates on an ambiguous value usually result in a new ambiguous value to generated. Furthermore, If a conditional jump need to be taken based on some condition on ambiguous value, a execution branch path is chosen at random.

To reduce the propagation of ambiguous values in our simulation, we also separate functionally pure methods in the Android-Runtime core libraries which will refer to them as *pure* methods. Pure methods are

methods that do not change the state of passed arguments or the system, i.e., do not assign a new value to any field or invoke another method which would change state of an Object. Pure methods only calculate and return a new value based on the inputs such as toString() methods. Hence, by detecting pure methods, the unnecessary propagation of ambiguous value on method arguments is stopped.

To separate safe and unsafe classes and methods from the core library classes bundled with the Android-Runtime, we first extract the Android-Runtime core library files from an Android device and then converting them to Smali format. Then we create an initial set of safe classes manually and then we complement it with the new safe classes and safe methods by analyzing core library classes iteratively. The logic of this algorithm is simple, any method that only uses safe methods is safe and others are not safe, however this gets complicated by considering inheritance and polymorphism between types. To account for polymorphism and overridden methods, we divide safe methods into two sub-groups: 1) static safe methods, and 2) instance safe methods. The definition of static safe methods and instance safe methods are as follow:

1. *Static safe method* is a method where all its arguments are a either from a static safe class or a instance safe class. Furthermore, all the invoked methods inside this method body are safe. Consequently, a class is considered a static safe class if all of its static methods are safe. In this definition we assume all classes have a safe "<clinit>" function.

2. *Instance safe method* is a method which all of its arguments including the reference object are an instance safe class and all the method invocations inside the its body are safe. In this definition we also include constructors as a method. Consequently, a class is considered instance safe if 1) all of the methods it defines are safe, 2) its super class is safe and 3)all of its child types are also safe.

To create the list of safe methods and classes, we start by first finding static safe methods and classes and then we will use its results to extract instance safe methods and classes. For extracting instance safe methods and classes, one challenge is that the second and third requirement of safe classes are dependent on each other meaning a super class can be marked as safe only when its child classes are marked safe and the child classes are marked as safe when their super class is marked as safe. To solve this we break the logic of finding instance safe classes in two to parts, first we find classes that only use safe methods and also inherit from another safe class, and then analyze them one more time to separate classes that also meet the third requirement which is having all safe subclasses. Algorithm 2 depicts the process of extracting static safe methods and static safe methods and Algorithm 3 depicts the how instance safe methods and instance safe classes are extracted.

For extracting pure methods, similarly to the safe methods we start by manually creating a set of pure methods. Then, we proceed to find other pure methods by performing a static analysis on the extracted Smali code of Android framework core library classes. We define any method pure if it does not change any field value, does not write new values to an array, and does not invoke other non-pure methods. Algorithm 4 presents a flow for extracting pure methods.

Algorithm 2 Extracting static safe classes and methods

```
staicSafeClasses ← getInitialSafeTypesList()
staticSafeMethods ← []
instanceSafeClasses ← getInitialSafeTypesList()
instanceSafeMethods ← []

function extractStaticSafeClassesAndMethods
    smaliTypes ← parseAllJavaAndAndroidAPIsSmaliFiles()
    wasUpdated ← updateStaticSafeClassesAndMethods(smaliTypes)
    while wasUpdated do
        wasUpdated ← updateStaticSafeClassesAndMethods(smaliTypes)

function updateStaticSafeClassesAndMethods(smaliTypes)
    wasUpdated ← False
    for all smaliType in smaliTypes do
        if smaliType in staicSafeClasses then
            continue
        for all smaliMethod in smaliType.allStaticMethods do
            if smaliMethod in staticSafeMethods then
                continue
            if isSaticSmaliMethodSafe(smaliMethod) then
                staticSafeMethods.add(smaliMethod)
                wasUpdated ← True
        if smaliType.allStaticMethods in staticSafeMethods then
            staicSafeClasses.add(smaliType)

function isStaticMethodSafe(smaliMethod)
    if smaliMethod.definingClass in staticSafeClasses or smaliMethod in staticSafeMethods then
        return True
    if smaliMethod.isNative then
        return False
    for all argType in smaliMethod.argTypes do
        if !argTypeIsStaticSafe(argType) then
            return False
    for all smaliInstruction in smaliMethod.instructions do
        if smaliInstruction instanceOf invokationInstruction then
            if smaliInstruction instanceOf invokeStaticInstruction then
                if !isStaticMethodSafe(smaliInstruction.targetMethod) then
                    return False
            else
                if smaliInstruction.targetMethod.isNative then
                    return False
                if !smaliInstruction.targetMethod.isInheritedFromObjectType then
                    return False
    return True

function argTypeIsStaticSafe(argType)
    if argType.isPrimitive then
        return True
    if argType.isArray then
        toCheckType ← argType.baseArrayType
    else
        toCheckType ← argType
    if toCheckType in staticSafeClasses or toCheckType in instanceSafeClasses then
        return True
    return False
```

Algorithm 3 Extracting instance safe classes and methods

```
staicSafeClasses ← getInitialSafeTypesList()
staticSafeMethods ← []
instanceSafeClasses ← getInitialSafeTypesList()
instanceSafeMethods ← []
extractStaticSafeClassesAndMethods()
function extractInstanceSafeClassesAndMethods
    smaliTypes ← parseAllJavaAndAndroidAPIsSmaliFiles()
    semiSafeClasses ← instanceSafeClasses.clone()
    semiSafeMethods ← []
    wasUpdated ← updateInstanceSemiSafeClassesAndMethods(smaliTypes, semiSafeClasses, semiSafeMethods)
    while wasUpdated do
        wasUpdated ← updateInstanceSemiSafeClassesAndMethods(smaliTypes, semiSafeClasses, semiSafeMethods)
    instanceSafeClasses = createRawSafeClassesFromSemiSafeClasses()
    instanceSafeMethods = instanceSemiSafeMethods
    wasUpdated ← removeSafeAndPartialSafeClassesUsingNewUnsafeMethods()
    while wasUpdated do
        wasUpdated ← removeSafeAndPartialSafeClassesUsingNewUnsafeMethods()
function updateInstanceSemiSafeClassesAndMethods(smaliTypes, semiSafeClasses, semiSafeMethods)
    wasUpdated ← False
    for all smaliType in smaliTypes do
        for all smaliConstructor in allConstructors do
            if !isInstanceMethodOrConstructorSafe(smaliMethod, semiSafeClasses, semiSafeMethods) then
                continue
            else
                semiSafeMethods.add(smaliConstructor)
                wasUpdated ← True
        for all SmaliMethod in allMethods do
            if isInstanceMethodOrConstructorSafe(smaliMethod, semiSafeClasses, semiSafeMethods) then
                semiSafeMethods.add(smaliMethod)
                wasUpdated ← True
    return wasUpdated
function isInstanceMethodOrConstructorSafe(smaliMethod, inputSafeClasses, inputSafeMethods)
    if smaliMethod.isNative then
        return False
    for all argument in smaliMethod.arguments do
        if !instanceArgTypeSafe(argument.type, inputSafeClasses) then
            return false
    for all instruction in smaliMethod.instructions do
        if instruction instance invokeStaticInstruction then
            targetMethod ← invokeStaticInstruction.targetMethod
            if !(targetMethod.definingClass in staticSafeClasses) and !(targetMethod in staticSafeMethods) then
                return False
        else if instruction instance invokeInstruction then
            targetMethod ← invokeInstruction.targetMethod
            if !(targetMethod.definingClass in inputSafeClasses) and !(targetMethod in inputSafeMethods) then
                return False
    return True
function createRawSafeClassesFromSemiSafeClasses
    rawSafeTypes ← []
    for all semiSafeType in semiSafeTypes do
        if semiSafeType.allChildTypes in semiSafeTypes then
            rawSafeTypes.add(semiSafeType)
    returnSafeTypes
function removeSafeAndPartialSafeClassesUsingNewUnsafeMethods
    for all smaliClass in instanceSafeClasses do
        for all smaliMethod in smaliClass.methods do
            if !isInstanceMethodOrConstructorSafe(smaliMethod, instanceSafeClasses, instanceSafeMethods) then
                instanceSafeClasses.remove(smaliClass)
            else
                instanceSafeMethods.add(smaliMethod)
    for all instanceSafeMethod in instanceSafeMethods do
        if !isInstanceMethodOrConstructorSafe(smaliMethod, instanceSafeClasses, instanceSafeMethods) then
            instanceSafeClasses.remove(smaliClass)
function instanceArgTypeSafe(argType, semiSafeClasses)
    if argType.isPrimitive then
        return True
    if argType.isArray then
        toCheckType ← argType.baseArrayType
    else
        toCheckType ← argType
    if toCheckType in semiSafeClasses then
        return True
    return False
```

