The Pennsylvania State University
The Graduate School

TOWARD OBFUSCATION-RESILIENT PLAGIARISM DETECTION

A Dissertation in
Computer Science and Engineering
by
Fangfang Zhang

Submitted in Partial Fulfillment
of the Requirements
for the Degree of
Doctor of Philosophy

August 2014

The dissertation of Fangfang Zhang was reviewed and approved* by the following:

Sencun Zhu
Associate Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Peng Liu
Professor of Information Sciences and Technology

Guohong Cao
Professor of Computer Science and Engineering

David J. Miller
Professor of Electrical Engineering

Lee Coraor
Associate Professor of Computer Science and Engineering
Graduate Officer

*Signatures are on file in the Graduate School.

Abstract

In the field of software development, plagiarism is an act of violating intellectual property rights. Plagiarists either illegally copy others' source/binary code (also known as *software plagiarism*) or steal others' algorithms and covertly implement them (called *algorithm plagiarism*). Code obfuscation techniques are often applied by plagiarists to evade detection. Plagiarism has become a serious concern for honest software companies and the open source community. Besides, along with the wide use of mobile devices such as smartphones and tablets and the rapid growth of mobile application (app) markets, mobile app repackaging, as a new kind of software plagiarism, has emerged. It not only harms the health of app markets but also hurts the security of mobile users. As a result, computer-aided, automated plagiarism detection is desired. There are two common requirements for a good plagiarism detection scheme: (R1) Capability to work on suspicious executables without the source code; (R2) Resiliency to code obfuscation techniques.

In this dissertation, we propose an obfuscation resilient plagiarism detection architecture, which satisfies the above requirements. It contains three components: *LoPD*, a program logic-based approach to software plagiarism detection, *ValPD*, a dynamic value-based approach to algorithm plagiarism detection, and *ViewDroid*, a user interface-based approach for Android application repackaging detection.

LoPD is a program logic-based software plagiarism detection method. Instead of directly comparing the similarity between two programs, LoPD searches for any dissimilarity between two programs by finding an input that will cause these two programs to behave differently, either with different output states or with semantically different execution paths. As long as we can find one dissimilarity, the programs are semantically different; otherwise, it is likely a plagiarism case. We leverage symbolic execution and weakest precondition reasoning to capture the semantics of execution paths and to find path dissimilarities. LoPD is resilient to current automatic obfuscation techniques. In addition, since LoPD is a formal program semantics-based method, we can provide a formal guarantee of resilience against most known obfuscation attacks. Our evaluation results indicate that

LoPD is both effective and efficient in detecting software plagiarism.

In the ValPD component, we propose two dynamic value-based approaches, namely *N-version* and *annotation*, for algorithm plagiarism detection. Our approaches are motivated by the observation that there exist some critical runtime values which are irreplaceable and uneliminatable for all implementations of the same algorithm. The N-version approach extracts such values by filtering out non-core values. The annotation approach leverages auxiliary information to flag important variables which contain core values. We also propose a value dependence graph-based similarity metric to address the potential value reordering attack. A prototype is implemented and evaluated. The results show that our approaches to algorithm plagiarism detection are practical, effective and resilient to many automatic obfuscation techniques.

Lastly, we propose ViewDroid, a user interface-based approach to smartphone application repackaging detection. Android applications are user interaction intensive and event dominated; the interactions between users and apps are performed through user interface (i.e., views). This inspired the design of our new birthmark for Android applications, namely, *feature view graph*, which captures user's navigation behavior across application views. Our experimental results demonstrate that this birthmark can characterize Android applications from a higher abstraction, making it resilient to code obfuscation. It can detect repackaged apps in large-scale scenarios both effectively and efficiently. Manual verification for the reported pairs shows that the false positive rate and false negative rate of ViewDroid are very low.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere gratitude to my advisor, Dr. Sencun Zhu. It is his constant encouragement and insightful advice that inspired me and guided me to finish this dissertation. His generosity, kindness and enthusiasm also influence me a lot. It has been my great honor to work with him.

I also want to thank Dr. Peng Liu and Dr. Dinghao Wu for their inspired discussions and insightful suggestions. I am so grateful that they spent their valuable time helping me revise my papers.

My gratitude also goes to my other committee members, Dr. Guohong Cao and Dr. David Miller for attending my defense, reviewing my dissertation and giving me critical comments to improve the dissertation.

Finally, I want to thank my parents and my husband. Without their unconditional love and support, I cannot finish this work.

Dedication

To my husband Wei and my son Ethan.

Chapter 1

Introduction

In the field of software development, plagiarism is an act of violating intellectual property rights. There are two ways to perform plagiarism. The first one is that plagiarists copy and reuse others' source/binary code and apply some obfuscation techniques to evade detection, also known as *software plagiarism*. The second way is to steal others' algorithm and implement it, but claim it is their own product. This is called *algorithm plagiarism*.

Along with the rapid growing of software industry and the burst of open source projects (e.g., SourceForge.net has over $430,000$ registered open source projects with 3.7 million developers and more than 4.8 million downloads a day, as of March 2014 [1]), plagiarism has become a very serious concern for honest software companies, developers and open source communities. There are some billion dollar lawsuits dealing with the plagiarism cases. As an example, in the settlement of an intellectual property lawsuit that Compuware filed against IBM in 2005, the latter should pay the former "$140 million to license its software and $260 million to purchase its services" [1], because it was discovered that some IBM products copied code from Compuware. As a result, computer-aided, automated plagiarism detection is desired for the purpose of intellectual property protection.

Meanwhile, along with the wide use of mobile devices (such as smartphones and tablets) and the fast growing of mobile application (app) markets, a special kind of software plagiarism, smartphone app repackaging, has emerged and drawn researchers' attention. Zhou et al. [2] found 5% to 13% of apps in third-party

---

[1] `http://sourceforge.net/about`

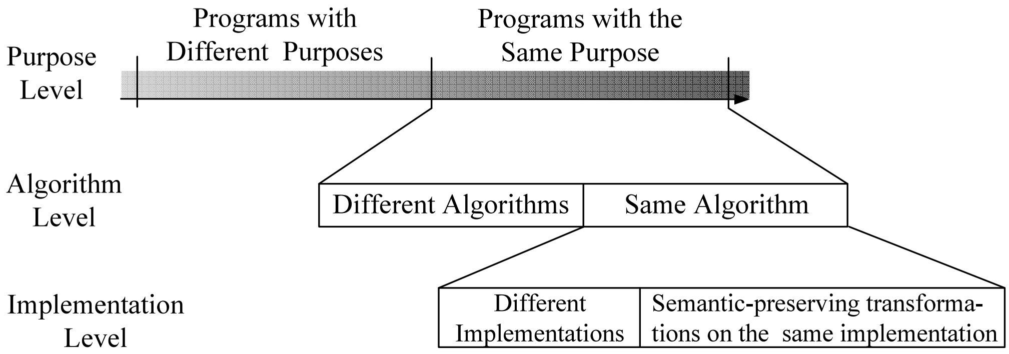

Purpose Level
Programs with Different Purposes
Programs with the Same Purpose
Algorithm Level
Different Algorithms
Same Algorithm
Implementation Level
Different Implementations
Semantic-preserving transformations on the same implementation

Figure 1.1. The spectrum of program similarity

app markets repackaged apps from the official Android app market. Moreover, a repackaged app can mimic a popular app but is used to propagate malware. In order to maintain the health of app markets and the security of mobile users, app repackaging detection is a critical issue to be addressed.

However, automated plagiarism detection is very challenging. For one reason, source code of suspicious programs is usually not available to plaintiff. The analysis of executables is much harder than the source code analysis. Besides having no access to source code of suspicious programs, code obfuscation techniques are also a huge obstacle to automated plagiarism detection. Code obfuscation is a technique to transform a sequence of code into a different sequence that preserves the semantics but is much more difficult to understand or analyze. Based on above two facts, there are two common requirements for a good plagiarism detection scheme [3]: (R1) Capability to work on suspicious executables without the source code; (R2) Resiliency to code obfuscation techniques. Moreover, due to the large number of apps on app markets, app repackaging detection has an addition requirement: (R3) Efficiency and scalability to a large scale app set.

In this work, we are focusing on three problems: software plagiarism detection, algorithm plagiarism detection and smartphone app repackaging detection. As shown in Figure 1.1, the similarity between programs can be reflected at different abstraction levels, including purpose level, algorithm level and implementation level. Both software plagiarism detection and smartphone app repackaging detection are on the implementation level, while the algorithm plagiarism is on the algorithm level.

Table 1.1. The code obfuscation resilience comparison of different detection approaches

	C1	C2	C3	C4	C5	LoPD
Noise instruction			✓	✓	✓	✓
Statement reordering			✓	✓	✓	✓
Instruction splitting/aggregation			✓	✓	✓	✓
Value splitting/aggregation			✓	✓		✓
Opaque predicate				✓	✓	✓
Control flow flattening				✓	✓	✓
Loop unwinding				✓	✓	✓
API implementation embedding			✓		✓	✓

1.1 Software Plagiarism Detection

The detection of software plagiarism has been discussed in many literatures, which can be divided into the following categories: (C1) static source code comparison methods [4, 5, 6]; (C2) static executable code comparison methods [7, 8]; (C3) dynamic control flow-based methods [9]; (C4) dynamic API-based methods [10, 11]; (C5) dynamic value-based approach [3, 12]. First, C1 does not meet R1 because it has to access source code. Second, none of them satisfy requirement R2 because they are vulnerable to some code obfuscation techniques as shown in Table 1.1.

In Chapter 3, we propose a novel software plagiarism detection approach, called LoPD. LoPD does not need the source code of tested programs. In addition, it is resilient to existing automatic code obfuscation techniques. Instead of directly measuring the similarity between two programs, LoPD is based on an opposite philosophy: *searching for any dissimilarity between two programs.* As long as we can find one dissimilarity, the tested programs are semantically different; but if we cannot find any dissimilarity, it is likely a plagiarism case.

Based on our design philosophy, LoPD tries to rule out dissimilar program pairs by finding input that will cause these two programs to behave differently, either with different output states or with different computation paths. The output states can be directly compared, but the comparison of computation paths is not such straightforward. Our idea is to find *path deviation*, i.e., given two different inputs, one program will follow the same execution path, whereas the other one will execute two different paths with these two inputs. In this case, at least one of these two inputs makes the two programs have different computation paths and behave differently. As long as we find a path deviation, we can claim the two programs in consideration are not semantically the same. Detecting path deviation transforms the comparison of different programs' execution paths with the same input to the comparison of the same program's execution paths with different inputs. The latter is much more straightforward and accurate. We leverage symbolic execution [13] and weakest precondition [14, 15] to systemically find such path deviations.

1.2 Algorithm Plagiarism Detection

Detection of algorithm plagiarism is desired in many practical scenarios. For example, when an algorithm is protected by patent right, the owners of this algorithm need to defend their proprietary by examining the plagiarism of this algorithm in other programs. Another scenario is that software companies often need to verify that their software products do not plagiarize any patent protected algorithms before release, to avoid lawsuits. In addition to its commercial potential, algorithm plagiarism detection can also provide important insight into the identification of essential characteristics of an algorithm. However, to the best of our knowledge, there has been little previous work focusing on this topic.

Although both algorithm plagiarism detection and software plagiarism detection rely on assessing the similarity between programs, they are fundamentally different. If two software products are independently developed by two companies using the same algorithm, there exists no software plagiarism because of the independence. Any valid software plagiarism detection tool should indicate the same conclusion. However, if the underlying algorithm belongs to one company and is implemented stealthily by the other, there exists algorithm plagiarism, which apparently cannot be detected by any software plagiarism detection tools. Therefore, we cannot apply software plagiarism detection approach to detect software plagiarism.

In fact, algorithm plagiarism detection is even more challenging than software plagiarism detection. A major reason is that an algorithm can be independently implemented in different ways by different programmers in different programming languages. These implementation processes involve human intelligence, coding style and creativity, which generate a lot of diversities in the resulted code. These diversities are hard to be described formally and can cause two programs implementing the same algorithm to appear dramatically different from each other. As a result, how to "peel off" these diversities and how to capture the essential code-level characteristics of an algorithm remain big challenges. In contrast, the diversities caused by software plagiarism assisted by automatic code obfuscation tools can be filtered out through birthmarks and structural features [3, 4, 16, 9, 10, 17, 18, 11]. In other words, the gap between essential characteristics of an algorithm and the (static/dynamic) exhibition of the algorithm implementations is much larger than the gap between the (static/dynamic) exhibition of a program and that of the obfuscated versions of the program.

In Chapter 4, we develop a dynamic value-based methodology to effectively detection algorithm plagiarism. We use *core values*, i.e., the critical runtime values that are irreplaceable and uneliminatable for different implementations of the same algorithm, as the signature of an algorithm. Then we propose two novel approaches to extract core values from programs' runtime values: the N-version approach and the annotation approach. The N-version approach tries to find the common values of different implementations of the same algorithm. The annotation approach leverage auxiliary information to identify core values. After that, we propose two metrics: the longest common subsequence (LCS) and the value dependence graph (VDG) to assess the similarity between core values extracted from an algorithm's plaintiff implementation and its suspicious implementation.

1.3 Smartphone Application Plagiarism Detection

There were over $1,100,000$ apps available on the Google Play Android app market [19] on March 2014. Since popularity has become the core value among mobile platforms, many popular Android apps have been "copied," or repackaged, as reported by Gibler et al. [20]. For example, when a famous smartphone game app Flappy Bird was removed from Google Play app market by its developer early this year, a lot of repackaged Flappy Bird apps appeared in third-part app markets for users to download.

One of the major reasons behind the emerging of Android app repackaging is that it is easy to reverse-engineer an Android app. When a user purchases and downloads an Android app, the installation package (i.e., the .apk file) is downloaded and stored on the user's mobile device. Given the openness of the Android platform, it is very easy to obtain the installation package from the mobile device. After that, reverse engineering can be performed based on readily available tools such as apktool [21] and Baksmali/Smali [22], which can dissemble the compiled Dalvik EXecutable (dex) from the .apk file into a human readable Dalvik bytecode format (e.g., .smali files). At this point, the content of an app can be easily manipulated, modified, repackaged and signed into a re-publishable apk file. To make things worse, the signing is not required to be bound with any official real ID of the developer and there is no certificate authority to sign apps. Moreover, due to the popularity of the Android platform, many unofficial app markets exist. Most of them do not enforce sanity checks on the apps listed on their web pages. As a result, the severity of app repackaging in the Android platform has been observed higher than in any other mobile platforms.

Generally speaking, there are two types of smartphone app repackaging. The purpose of the first type is to use other developers' apps to earn pecuniary profits. An attacker can easily repackage an app under his own name or embed different advertisements to gain ad benefits. The second type is related to malware, where attackers modify a popular app by inserting some malicious payload, e.g., sending out users' private information and purchasing apps without users' awareness, into the original program. They leverage the popularity of the original program to accelerate the propagation of the malicious one. According to a recent study [2], 1083 (or 86.0%) of 1260 malware samples were repackaged versions of legitimate apps with malicious payloads, indicating repackaging is a favorable vehicle for mobile malware propagation.

Recently, several research works have discussed the detection of app repackaging, including Fuzzy Hashing-based detection [2], Program Dependence Graph (PDG)-based detection [23, 24], Feature Hashing-based detection [25], module decoupling-based detection [26], and Normal Compression Distance (NCD)-based detection [27]. These approaches can identify the repackaging apps efficiently based on certain "invariants" extracted from the app code. Such invariants are called *software birthmark* as in the software engineering research. A *software birthmark* is defined as a unique characteristic that a program or smartphone app inherently possesses, and can be used to uniquely identify the program. All the above approaches use code-level birthmarks to characterize an app.

In Chapter 5, we propose a novel app repackaging detection system called *ViewDroid*, which leverages user interface-based *birthmark* to detect repackaged app pairs on Android platform. ViewDroid is a nice alternative to code-level detection approaches. It was motivated by two observations. First, smartphone apps are user behavior intensive and Android event-driven. The interactions between users and apps are performed through user interfaces (i.e., app views). Some characters of views (e.g. the navigation between views) are unique for each independently developed app. Second, in both types of repackaging, because attackers want to leverage the popularity of the target app, they will keep the repackaged apps' look-and-feel similar to the original one in the user interface level. Specifically, ViewDroid is built upon a robust birthmark called *view graph*, which is a graph constructed from all views through static analysis and catches the navigation relation among app views.

1.4 Contributions

This dissertation has the following contributions:

LoPD, logic-based software plagiarism detection.

- We present LoPD, a program logic-based software plagiarism detection approach, which applies symbolic execution and weakest precondition reasoning to find dissimilarities between programs.
- LoPD is resilient to most current code obfuscation techniques. In addition, LoPD can provide a formal assurance of resilience against many types of obfuscation attacks.
- LoPD theoretically guarantees high detection accuracy.

ValPD, value-based algorithm plagiarism detection.

- To the best of our knowledge, this work is the first one on algorithm level similarity assessment.
- We innovatively apply the idea of N-version programming in plagiarism detection.
- We also propose a novel approach that leverages auxiliary information to extract core values, namely the annotation approach. We can do both manual annotation and automatic annotation. Manual annotation is more accurate while automatic annotation is more efficient.
- Besides the LCS similarity metric, we propose to use VDG to measure algorithm level similarity as well. VDG can effectively defend against value reordering attacks.
- The evaluation results show that our approaches to algorithm plagiarism detection are practical, effective and resilient to many automatic obfuscation techniques.

ViewDroid: user interface-based Android app repackaging detection.

- We propose view graph, a user interface-based birthmark for Android apps. To the best of our knowledge, it is the first user interface level birthmark for software plagiarism or app repackaging detection.
- We propose ViewDroid, an Android app repackaging detection system based on view graph. ViewDroid is robust to many code obfuscation techniques. It is efficient and scalable. ViewDroid is a nice complementary approach to current code-level repackaging detection methods.
- We evaluated the obfuscation resilience of ViewDroid by 39 obfuscators from SandMarks [28] and KlassMaster [29], based on the evaluation framework proposed by Huang et al. [30]. The experiment results also show that ViewDroid outperforms Androguard [27] in terms of obfuscation resilience.
- We tested ViewDroid on $10,311$ real-world apps ($573,872$ app pairs) from the Android market. It is detected that about 4.7% apps are repackaging cases. We also evaluated the false negative of ViewDroid.

The false negative rate is 1.3%. The large scale evaluation demonstrates the efficiency and effectiveness of ViewDroid.

Chapter 2

Related Work

The plagiarism detection methods can be categorized as in Table 2.1. The methods LoPD, ValPD and ViewDroid are proposed in this dissertation. The other methods are discussed in the following sections.

2.1 Software Plagiarism Detection

We roughly group the existing software plagiarism detection methods into the following two categories.

2.1.1 Static birthmark-based plagiarism detection

Liu et al. [4] proposed a program dependence graph (PDG)-based approach, which extracts the data and control dependencies from source code. It is vulnerable to some obfuscation techniques such as control flow flattening and opaque predicates. Park *et al.* [8] developed a static API-based birthmark for Java applications. Lim *et al.* [31] used stack pattern-based birthmark, which is only suitable for Java applications. Tamada et al. [32] proposed four static birthmarks for Java programs: Constant Values in Field Variables (CVFV), Sequence of Method Calls (SMC), Inheritance Structure (IS) and Used Classes (UC). These birthmarks can be changed by some obfuscation techniques, such as method call reordering. Myles et al. [7] statically analyzed executables to obtain instruction sequences of length k. Then they used K-gram techniques to measure the similarity. This approach is vulnerable to instruction reordering and junk instruction insertion.

Most above static analysis methods require the source code of the analyzed programs. This limits their practicability since the source code of a suspicious program is not always available. Some of them are platform specific. In addition, they are easy to be bypassed by applying obfuscation techniques.

	PC Apps	Smartphone Apps
User Interface	–	**ViewDroid**
Code Logic	Static source code-based: [32] Static opcode-based: [7] Whole program path-based: [9] PDG-based: [33], GPLAG [4] API-based: [10, 34, 8, 11]. System call-based: [18, 17] Clone Detection: [36, 37, 38, 5, 39], etc	Opcode-based: DroidMOSS [2], Juxtapp [25] AST-based: [35] PDG-based: DNADroid [24]
Program Semantics	VaPD [3], **LoPD**	–
Algorithm-level	**ValPD**	–

Table 2.1. Categories of plagiarism detection methods. Methods proposed in this dissertation are in **Bold**

2.1.2 Dynamic birthmark-based plagiarism detection

Jhi et al. [3] proposed to use core values as birthmark to detect software plagiarism. This approach is lack of formal guarantee, since core value is hard to define. Lu et al. [16] presented a dynamic opcode n-gram birthmark, which is vulnerable to instruction reordering and irrelevant instructions insertion. Myles et al. [9] developed a whole program path (WPP) birthmark, which is robust to some control flow obfuscations such as opaque prediction, but is still vulnerable to many semantic-preserving transformations such as loop unwinding. Tamada et al. [10] used dynamic API birthmark for windows applications. Their approach relied on the sequence and the frequency of API invocations, both of which can be easily changed by reordering APIs or embedding API implementations into the program. Schuler et al. [11] proposed a dynamic API-based birthmark for Java. Wang et al. [18, 17] introduced a system call-based birthmark. Their approach is not suitable for programs that invoke few system calls.

2.2 Smartphone Application Plagiarism Detection and Security

2.2.1 Smartphone App Plagiarism Detection

The smartphone app repackaging problem has drawn great attention from the research community. There are several relevant works on measuring the similarity between Android apps on code level. DroidMOSS [2] leverages fuzzy hash to detect app repackaging. A hash value is computed for each local unit of opcode sequence of the classes.dex, instead of computing a hash over the entire program opcode set. It can efficiently and effectively identify the opcode segments that were left untouched by the lazy repackager and works well when the bytecode is only manipulated at a few interesting points (e.g., the string names or hard-coded URLs). However, some obfuscation, such as noise injection, can evade the detection. DNADroid [24] proposed a program dependence graph (PDG)-based detection approach, which considers the data dependency as the main characteristic of the apps for similarity comparison. DNADroid compares the PDGs within a pre-computed cluster of Android apps using graph isomorphism algorithms. The efficiency of the comparison is further improved in AnDarwin [23] by building semantic vectors from PDG for each method. In general, PDG is resilient against several control flow obfuscation techniques and noisy code insertion attacks that do not modify the data dependency. However, some specific data dependence obfuscations can be designed to evade this approach. For example, PDG can be changed by inserting intermediate variable assignment instructions into the code. Juxtapp [25] proposed a code-reuse evaluation framework which leverages k-grams of opcode sequences to build feature for the feature hashing approach. Features are defined based on the k-grams of various opcode sequence patterns within each basic block. A sliding window will move within each basic block to map the features into bit vectors, which are further combined into a feature metric to help birthmark each app. This detection scheme is able to effectively detect different code reuse situations, including piracy and code repackaging, malware existence, vulnerable code. Special designed code manipulation can potentially destruct the normal opcode pattern of Dalvik bytecode in a very dense fashion. Chen et al. [40] propose a novel app birthmark, which is the geometry-characteristic-based encoding of control flow graph. This approach can effectively and efficiently detect Type 2 and Type 3 clones, where cloned code is syntactically similar with the original code. However, it cannot deal with app repackaging using code obfuscation techniques.

2.2.2 Smartphone App Security.

There are several publications in this category related to ViewDroid. SmartDroid [41] leverages user interfaces to find user interface interactions that will trigger sensitive APIs. It combines the static analysis and dynamic analysis. Chen et al. [42] developed a Permission Event Graph (PEG) to detect, or prove the absence of malicious behavior that is not authorized by users. Zhou et al. [26] proposed a module decoupling method to partition an app's code into primary and non-primary modules and thus to identify the malicious payloads reside in the benign apps. They also develop an approach to extracting feature vectors from those piggy backed apps to help improve the efficiency of the piggyback relationship detection. Grace et al. [43] developed AdRisk to identify risks in ad libraries.

Anand et al. [44] applied concolic testing on Android platform to generate events to trigger view navigation. Our approach pays more attention to the repackaging detection from the primary functionalities of the Android apps and takes into account obfuscation resilience. We also identify certain features for the nodes and edges during the view graph construction to improve the efficiency of our detection.

2.3 Clone Detection

Clone detection is a technique to find duplicate code. Besides being used to decrease code size and facilitate maintenance, clone detection can also be used to detect software plagiarism. Existing source code clone detection techniques include String-based [36], Tree-based [45, 38], Token-based [37, 5, 39] and PDG-based [46, 47, 33]. Sæbjørnsen et al. [48] proposed a tree-based clone detection in binary code. Since most clone detection techniques do not take code obfuscation into consideration, they are not robust to many obfuscation techniques. As a result, when being applied to detect software plagiarism detection, clone detection approach can be easily evaded by attackers.

2.4 Software Watermarking

There exists a large volume of literatures on software watermarking (e.g. [49, 50, 51, 52, 53].). Software watermarking embeds secret information into a program to protect intellectual property. The embedded information is added to program during the implementation and can be extracted to identify the ownership of the program. Software Watermarking is a prevention scheme against software plagiarism. However, most of the commercial and open source programs do not have watermarking embedded. As a result, the detection scheme of plagiarism is necessary.

2.5 Path Deviation Detection

Brumley et al. [54] first proposed the path deviation idea and used it to find protocol errors in different implementations. We adopted their path deviation idea and applied it to a new context of software plagiarism detection. [54] only compares the output of executions. This is not sufficient for software plagiarism detection, because independent software products may have the same functionality, i.e. the same input-output pairs. As a result, in addition to output, we need to compare the execution paths, which is more challenging. We propose new techniques such as path equivalence detection to deal with automatic code obfuscation attacks and eliminate false positives and false negatives. We have evaluated path deviation and path equivalence detection in this new context with presence of automatic obfuscation attacks and obtained promising results.

2.6 N-version programming

N-version programming [55, 56, 57, 58] is defined as independently developing multiple functionally equivalent programs following the same specification. It explores the diverse characteristics of multiple independent implementations of the same specification. In recent years, the N-version programming approach started to be applied in security vulnerabilities detection area. Nagy et al. [59] used the concept of N-version Programming to detect zero-day exploits in web applications. Cox et al. [60] extended the idea of N-version programming to N-Variant Systems. They automatically diversified the original program such that the client-observable behavior remained the same on normal inputs, but became different on abnormal inputs corresponding to a particular class of attacks. The system was able to detect attacks exploiting injected code and absolute memory addresses.

2.7 Test input generation

Both LoPD and ValPD relies on input. There are a number of choices on how to generate an input. The first option is to generate a random input, ideally independent, for each run using methods such as fuzz testing [61, 62]. The second option is symbolic execution [13] and automatic test case generation using systematically white-box exploration (also called, concolic testing, directed systematic path exploration, etc) [63, 64, 65, 66, 67]. Path constraints are collected and manipulated to cover different paths, and a constraint solver [68, 69] is usually used to generate the input that satisfies the corresponding path constraints. By doing this, each run is guaranteed to hit a different path.

Chapter 3

LoPD: Logic-based Software Plagiarism Detection

In this chapter, we propose a logic-based software plagiarism detection approach, which satisfies all three requirements discussed in Chapter 1.

LoPD is based on the philosophy: *search for any dissimilarity between two programs and if not found, it is likely they are the same.* We leverage symbolic execution [13] and weakest precondition [14, 15] to systemically find such dissimilarity.

LoPD is resilient to most automatic obfuscation techniques, which change the syntax of a program but preserve its semantics. Since the symbolic formula and weakest precondition, applied by LoPD, can capture semantics and constraint of an execution path of the tested program, as long as the semantics are not changed, LoPD will detect the semantics equivalence of the execution paths of the plaintiff program and the suspicious program. In addition, since LoPD is a formal program semantics-based approach, we can provide a formal assurance of the resilience against most types of known obfuscation attacks, as discussed in Section 3.3. Moreover, LoPD provides theoretical guarantees of the high detection accuracy, subject to the limitations of the current symbolic execution tools and constraint solvers.[1]

---

[1] According to the Rice's Theorem, testing any non-trivial computer program property is undecidable. We do not aim to solve this undecidable problem, but rather to develop tools for practical use with some degree of formal guarantee. All the conclusions, we draw from this research are subject to the limitations of automated theorem proving or constraint solving and other undecidable factors.

3.1 Overview

3.1.1 Problem Statement

The goal of our work is to automatically detect software plagiarism for nontrivial programs in the presence of automatic code obfuscation. To be more specific, given a plaintiff program P and a suspicious program S, our purpose is to detect if S is generated by applying *automatic* semantics-preserving transformation techniques on P. That is, we provide a Yes/No answer to the question: are S and P semantically equivalent? Automatic semantics-preserving transformation changes the syntax of the source code or binary code of a program but keeps the function and the semantics of the program by automated tools (e.g., Loco [70], SandMark [71]) with little human effort. The reason that we only focus on automatic code transformation is as follows. Although an exceptionally sedulous and creative plagiarist may manually obfuscate the plaintiff code to fool any known detection technique, the cost is sometimes higher than rewriting his own code, which conflicts with the intention of software theft. After all, software theft aims at code reuse with disguises, which requires much less effort than writing one's own code.

We have two assumptions: (1) we have preknowledge about the plaintiff program, e.g., the input space; (2) while we do not require access to the source code of the suspicious program, we assume its binary code is available.

3.1.2 Basic Idea

Our basic idea is to search for any difference between the plaintiff program and the suspicious program, and if differences are found, there is no plagiarism; otherwise, it is likely that plagiarism exists.

At high level, three things characterize program behavior—input, output, and the computation used to achieve the input-output mapping. Based on our design philosophy, LoPD tries to rule out dissimilar programs by finding an input that will cause these two programs to behave differently, either with different output states or with different computation paths. Whenever we find such an input, we can assert that the plaintiff program and the suspicious program are either functionally or computationally different and is thus not plagiarism via automated code obfuscation.

Given an input, the comparison between output states is relatively straightforward: since the plaintiff has the preknowledge of his own software, he can specify which output variables and states are semantics-relevant (e.g., the terminal output or the file modification) and how to measure the similarity between output states (e.g., the mathematic computation programs require the exactly same result, while the error messages from Web servers can tolerate some literal differences).

The challenge is how to compare the semantics of computation paths. Computation path, also known as execution path, is a sequence of all instructions executed during one round execution. The *semantics* of an execution path can be captured by symbolic execution. To be more specific, symbolic expressions of output variables in terms of input variables along with a path constraint represent the semantics of an execution path. The following is an example. n is the input variable and a is the output variable. There are two execution paths. The semantics of path 1 is the path constraint "$n > 0$ is true" along with the output expression $a = n - 1$. In path 2, the semantics is the path constraint "$n > 0$ is false" and the output expression $a = 2n + 2$.

The code	**Path 1**	**Path 2**
n = read()	input: $n > 0$	input: $n <= 0$
if $n > 0$ **then**	True	False
$a = n - 1$	$a = n - 1$	
else		
$a = n + 1$		$a = n + 1$
$a = a * 2$		$a = (n + 1) * 2$
end if		
print a	output: $a = n - 1$	output: $a = 2n + 2$

Instead of directly comparing two execution paths, we propose a novel approach based on the concept of path-deviation [54]. It is motivated by the fact that if one program is an automatic semantics equivalent transformation of another program, these two programs would have *one-to-one (1:1) path correspondence*, as defined in Definition 1. That is, given the same input, the execution of each program follows a certain path, respectively, and when given a different input, the programs should either both follow their original path or both execute new paths. Note that there is one exception: when an execution path of one program is split into two semantically equivalent paths for the obfuscation purpose, there would be no one-to-one path correspondence, but it is still a software plagiarism case. We will therefore also handle this semantically equivalent path splitting problem in our detection system.