Algorithm 4 Extracting Android pure methods

```
1: function extractPureMethods
2:     pureMethods ← getInitialListOfPureMethods()
3:     allJavaAndAndroidSmaliTypes ← parseAllJavaAndAndroidSmaliFiles()
4:     wasUpdated ← updatePureMethods()
5:     while wasUpdated do
6:         wasUpdated ← updatePureMethods()
7:     for all pureMethod in pureMethods do
8:         if isMethodOveriddenByUnPureMethodInChildTypes() then
9:             pureMethods.remove(pureMethod)
10: function updatePureMethods
11:     for all smaliType in smaliTypes do
12:         for all smaliMethod in smaliType do
13:             if !smaliMethod.isAbstract And !smaliMethod.isNative then
14:                 for all instruction in smaliMethod.instructions do
15:                     if instruction instanceOf StaticFieldPutInstruction  then
16:                         continue
17:                     if instruction instanceOf InstanceFieldPutInstruction then
18:                         continue
19:                     if instruction instanceOf ArrayPutInstruction then
20:                         continue
21:                     if instruction instanceOf invokeMethodInstruction then
22:                         if !invokeMethodInstruction.targetMethod in pureMethods then
23:                             continue
24:                 pureMethods.add(smaliMethod)
25: function isMethodOveriddenByUnPureMethodInChildTypes(smaliMethod)
26:     methodDefiningClass ← smaliMethod.definingClass
27:     for all smaliType in methodDefiningClass.allChildClasses do
28:         if smaliType.hasMethod(smaliMethod.signature) And (!smaliMethod in pureMethods) then
29:             return True
30:     return False
```

After solving class loading challenge and separating safe methods and pure methods, we can implement our Smali simulator. Our simulator helps us to generate possible execution traces of a Smali method enhanced with the corresponding runtime metadata. Essentially each execution trace captures a sequence of instruction corresponding with one execution path with some information on the values used during execution of each instruction. Figure 4.2 depicts a sample code with one possible execution trace of it. In this example due to lack of space the runtime information are not included.

4.4 Execution Trace Analysis

For a given app, the simulator generates possible execution traces. These raw execution traces require further analysis to extract semantically significant parts of code for detection of code clones. The analysis of execution traces consists of four steps as follow.

4.4.1 Nested execution trace extraction

The first step of analyzing execution traces is to separate all the nested execution traces in our execution traces. During the simulation of a method, if the method invokes another user-defined method, the execution trace of the called method is also included in the generated execution trace of the caller method. In the later stages of our execution trace analysis, we consider all the instructions in an execution trace as if they all were executed by the outermost method. Thus, in order to analyze nested execution traces for code clone detection, we extract them and analyze them independently.

```
// App: com.gabm.fancyplaces:9
package com.gabm.fancyplaces.functional;

class LocationHandler {

...
protected Boolean
isValidLocation(Location location){
     if (location == null) {
         return false;
     }
     Time now = new Time();
     now.setToNow();
     return (now.toMillis(true) -
location.getTime()) <= TWO_MINUTES;
}

…

}
```

```
const/4 v1 1 &0
const/4 v2 0 &1
if-nez v9 @6 &2
new-instance v0 Landroid/text/format/Time; &6
invoke-direct v0 Landroid/text/format/Time;-><init>()V &7
invoke-virtual v0 Landroid/text/format/Time;->setToNow()V &8
invoke-virtual v0 v1 Landroid/text/format/Time;->toMillis(Z)J &9
move-result-wide v4 &10
invoke-virtual v9 Landroid/location/Location;->getTime()J &11
move-result-wide v6 &12
sub-long/2addr v4 v6 &13
const-wide/32 v6 120000 &14
cmp-long v3 v4 v6 &15
if-gtz v3 @20 &16
invoke-static v1 Ljava/lang/Boolean;->valueOf(Z)Ljava/lang/Boolean; &17
move-result-object v1 &18
goto @5 &19
return-object v1 &5
```

Figure 4.2: A sample method and one possible execution trace

4.4.2 Execution trace simplification

The next step is to normalize and simplify the execution trace to purge any side effects obfuscation has had on the execution trace. The applied simplification and normalization are essentially a series of transformations that are performed on the obtained execution traces to remove the differences caused by compilers, optimizations, and obfuscations. Applying simplifying transformations on execution traces is generally simpler than applying them on code since first, an execution trace is a sequential set of instructions and thus much easier to analyze than code and second, our execution traces containing runtime information of each value in the registers used by each instruction. The simplification transformations we perform are:

- *Execution trace flattening*: In this transformation, an execution trace is modified so that the instructions in nested traces can be treated as if they were executed in their parent executions. To achieve this the register number used by nested execution traces has to be updated to new unused register numbers. In this transformation, we assume an infinite number of registers exists and any register can be addressed in any instruction. Furthermore, the invocation call in the parent execution trace and the return instruction in the nested trace is removed and instead of them, new move instructions are added to connect the data dependency between nested and parent execution trace.

- *Removing register aliasing*: In this transformation we any register aliasing in the execution trace that have resulted from move instructions. This is necessary since we need to perform backward slicing later which to perform correctly needs the execution trace to not include any aliasing between registers.

- *Reflection normalization*: In this transformation we need to replace any method invocation done using reflection with a normal method invocation. This transformation is not trivial since by changing the method invocation from reflection to a normal method invocation we need to use new registers and add

move instructions in the correct places to connect the data dependency from instructions that create the argument values to the new added registers.

- *Constant transformation simplification*: In this transformation, try to find any method invocation that takes constant primitive values or String as inputs and computes and returns a new primitive value or String based on the inputs. After finding such methods we replace the consequent Smali "move-result" instruction in the parent execution with a "const" Smali instructions.

4.4.3 Important instruction extraction

The third step of analyzing execution traces is separating their *important instructions*. Knowing that not all instructions in an execution trace bear the same significance, our first task is to define what type of instructions we consider semantically important. This is also a key step to only extract semantically important parts of an execution trace in order to eliminate the differences in code instances caused by different compiler versions, compiler optimization, and obfuscation. When analyzing execution traces we informally define an important instruction as any instruction that can affect the system outside of the method's scope. We formally define an important instructions as:

- A methods return instruction which does not void.
- Changing the value of a static fields of classes.
- Changing the value of an instance fields of an object.
- Invocation of Java/Android API methods that change global state such as reading from Input or writing to Output.
- Invocations of Java/Android API methods that change the state of an object not created in local method scope.

Separating the return instruction of a method or instructions that assign a new value to a field is straightforward, however, separating method invocations that can generate side effects is much more difficult. In our work for finding Java/Android API invocations that change the global state of an application such as printing some value or writing a value to a file, we reused our list of safe classes and safe methods. Essentially in our current implementation, any method invocation on unsafe methods is considered to cause a side effect. For method invocations that change the state of a non-local object, we reused our list of pure methods. In this case, any non-pure method invocation on a non-local object is also considered to change its state. For example, adding a new value to an ArrayList object that has not been instantiated inside a method. In the Figure 4.2 the important instruction has been marked as bold.