Definition 1. *Given two programs P, S, their input spaces are I_P and I_S, respectively. $\forall x_1, x_2 \in I_P \cup I_S$, the execution paths of P with input x_1, x_2 are e_{p1}, e_{p2}, respectively and the execution paths of S with input x_1, x_2 are e_{s1}, e_{s2}, respectively. If $e_{p1} = e_{p2} \leftrightarrow e_{s1} = e_{s2}$, P and S have* ***one-to-one (1:1) path correspondence****.*

If we can find two inputs which could cause one program to execute the same path with both inputs, while the other program to execute two different paths with these two inputs, we can rule out the case; that is, the suspicious program will not be considered as a plagiarized one. We call such two programs having *path deviation*, whose formal definition is:

Definition 2. *Given two programs P, S, their input spaces are I_P and I_S, respectively. $\exists x_1, x_2 \in I_P \cup I_S$, the execution paths of P with input x_1, x_2 are e_{p1}, e_{p2}, respectively and the execution paths of S with input x_1, x_2 are e_{s1}, e_{s2}, respectively. If $(e_{p1} = e_{p2} \wedge e_{s1} \neq e_{s2}) \vee (e_{p1} \neq e_{p2} \wedge e_{s1} = e_{s2})$, P and S have* ***path deviation****.*

Figure 3.1 illustrates this path deviation idea. Given the same input x_1, program P and S take the execution path e_{p1} and e_{s1}, and output O_p and O_s, respectively. If $O_p \neq O_s$, it means e_{p1} is different from e_{s1}, so it is not a plagiarism case. If $O_p = O_s$, our next step is to try another input x_2, hoping that (1) P will take the same path e_{p1} but S will take a different path e_{s2} given x_2 or (2) the output $O'_p \neq O'_s$. In either case, it is not a plagiarism case. If neither of the above two cases occurs, we will try another input. If after many iterations we still cannot find such a deviation-revealing input, it indicates the two programs are likely to be the same.

However, a path deviation may be caused by the path splitting obfuscation, that is, e_{s1} and e_{s2} in Figure 3.1 are semantically the same. Therefore, when we find a deviation, we need to check the semantics equivalence of the deviated paths (e.g. e_{s1} and e_{s2}). Only when semantics differences exist between the two paths, we claim that the two programs have *true path deviation* and they are dissimilar.

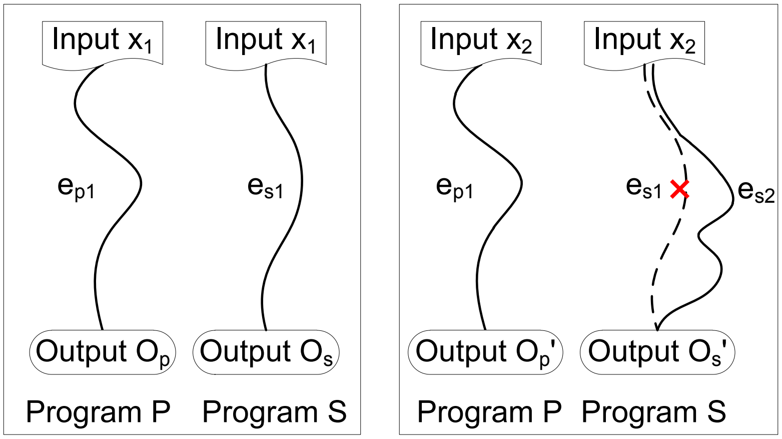


Figure 3.1. Path deviation example

Table 3.1. The tabular representation of relations between the reality and the detection results.

			Reality	
			a.Plagiarism	b.Not Plagiarism
Detection Result	Case I. Same Output	I.1. PD [1]	FN [2]	TN [3]
		I.2. No PD	TP [4]	FP [5]
	Case II. Diff Output	N/A	-	TN

[1] path deviation [2] false negative [3] true negative [4] true positive [5] false positive

We leverage the techniques of logic-based execution path characterization including symbolic execution, weakest precondition calculation, and constraint solving (e.g., STP [68, 69]) to find path deviation and to measure the semantics equivalence of two execution paths.

To ensure the effectiveness of our approach, we analyze the possible false detection cases based on the results of output similarity measurement and path deviation detection. The relations between the reality and the detection results are shown in Table 3.1:

- **Case I:** Given the same input, P and S generate the same output.

  Case I.1: Detection result: P and S have path deviation.

  Case I.1.a (False Negative): P and S are indeed software plagiarism. We check the semantics equivalence of e_{s1} and e_{s2} when we find a path deviation. Only when a semantics deviation exists between the two paths, we call the two programs dissimilar and conclude non-plagiarism. Since path equivalence checker applies the weakest precondition (a symbolic formula) that captures formal semantics of a path, and constraint solver that checks the equivalence of symbolic formula, we ensure that there is no false negative caused by the approach. However, this is subject to the limitations of the constraint solving or theorem proving, which we will discuss in the limitation section.

Case I.1.b (True Negative): P and S are indeed not software plagiarism.

Case I.2: Detection result: P and S do not have path deviation.

Case I.2.a (True Positive): P and S are indeed software plagiarism.

Case I.2.b (False Positive): P and S are indeed not software plagiarism. In practice, it is hard to image that two independent nontrivial software will have one-to-one semantically equivalent path correspondence. Therefore, in practice we do not have false positive. The case due to the limitations of the constraint solving will be discussed in the limitation section.

- **Case II (True Negative):** Given the same input, P and S generate different output. P and S are indeed not software plagiarism.

Based on the above analysis, LoPD tries to find a path deviation first and then checks the path equivalence to make sure that such a deviation is a real semantics deviation, not caused by obfuscation. Next we introduce the design details of LoPD.

3.2 Design

3.2.1 Architecture

The overview of the system design is shown in Figure 3.2. We tackle the problem by two phases: Path Deviation Detection and Path Equivalence Checking. In the first phase, we detect whether there exists any path deviation between the

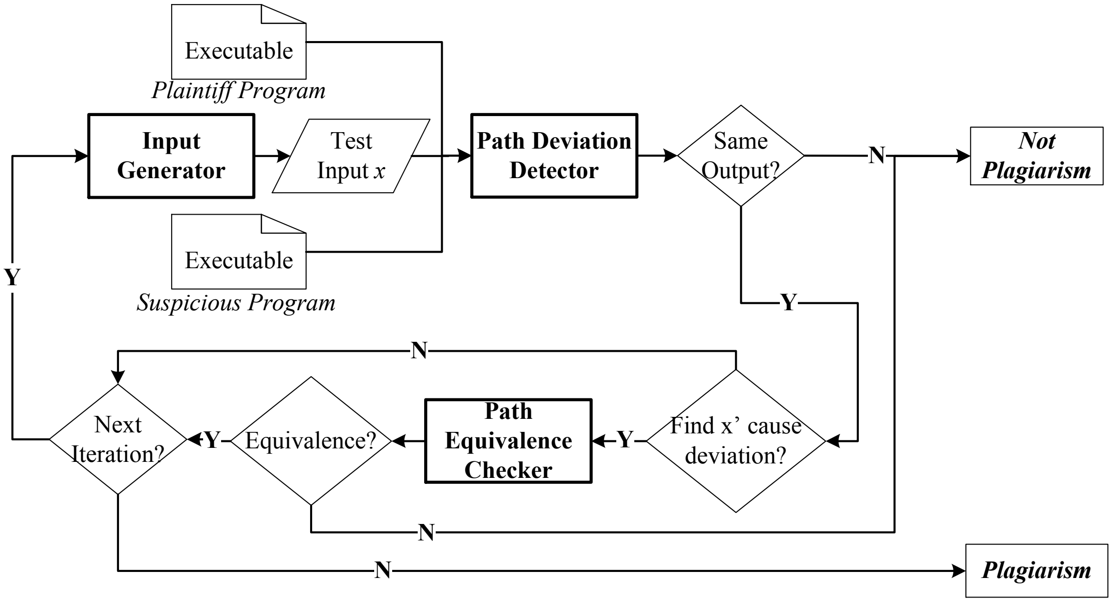


Figure 3.2. LoPD system design

plaintiff and suspicious programs. If there is no path deviation, we conclude that this is a plagiarism case, because we believe it is impossible that two nontrivial independent programs have 1:1 path correspondence. If there is a path deviation, we check whether the deviated path is a semantically equivalent path split from the original one, if yes, this is likely by obfuscation and it is a fake path deviation. If no, it is a true path deviation and thus we conclude it is not a software plagiarism.

In each iteration, the *input generator* generates a test input x. The *path deviation detector* symbolically executes both the plaintiff executable and the suspicious executable in a monitored environment with x as input, records the execution path of each program, and extracts a symbolic formula representing each path. First, we compare the output states of two programs and if they are different, we can claim these two programs are not plagiarism. If the output states are the same, we need to further detect whether there is a path deviation. Based on the two symbolic formulas representing the execution paths, we generate a special "check" formula (Formula (3.1) in Section 4.3), to discover potential path deviations by running the check formula through a constraint solver.

If an input x' that causes path deviation is found, we apply *path equivalence* *checker* to ensure this path deviation is not caused by a path-splitting or path-merging attack. Let d represent the program that executes different paths with x and x'. The path equivalent checker symbolically runs d with x' as the input and extracts a new formula representing the current path. If the two execution paths of d with inputs x and x' are semantically equivalent, it is not a true path derivation. Once we verify the path deviation is not caused by path-splitting, we rule out plagiarism and stop. If we cannot find a path deviation or the path deviation is caused by path-splitting, we start a new iteration by generating a new input to cover a different path. This process can be repeated for a number of iterations. We set a threshold on the maximum number of iterations. The tool terminates either after a true path deviation is found or the number of iterations reaches the threshold.

The detection procedure is described in Algorithm 1. The details of each component are described below.

3.2.2 Input generator

There are several ways to generate an input x for each iteration. The first option is to generate a random input, ideally independent, for each iteration using methods such as fuzz testing [61]. However, random input generation might not be desired. We adopt symbolic execution [13] and automatic test case generation using systematically white-box exploration (also called, concolic testing and directed systematic path exploration) [63, 64, 65, 66, 67]. In this way, each iteration is guaranteed to hit a different path.

We first randomly generate an initial input from the input space. Path constraints are collected during the program execution with the initial input and are manipulated to cover different paths. Then a constraint solver is used to generate the input that satisfies the corresponding path constraints.

3.2.3 Path Deviation Detector

The path deviation detector is used to detect if two tested programs have path deviation, which is formally defined in Definition 2. Generally speaking, given an input x, we are trying to find another input x' that causes one of the program to

Algorithm 1 Path Deviation-based Software Plagiarism Detection

```
Require: Plaintiff Program $P$, Suspicious Program $S$
Ensure: Plagiarism / Not Plagiarism.
for $i = 1$ to max_iteration do
    Generate input $x$ by Input generator.
    $P$, $S$ and $x$ are given to the Path deviation detector. The output states are $O_p$ and $O_s$, respectively. The execution paths are $e_p$ and $e_s$
    if $O_p = O_s$ then
        if The Path deviation detector can find another input $x'$ cause $P$ and $S$ path deviated. then
            The execution paths of $P$ and $S$ with input $x'$ are $e'_p$ and $e'_s$.
            $d \leftarrow P$ or $S$, the one executes different paths with $x, x'$.
            The Path equivalence checker checks the sematic equivalence of $e_d$ and $e'_d$
            if $e_d$ and $e'_d$ are semantically equivalent then
                continue
            else
                return "Not Plagiarism"
            end if
        else
            continue
        end if
    else
        return "Not Plagiarism"
    end if
end for
return "Plagiarism"
```

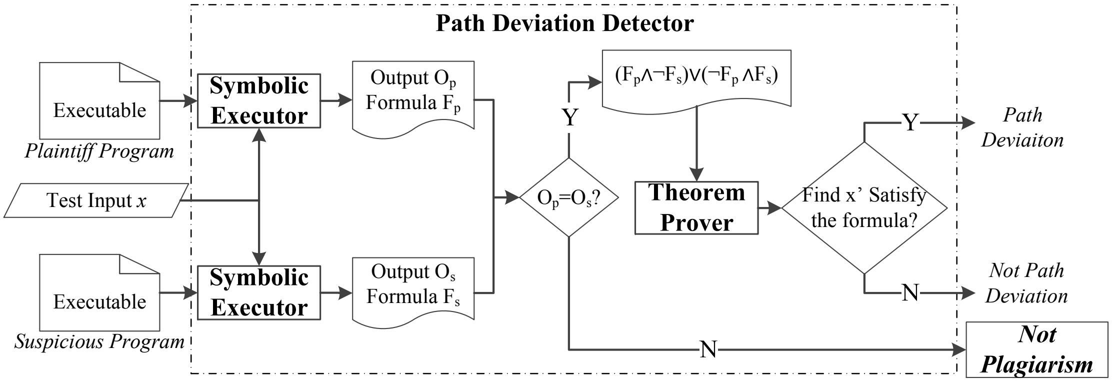


Figure 3.3. Path deviation detector

execute the same path as taking x as input, while the other program to follow a different path from the one taking x as input. We leverage symbolic execution [13] and weakest precondition [14, 15] to find such x'. The design of path deviation detector is shown in Figure 3.3.

The symbolic executor performs a mixed concrete and symbolic execution [66, 72] for each tested program with x as input. In other words, the tested program is first concretely executed with the input x in the executor, which is a monitored environment with taint analysis. The input is the taint seed. The whole execution path is logged, including the executed instructions, the taint information and the output states.

The output states can be specified by the domain experts or the owner of the plaintiff program. They may include the terminal output, the network interface and the modification in file system, etc. Their output states are represented as O_p and O_s, respectively. If $O_p \neq O_s$, programs P and S are semantically different. As a result, we can get the correct conclusion that they are not software plagiarism.

The symbolic execution is operated on the logged concrete execution path. We build a symbolic formula in terms of input variables to express each path constraint. This formula reflects both the semantics of the execution path and the conditions which make the program execute this particular path. We denote the execution paths of plaintiff program and suspicious program with input x as e_p and e_s, respectively. The two formulas that we build based on these two paths are $F_p^O(I)$ and $F_s^O(I)$ parameterized with the input variables I, based on the output state O ($O = O_p = O_s$). These two formulas are built using the technique of *weakest precondition* and have the property that they are true with some truth assignment i ($i \in$ input space) if and only if the program executes the corresponding path on the input i and ends with output state O; i.e., the path is feasible on input i and leads to output O:

$$F_p^O(i) \text{ is true } \textit{iff } e_p \text{ is feasible on input } i \text{ and ends with output } O.$$

Given any input that satisfies the formula, the execution of the program will follow the original path, while given any input that does not satisfy the formula, the execution will follow a different path. As a result, to find a path deviation of plaintiff program and suspicious program, we need to find an input x', which makes the execution path of one program remain the same as its execution path with input x, and the execution path of the other program be different from its path with input x. As a result, we check the satisfiability of Formula (3.1), as used by Brumley et. al. [54], via a theorem prover STP [68, 69].

$$(F_p^O(I) \wedge \neg F_s^O(I)) \vee (\neg F_p^O(I) \wedge F_s^O(I)) \tag{3.1}$$

If Formula (3.1) is satisfiable, STP will return an assignment that satisfies the formula.[2] Without loss of generality, assume the assignment x' satisfies the first part of the disjunction, $F_p^O(I) \wedge \neg F_s^O(I)$. This means that the input x' will cause the first program to follow path e_{p1}, while the path e_{s1} is infeasible in the second program, as shown in Figure 3.1. That is, two programs behave differently on input x and x', unless paths e_{s1} and e'_{s2} are semantically equivalent. If Formula (3.1) is not satisfiable, it means that there exists no input that can deviate the programs from these two paths.

Example. Consider the following two programs: one checks for condition $n > 0$ and the other checks for condition $n > 1$:

$$\begin{aligned} f(n) &= \text{if } (n > 0) \text{ then } 2 \text{ else } 1 \\ g(n) &= \text{if } (n > 1) \text{ then } 2 \text{ else } 1 \end{aligned}$$

Given an input 0 or any negative number, the path constraint formula of f is $\neg(n > 0)$ and the formula of g is $\neg(n > 1)$. The check formula is:

$$(\neg(n > 0) \wedge (n > 1)) \vee ((n > 0) \wedge \neg(n > 1))$$

A constraint solver can solve it with a satisfiable assignment $n = 1$, which causes f to execute a different path but not g. If given an initial input 1, the two programs have different output and we can directly conclude they are different programs If we select a positive number as the initial input, the constraint solver could not find a path deviation and we continue with white-box symbolic exploration to generate

---

[2] When STP cannot solve the formula to give a definite yes or no answer, we simply ignore the case and try next one. We apply the same strategy for the path equivalence checker presented in the next subsection.