4.4.4 Slice and fragment extraction

Backward slicing

Each important instruction depicts a single side effect seen in an execution trace of a method. Each important instruction usually depends on multiple instructions executed before it. By extracting all instructions that are needed for one important instruction to be executed based on the data-dependency, we can extract a set of instructions that correlate with a single behaviour in execution trace. The data-dependency between instructions is computed based on def-use chain of registers used in the important instruction and its dependent instructions. We use backward slicing on data-dependent instructions of an execution trace for two reasons: 1) control-flow can be modified by obfuscation which would be difficult to simplify using only execution traces. 2) Because Smali is an object-oriented instruction set, data-dependency between instructions includes significant semantic information.

Fragmenting slices

Our next step is to break the slices to smaller pieces in order to capture small but semantically important code segments that can be shared between different methods. In an slice, the instruction are operating on multiple registers containing different objects or primitive values. The goal here is to separate instructions that correlate with a single value into a single group that we refer to as a *Fragment*.

Fragment normalization

After separating our fragments we apply different normalizations on them to remove minor differences caused by different literal values, user-defined type names, register numbers, and repeated instruction caused by loops and recursion. In this part, we remove all literal values and the used register numbers for each instruction. Furthermore, we remove type identifiers from instructions if they are not user-defined. Finally, we discard any repeated instructions in a fragment to remove any differences in fragments caused by loops and recursion. After normalizing the fragments we save the results for further analysis. Figure 4.3 shows the slices and fragments extracted from the important instruction of the execution trace in Figure 4.2.

4.5 Code clone detection

Once different methods have been analyzed and summarized in the form of slices and their corresponding fragments, the next step is to extract code clones. In this works we propose code clone defection at two levels, method-level and slice-level.

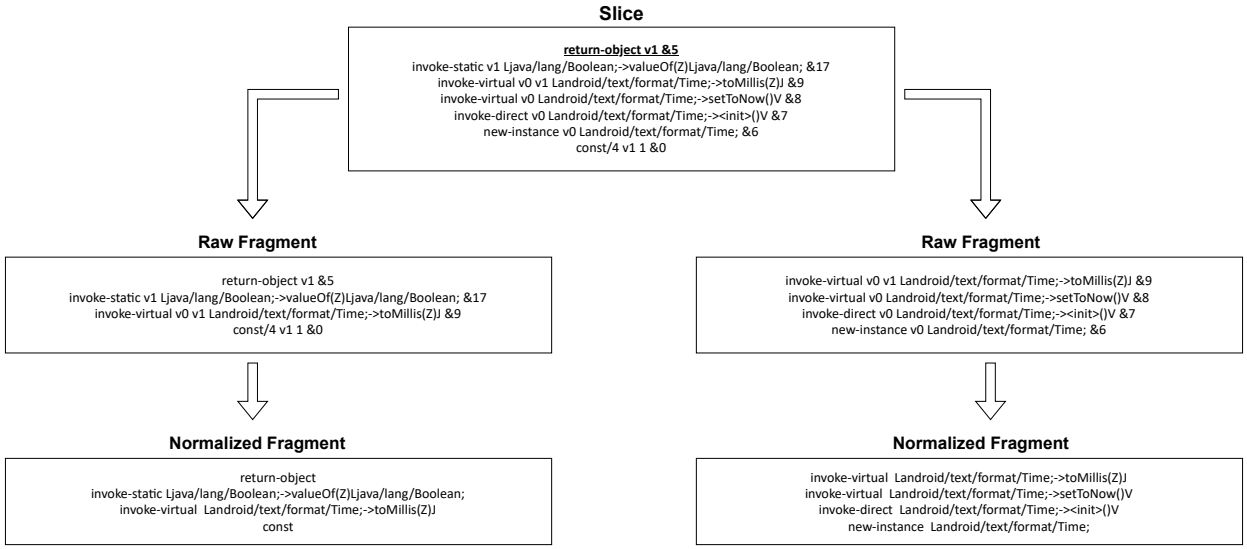

Figure 4.3: Sample slice and fragments

4.5.1 Method-level code clone detection

We measure the similarity between methods by employing the Jaccard index on the sets of fragments extracted from two methods. We consider a pair of two methods that show similarity above the specified threshold as a clone. Given two sets of fragments, A and B, the Jaccard index of these sets is calculated as follows:

$$JaccardIndex = \frac{|A \cap B|}{|A \cup B|} \tag{4.1}$$

4.5.2 Slice-level code clone detection

For finding small pieces of cloned code which often are non-contiguous, between methods, one important challenge is to define what is the smallest piece of code that can be considered as a code clone. In this work, we propose to use our slices as units we compare for finding small pieces of code clones. The reason for this is each slice represents a group of data-dependent instructions that perform one semantically significant task. To compare the similarity between slices, the Jaccard Index of their fragments can be measured.

5 Validation and Experiments

In this chapter, we elaborate on how we created the dataset we used for our experiments and how we validated our approach. Finally, we present our findings from the experiments we performed to investigate the presence of code clones in benign and malicious applications.

5.1 Setup configuration

Our simulator was implemented as an Android application for the Android version 26 or above. After installation, the simulator exposes APIs over the network which can be used to interact with it and perform Smali method simulation. After performing simulation on the methods needing to be analyzed, we copied their execution traces from the Android device to our computer and continue analyzing traces off-device. For our experiments, we installed our simulator on 6 emulated Android devices to perform our simulation in parallel, each device having 2 CPU cores and 4GB of memory.

5.2 Dataset creation

Due to the lack of a labeled dataset containing Android application code clones at byte-code level, with or without obfuscation, we ventured to create our set. From the open-source Android market F-Droid[1], we collected all the Android application source projects available to download at the time of creating our dataset, which compromised of 5045 projects, containing multiple versions for 1814 unique applications. Next, we attempted to compile these projects our self to create a consistent set of applications. To achieve this, first, we filtered apps containing code written in Kotlin, as our current implementation does not support applications written in the Kotlin programming language. We also removed applications that do not use the Gradle build tool, the default build tool used by Android Studio, the official Android application development environment. This was done due to the fact that the projects which not use Gradle as their build tool, could have not been compiled automatically. The remaining projects were compiled and those that were successfully compiled were used in our dataset for further analysis. A total of 1751 projects from 685 unique applications comprised our dataset. The summary of the employed dataset is shown in Table 5.1.

To obfuscate these applications, we employed Obfuscapk [7], a free and open source Android obfuscator that includes a range of obfuscation techniques such as control flow obfuscation, string encryption and

[1] https://www.f-droid.org/

Table 5.1: The summary of the employed dataset

Set name		**Num. of all apps**
Downloaded source code projects		5045
Successfully compiled projects		1751
successfully simulated applications		1603
successfully obfuscated applications	Control-flow obfusctions	1320
	Reflection obfuscations	1741
	String encryption obfuscations	1740
	All obfuscations	1085
Used obfuscated applications for evaluation		1085
Banking Malware applications		1076

reflection. Table5.2 depicts the obfuscation techniques provided by Obfuscapk.

To explore the obfuscation resiliency of our code clone detection approach, we obfuscated the available Android projects with various obfuscation techniques, creating four different sets of obfuscated apps. The rationality for creating four sets was to better understand the performance of our approach under different obfuscation techniques. The obfuscation transformations applied to each set and their corresponding order are shown in Table 5.3. Unfortunately, Obfuscapk failed to successfully obfuscate some of the apps. For the control-flow obfuscation set, Obuscapk was able to successfully obfuscate 1320 out of 1751 .apk files, the reflection was only applied to 1741 .apk files, and the string encryption to 1740 apps. To ensure a proper comparison, for the further evaluation experiments, we used 1085 apps in the intersection of all these sets. We successfully applied all obfuscation transformations to these 1085 apps. The summary of the obfuscated applications is shown in Table 5.1.

Table 5.2: Obfuscapk transformation

Type	Name	Description
Trivial	NewSignature	Re-sign the application with a new custom signature.
	NewAlignment	Realign the application.
	Rebuild	Rebuild the application.
Rename	ClassRename	Change the package name and rename classes (even in the manifest file).
	MethodRename	Rename methods.
	FieldRename	Rename fields.
Code	ArithmeticBranch	Insert junk code. In this case, the junk code is composed by arithmetic computations and a branch instruction depending on the result of these computations, crafted in such a way that the branch is never taken.
	CallIndirection	This technique modifies the control-flow graph without impacting the code semantics: it adds new methods that invoke the original ones. For example, an invocation to the method m1 will be substituted by a new wrapper method m2, that, when invoked, it calls the original method m1.
	DebugRemoval	Remove debug information.
	Goto	Given a method, it inserts a goto instruction pointing to the end of the method and another goto pointing to the instruction after the first goto; it modifies the control-flow graph by adding two new nodes.
	MethodOverload	It exploits the overloading feature of the Java programming language to assign the same name to different methods but using different arguments. Given an already existing method, this technique creates a new void method with the same name and arguments, but it also adds new random arguments. Then, the body of the new method is filled with random arithmetic instructions.
	Nop	Insert junk code. Nop, short for no-operation, is a dedicated instruction that does nothing. This technique just inserts random nop instructions within every method implementation.
	Reorder	This technique consists of changing the order of basic blocks in the code. When a branch instruction is found, the condition is inverted (e.g., branch if lower than, becomes branch if greater or equal than) and the target basic blocks are reordered accordingly. Furthermore, it also randomly re-arranges the code abusing goto instructions.
	Reflection	This technique analyzes the existing code looking for method invocations of the app, ignoring the calls to the Android framework (see AdvancedReflection). If it finds an instruction with a suitable method invocation (i.e., no constructor methods, public visibility, enough free registers, etc.) such invocation is redirected to a custom method that will invoke the original method using the Reflection APIs.
	AdvancedReflection	Uses reflection to invoke dangerous APIs of the Android Framework. To find out if a method belongs to the Android Framework, Obfuscapk refers to the mapping discovered by Backes et al.
Encryption	AssetEncryption	Encrypt asset files.
	LibEncryption	Encrypt native libs.
	ConstStringEncryption	Encrypt constant strings in code.
	ResStringEncryption	Encrypt strings in resources (only those called inside code).
Resource	RandomManifest	Randomly reorder entries in the manifest file.

Table 5.3: The applied obfuscation transformations.

Obfuscation set name	**Applied Transformations**
Control-flow obfuscation	1-CallIndirection 2-MethodOverload 3-ArithmeticBranch 4-Reorder 5-Goto 6-Nop
Reflection obfuscated	1-Reflection 2-AdvancedReflection
String encryption obfuscation	1-ConstStringEncryption
All obfuscations	1- All control-flow obfuscations 2- All reflection obfuscations 3- ConstStringEncryption

5.3 Code clone detection validation

For our first experiment, we examined the effectiveness of our approach in finding different types of code clones. In this experiment, we manually injected 20 code clones for 5 different code snippets to a sample Android application. For each code snippet, 4 code clones were created corresponding to a code clone type which was analyzed for code clone detection.

To create these clones we first created four code clone samples at source code as depicted in Figure 5.1 and then they were compiled to Smali code. However, since the Smali code is what matters for our approach and some differences in source code such as variable names are not present in the Smali code, we further modified some of the compiled Smali code to create correct code clones. For creating type-1 clones from the Smali code of these code snippets we removed the debugging information in the Smali code and for type-2 clones, we changed the register numbers in the code samples. For type-3 and type-4 clones, no further change was done to the Smali code since the differences in the source code were enough for creating type-3 and type-4 clones at the Smali code level.

With our code clone samples created and injected to a sample Android application, we measured the similarity of these code clones. Table 5.4 shows the results of our approach for the injected code clone samples. The similarity results shows that our approach is successful in finding type-1 to type-3 clones and also can detect some type-4 clones.

Table 5.4: Code similarity of added code clones

Code clone Name	Type-1 similarity	Type-2 similarity	Type-3 similarity	Type-4 similarity
appendTextToFile	100%	100%	100%	44%
isTreeFull	100%	100%	100%	100%
removeDuplicatedInIntegerArrayList	100%	100%	100%	0%
reverseNumber	100%	100%	100%	26%
removeMiddleNodeInLinkedList	100%	100%	100%	100%

5.4 Obfuscation resistance validation

To evaluate the obfuscation resistance of our approach against different obfuscation techniques we used our four obfuscated datasets (Table 5.3). When analyzing the applications in our dataset, for some of the apps our simulator failed to simulate some of their entry point methods. In our validation, we discarded any application that any of its methods failed to simulate properly. When manually analyzing the errors for such failures and we noticed most of the failures are caused by our tool failing dynamically creating classes using ByteBuddy because some Android applications contain classes with fields or method declarations referring to types that do not exist in the applications .dex files. This may cause a problem in our current implementation but it works fine in Android, since 1) the methods or fields with non-existent types are not invoked during execution and 2) the ART engine loads classes lazily. However, in our current implementation, when defining classes using ByteBuddy, we have to declare and load classes eagerly which when loading such classes, causes an error to be thrown.

Considering the original set of non-obfuscated apps as the ground truth for evaluating the resistance of our approach against obfuscation, for each obfuscation set, we compared the fragments of each method of each app with its corresponding fragments in the non-obfuscated set. If a method only exists in one of the non-obfuscated or the obfuscated set, it was ignored since it could not be compared properly. The reason for the existence of such cases is that our approach may traverse different execution paths in each analysis since it takes a random path when branching on Ambiguous values. In the case of reflection and string encryption since not all methods were affected by these obfuscations, when comparing the results of our approach on obfuscated and non-obfuscated applications we only considered the methods that were affected by these obfuscations.

The results of these comparisons are shown in Table 5.5. As the results show, string encryption has the highest similarity (87%), i.e., 87% of fragments from non-obfuscated set are similar to the application set obfuscated with String encryption. Among obfuscated sets, all obfuscation techniques generated the most comparable methods (53564) from 846 apps and the minimum similarity (78%) which is expected due to the complexity of obfuscation techniques. Moreover, after obfuscating applications with Control-flow and Reflection techniques, our approach performs 84% and 81% similarly to the non-obfuscated applications, respectively.

5.5 Code clone detection among all applications

In this experiment, we investigate the presence of method-level code clones among all of our 1603 successfully simulated applications from our non-obfuscated set. To perform such comparison effectively, instead of comparing application pairs and extracting their code clones, we tried finding code clone clusters. In this experiment, we define a code clone cluster as a group of methods that were detected as identical to each other

Table 5.5: Obfuscation resistance validation

Obfuscation set	**Successfully Analyzed Apps**	**Compared Methods**	**Average Similarity**
Control-flow obfuscations	989	72638	84%
Reflection obfuscations	1017	7613	81%
String encryption obfuscation	1035	13836	87%
All obfuscations	846	53564	78%

or a pair of identical method clusters whose similarity was greater than 80%. Table 5.6 shows the number of code clone clusters we were able to extract. Moreover, when clustering methods together, we discard any methods that had less than four fragments to ignore very small methods such as getters and setters.

It is very common for Android applications to include a group of libraries called Android support libraries in their build process. These libraries are developed by Google in order to improve compatibility with older devices. Thus, after extracting code clone clusters we filtered the clusters that have been detected in the Android support library classes. This was done to investigate the code clones detected in Android applications that have been generated by user-defined code or any common library that the developer has used. Table 5.6 depicts our extracted code clones. We further explored the extracted code clones among all applications without Android support libraries (see Table 5.7). Out of 44776 code clones, 66% and 34% are associated to different versions of the apps and different apps, respectively.