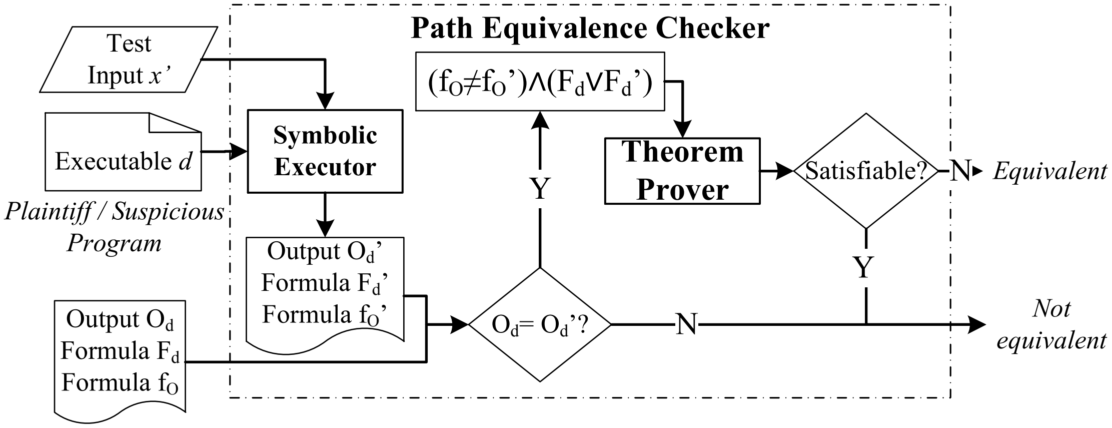

Path Equivalence Checker
Test Input x'
Executable d
Plaintiff / Suspicious Program
Output O_d Formula F_d Formula f_O
Symbolic Executor
Output O_d' Formula F_d' Formula f_O'
$(f_O \neq f_O') \wedge (F_d \vee F_d')$
Y
$O_d = O_d$'?
N
Theorem Prover
Satisfiable?
N
Equivalent
Y
Not equivalent

Figure 3.4. Path equivalence checker

a new input for next round. This process repeats until it hits 0 or a negative number. With symbolic exploration we can reach this desired input in one step, since one of the path constraints is flipped to hit a different path.

3.2.4 Path Equivalence Checker

As discussed above, when we find a path deviation, we need to check whether these two deviated paths are semantically equivalent path splitting to avoid false negative. The following is a simple example of semantically equivalent path splitting. The left is the original code. The right is the code after path splitting, where the value of n decides the path to go but both paths are *semantically equivalent.*

```
$a = n$
```

```
if $n > 0$ then
    $a = n$
else
    $a = n + 1$
    $a = a - 1$
end if
```

The detection of path equivalent is done by path equivalence checker, which is shown in Figure 3.4. The new test input x' is a satisfiable assignment of Formula (3.1) returned by constraint solver in the path deviation detection step, and d represents the program that has different execution paths with input x and x'.

That is, d is either P or S. Taking Figure 3.1 as an example, $d = S$. In other words, in the path deviation detection step, if the first part of the disjunction of Formula (3.1), $F_p^O(I) \wedge \neg F_s^O(I)$, is satisfiable, $d = S$, while if the second part, $\neg F_p^O(I) \wedge F_s^O(I)$, is satisfiable, $d = P$. We compare the semantics equivalence of d's two execution paths, which take x and x' as input, respectively. If these two paths are semantically equivalent, the path deviation is caused by path splitting. We take the next iteration, as shown in Figure 3.2. Otherwise, we can conclude that P and S are not software plagiarism and call such path deviation as a *true path deviation*.

We still apply symbolic execution and weakest precondition to detect path equivalence. Program d is executed with input x' in the symbolic executor, which is the same one as in the path deviation detector. A path constraint formula F_d' and a symbolic formula of output states f_O' are generated. Both of them are in terms of input variables. F_d' captures the conditions that make d follow the same execution path as input x'. f_O' captures the semantics of such execution path. The formulas (F_d, f_O) for the execution path of input x have already been generated in the path deviation detection step.

In an execution path, the truthness and the target of a conditional branch are fixed. By ignoring such conditional branches, we can force a program to follow a particular execution path with any input, although some inputs may cause the program to crash or to get a wrong output. In such way, we can pick any input that satisfies either of the above path constraints (F_d or F_d'), and give it to both execution paths. If these two paths are equivalent, they should get the same results with such input. In other words, if an input assignment satisfies at least one of the path constraint formulas: F_d or F_d', f_O and f_O' should be equal with this input assignment:

$$\begin{aligned}
\text{Path Equivalent} \quad &\Leftrightarrow \quad (F_d \vee F_d') \rightarrow (f_O = f_O') \\
&\Leftrightarrow \quad (f_O = f_O') \vee \neg(F_d \vee F_d') \\
\text{Not Path Equivalent} \quad &\Leftrightarrow \quad \neg((f_O = f_O') \vee \neg(F_d \vee F_d')) \\
&\Leftrightarrow \quad (f_O \neq f_O') \wedge (F_d \vee F_d') \qquad (3.2)
\end{aligned}$$

We check the satisfiability of Formula (3.2) via a constraint solver STP [68, 69].

If it is satisfiable, these two execution paths are not equivalent.

Example. Consider the same path splitting example in this section. Assume n is the input variable, initial input x is $n = 10$ and x' is $n = -1$:

$$\begin{aligned} f_O(n) &= n & F_d &= (n > 0) \\ f'_O(n) &= n + 1 - 1 = n & F'_d &= \neg(n > 0) \end{aligned}$$

Formula (3.2) is $(n \neq n) \wedge (n > 0 \vee \neg(n > 0))$, which is not satisfiable. As a result, the two paths are equivalent.

3.3 Counterattack Analysis

Since our logic-based method captures path semantics by weakest precondition, in theory it is resilient to most known attacks such as noise injection, statement reordering, register and constant splitting and opaque predication. In practical implementation, we need to take into consideration the limitations of symbolic execution, theorem proving, and weakest precondition calculation.

Noise instruction/data injection: Suppose an irrelevant statement S_1 is inserted right after statement S_0. Given a postcondition R, the weakest precondition for the original program is $wp(S_0, R)$, while the weakest precondition for the new program is $wp(S_0; S_1, R)$. Because S_1 is an irrelevant statement we have $wp(S_0; S_1, R) = wp(S_0; wp(S_1, R)) = wp(S_0, R)$. Similarly the equation also holds in the cases of inserting multiple instructions. As a result, LoPD is resilient to noise injection.

Statement reordering: Two instructions S_1 and S_2 can be reordered only when there is no data or control flow between them: $wp(S1; S2, R) = wp(S2; S1, R)$. Similarly, the weakest precondition also remains the same when reordering multiple instructions. So LoPD is resilient to instruction reordering.

Instruction splitting and aggregation: Two instructions S_1 and S_2 could be merged into one instruction S_0; in the other direction, instruction S_0 could be split into two instructions S_1 and S_2. Since they are semantically equivalent, there is $wp(S_0, R) = wp(S_1; S_2, R)$. Hence, LoPD is resilient to instruction splitting and aggregation obfuscation.

Opaque predicate: One opaque predicate E is inserted right before statement S_0. If E is an always true predicate, $wp(\text{if } E \text{ then } S_0 \text{ end}, R) = E \Rightarrow wp(S_0; R) = wp(S_0, R)$. Similarly, the weakest precondition also remains the same when S_0 represents multiple instructions or E is an always false predicate.

Path splitting and merging: By applying symbolic execution and weakest precondition, we can effectively detect semantically equivalent path splitting / merging.

In summary, LoPD provides a formal guarantee of resilience against most types of obfuscation attacks.

3.4 Implementation and Evaluation

We implement a prototype system. The symbolic executor is built atop Bitblaze infrastructure [72, 73]: we leverage their whole-system emulator, TEMU, to concretely execute the tested programs and record the whole execution path; we use vine, the static analysis component, to analyze the execution paths and extract their symbolic formulas. We apply STP [68, 69] as the constraint solver to solve path deviation Formula 3.1 and path equivalence Formula 3.2. STP is a decision procedure whose output indicates whether the formula is satisfiable or not. If so, it also provides an assignment to variables that satisfies the input formula. We integrate all the above components and implement an automatic software plagiarism detection system in C and Python.

Our evaluations are in two categories: software plagiarism case and different program case. The evaluation is performed on a Linux machine with Intel Centrino duo 1.83GHz CPU and 2 GB RAM.

Table 3.2. The tested programs and their running time per iteration for the same program case (in seconds).

Name	Type	Execution Time (s)							
		IG[1]	PDD[2]			PEC[3]			Total
			DR[4]	FE[5]	SS[6]	DR[4]	FE[7]	SS[6]	
THTTPD	HTTP server	1.08	6.21	10.32	1.17	6.34	12.32	2.13	22.78
mini_httpd	HTTP server	0.92	6.98	8.04	1.08	6.59	11.52	3.42	21.55
7za	File archiver	12.68	48.53	28.39	12.96	43.73	30.46	18.72	107.47
gzip	File archiver	4.89	13.87	2.53	5.07	14.83	3.69	7.02	30.36
Ford-Fulkerson	Maximum flow	1.62	6.11	7.18	1.52	5.78	9.27	3.71	18.43
tcc	C compiler	2.89	58.91	27.25	3.30	62.91	32.83	5.36	112.57

[1] Input Generator [2] Path deviation detector [3] Path equivalence checker [4] Dynamic Running on TEMU [5] Formula (3.1) extraction [6] STP slover [7] Formula (3.2) extraction

3.4.1 Case Study I: the Same Programs

In this section, we evaluate the effectiveness of LoPD in the software plagiarism case, where one program is a semantics-preserving transformation of the other program. We have 6 tested programs as shown in Table 3.2: thttpd[3], mini_httpd[4], 7-Zip[5], gzip[6], Ford-Fulkerson maximum flow implementation [74] and tcc [7]. The input variables of thttpd and mini_httpd are the HTTP requests and the output states are the HTTP response according to a particular request. The input variables of the Ford-Fulkerson maximum flow implementation are a flow network and the output state is the calculated maximum flow. For the other three programs, the input variables are the input files and the output states are the generated new files.

For each program, we generate different semantics-preserving executable files by compiling the source code using gcc/g++ (with different optimization options: -O0, -O1, -O2, -O3 and -Os) and tcc. Besides, we apply Diablo, a link-time optimizer [75] and Loco [70], an obfuscation tool based on Diablo to generate two additional executables. Different compilers and different levels of optimization can change the syntax of executables, e.g., "-freorder-blocks" reorders basic blocks, "-funroll-loops" unwinds loops and "-finline-small-functions" inserts small functions' definitions in their caller [76]. Diablo rewrites the binaries during link-time. Loco can obfuscate binaries by control flow flattening and opaque predicate. Hence, we have 8 different executables for each program.

We use LoPD to do pairwise comparison of the generated 8 executables for each program in Table 3.2. We set the threshold of the maximum number of iterations to be 100. For all 168 tested pairs (28 executable pairs for each of the 6 tested programs), LoPD do not find any true path deviation. That is, LoPD draws the right conclusion that they are software plagiarism cases. There is no false negative.

Path splitting resilience check. In order to test the resilience of LoPD to semantically-equivalent-path splitting/ merging attacks, we manually add 2 to 3 such split paths in the source code of each program in Table 3.2. Briefly, we find a code segment $s_1,\ s_2,\ ...\ s_n$, (s_i could be any type of statement, e.g., assignment, declaration, conditional branch, etc). We obfuscate this segment by independent statement reordering, variable splitting/merging, opaque predicate, etc. Then we

---

[3] `http://www.acme.com/software/thttpd/`

[4] `http://www.acme.com/software/mini_httpd/`

[5] `http://www.7-zip.org/`

[6] `http://www.gnu.org/software/gzip/`

[7] `http://bellard.org/tcc/`

add the `if...else` statement, where if c is true, the original segment will be executed; otherwise, the obfuscated segment will be executed. As demonstrated in the following example, the left part is the original code and the right part is the code after path splitting. We compile the new code into executable and compare it with one of the original executables by LoPD. LoPD finds no dissimilarity between the obfuscated and original executables within 100 iterations. It indicates the two programs are software plagiarism, as expected.

```
...
$s_1$;
$s_2$;
...
$s_n$;
...
```

```
...
if $c$ then
   $s_1;\ s_2;\ ...\ s_n;$
else
   $obf(s_1;\ s_2;\ ...\ s_n;)$
end if
...
```

The execution time per iteration is also shown in Table 3.2. The listed time is the average running time of 28 executable pairs for each program and the path splitting experiment. The execution time per iteration is within two minutes for test cases. Note that, the average total time for each iteration is not the sum of the other running time in this line, because path equivalence checker is only needed when there is a path derivation. The total execution time of 100 iterations is within three hours, which is reasonable for offline detectors.

3.4.2 Case Study II: Different Programs

In this section, we evaluate the effectiveness of LoPD in determining non-plagiarism cases. In the first part of this evaluation, we evaluate different programs that have the same purpose and are supposed to generate the same output when given the same input, but there may exist some inputs that cause two programs to generate different outputs, due to either implementation errors or functional extension. The first three lines in Table 3.3 are such program pairs.

Table 3.3. The tested programs and their running time per iteration for the different programs case (in s).

ID	Program P	Program S	Execution Time (s)								
			IG [1]	PDD [2]				PEC [3]			Total
				DR_P [4]	DR_S [5]	FE [6]	SS [7]	DR_d [8]	FE [9]	SS [7]	
1	THTTPD	mini_httpd	1.08	6.21	6.98	10.23	1.38	6.83	12.71	2.35	32.87
2	7za	gzip	12.68	48.53	13.78	18.65	10.19	22.80	20.81	12.39	124.83
3	Ford-Fulkerson	Push-relabel	1.62	6.11	6.95	10.41	1.45	6.94	11.83	3.21	48.52
4	Ford-Fulkerson	Dijkstra shortest path	1.62	6.11	5.26	7.86	2.12	-	-	-	22.97
5	THTTPD	gzip	1.08	6.21	13.87	7.27	1.32	-	-	-	29.75
6	tcc	gzip	2.89	58.91	13.87	17.49	5.12	-	-	-	98.28
7	Ford-Fulkerson	7za	1.62	6.11	48.53	20.90	13.21	-	-	-	90.37

[1] Input Generator [2] Path deviation detector [3] Path equivalence checker [4] Dynamic Running of P on TEMU

[5] Dynamic Running of S on TEMU [6] Formula (3.1) extraction [7] STP slover

[8] Dynamic Running of d ($d = S$ OR P) on TEMU [9] Formula (3.1) extraction

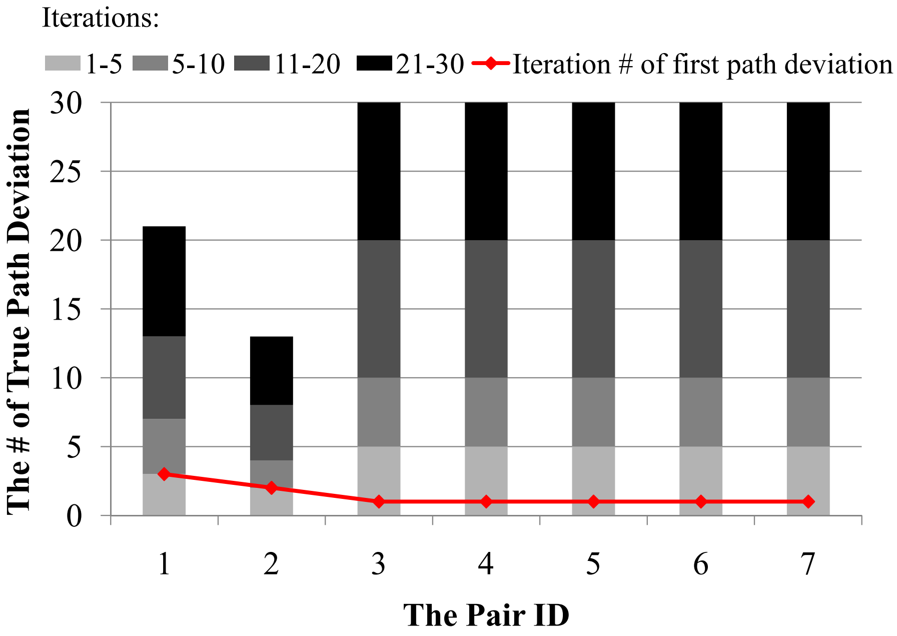


Figure 3.5. The number of path deviations discovered within the first N iterations.