To further investigate and validate the extracted code clones, we randomly selected 15 clusters with different similarity degrees and analyzed them manually. From the selected clusters 11 of them were indeed code clones, 3 of them were not code clone and 1 was a clone with the wrong similarity number. For the four samples that were not correctly identified, all of them shared one problem which was very low code coverage for methods that use the Android support library classes. Many applications use support library classes instead of types bundled in Android-Runtime environment to improve compatibility with older devices. The methods defined in support library classes perform many checks to validate the state of the device and if one of the checks fails, the method throws an exception. In the current implementation of our approach since we lack proper input generation and an Ambiguous value is used for all of the method arguments including the reference object of each method, at each check a random path is taken which in most cases causes an exception to be thrown and thus the user-defined code rarely is executed. This problem in the execution path causes many methods to be detected as clones since all of them show a similar execution trace leading to the throw exception.

Table 5.6: Code clone detection among all applications

All Code Clone Clusters	number of all code clone clusters	54319
	clones with 100% similarity	19865 (36.57%)
	clones with [96%:100%) similarity	2537 (4.67%)
	clones with [92%:96%) similarity	4982 (9.17%)
	clones with [88%:92%) similarity	7136 (13.13%)
	clones with [84%:88%) similarity	7817 (14.39%)
	clones with [80%:84%) similarity	11982(22.05%)
Code Clone Clusters Without Android Support Libraries	number of all code clone clusters	44776
	clones with 100% similarity	17638 (39.39%)
	clones with [96%:100%) similarity	2145 (4.79%)
	clones with [92%:96%) similarity	3982 (8.89%)
	clones with [88%:92%) similarity	5626 (12.56%)
	clones with [84%:88%) similarity	6125 (13.67%)
	clones with [80%:84%) similarity	9260 (20.68%)

Table 5.7: Code clone detection among all applications without Android support libraries

Code Clone Clusters in different versions of apps	number of all code clone clusters	29744(66%)
	clones with 100% similarity	14076(47.32%)
	clones with [96%:100%) similarity	1802(6.06%)
	clones with [92%:96%) similarity	2895(9.73%)
	clones with [88%:92%) similarity	3504(11.78%)
	clones with [84%:88%) similarity	3447(11.59%)
	clones with [80%:84%) similarity	4020(13.52%)
Code Clone Clusters in different apps	number of all code clone clusters	15032(34%)
	clones with 100% similarity	3562(23.7%)
	clones with [96%:100%) similarity	343(2.28%)
	clones with [92%:96%) similarity	1087(7.23%)
	clones with [88%:92%) similarity	2122(14.12%)
	clones with [84%:88%) similarity	2678(17.82%)
	clones with [80%:84%) similarity	5240(34.86%)

5.6 Comparative analysis

To further investigate the ability of our approach in finding code clones, we compared the findings of our approach with a source code clone detection tool at method level. We chose Nicad [18] for this comparison because first, it can analyze Java code which is the language used to develop Android applications and second, for each clone method that it finds, it also reports a similarity degree which can be used for comparing the results of our approaches.

To perform such comparison, our first step was to separate the source code files that are relevant for these comparisons and should be analyzed by Nicad. This is needed because Android application projects, contain many different types of code, such as: source code files, test cases, automation scripts, and etc. In case of projects that build different flavors for an application, they may also define multiple implementation for some classes which is problematic since the source code file corresponding to the implementation used in the .apk file we used for our analysis has be separated. To first problem can be solved by analyzing the

directory structure of Android application projects and also matching the types defined in a source file with types used in the compiled .apk files. The second problem is more challenging to solve since we needed to manually select what source file should be used in our analysis.

Another key challenge for comparing the results of our approach with Nicad is that since our work operates on Smali byte-code, it also considers all the library codes included in an app which causes much more code clones be detected compared to Nicad. For this experiment because we had compiled the applications of our dataset our self, we were able to leverage the name of types to separate user-defined code from library codes bundled in an .apk file which is needed to be able to properly compare our approach with Nicad. However, when working with obfuscated applications where identifier renaming has been applied, it is much more challenging to separate used-defined code from library code since to do so a library profile has to be built before hand by analyzing different libraries using our tool.

In this experiment, we only compared different versions of application with each other. The reason for such choice is that 1) Different version of an applications are expected to share more codes. 2) Comparing any combination of two applications would have resulted in an explosion in the number of comparisons and time required to perform such comparisons. After creating the application-version pairs needed for our analysis, We used pairs of different versions of an application to be used as a benchmark for comparing our approach with Nicad. In this experiment 1555 application-version pairs were analyzed using Nicad which then were compared with our approach at method-level code clone detection. We used the threshold of 70% for extracting cloned methods for both our approach and Nicad. Table 5.8 shows our findings after comparing our approach with Nicad. For any method-clone pairs that the difference of similarity degree from Nicad and our approach was less than 15% we marked it as "Agreed Similarity Clone", if the Similarity difference was greater than 15% we marked it as "Disagreed Similarity Clone". For any clones that was only detected by one tool or the other we marked them as "Not detected as clone by Nicad" or "Not detected as clone by our approach". These two groups depict code clone that either of the tools failed to analyze or after analyses their similarity degree was less than our 70% threshold.

Table 5.8: Comparison results with Nicad

Agreed similarity clones	15865 (81%)
Disagreed similarity clones	3565 (19%)
Not detected as clone by Nicad	312906
Not detected as clone by our approach	455624
Not detected as clone by our approach due to code coverage	443393 (%)

5.7 Malware detection using code clone fragments

In this experiment we investigate the application of our approach in detecting code clone in malware applications. To explore the performance of our approach on a set of Android malware apps, we used a set of Android banking malware extracted from [40]. The dataset consists of 103 benign 973 malicious Android banking applications and distributed across 10 families of malware: BankBot, Binv, Citmo, FakeBank, Sandroid, SMSspy, Spitmo, Wroba, ZertSecurity and Zitmo.

In order to evaluate our approach we tried finding code clone clusters among all applications. We define a code clone cluster as a group of methods that were detected as identical to each other, or a pair of identical method clusters that their similarity was greater than 75%. We discard any methods that had less than 3 fragments to highlight longer and more meaningful methods. In total we find 60,057 code clone clusters among all applications. We categorize these clusters in four groups:

- Identical code clones that have been seen in one malware family or only benign apps
- Similar code clones that have been seen in one malware family or only benign apps
- Identical code clones that have been seen in more than one malware family or benign apps
- Similar code clones that have been seen in more than one malware family or benign apps

We are interested in the first and second group of code clones since they can be considered as unique clones to a malware family or the benign applications. These extracted code clone can further be used to detect new variant of the studied malware.

Table 5.9 shows the number of code clone clusters we were able to extract for each malware type. 92.9% of code clone clusters have been seen in apps belonged to one type of malware and 7.1% of code clone clusters are belonged to apps in different malware family. Among all malware types, SMSspy, FakeBank, Sandroid have the highest similar and identical code clones among their apps. However due to small number of samples in Binv, ZertSecurity and Citmo our approach finds less code clones among their apps.