Instead of terminating the detection process as long as we find a true path deviation, we repeat 30 iterations and count the number of path deviations we discover for each program pair. The results are shown in Figure 3.5. The x-axis are different program pairs, whose IDs are the same as in Table 3.3. The bars indicate the count of true path deviations LoPD finds within N ($N = 5, 10, 20, 30$) iterations. The red line shows the number of iterations when LoPD find the first true path deviation.

Thttpd and mini_httpd are two HTTP servers. If their settings are the same, both of them should give the same response when receiving the same request. The first path deviation happens in the $3rd$ iteration. We find total 21 true path deviations within 30 iterations. The deviations are caused because one of the programs does not follow the HTTP protocol specifications and has bugs in its implementation. A path deviation example is shown in Figure 3.6. When given request x, both of them normally response “`200 Ok`”. Based on x, LoPD finds another input x' that causes path deviation, where mini_httpd still returns “`200 Ok`”, but thttpd returns “`400 Bad Request`”.

```
 input x :   00000000 48 45 41 44 20 2F 69 6E 64 65 78 2E 68 74 6D 6C 20 48 54 54 50 2F 31 2E 30 0A 0A 0A HEAD /index.html HTTP/1.0...
input x′ :   00000000 48 45 41 44  20 2F 69 6E  64 65 78 2E  68 74 6D 6C  20 48 01 01  10 FF FF 02  01 0A 0A 0A HEAD /index.html H.........
```

Figure 3.6. Path deviation example of THTTPD vs. mini_httpd.

The second program pair, 7za and gzip, are two file compression tools. If given the particular parameters (e.g., no parameter for gzip and `a -tgzip` for the 7za), they can generate the same output file when operating on the same input file. The first path deviation is found in the second generation. There are 13 path deviations out of 30 iterations. More specifically, when using them for file compression, there is no path deviation for these two programs, but when using them for file decompression, we can find a path deviation in most iterations. One example of the path deviation is: the original input x is a normal `.gz` file, which both programs compress correctly; LoPD generates a new input file x' based on x; both 7za and gzip report a CRC-Failed upon x'. After that, gzip terminates without decompression, whereas 7za continues and generates a decompressed file anyway.

Ford-Fulkerson and Push-relabel are two maximum flow implementations, using different algorithms. Given the same flow network, they should always calculate the same maximum flow. This is an example of the case that the same outputs do not indicate the software plagiarism, because the two programs are computationally different, i.e. applying different algorithms in this case. LoPD can find true path deviations in all 30 iterations, although they can always get the same output. The reason of the existence of true path deviation along with identical output can be explained by a simplified analogy as follows. For any input $x \neq 1$, we can find an input $x = 1$ to cause a path deviation. For example, with an initial input $x = 2$, both programs execute $x = x*2$; a new input $x = 1$ will make the left program still execute $x = x*2$ while the right program goes to another path $x = x + 1$. Their results remain equal. However, it is a true path deviation, because Formula (3.2), $(x * 2 \neq (x+1)) \wedge ((x > 1) \vee \neg(x > 1))$ in this case, is satisfiable and then the two execution paths of the right program are not semantically equivalent.

For all evaluation in this part, within 3 iterations, LoPD draws the right conclusion that they are two different programs.

```
int $x$
...
if $x > 0$ then
    $x = x * 2$
else
    $x = x + 1$
end if
```

```
int $x$
...
if $x > 1$ then
    $x = x * 2$
else
    $x = x + 1$
end if
```

The second part of the evaluation is on different programs that may or may not have the same purpose, but generate different outputs by given the same input. Because LoPD relies on two programs taking the same input, but for some program pairs, the intersection of two programs' input spaces is empty, e.g., thttpd vs. tcc, we can easily rule out software plagiarism case when one program crashes or returns an error message and the other program executes normally. Hence we only choose certain pairs that have common inputs. The last 4 lines in Table 3.3 are such program pairs. Since in most cases, two programs of such pairs cannot generate the same output regarding the same input, we can simply draw the conclusion that they are different programs by comparing the outputs. However, in order to evaluate how different the paths are in this case, we use LoPD to find path deviation regardless of their outputs. Similar to previous evaluation, we do not terminate the detection when we find a path deviation, although we have already gotten the right conclusion that they are different programs. LoPD continues until finishing 30 iterations.

The results are shown in Figure 3.5 with pair ID $4-7$. For each pair, LoPD can find true path deviation in all 30 iterations. The results are as expected, since two programs in each pair have different functionalities and it is not hard to imagine that both their path constraints and output states are different.

The execution time is shown in Table 3.3. The total running time per iteration is longer than the software plagiarism case, because in most iterations path equivalent checker is invoked. In real case, we do not need to run all 30 iterations as in this experiment. As long as we find a true path deviation, LoPD will terminate. For all 7 tested pairs, the first path deviation is discovered within 5 minutes.

Summary. We evaluated the effectiveness and efficiency of LoPD in both the same program and different program cases in this section. The evaluation result demonstrates that LoPD can effectively and efficiently detect the software plagiarism. LoPD can quickly find the dissimilarity between two different programs. It sheds some light on the selection of the maximum iteration threshold. Since in the evaluation of different program cases LoPD can find the first true path deviation within 3 iterations and more than 10 true path deviations within 30 iterations, we believe normally 100 iterations is a reasonable tradeoff between the accuracy and efficiency.

3.5 Discussion

First, LoPD is not suitable for small programs, because when the program logic and semantics are too simple, it is possible that two programs, such as bubble sort and quick sort, have the one-to-one path correspondence. However, for nontrivial software products, it is unlikely that two independent programs have such path correspondence. Therefore, in practice, we do not need to concern about these potential false positive cases.

Second, LoPD is limited theoretically by the capability of the constraint solving or theorem proving. In the path deviation detector, when the constraint solver finds a satisfying assignment to the formula, it is surely correct. However, when it says *no*, it could really mean the formula is not satisfiable, or the solver cannot find a satisfying assignment due to its limited capability. In this case, it can potentially lead to false positives. Our solution is to iterate many rounds on path deviation detection. It would be practically not possible that for a large number of rounds, with a large number of different paths, the constraint solver will consistently report no on satisfying assignments. In the path equivalence checker, the similar can happen and LoPD can theoretically report false negative. In our experiments, we have not seen such false positives or false negatives.

Third, LoPD needs to repeat the iteration until a true path deviation is found or the maximum number of iterations is reached. Therefore, the threshold of such number is a tradeoff between the accuracy and the efficiency. A low threshold takes less time but may cause false positive, while a high threshold decreases the possibility of false positive but takes more time. The evaluation results in Section 3.4 give us some hints about threshold selection: LoPD can quickly find the true path deviation for two different programs (within 3 iterations in all evaluated cases). As a result, we believe setting the threshold at 100 is reasonable. We can also leverage the preknowledge of the plaintiff program to customize the threshold, e.g., for programs with less input dependent conditional branches, we choose a lower threshold and otherwise we set a higher threshold.

Fourth, LoPD may find path deviations for two versions of the same software, if one fixed some bugs in the other one or added new functions. LoPD reports that they are not semantically equivalent. This is true. A similar situation happens when an attacker steals a program and improves it. In fact, LoPD comes to the right conclusion that the two programs are not semantically equivalent, even if the they may be quite similar. Note that in this case the transformation is not achieved automatically but involves human efforts. In order to be resilient to manual modification on plaintiff programs, LoPD could provide a user interface that presents the dissimilarity it finds (e.g., differences in the outputs, the input that causes path deviation) to users and let users make a decision about whether to continue the detection to find another difference or to terminate the process and draw the conclusion. A possible alterative solution is to find all different outputs and path deviations within the maximum count of iterations and calculate a dissimilarity score, which can help users to make a final judgment.

In addition, LoPD focuses on the detection of whole-program plagiarism, where a plagiarist copies the whole plaintiff program and uses it as a finished software product. Whole-program plagiarism detection is very useful in real world. For example, on Android application market, many software plagiarism cases are just repackaging, which are the whole-program plagiarism cases. We view our proposed whole program plagiarism detection approach, based on formal program semantics foundation, as a major milestone towards solving the partial software plagiarism problem. Without a deep understanding of the whole program plagiarism problem, the partial software plagiarism problem probably won't be solved with rigorous soundness and completeness.

3.6 Summary

In this chapter, we propose LoPD, a novel logic-based software plagiarism detection approach. LoPD leverages symbolic execution and weakest preconditions to capture the semantics of execution paths. In addition to formal assurance of resilience against most types of known obfuscation attacks, LoPD provides theoretical guarantee of the high detection accuracy. Our evaluation results indicate that LoPD is both effective and efficient in detecting software plagiarism.

Chapter 4

ValPD: Value-Based Algorithm Plagiarism Detection

In order to detect algorithm plagiarism, we need to find an algorithm's distinct code-level characteristics that cannot be concealed. Such distinct characteristic is considered as a *signature*. There are two key challenges: (1) what is a good signature of an algorithm; (2) how to extract the signature of an algorithm from its implementations. In this work, we develop a dynamic value-based plagiarism detection methodology that addresses both challenges. First, we use *core values*, i.e., the critical runtime values that are irreplaceable and uneliminatable for all implementations of the same algorithm, as the "signature" of an algorithm. Then we propose two novel approaches to extract core values from programs' runtime values: the N-version approach and the annotation approach. After that, we propose two metrics: the longest common subsequence (LCS) and the value dependence graph (VDG) to assess the similarity between core values extracted from an algorithm's plaintiff implementation and its suspicious implementation.

To the best of our knowledge, there was no previous work discussing similarity assessment on the algorithm level.

4.1 Problem Statement

The goal of our work is to automatically detect algorithm plagiarism, i.e., given one (or more) implementation(s) of a plaintiff algorithm and one suspicious program, the proposed methods can automatically assess their algorithm-level similarity. In real-world, the verdict of an algorithm plagiarism case is often algorithm specific, therefore we believe it is more meaningful to provide an algorithm-level similarity score than to draw a simple yes/no conclusion in algorithm plagiarism detection. We also realize that no universal detection threshold can fit all algorithm plagiarism cases because the potential threshold for each case may vary due to the algorithm specific factors, e.g., how complex the algorithm is, how specific it is described, etc. Therefore, instead of giving a yes/no answer, our approach provides users with similarity scores between programs and lets users make their own decisions.

The scope of this work: we only focus on the computational algorithms, that is the output of an algorithm is the computational calculation result of the input variables. Those algorithms whose output is a selection or a permutation of the input, e.g. sorting algorithms, are out of the scope of our work.

This work is based on the following assumptions: (1) We have the source code of at least one implementation of the plaintiff algorithm; (2) We have preknowledge (e.g., input and output) about the implementation(s) of the plaintiff algorithm; (3) We assume the plaintiff has no access to the source code of the suspicious program, but can provide the executable file of the suspicious program to the detector. These assumptions are reasonable in the real world. In most cases, the owner of an algorithm must have implemented the algorithm and is willing to provide the source code in order to win a plagiarism lawsuit. In addition, the owner must have preknowledge on her/his own algorithm.

4.2 Signature Selection

The first challenge in this work is to identify and represent the signature of an algorithm. To address this challenge, we first discuss and compare several candidates, and then explain why core values are selected as the signature of an algorithm.

4.2.1 Signature Candidates

There exist a wide range of properties that may be used as a potential signature to characterize an algorithm.

System call sequence/graph is an essential characteristic of a program that invokes many system calls. However, an examination of the algorithms listed in the "*Algorithm Design*" book [77] indicates that few of these algorithms involve system calls. This indicates that system call sequence/graph is not suitable to characterize an algorithm.

Function call sequence/graph. Since most algorithms use functions to reduce code duplication and to improve modularity and readability, function calls are better than system calls in this aspect. However, programmers have huge flexibility to choose when and how to use functions. In addition, function call sequence/graph can be easily changed by splitting or merging functions, or by inserting useless functions.

Control flow graph (CFG) represents the control flow between basic blocks. When an algorithm is implemented by different programmers, the implementation details could cause significant differences in CFGs. Implementations in different programming languages can also lead to different CFGs. In addition, attackers can apply obfuscation techniques, such as opaque predicates, control flow flattening and loop unwinding, to change CFGs.

Data flow graph is similar to CFG. Graphs are used to represent data flows between basic blocks. Similar to CFGs, basic blocks as well as their relations in data flow graphs are not stable when an algorithm is implemented in different ways. Moreover, this property could be easily manipulated by basic block splitting, irrelevant basic block injection, etc.

Instruction level control dependence characterizes the instruction level control relations in a program. It suffers the same problem as the CFG.

Instruction level data dependence characterizes the relations among *runtime values.* We observe that when feeding different implementations of the same algorithm with the same input, some runtime values cannot be replaced or eliminated. Therefore, these runtime values along with their dependence, e.g., value sequence or value dependence graph, are a good candidate to characterize an algorithm.

Given the comparison results, we choose the irreplaceable and uneliminable values, namely *core values*, as the signature to characterize an algorithm.

4.2.2 Core Values

Runtime values are the values in the output operands of machine instructions executed during runtime. Given an input, the *core values* of an algorithm are a subset of the runtime values of its implementations. They are derived from the input and cannot be replaced or eliminated by implementing the same algorithm in different ways.

Jhi et al. [3] have demonstrated that core values exist at implementation level. Our experiments in Section 6 demonstrate the existence of core values at algorithm level. The approach used to extract program's core values by Jhi et al. [3] is not suitable to obtain core values at algorithm level, since some of program's core values are not core values of the algorithm behind this program. Consider two programs independently implementing the same algorithm, the core values of the programs may be different, but the core values of the algorithm should remain the same.

In the next section, we propose two novel approaches to extract algorithm-level core values.

4.3 Our Approaches

In Section 4.2, we show that core values are a signature of an algorithm implemented in a program. The next challenge is how to extract core values from a program.

In principle there could be two ways to find core values. First, if we know what core values are, we can directly identify them. Second, if we do not know what core values are but we do know what core values are not, we can prune the non-core values and hopefully the remaining set of values mainly contains core values, if not all. Based on these two ways, we propose the *N-version* approach to indirectly extract core values and the *annotation* approach to directly extract core values.

4.3.1 N-version Approach

The *N-version* approach is inspired by N-version programming [56, 78]. We use this approach to filter out the diversities in independent implementations of the same algorithm while keeping the persistent runtime values.

We identify a subset of non-core values and then refine runtime values by filtering out these non-core values. Let $P_{\mathcal{A}}$ be an implementation of an algorithm $\mathcal{A}$, $v_{P_{\mathcal{A}}}$ be a runtime value of $P_{\mathcal{A}}$ taking I as input, and $Q_{\mathcal{A}}$ be any other implementation of $\mathcal{A}$. Then, the non-core values satisfy at least one of the following properties:

- If $v_{P_{\mathcal{A}}}$ is not derived from I, $v_{P_{\mathcal{A}}}$ is a non-core value of $\mathcal{A}$.
- If $v_{P_{\mathcal{A}}}$ is not in the set of runtime values of $Q_{\mathcal{A}}$ taking input I, $v_{P_{\mathcal{A}}}$ is a non-core value of $\mathcal{A}$.

The N-version approach leverages both above properties to eliminate non-core values. To leverage the first property, i.e., values are not derived from input, we apply dynamic taint analysis. With input as taint seed, all tainted values are derived from input, while others are not. To leverage the second property, we require multiple independent implementations of the plaintiff algorithm. After extracting runtime values from each implementation with the same test input, we filter out non-common runtime values. We also utilize the relations among common values, e.g., the sequences of values or the dependence among values, to characterize an algorithm. The final remaining values as well as their relations are considered as the signature of the algorithm.

The architectural view of the N-version approach is shown in Figure 4.1. Here, the plaintiff provides N ($N \geq 2$) implementations of the algorithm. These implementations can be source code or executables. For each implementation, the *value sequence extractor* extracts a refined value sequence, which only contains runtime values derived from the input. Then the *LCS (Longest Common Subsequence) extractor* generates a common value subsequence out of all these refined value sequences. This common value subsequence is considered as a signature of the plaintiff algorithm. For a suspicious program, we usually have only one executable. We also apply the *value sequence extractor* to extract its value sequence, with the same test input as that used in plaintiff implementations. Finally, the *similarity detector* compares the signature of the plaintiff algorithm with the value sequence of the suspicious program to calculate a similarity score. Next we explain the details of each component.

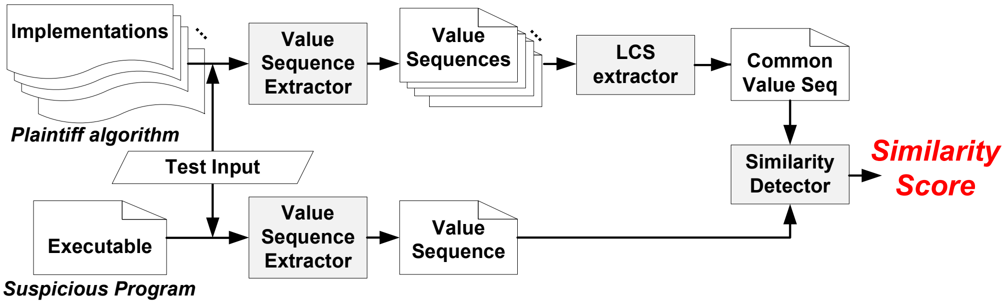


Figure 4.1. The design of N-version approach

Value sequence extractor extracts refined value sequence of programs. The same extractor is also used in our previous work [3]. This component first leverages the dynamic taint analysis technique [79] to only preserve the runtime values derived from input. We run a program in a virtual machine environment with the input as the taint seed. The dynamic taint analyzer monitors the taint propagation from the taint seed to registers and memory cells. Registers and memory cells are tainted if they appear in destination operands of any instructions that take values from tainted registers or tainted memory locations as input. The output values in these tainted destination operands are appended into a value sequence.

Besides dynamic taint analysis, we also employ several other schemes to further refine the value sequence:

- *Value-updating instructions only.* A value-updating instruction is a machine instruction that does not preserve input in its output. For example, `add` is a value-updating instruction, while `mov` is not. The value sequence should only contain the output of value-updating instructions.

- *Sequential reduction.* If the value of a register or memory cell is sequentially updated, the intermediate results, which are never read, will not be added into the value sequence.

- *Optimization-based refinement* is only applied on plaintiff programs. It is used to filter out the values that vary because of different compiler options. We use several different optimized executables of the same program to generate value sequences. (e.g. GCC have five optimization flags -O0, -O1, -O2, -O3, and -Os, which could create five different executables for the same pro-

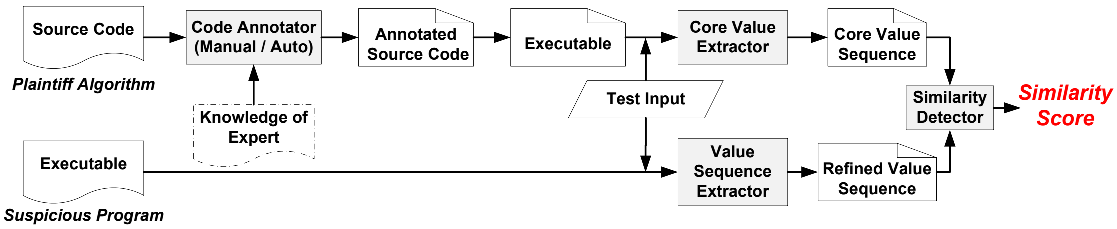


Figure 4.2. The design of annotation approach

gram). Then, we calculate the longest common subsequence of all these value sequences to retain only the common values in the resulting value sequence.

- *Address removal.* Memory addresses are not core values, because they may be changed by binary transformation techniques, such as word alignment and local variable reordering. Hence, we refine the value sequence by removing addresses. Array index is essentially the offset from the base address of the array. It may vary from implementations, so we also eliminate array indexes.

LCS extractor generates the LCS of all refined value sequences for N implementations. Note that subsequence is not necessary to be a consecutive part of the original value sequences. This common subsequence is considered as core values of the algorithm, since each value in the subsequence is derived from the input and is present in all plaintiff implementations.

Similarity detector compares the LCS of the N plaintiff implementations with the refined value sequence of the suspicious program. It measures their similarity and calculates a similarity score. The similarity metric is described in Section 4.3.3.

4.3.2 Annotation Approach

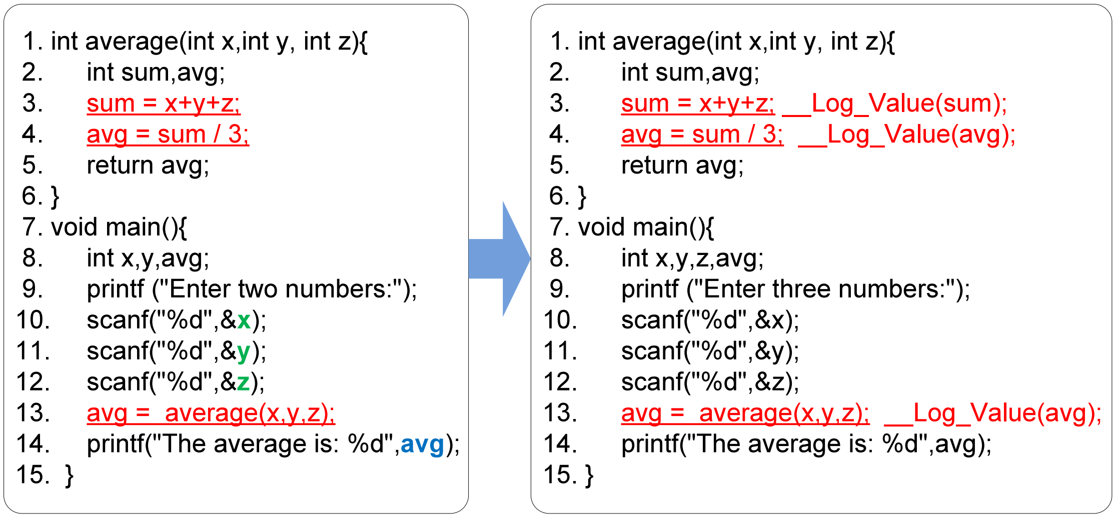
N-version approach requires multiple independent implementations of a plaintiff algorithm. Such requirement may limit its application in practice. Given this, we propose the second approach, the annotation approach, to extract core values. It only requires the source code of one implementation of the plaintiff algorithm. Instead of obtaining core values by filtering out “noise”, annotation approach resorts to auxiliary information to identify core values directly. The basic idea is to utilize the auxiliary information to identify critical statements that generate core values. We insert annotations at these statements and then compile and run the annotated code to extract the core values from the annotated variables.

The scheme is shown in Figure 4.2. The *code annotator* adds annotations to the source code either automatically or based on knowledge of domain experts. These annotations identify which variables in which statements will contain algorithm-level core values during execution. The *core value extractor* executes the annotated source code with a specific input and records all runtime values flagged by annotations. It also tracks the relations (e.g., the order of presence) among core values in runtime. These values are core values, the sequence of which is the signature of the plaintiff algorithm. Meanwhile, the *value sequence extractor* generates a refined value sequence during the execution of a suspicious program given the same input as the plaintiff algorithm. After execution, the core value sequence of a plaintiff algorithm and the refined value sequence of a suspicious program are compared by the *similarity detector* which provides a similarity score. Note that in annotation approach the value sequence extractor and similarity detector are the same as in the N-version approach.

We choose to annotate on source code instead of on binary code because of the following reasons. First, every core value is the runtime value of a variable in the source code. That is to say an adequate annotation at source code level is sufficient to extract all core values. Second, source code can liberate the scheme from the large amount of intermediate non-core values generated by machine operations. Third, source code is written by programmers based on algorithm descriptions, whereas binary code is generated from source code by compilers. Therefore, the abstraction gap between binary code and algorithm is larger than that between source code and algorithm.

In the rest of this section, we will explain the design of key components.

Code annotator and annotation methods. *Code annotator* adds annotations to source code. These annotations can be generated by different methods. One method is manual annotation based on the knowledge of domain experts (e.g., author of the algorithm). We can leverage experts' full understanding of the algorithm and its implementation to point out which variables reflect the critical logic of the algorithm. This knowledge based annotation could be very accurate for simple algorithms. However, as a manual process, it becomes extremely time-



```
1. int average(int x,int y, int z){
2.      int sum,avg;
3.      sum = x+y+z;
4.      avg = sum / 3;
5.      return avg;
6. }
7. void main(){
8.      int x,y,avg;
9.      printf ("Enter two numbers:");
10.     scanf("%d",&x);
11.     scanf("%d",&y);
12.     scanf("%d",&z);
13.     avg =  average(x,y,z);
14.     printf("The average is: %d",avg);
15.  }
```

```
1. int average(int x,int y, int z){
2.      int sum,avg;
3.      sum = x+y+z; __Log_Value(sum);
4.      avg = sum / 3;  __Log_Value(avg);
5.      return avg;
6. }
7. void main(){
8.      int x,y,z,avg;
9.      printf ("Enter three numbers:");
10.     scanf("%d",&x);
11.     scanf("%d",&y);
12.     scanf("%d",&z);
13.     avg =  average(x,y,z);  __Log_Value(avg);
14.     printf("The average is: %d",avg);
15. }
```

Figure 4.3. Forward slicing and backward slicing annotation example

consuming when the plaintiff algorithms become complex.

Given the drawback of manual annotation, we propose an automatic annotation method. It combines the techniques of static backward slicing and static forward slicing [80, 81]. Backward slicing starts from the output variables and ends at the beginning of the implementation. The result of backward slicing is the set of statements which affect the output. In the opposite, forward slicing initiates from the input variables and terminates at the end of the implementation. The result of forward slicing is the set of statements affected by the input. The intersection of these two sets is the statements that derive the output from the input. After finding these important statements, we add an annotation to the result variable in each statement.

Figure 4.3 shows an example. The left part is the original source code. First, we specify input variables and output variables by preknowledge. x, y, z are the input variables and avg is the output variable. The second step is to apply static forward slicing and backward slicing based on the input and output variables. Line 3, 4 and 13 are statements in the intersection of resulting sets from forward slicing and backward slicing. Based on the slicing result, we add annotations to these statements. The right part is the annotated code.

Although automatic annotation method is not as accurate as the manual one in identifying core values, it is often effective enough to detect algorithm plagiarisms while much more efficient and scalable. We can also employ a hybrid annotation scheme by combining both methods. That is, we can apply backward slicing and forward slicing to get a candidate set of annotations and then manually remove annotations on insignificant variables or add annotations on other crucial variables.

Core value extractor is used to extract core values from an implementation of the plaintiff algorithm. The extractor runs in a virtual machine environment, where a special system call is inserted to handle the annotation. The parameter of this system call is the annotated variable. When an annotation is encountered in execution, this system call will be invoked to record the runtime value of the variable.

4.3.3 Similarity Metric

After extracting the signature (i.e., core value sequence) of a plaintiff algorithm and the value sequence of a suspicious program, the similarity detector measures their similarity in terms of the proportion of values common to both sequences. We apply the longest common subsequence (LCS) algorithm to obtain the common value sequence of two programs. Let $|s|$ be the length of a sequence s, then the similarity score is calculated by the following formula. Since our approach is input sensitive, we randomly choose multiple inputs and the final similarity score is the average of all similarity scores.

$$\text{Similarity score} = \frac{|\text{common value seq}|}{|\text{signature seq}|}$$

4.4 Address Reordering Problems

Although the LCS metric is efficient, it is sensitive to value reordering. For example, an adversary can reduce the length of the LCS by exchanging the order of independent instructions or independent basic blocks. As shown in Figure 4.4, two code segments are semantically equivalent, but the length of their LCS is only 2. To defend this attack, we propose a technique to organize a value sequence into subsequences showing unchangeable partial ordering of the values. To get such reordering-intolerant subsequences, we build dynamic value dependence graphs

```
n1 = ( (uint32) b[i]    << 24 );
n2 = ( (uint32) b[i+1] << 16 );
n3 = ( (uint32) b[i+2] <<  8  );
n4 = ( (uint32) b[i+3]          );
n   = n1 | n2 | n3 | n4;
```

```
n1 = ( (uint32) b[i+3]          );
n2 = ( (uint32) b[i+2] <<  8  );
n3 = ( (uint32) b[i+1] << 16 );
n4 = ( (uint32) b[i]    << 24 );
n   = n1 | n2 | n3 | n4;
```

Figure 4.4. Reordering problem example

(VDGs) of the core values. Then we use a novel path comparison technique to check whether the reordering-intolerant subsequences of the plaintiff program are similar to any paths in the VDG of the suspect program.

Definition 1. (Value Dependence Graph). *Given a program P, its value dependence graph* VDG(P) *is a directed acyclic graph $G(V_P, E_P)$, where V_P is a set of vertices each of which represents a runtime value that is the output of some instruction of P, E_P is a set of edges (a, b) such that $a \in V_P$, $b \in V_P$, $a \neq b$, and the runtime value represented by b is derived from the runtime value represented by a.*

Since we have implementations of plaintiff algorithm $P_{\mathcal{A}}$ and the suspicious program S, we can construct VDGs from their runtime values (refined to expose the core values). Both VDG(P) and VDG(S) are acyclic graphs. Then, if there is no path between two nodes in the VDG, they are independent. In other words, all values on a path from the root node to a leaf node have ordering dependence, so reordering techniques cannot change their orders.

4.4.1 VDG Comparison

Both VDG(P) and VDG(S) are constructed during runtime given the same test input. The runtime values along with their dependence are recorded. Each value is represented by a node and dependence among values is represented by edges. Once VDG(P) and VDG(S) are extracted, we propose a dynamic programming algorithm to check whether dynamic data dependence paths in the VDG(P) are similar to any path in the VDG(S). These paths transform an initial input to a final output. For each such path p in VDG(P), we find a path in VDG(S) which has the largest LCS with p. Since all the values contained in p have partial ordering dependence, they cannot be reordered.

We calculate the longest matched path in VDG(S) of p following Formula (4.1) and (4.2), where p_i is the ith node in p, n_j is a node in VDG(S), v_{p_i} and v_{n_j} represent the values of p_i and n_j, respectively. The computational complexity is $O(|p||V_S|^2)$, where $|p|$ is the length of path p and $|V_S|$ is the number of nodes in VDG(S).

$$\text{LCS}(p_i, n_j) = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = \text{ROOT} \\ \max_t(\text{LCS}(p_{i-1}, n_t)) + 1, & \\ & \text{if } v_{p_i} = v_{n_j}, t \in \{\text{parents of } j\} \\ \max\{\text{LCS}(p_{i-1}, n_j), \max_t(\text{LCS}(p_i, n_t))\}, & \\ & \text{if } v_{p_i} \neq v_{n_j}, t \in \{\text{parents of } j\} \end{cases} \tag{4.1}$$

$$\begin{aligned} \text{LCS}(p, \text{VDG}(S)) = & \max_{n_l}(\text{LCS}(p_{|p|}, n_l)), \\ & n_l \in \{\text{leaf node of VDG}(S)\} \end{aligned} \tag{4.2}$$

4.4.2 VDG Reduction

We further improve the performance of VDG comparison by removing useless nodes and edges from VDG(S). We remove the nodes whose values do not appear in VDG(P) by merging the nodes to the nearest predecessors or successors if possible. When such node has only one predecessor/successor, we merge it to its predecessor/successor. Both the construction of VDG(S) and the reduction can be done in $O(|E_S| + |V_S|)$ time, where $|E_S|$ is the number of edges in VDG(S) and $|V_S|$ is the number of nodes in VDG(S). Since the computational complexity of path comparison is $O(|p||V_S|^2)$, reducing node size will significantly improve its performance.