Sample 1:
appending text to a file

Original sample

```
private static void
appendTextToFile(String text, File file) throws IOException {
    // check file exists and it is indeed a file
    if (!file.isFile()) return;
    // creating FileWriter for the given file with append mode as true
    FileWriter fileWriter = new FileWriter(file,true);
    // creating a BufferedWriter from the FileWriter
    BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
    // writing the text and closing the bufferWriter
    bufferedWriter.write(text);
    bufferedWriter.close();
}
```

Type-1 clone

```
private static void
appendTextToFile(String text, File file) throws IOException {
    if(!file.isFile()) return;
    FileWriter fileWriter = new FileWriter(file,true);
    BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);
    bufferedWriter.write(text);
    bufferedWriter.close();
}
```

Type-2 clone

```
private static void x(String s, File f) throws IOException {
    if(!f.isFile()) return;
    FileWriter fw = new FileWriter(f,true);
    BufferedWriter bw = new BufferedWriter(fw);
    bw.write(s);
    bw.close();
}
```

Type-3 clone

```
private static void x(String s, File f) throws IOException {
    if(f.isFile()) {
        BufferedWriter bw =
            new BufferedWriter(new FileWriter(f, true));
        bw.write(s);
        bw.close();
    }
}
```

Type-4 clone

```
private static void x(String s, File f) throws IOException {
    if(f.isFile()) return;
    FileWriter fw = new FileWriter(f,true);
    fw.write(s);
    fw.close();
}
```

Sample 2:
checking if a binary tree is a full tree

Original sample

```
private static boolean
isTreeFull(BinaryTree.Node node){
    // null node
    if(node==null)return true;
    // both null childs
    if(node.left == null && node.right == null) return true;
    // if node left == null || node right == null return false;
    // both child nodes not null, check the recursively
    return isTreeFull(node.left) && isTreeFull(node.right);
}
```

Type-1 clone

```
private static boolean isTreeFull(BinaryTree.Node node){
    if(node.left == null || node.right == null) return true;
    if(node.left == null && node.right == null) return false;
    return isTreeFull(node.left) && isTreeFull(node.right);
}
```

Type-2 clone

```
private static boolean x(BinaryTree.Node n){
    if(n.left == null || n.right == null) return true;
    if(n.left == null && n.right == null) return false;
    return x(n.left) && x(n.right);
}
```

Type-3 clone

```
private static boolean x(BinaryTree.Node n){
    if((n.left == null) ^ (n.right == null)) return true;
    if(n.left == null || n.right == null) return false;
    return x(n.left) && x(n.right);
}
```

Type-4 clone

```
private static boolean x(BinaryTree.Node n){
    if((n.left == null) ^ (n.right == null)) return true;
    if(n.left == null) return false;
    return x(n.left) && x(n.right);
}
```

Sample 3:
removing duplocated numbers in an ArrayList of integers

Original sample

```
private static void
removeDuplicatedInIntegerArrayList(ArrayList<Integer> arrayList){
    int arraySize = arrayList.size();
    // HashSet to keep the unique numbers we see
    Set<Integer> seenNumbers = new HashSet<>();
    for(int i=0; i < arrayList.size(); i++){
        int n = arrayList.get(i);
        // if the number is repeated,
        if(seenNumbers.contains(n)){
            // remove it and change counters accordingly
            arrayList.remove(i);
            i--;
            arraySize--;
        }
        // if the number we see is unique add it to the HashSet
        else {
            seenNumbers.add(n);
        }
    }
}
```

Type-1 clone

```
private static void
removeDuplicatedInIntegerArrayList(ArrayList<Integer> arrayList){
    int arraySize = arrayList.size();
    Set<Integer> seenNumbers = new HashSet<>();
    for(int i=0; i < arrayList.size(); i++){
        int num = arrayList.get(i);
        if(seenNumbers.contains(num)){
            arrayList.remove(i);
            i--;
            arraySize--;
        }
        else {
            seenNumbers.add(num);
        }
    }
}
```

Type-2 clone

```
private static void x(ArrayList<Integer> a){
    int s = a.size();
    Set<Integer> sn = new HashSet<>();
    for(int i=0; i < s; i++){
        int num = a.get(i);
        if(sn.contains(num)){
            a.remove(i);
            i--; s--;
        }
        else {
            sn.add(num);
        }
    }
}
```

Type-3 clone

```
private static void x(ArrayList<Integer> a){
    Set<Integer> sn = new HashSet<>();
    int i = 0;
    while (i < a.size()){
        int num = a.get(i);
        if(sn.contains(num)){
            a.remove(i);
        }
        else {
            sn.add(num);
            i++;
        }
    }
}
```

Type-4 clone

```
private static void x(ArrayList<Integer> arrayList){
    Set<Integer> seenNumbers = new HashSet<>
    (arrayList);
    arrayList.clear();
    arrayList.addAll(seenNumbers);
}
```

Sample 4:
removing the middle node in a linked list

Original sample

```
private static void
removeMiddleNodeInLinkedList(MyLinkedList linkedList){
    MyLinkedList.Node curr = linkedList.getHead();
    MyLinkedList.Node prevMiddleNode = null;
    MyLinkedList.Node middleNode = curr;
    // if LinkedList is empty return
    if(curr==null) return;
    boolean incrementMiddleNode = false;
    // traverse the Linked list and for every two traversed nodes
    // move the middle node pointer
    while (cur != null){
        if(incrementMiddleNode){
            prevMiddleNode = middleNode;
            middleNode = middleNode.next;
        }
        incrementMiddleNode = !incrementMiddleNode;
        cur = cur.next;
    }
    if(prevMiddleNode != null){
        prevMiddleNode.next = middleNode.next;
    }
}
```

Type-1 clone

```
private static void
removeMiddleNodeInLinkedList(MyLinkedList linkedList){
    MyLinkedList.Node cur = linkedList.getHead();
    MyLinkedList.Node prevMiddleNode = null;
    MyLinkedList.Node middleNode = cur;
    if(cur==null) return;
    boolean incrementMiddleNode = false;
    while (cur != null){
        if(incrementMiddleNode){
            prevMiddleNode = middleNode;
            middleNode = middleNode.next;
        }
        incrementMiddleNode = !incrementMiddleNode;
        cur = cur.next;
    }
    if(prevMiddleNode != null){
        prevMiddleNode.next = middleNode.next;
    }
}
```

Type-2 clone

```
private static void x(MyLinkedList l){
    MyLinkedList.Node c = l.getHead();
    MyLinkedList.Node p = null;
    MyLinkedList.Node m = c;
    if(c==null) return;
    boolean i = false;
    while (c != null){
        if(i){
            p = m;
            m = m.next;
        }
        i = !i;
        c = c.next;
    }
    if(p != null){
        p.next = m.next;
    }
}
```

Type-3 clone

```
private static void x(MyLinkedList l){
    MyLinkedList.Node c = l.getHead();
    if(c==null) return;
    boolean i = false;
    MyLinkedList.Node m = c;
    MyLinkedList.Node p = null;
    while (true){
        if(i){
            p = m;
            m = m.next;
        }
        i = !i;
        c = c.next;
        if(c == null) break;
    }
    if(p != null)p.next = m.next;
}
```

Type-4 clone

```
private static void x(MyLinkedList l){
    MyLinkedList.Node head = l.getHead();
    if(head==null) return;
    MyLinkedList.Node fastPointer = head;
    MyLinkedList.Node slowPointer = head;
    MyLinkedList.Node prev = null;
    while (fastPointer != null) {
        fastPointer = fastPointer.next.next;
        prev = slowPointer;
        slowPointer = slowPointer.next;
    }
    prev.next = slowPointer.next;
}
```

Sample 5:
reversing a number

Original sample

```
public static int reverseNumber(int num){
    int reversed = 0;
    // run loop until num becomes 0
    while(num != 0) {
        int digit = num % 10; // get last digit from num
        reversed = reversed * 10 + digit;
        num /= 10; // remove the last digit from num
    }
    return reversed;
}
```

Type-1 clone

```
private static int reverseNumber(int num){
    int reversed = 0;
    while(num != 0) {
        int digit = num % 10;
        reversed = reversed * 10 + digit;
        num /= 10;
    }
    return reversed;
}
```

Type-2 clone

```
private static int x(int n){
    int r = 0;
    while(n != 0) {
        int d = n % 10;
        r = r * 10 + d;
        n /= 10;
    }
    return r;
}
```

Type-3 clone

```
private static int x(int n){
    int r = 0;
    while(n != 0) {
        int digit = n % 10;
        r = r * 10 + digit;
        n /= 10;
    }
    return r;
}
```

Type-4 clone

```
private static int x(int num){
    int result = 0;
    char[] charArray = Integer.toString(num)
                        .toCharArray();
    for (int c = charArray.length - 1; c > -1 ; c--) {
        result = result * 10 + c - 48;
    }
    return result;
}
```