4.4.3 VDG Similarity Metric

When p, a path of VDG(P) is compared to VDG(S), the *per-path* similarity score is computed as follows:

$$\text{PSS}_{path}(p, \text{VDG}(S)) = \frac{\text{LCS}(p, \text{VDG}(S))}{|p|}$$

Given a set of paths extracted from VDG(P), we use the weighted average of per-path similarity scores as the path comparison score of two graphs, because long paths are more likely to serve the main purpose of P and to reduce the chance of false positives. Since P is provided by the plaintiff, we have control over the source code and the compilation process to make sure that P would not contain a large number of dummy instructions. The path comparison score of VDG(P) and VDG(S) is calculated as follows:

$$\text{PCS}(\text{VDG}(P), \text{VDG}(S)) = \sum_{i=1}^{|\rho|} \omega_i \text{PSS}_{path}(p_i, \text{VDG}(S))$$

where ρ is the set of paths selected from VDG(P), $|\rho|$ is the number of paths in ρ, $p_i \in \rho$ and $|p_i|$ is the length of path p_i. ω_i, the weight of ith path is defined as

$$\omega_i = \frac{|p_i|}{\sum_{k=1}^{|\rho|} |p_k|}$$

4.5 Implementation and Experiment

We implement the value sequence extractor inside QEMU 0.9.1 [82, 83] by adding dynamic taint analyzer. The static backward slicing and forward slicing utilize the CodeSurfer 2.1 API [84]. Core value extractor is implemented by adding a new system call which is dedicated to flag core values.

We evaluate the effectiveness of our approaches by conducting proof-of-concept experiments. For each approach, we measure the similarities in the following three cases: (1) implementations of the same algorithm (2) implementations of different algorithms with the same purpose, and (3) implementation of different algorithms with different purposes. More experiments are conduct for the automatic annotation approach, since it is more practical. All tested algorithms are representative and well-known. The evaluation is performed on a Linux machine with Intel Pentium 4 2.80 GHz CPU and 1 GB RAM.

4.5.1 Effectiveness of the N-version Approach

In this part of the evaluation, we use three algorithms: MD5, AES and network flow. We obtain multiple implementations of each plaintiff algorithm. All implementations are from open source libraries. To assure that these implementations

Table 4.1. The similarity scores in MD5 experiment with various inputs

	# of plaintiff implementations				
	1	2	3	4	5
Min	0.609	0.629	0.720	0.731	0.814
Max	0.832	0.997	1.000	1.000	1.000
Avg	0.729	0.891	0.934	0.962	0.980

Table 4.2. The similarity scores in AES experiment with various inputs

	# of plaintiff implementations		
	1	2	3
Min	0.206	0.337	0.480
Max	0.536	0.963	1.000
Avg	0.413	0.623	0.826

Table 4.3. The similarity scores in max flow experiment with various inputs

Similarity of same algorithm			
	# of plaintiff implementations		
	1	2	3
Min	0.452	0.521	0.787
Max	0.992	1.000	1.000
Avg	0.676	0.787	0.886
Similarity of different algorithms			
	1	2	3
Min	0.011	0.077	0.113
Max	0.148	0.164	0.164
Avg	0.102	0.133	0.128

are independent, we use MOSS, an online software plagiarism detection service [85], and VaPD, a dynamic value-based software plagiarism detection system [3], to measure their pair-wise similarities. A low pair-wise similarity would suggest independent. Therefore, we filter out the implementations with high similarity scores.

MD5. We have 6 independent implementations of MD5. First, to verify the existence of algorithm level core values that present in all implementations, we obtain the common value sequence of the first n ($1 \leq n \leq 6$) implementations, respectively (when $n = 1$, the common value sequence is itself). As shown in Figure 4.5, the length of common value sequence converges quickly as N increases and eventually becomes stable. Similar results are observed for the other two algorithms. This indicates that the irreplaceable and uneliminable core values of an algorithm do exist.

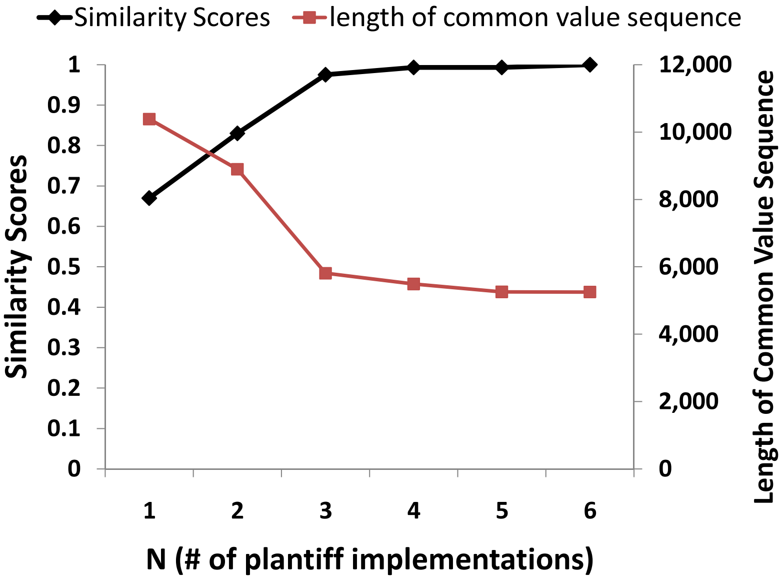


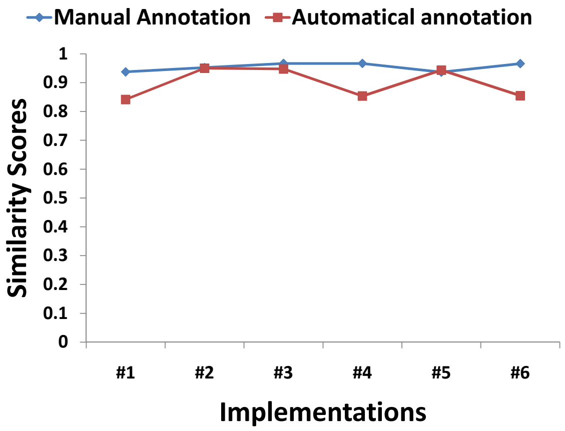
Figure 4.5. The similarity scores and lengthes of common value sequences for MD5

To test the effectiveness of N-version approach, we randomly choose N ($2 \leq N \leq 5$) implementations as plaintiff programs, while the rest are suspicious. Some statistics of similarity scores are shown in Table 4.1. These results demonstrate that as N increases, the similarity scores between plaintiff algorithm and plagiarized program increase as well. In other words, the ability to detect algorithm plagiarism is improved. When $N = 3$, the minimum similarity is 0.720, which is enough to identify algorithm plagiarism. Figure 4.5 also indicates that as N increases, similarity scores increase and converge to be stable.

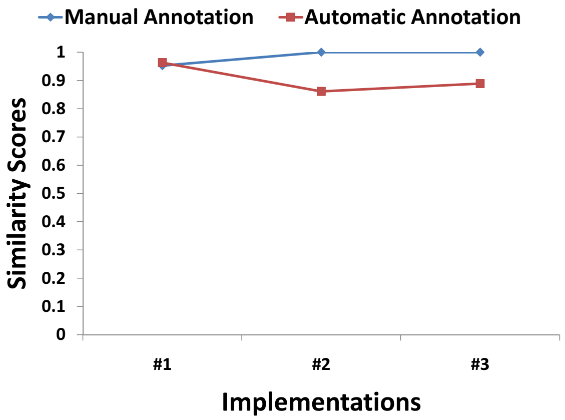
AES. We use 3 implementations as plaintiff programs and the other one as the suspicious program. The statistics of similarity scores are shown in Table 4.2. Some similarity scores are not high enough to distinguish the same algorithm from different algorithms. The reason is that in AES, a lot of intermediate values are independent, so they could be in any order. Figure 4.4 shows an example. It may result a false negative. In Section 4.5.3, we will show that by using a VDG-based metric we are able to eliminate this false negative.

Both MD5 and AES have only one algorithm each, so we cannot test the false positive.

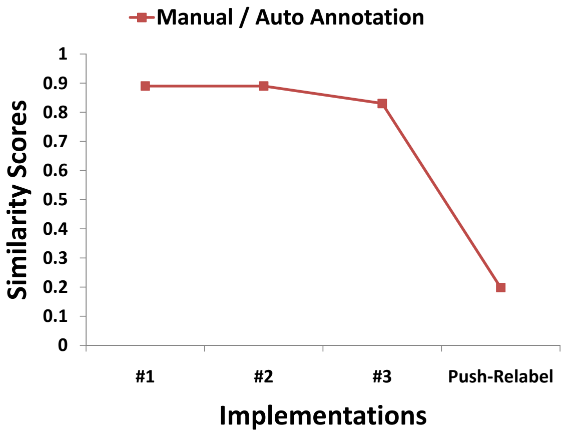
Maximum flow algorithms. We have four implementations of the Ford-Fulkerson algorithm and another implementation of the push-relabel algorithm.



(a) MD5



(b) AES



(c) Max Flow

Figure 4.6. The similarity scores of the annotation approach

The result is shown in Table 4.3. We are able to distinguish the case of the same algorithm from that of different algorithms with the same purpose. For different algorithms, the similarity scores are all very low, irrespective of N.

Conclusion. Results in Table 4.1, 4.2 and 4.3 demonstrate that as N increases, similarity scores of implementations of the same algorithm increase, while similarity scores of implementations of different algorithms are not affected. This indicates that applying multiple implementations can significantly reduce noises in core value extraction. The results also show that based on the LCS metric, false negative exists due to the value reordering problem.

4.5.2 Effectiveness of the Annotation Approach

4.5.2.1 Manual Annotation Approach

We perform proof-of-concept experiments on algorithms of MD5, AES and network maximum flow. For each algorithm, we randomly choose one implementation as plaintiff and manually annotate its source code. The rest of implementations are treated as suspicious programs. The results are shown in Figure 4.6. All similarity scores are higher than 0.85 when the plaintiff program and suspicious program implement the same algorithm. The similarity scores between implementations of different maximum flow algorithms are all around 0.25. The results indicate that the manual annotation approach can distinguish the case of the same algorithm from the case of different algorithms.

4.5.2.2 Automatic Annotation Approach

First, we conduct the same experiments as for the manual annotation approach, except that the plaintiff programs are automatically annotated through static forward slicing and backward slicing. The results are also shown in Figure 4.6. For both MD5 and AES algorithms, the similarity scores between implementations of the same algorithm are slightly lower than those measured by the manual annotation approach, but are still higher than 0.80. For the max flow algorithms, automatic annotation annotates the same variables in the plaintiff program as the manual annotation does, since max flow algorithms only contain calculation operations in very few statements. The automatic annotation approach is effective for

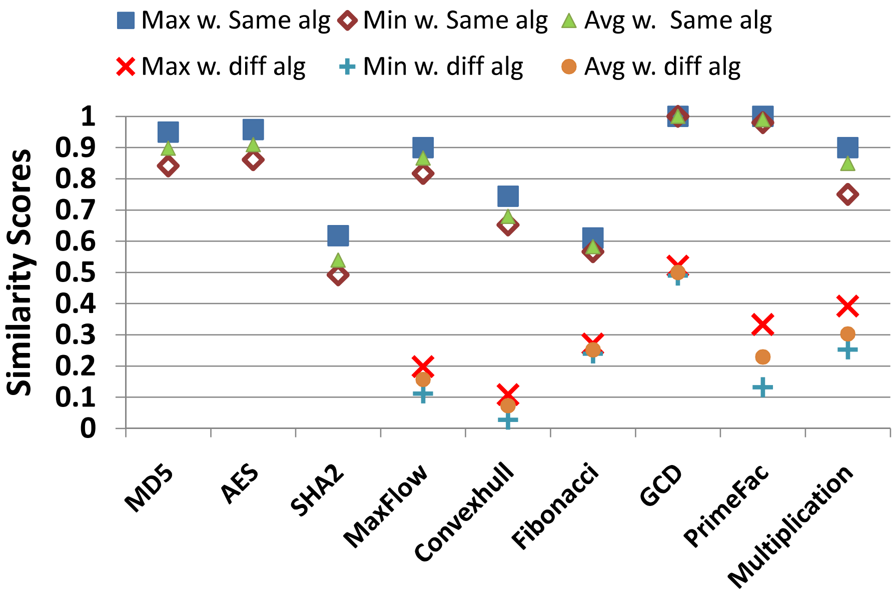


Figure 4.7. The similarity scores of automatic annotation with the LCS Metric

all above three applications.

Besides the above three applications, we also conduct experiments on 6 other applications. The list of all applications and their algorithms is shown in Table 4.4.

Algorithms with the same purpose. We first compare the similarities in two cases: (1) the same algorithm and (2) different algorithms with the same purpose. For each application, we choose one plaintiff implementation and two suspicious implementations, one of which implements the same algorithm with the plaintiff implementation and the other does not. We measure their similarity

Table 4.4. The List of Applications and their algorithms

Applications	Plaintiff Algorithm	The Different Algorithm
MD5	MD5	-
AES	AES	-
SHA2	SHA2	-
MaxFlow	Ford-Fulkerson	Push-relabel
Convex hull	Monotone chain	Graham scan
Fibonacci	Exponentiation by squaring	Iterative
Greatest common divisor	Extended Euclidean	Brute force
Prime factorization	Wheel	Fermat
Multiplication	Karatsuba	Long

scores by giving 10 randomly generated inputs to each of them. Figure 4.7 shows the experiment results. It indicates that for each application, there is a significant gap between similarity scores of the same algorithm and those of different algorithms, although no universal threshold can be applied for all applications. The low similarity scores (< 0.7) for the same algorithm in the SHA2 and Fibonacci applications are caused by the value reordering problem, which will be solved in Section 4.5.3. The high similarity score of different algorithms for greatest common divisor (GCD) application is caused by the reason that the brute force algorithm goes through every integer until the GCD is found. As a result, all integers between the GCD and the smaller integer are in its value sequence, therefore there are false matches, which will be eliminated by the VDG metric. The same algorithm for the convex hull application does not achieve high similarity scores (around 0.65) because the suspicious program optimizes the algorithm and does less calculations. Even though, the differences between similarity scores of the same algorithm and those of different algorithms are still large enough to distinguish them from each other. As a result, the automatic annotation approach is effective to distinguish the same algorithm from different algorithms with the same application.

Algorithms with different purposes. We evaluate the similarities between algorithms of different applications in Table 4.4. Since our approach is input sensitive, we choose 28 pairs of different algorithms, each pair of which can accept the same input. The results are quite positive: 20 pairs have the similarity scores lower than 1%. The other 8 pairs have the similarity scores between 1% and 30%. These higher similarity scores are all caused by the reason that the plaintiff algorithm is simple and has short core value sequence, while the suspicious one is complicated with much longer core value sequence, which increases the chance of false matches. As a result, for simple plaintiff algorithms, the plaintiff can use manual annotation to assure the accuracy of core value annotation and to reduce false matches. Even applying automatic annotation, all the similarity scores between programs implementing algorithms with different purposes are low enough to be distinguished from those between programs implementing the same algorithm.

Conclusion. Manual annotation is more effective but needs domain experts to annotate the source code manually. Although automatic annotation is not as accurate as manual annotation, the detection accuracy is good enough to tell the

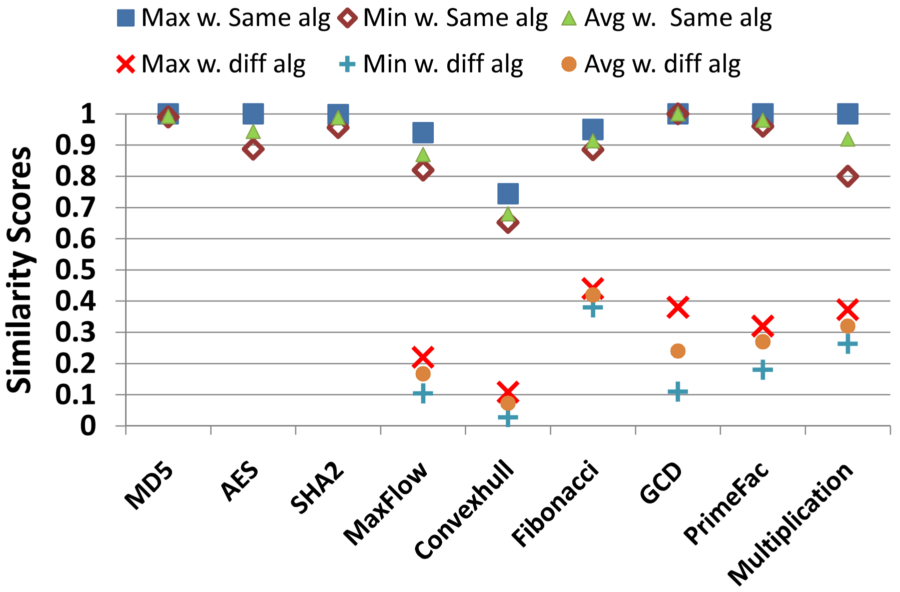


Figure 4.8. The similarity scores of automatic annotation with the VDG metric

same algorithm from different algorithms.