Figure 5.1: Code clone samples added for code clone detection validation

Table 5.9: Code clone clusters in malware Android apps

Cluster type	**Malware Type**	**Num of Sample**	**Identical Clones**	**Similar Clones**
Clusters with same malware type	Spitmo	191	55	704
	BankBot	136	224	581
	Zitmo	142	531	837
	SMSspy	131	1458	27540
	Wroba	152	344	1004
	FakeBank	151	1216	1887
	Sandroid	61	830	5620
	Binv	2	8	5
	ZertSecurity	4	2	7
	Citmo	3	42	10
	Benign	103	2719	10189
	All	1076	7429(12.4%)	48384(80.5%)
Clusters with different malware type	All	-	938(1.6%)	3306(5.5%)

6 Discussion

One major challenge we faced in our work was defining what should be considered as a code clone, especially when analyzing a code that is obfuscated and traditional heuristics can not be employed. When studying previous works done on analyzing obfuscated programs, and code clone detection, we clearly noticed a semantical gap. Most of the code clone detection works we studied defined code clones as code pieces that essentially look similar in some form and do not focus on the behaviour of the code which makes those approaches prone to obfuscation. On the other hand, proposed obfuscation-resistant approaches in the literature analyze a given code very abstractly and only focused at the captured behavioural features of some code at runtime such as invoked system calls or the captured I/O behaviour which is not particularly useful for finding code clones and measuring code reuse. In our approach we proposed analysis of execution traces of to achieve an automated and obfuscation-resistant code clone detection approach that does not assume anything about a given code. We also implemented a prototype of Smali simulation environment that can generate possible execution traces of an Android application, enhanced with its runtime metadata. Finally, we proposed a code clone detection approach for extracting semantic code clones which can further be used for other applications such as Android library detection and profiling.

The current implementation of our approach suffers from three limitations: limited code coverage, scalability and false-positive cases.

Limited code coverage: Currently, the most significant issue of our work is its limited code coverage, especially for applications that use Android support library classes. This is caused by the fact that our approach uses ambiguous value as all of its inputs and thus traverse undesired paths. In addition, during simulation, when a condition is checked against an ambiguous value, the ambiguous value does not capture the result of the checked condition and thus does not keep track of its corresponding constraints. This can lead to some invalid paths being taken during simulation.

Scalability: Another limitation of our approach is its needed computational resources, its analysis speed and scalability. Due to the fact that our approach is a hybrid approach that tries to incorporate elements of dynamic analysis, it makes it slower than most static analysis approaches, however, it is faster and more efficient than most purely dynamic approaches since it can analyze any method thrown at it and does not require all the given inputs to be completely valid and can abstract over the I/O operations executed in an Android application. During our experiments we noticed that our tool performs slowly when dynamically creating classes that contain very large number of methods (e.g. 2000 methods) which can happen in obfuscated applications. This bottleneck is caused by how ByteBuddy create and dynamically loads classes into

the system. However, we believe that this is a matter of implementation and our tool can be significantly optimized in the future.

False-positives: Our proposed approach can lead to false-positive cases, particularly if two user-defined code samples use a common library. This is caused by the fact that the execution traces of two methods share many similarities because of the shared logic from the library while the code samples defined by the user might be vastly different. This was a trade-off we deliberately chose to achieve obfuscation resistance against common obfuscation techniques such as method outlining, without assuming anything about a given code and any possible obfuscations applied to it.

7 Conclusion

Code clone detection is a challenging form of software analysis that is widely used in both software development and software security. From a software security perspective, code clone detection can be used for a variety of use cases such as malware detection and classification or vulnerability detection. However, one key challenge for effective code clone detection in binary code is the presence of code obfuscation. Due to ease of reversing and modifying Dalvik byte-code, code obfuscation has become widely used in Android applications for software protection purposes. Even though code obfuscation is a very common form of software protection, it also plays an important role in the creation of new malware as it helps malware authors to hide the logic and intents of their created samples.

Due to the fact that purely static approaches do not perform well on obfuscated code and dynamic approaches are very expensive to perform and automate, we proposed a hybrid approach that tries to combine the best of both worlds. In this work, we introduced an obfuscation-resistant code clone detection approach based on two key ideas, first is the analysis of execution traces of Android apps instead of their code, and second, using Smali simulation to generate possible execution traces for an app. Using these techniques we were able to achieve resistance against different types of obfuscation techniques used for Android applications even for more challenging obfuscation techniques such as string encryption and reflection. In this work, we confirmed that code simulation and the analysis of execution traces are effective techniques for overcoming code obfuscation. However, they can have their own drawbacks such as being more expensive than most common static analysis techniques and suffering from the code coverage problem.

7.1 Future work

This research can be extended in many other interesting directions such as increasing the overall performance and accuracy, improving scalability, and applying the proposed approach for different application analysis purposes. In this section, some of these possible directions are enumerated.

- **Adding input generation :** The addition of input generations techniques can be used when simulating Smali methods. Improving input generation can increase the accuracy and obfuscation resistance of our approach. Sophisticated input generation techniques are generally expensive to perform, however, since our approach can also take an ambiguous value as an input, this becomes less challenging since sophisticated input generation can be used only for some specific arguments which are critical to the overall performance of our approach.

- **Integration with SMT solver:** One limitation of the ambiguous value is that as a random path is taken based on the result of some condition on an ambiguous value, the corresponding constraint is not captured. By capturing constraints of ambiguous values along different execution paths we can significantly reduce the number of invalid execution paths generated. By integrating our approach with an SMT solver, much more information can also be inferred from Android applications such as on what conditions a code is executed.

- **Improving scalability:** By optimizing the simulation process and how execution traces are analyzed and compared the speed and scalability of our approach can be greatly improved. One key challenge of code clone detection on large datasets is scalability. This issue can be improved by using different heuristics such as the size of code in order to reduce the number of comparisons needed to be made by eliminating improbable clones.

- **Android library profiling:** Our proposed approach can be leveraged for the detection of Android library codes in presence of code obfuscation. Detection of Android library codes is beneficial since it can be used to separate user-defined code, improve application comprehension, and even vulnerability detection. Android libraries are bundled in an apk file during the compile process which after obfuscation becomes very hard to distinguish from user-defined code.

- **Vulnerability detection:** Code clone detection has been proved to be an effective way for finding software vulnerabilities in x86 programs, however, this has also been left unexplored for Android. An effective code clone detection approach that can operate on binary code instead of source code and is resilient to obfuscation can be further used for vulnerability discovery on applications present in Android markets.

References

[1] Android runtime (art) and dalvik : Android open source project. `https://source.android.com/devices/tech/dalvik`. Accessed: 2021-11-01.

[2] Application fundamentals : Android developers. `https://developer.android.com/guide/components/fundamentals`. Accessed: 2021-11-01.

[3] Platform architecture : Android developers. `https://developer.android.com/guide/platform`. Accessed: 2021-11-01.

[4] Junaid Akram, Zhendong Shi, Majid Mumtaz, and Ping Luo. Droidcc: A scalable clone detection approach for android applications to detect similarity at source code level. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 100–105. IEEE, 2018.

[5] Shahid Alam, Ryan Riley, Ibrahim Sogukpinar, and Necmeddin Carkaci. Droidclone: Detecting android malware variants by exposing code clones. In *2016 Sixth International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, pages 79–84. IEEE, 2016.

[6] Hakam W Alomari and Matthew Stephan. Srcclone: Detecting code clones via decompositional slicing. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 274–284, 2020.

[7] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. Obfuscapk: An open-source black-box obfuscation tool for android apps. *SoftwareX*, 11:100403, 2020.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[9] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, and Francesco Mercaldo. Detection of obfuscation techniques in android applications. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–9, 2018.

[10] Vivek Balachandran, Darell JJ Tan, Vrizlynn LL Thing, et al. Control flow obfuscation for android applications. *Computers & Security*, 61:72–93, 2016.

[11] Richard Baumann, Mykolai Protsenko, and Tilo Müller. Anti-proguard: Towards automated deobfuscation of android apps. In *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*, pages 7–12, 2017.

[12] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 343–355, 2016.

[13] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. Horndroid: Practical and sound static analysis of android applications by smt solving. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 47–62. IEEE, 2016.

[14] Jian Chen, Manar H Alalfi, Thomas R Dean, and Ying Zou. Detecting android malware using clone detection. *Journal of Computer Science and Technology*, 30(5):942–956, 2015.

[15] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186, 2014.

[16] Taejoo Cho, Hyunki Kim, and Jeong Hyun Yi. Security assessment of code obfuscation based on dynamic monitoring in android things. *Ieee Access*, 5:6361–6371, 2017.

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

[18] James R Cordy and Chanchal K Roy. The nicad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220. IEEE, 2011.

[19] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *European Symposium on Research in Computer Security*, pages 37–54. Springer, 2012.

[20] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *European Symposium on Research in Computer Security*, pages 37–54. Springer, 2012.

[21] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of semantically similar android applications. In *European Symposium on Research in Computer Security*, pages 182–199. Springer, 2013.

[22] Martijn de Vos and Johan Pouwelse. Astana: Practical string deobfuscation for android applications using program slicing. *arXiv preprint arXiv:2104.02612*, 2021.

[23] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Kam1n0: Mapreduce-based assembly clone search for reverse engineering. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 461–470, 2016.

[24] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[25] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, XiaoFeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In Raheem Beyah, Bing Chang, Yingjiu Li, and Sencun Zhu, editors, *Security and Privacy in Communication Networks*, pages 172–192, Cham, 2018. Springer International Publishing.

[26] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 303–317, San Diego, CA, August 2014. USENIX Association.

[27] Rochelle Elva and Gary T Leavens. Jsctracker: A semantic clone detection tool for java code. Technical report, University of Central Florida, Dept. of EECS, CS division, 2012.

[28] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, volume 52, pages 58–79, 2016.

[29] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *2014 Eighth International Conference on Software Security and Reliability (SERE)*, pages 78–87. IEEE, 2014.

[30] Mohammad Reza Farhadi, Benjamin CM Fung, Yin Bun Fung, Philippe Charland, Stere Preda, and Mourad Debbabi. Scalable code clone search for malware analysis. *Digital Investigation*, 15:46–60, 2015.

[31] Felix C Freiling, Mykola Protsenko, and Yan Zhuang. An empirical evaluation of software obfuscation techniques applied to android apks. In *International Conference on Security and Privacy in Communication Networks*, pages 315–328. Springer, 2014.

[32] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, page 431–444, New York, NY, USA, 2013. Association for Computing Machinery.

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

[34] Yoshiki Higo and Shinji Kusumoto. Code clone detection on specialized pdgs with heuristics. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 75–84. IEEE, 2011.

[35] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Binary code clone detection across architectures and compiling configurations. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 88–98. IEEE, 2017.

[36] Md R Islam and Minhaz F Zibran. A comparative study on vulnerabilities in categories of clones and non-cloned code. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, volume 3, pages 8–14. IEEE, 2016.

[37] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: finding unpatched code clones in entire os distributions. In *2012 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2012.

[38] Jihwan Jeong, Dongwon Seo, Chanyoung Lee, Jonghoon Kwon, Heejo Lee, and J. Milburn. MysteryChecker: Unpredictable attestation to detect repackaged malicious applications in Android. In *Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on*, pages 50–57, Oct 2014.

[39] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[40] Andi Fitriah A Kadir, Natalia Stakhanova, and Ali A Ghorbani. An empirical analysis of android banking malware. *Protecting mobile networks and devices: challenges and solutions*, 209, 2016.

[41] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[42] Iman Keivanloo, Chanchal K Roy, and Juergen Rilling. Sebyte: Scalable clone and similarity search for bytecode. *Science of Computer Programming*, 95:426–444, 2014.

[43] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In Patrick Cousot, editor, *Static Analysis*, pages 40–56, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[44] Aleksandrina Kovacheva. Efficient code obfuscation for android. In *International Conference on Advances in Information Technology*, pages 104–119. Springer, 2013.

[45] Renuka Kumar and Anjana Mariam Kurian. A systematic study on static control flow obfuscation techniques in java. *arXiv preprint arXiv:1809.11037*, 2018.

[46] Li Li, Daoyuan Li, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Trans. Inf. Forensics Secur.*, 12(6):1269–1284, 2017.

[47] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260. IEEE, 2017.

[48] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 242–251, 2014.

[49] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[50] Jan M Memon, Asghar Mughal, Faisal Memon, et al. Preventing reverse engineering threat in java using byte code obfuscation techniques. In *2006 International Conference on Emerging Technologies*, pages 689–694. IEEE, 2006.

[51] Omid Mirzaei, José María de Fuentes, J Tapiador, and Lorena Gonzalez-Manzano. Androdet: An adaptive android obfuscation detector. *Future Generation Computer Systems*, 90:240–261, 2019.

[52] Alireza Mohammadinodooshan, Ulf Kargén, and Nahid Shahmehri. Robust detection of obfuscated strings in android apps. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, pages 25–35, 2019.

[53] Michele Pasetto, Niccolò Marastoni, and Mila Dalla Preda. Revealing similarities in android malware by dissecting their methods. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 625–634. IEEE, 2020.

[54] Mykola Protsenko and Tilo Müller. Pandora applies non-deterministic obfuscation randomly to android. In *2013 8th International conference on malicious and unwanted software:" The Americas"(MALWARE)*, pages 59–67. IEEE, 2013.

[55] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354–365, 2018.

[56] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168, 2016.

[57] Abdullah Sheneamer and Jugal Kalita. Semantic clone detection using machine learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1024–1028. IEEE, 2016.

[58] Sonatype. 2017 DevSecOps Community Survey of 2,292 IT professionals. Technical report, Sonatype, 8161 Maple Lawn Blvd 250, Fulton, MD 20759, 2017.

[59] Jeffrey Svajlenko and Chanchal K Roy. Evaluating clone detection tools with bigclonebench. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 131–140. IEEE, 2015.

[60] Zoran urić and Dragan Gašević. A source code similarity system for plagiarism detection. *The Computer Journal*, 56(1):70–86, 2013.

[61] Yan Wang and Atanas Rountev. Who changed you? obfuscator identification for android. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 154–164. IEEE, 2017.

[62] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)*, 21(3):1–32, 2018.

[63] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. A large scale investigation of obfuscation use in google play. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 222–235, 2018.

[64] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 87–98. IEEE, 2016.

[65] Rafael Winterhalte. Byte buddy - runtime code generation for the java virtual machine. `https://bytebuddy.net`.

[66] Jun Yang, Yu Xiong, and Jinxin Ma. A function level java code clone detection method. In *2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, volume 1, pages 2128–2134. IEEE, 2019.

[67] Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang Shi. Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection. *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun 2021.

[68] WooJong Yoo, Myeongju Ji, MinKoo Kang, and Jeong Hyun Yi. String deobfuscation scheme based on dynamic code extraction for mobile malwares. *IT Convergence Practice*, 4(2):1–8, 2016.

[69] Geunha You, Gyoosik Kim, Jihyeon Park, Seong-je Cho, and Minkyu Park. Reversing obfuscated control flow structures in android apps using redex optimizer. 2020.

[70] Dongjin Yu, Jie Wang, Qing Wu, Jiazha Yang, Jiaojiao Wang, Wei Yang, and Wei Yan. Detecting java code clones with multi-granularities based on bytecode. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 317–326. IEEE, 2017.

[71] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *7th ACM Conference on WiSec*, Oxford, United Kingdom, 2014.

[72] Haibo Zhang and Kouichi Sakurai. A survey of software clone detection from security perspective. *IEEE Access*, 9:48157–48173, 2021.

[73] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, page 317–326, New York, NY, USA, 2012. Association for Computing Machinery.