4.5.3 VDG-Based Metric

Both the N-version and the annotation approach can adopt VDG as a metric to measure algorithm-level similarities. We perform the same experiments on the VDG metric as in Section 4.5.1 and 4.5.2. We show the results of automatic annotation using the VDG metric in Figure 4.8. Since both SHA2 and Fibobacci implementations suffer from the value reordering problem, their similarity scores are significantly increased. For the brute force algorithm of the GCD application, false matches are eliminated because its VDG is wide with all paths shorter than 3, which are not matched to the paths in the plaintiff VDG. The other results are similar to the results in Figure 4.7. Thus automatic annotation approach using VDG as the metric is effective in detecting algorithm similarity.

In addition, previously we have one false negative in Section 4.5.1, when the N-version approach is applied to detect similarity of the AES implementations with the LCS metric. The false negative is eliminated when we adopt the VDG metric. Its similarity scores are significantly increased (minimum score $= 0.757$, when $N = 3$).

We also evaluate the scalability of VDG-based comparison with large graphs. We use a large file as input to MD5 implementations. The VDG(P) has 75k nodes, with the maximum path length of 21k. The VDG(S) has 448k nodes originally and 75k nodes after reduction. The running time of the path comparison process is less than 4 hours. This result indicates the capability of our approach in handling large graphs.

4.5.4 Resiliency to Automatic Obfuscation

Plagiarists can exploit automatic obfuscation tools to obfuscate their implementation of the plagiarized algorithms to further evade detection. In this section, we evaluate the resiliency of our approaches to such cases.

We apply 3 different automatic obfuscation tools: Semantic Designs Inc's C obfuscator [86], Diablo link-time optimizer-based obfuscator (Loco) [70] and binobf [87]. The first one is source code-based while the latter two are binary-based. The features of Semantic Designs Inc's C obfuscator include, but are not limited to, identifier scrambling, format scrambling, loop rewriting, and if-then-else rewriting. Loco can obfuscate binaries by control flow flattening and opaque predicate. Binobf performs junk insertion, opaque predicate, jump table spoofing, etc.

We obfuscate the suspicious program of each application by these three tools and repeat the experiments in Section 4.5.2. The core value sequences of obfuscated suspicious programs are almost the same as those of original programs. The only differences are caused by several value reordering cases. As a result, the similarity scores are nearly the same as in Section 4.5.2. After we applied VDG as similarity metric to eliminate value reordering problem, the similarity scores become the same as in Section 4.5.3. The results show our approach is resilient to automatic obfuscation techniques.

4.6 Discussion

4.6.1 Counterattacks

Software obfuscation techniques. An attacker may apply obfuscation techniques to evade algorithm plagiarism detection. Since our approaches apply core values as signature, we mainly focus on obfuscation methods that manipulate runtime values. These methods include noise injection, irrelevant instruction reordering and data transformation [88].

Our approach is resilient to noise injection. Injecting noise to suspicious program may cause false matches and will raise the chance of accusation, so if a plagiarist knows the mechanism of our approach, he will never try to evade detection by injecting random noise. However, if a lot of noise is injected, the size of value sequence or VDG could dramatically increase. This will slow down the similarity score computation. Our solution filters out values that are not present in the value sequence or VDG of the plaintiff program before performing the similarity computation.

Our VDG metric is resilient to irrelevant instruction reordering as discussed in Section 4.4.

Data transformation is another possible counterattack to evade plagiarism detection. Splitting or merging variables can change the runtime values. For example, a single byte value b can be split into 8 bytes, each of which represents one bit of b. Another example is that an array of four bytes can be merged into one integer. However, whenever this value is used, the original value has to be assembled back, unless the adversary adopts complicated methods to convert all operations on the original variable type to operations on the new one, which is usually not practical. As long as these original values are restored, our approach can detect the plagiarism. Therefore, our approach is resilient to most variable splitting and merging attacks and raises the bar for plagiarism and increase its cost—simple data transformation attacks will be caught and sophisticated transformation has a high overhead for the plagiarists.

Optimization. Attackers can utilize different compilers or compiler optimization options to change the executables of their plagiarized programs. However, based on the definition of core values, runtime values that vary from different compilers and optimization options are not core values. Therefore, compiler optimization does not eliminate core values. Hence, our approaches are not affected by these optimization techniques.

Another way of changing runtime values is to optimize the algorithm for implementation. If a plagiarist optimizes a plaintiff algorithm and then implements it, the similarity score will decrease. This can be solved by applying manual annotation, because experts master complete knowledge about core values of an algorithm. For a complex algorithm, in order to reduce similarity score, a significant amount of optimization is required. The resulting algorithm after such optimization may no longer be considered as the same as the original algorithm.

4.6.2 Partial Plagiarism

Less self-disciplined developers may steal an algorithm by implementing and embedding it in a large program. Since only a small part of the whole program is plagiarized, it is difficult to detect. To detect partial plagiarism, we need to make sure the inputs to the plaintiff algorithm and the suspicious module are the same. Then we can search in value sequence of the suspicious program to find a subsequence that matches the sequence of plaintiff algorithm. To this end, a feasible solution on partial plagiarism detection must be able to identify suspicious modules in a suspicious program. One possible solution is to leverage some characteristics of a specified algorithm to provide a hint about the location of suspicious modules, such as invoking special system calls or APIs.

4.6.3 Limitations

First, our detection results rely on the selection of a similarity score threshold to decide whether or not an algorithm is plagiarized. However, there is no such universal threshold for all algorithms, because the threshold may vary for each algorithm depending on how complex it is, how specific it is described, etc. To this end, instead of giving a yes/no answer, our approach provides users with similarity scores between programs as initial evidences. Based on these evidences, users can take further investigations, which often involve non-technical actions.

Second, our value-based methods are input sensitive. This means it is possible that different algorithms handle certain input in the same way. This may cause false positives. Nevertheless, since we choose multiple inputs randomly, this risk would be effectively mitigated in practice.

Third, our approaches leverage dynamic taint analysis to extract values derived from input. It suffers from the common limitations of dynamic taint analysis techniques [89]. A plagiarist could use anti-taint-analysis techniques to hide core values. Solutions to this issue still remain an open question.

Finally, our value-based approaches are not applicable to all algorithms, since they rely on extracting runtime values from tainted value-updating instructions. Some other algorithms, where the output is a permutation or a selection of the input variables, e.g., sorting algorithms and finding minimum/maximum value in an array, contain very few of such instructions. As a result, ValPD can only detect computational algorithms.

4.7 Summary

In this chapter, we propose two dynamic value-based approaches, i.e., N-version and annotation, to detect algorithm plagiarism. To the best of our knowledge, our work is the first one focusing on algorithm plagiarism detection. We evaluate the proposed approaches on different algorithms. The evaluation results indicate that our approaches can detect algorithm plagiarism effectively. We believe our work has laid a foundation as a first step towards a practical solution to algorithm plagiarism detection for intellectual property protection.

Chapter 5

ViewDroid: User Interface-based Android Application Repackaging Detection

In this chapter, we propose a user interface-based Android application repackaging detection method, called as ViewDroid. ViewDroid extract the navigation relations among views as the signature of an app. It is a higher level representation of an app's behavior than the traditional code level birthmarks (e.g., opcode sequence, program dependence graph). In other words, ViewDroid does not need instruction-level details. Hence, it is resilient to code obfuscations such as noise instruction/data injection, instruction reordering, instruction splitting and aggregation and data dependence obfuscation, etc. In addition, the generation of view graph relies on statically analyzing Android specific APIs (e.g. `startActivity`, `startActivityForResult`). These APIs are provided by the Android system and are hard to be replaced or modified. Therefore, view graph, as the birthmark, is more robust to obfuscation techniques such as, API splitting, API renaming, API re-implementation.

5.1 Android Application Background

Android is a Linux-based platform for mobile devices. Users can download and install Android apps from various app markets. Android apps are published to the market in a compressed file format (i.e., .apk file). It contains a manifest file (i.e., AndroidManifest.xml), resource files (i.e., files in res directory), and compiled Dalvik Executable (i.e., classes.dex). The manifest file lists the package name, version number, critical components of the app, and the associate permissions to each component. The resource folder includes all the raw resource files, such as images and audio files, and the XML files which describe the layouts of user interfaces. The Dalvik executable contains all the classes that implement the functionality of all the primary components of an app. Some apps contain parts that are implemented by native languages. Since relatively few Android apps contain such components developed in the native languages C/C++ and they mostly serve as background services, our current ViewDroid design only takes into consideration the Dalvik executable, the relevant Android manifest file, and the layout files in the resource folder .

Components serve as the building blocks for Android apps. There are four types of components, namely, *Activity*, *Service*, *Broadcast Receivers*, and *Content Provider*. An *Activity* provides a screen for the user to interact with. An app requires one main activity to start but can have a number of other activities (roughly one per screen view). A stack is designed to organize activities. When a new activity starts, it goes to the top of the stack. A *Service* is a component that runs in the background, usually engaged in the performance of long-running tasks. In general, a service is used to perform any task that is asynchronous with respect to the main user interface. A *Broadcast Receiver* listens to special messages broadcasted by the system or individual apps and relays work to other services or activities. Finally, a *Content Provider* manages shared data and optionally exposes query and update capabilities for other components to invoke. A message-like *intent* is used to help the communications among components.

The execution sequence of an Android app usually starts from the main activity, specified in the manifest file. When the launching icon of an app is pressed by user, the main activity will be launched. It serves as the main entry point to the user interface. The app switches between activities by invoking platform APIs, `Context.startActivity()` or `Activity.startActivityForResult()` with `Intent` objects as parameter. An `Intent` object contains the information of the target activity. A user interface is loaded when an activity is initialized by the `onCreate()` method, which creates a new user view through APIs like `setContentView()`. The view is then put on the top of the view stack and becomes the running activity. Therefore, by analyzing the Android specific APIs within each `Activity` class, the user interface navigation relation information can be constructed to build our view graph. Note that, in our work, we only consider apps that have interaction with users (e.g., by key pressing, button clicking). Some other apps, which only have background services and do not interact with users, are out of our consideration.

5.2 Problem Statement

The most fundamental challenge of app repackaging detection is to find unique *birthmarks* to characterize an app. The proposed birthmark should be *accurate* and *unique* enough to identify an Android app. Moreover, as reported by Zhou et al. [2], the plagiarists and malware writers tend to use obfuscation on the repackaged apps to evade detection. Hence, to significantly raise the bar for stealthy repackaging, the designed detection scheme must be resilient against most code obfuscation techniques. Finally, since the Android app repackage problem is prevalent among most Android markets, it is very important to build a detection tool that can perform detection in large-scale scenarios.

Note that we only focus on non-trivial Android apps that interact with users through user interface and are implemented as Dalvik executables. Apps that contain components implemented by native-code languages are out of the scope of our research. Those only providing background services without user interactions are not under our consideration either.

5.2.1 Attack model

The general attack model in the Android app repackaging problem is: an attacker has access to the plaintiff Android app package (.apk file); he repackages the app by copying the code, making a few modifications (e.g., replacing the advertisement, attaching malicious payloads), and applying automatic code obfuscation techniques in order to evade detection; the repackaged app is then signed with a private key and republished to the app markets.

Based on the level of modification on the original APKs and the effort an attacker is willing to pay on the repackaging process, we further classify the repackaging attacks into the following three categories:

Lazy attack: A lazy attacker can make some simple changes over an app without changing its code. For instance, repackaging an app with a different author name or with different advertisements is such rudimentary lazy attack. Non-developers can be easily trained to perform such tasks manually. More knowledgeable lazy attackers may apply current automatic code obfuscation tools to repackage an app without changing its functionality, following the procedure similar to that shown in our evaluation section.

Amateur attack: An amateur attacker not only applies automatic code obfuscation but also changes/adds/deletes a small part of the functionalities. For example, an attacker can add some online social functionalities along with the online chat view to the original app. Attackers must pay more effort to understand and thus modify the code. For example, they have to read the Android manifest file to delete or append the components that they want to register for the app and to insert some interaction code into the original component to glue the newly added components.

Malware: A malware writer creates a malicious app that mimics a popular app by inserting some malicious payload into the original program. In this way, the malicious app can leverage the popularity of the original program to increase its propagation speed. With this purpose, the attacker tries to make the functionality and user interface of the repackaged apps similar to the original one. Under this circumstance, an attacker actually has to perform most of tasks that an Amateur attacker has to do. In addition, the attacker needs to write the malicious payload either in Java or C/C++ and stealthily insert the payload into the app.

We further analyze how well ViewDroid can detect these attacks and other potential advanced attacks in Section 5.5.

5.2.2 Design Goals

Accurate Birthmark: In order to measure the similarity between two Android apps, ViewDroid must select an accurate birthmark to characterize apps. This accurate birthmark should be able to reflect the primary semantics of Android apps and tell independently-developed apps apart. In other words, the designed birthmark for ViewDroid should cause very few false positives.

Obfuscation Resilience: Code obfuscation is a technique to transform a sequence of code into a different sequence that preserves the semantics but is much more difficult to understand or analyze. Obfuscation techniques can also be applied by attackers to evade repackaging detection. Hence, ViewDroid must be able to detect repackaging with the presence of various automatic code obfuscation techniques. In other words, the designed birthmark should be robust against various obfuscators. Obfuscation resilient birthmarks will ensure low false negatives.

Scalable Detection: Because there are a huge number of apps on different Android app markets, ViewDroid must be efficient and scalable enough to detect repackaging in such a large-scale scenario.

5.3 Design

5.3.1 Overview

It is critical while very challenging to identify an accurate and obfuscation resilient birthmark in the design of a repackaging detection tool. In the past, a good variety of birthmarks have been proposed and evaluated for different types of program languages (C, Java) and platforms (Linux, Windows). Some of these traditional birthmarks have been proposed to detect Android app repackaging [24, 25, 27, 2]. They are all code-level birthmarks. Instead of applying traditional software birthmarks, we propose a novel user interface-based one, namely *feature view graph*. It fully leverages a unique characteristic of smartphone apps – they are mostly UI intensive and event dominated [90]. Feature view graph represents a higher level abstraction of an Android app's semantics. Therefore, it has the potential to be

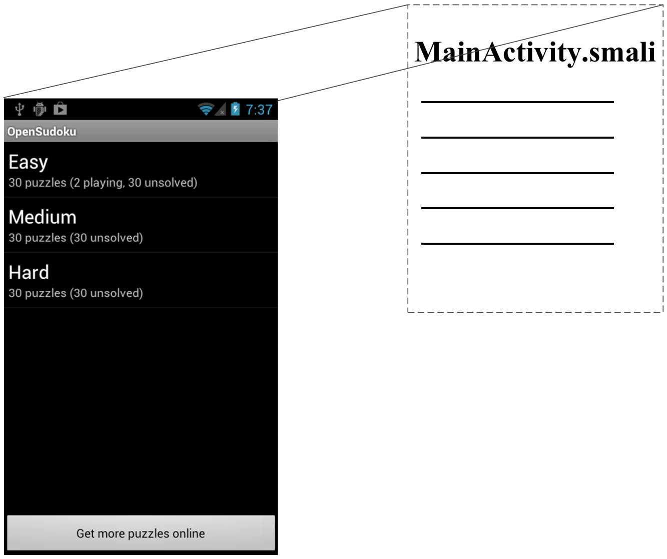


Figure 5.1. An example of a Sudoku game view

more robust to code obfuscation. In order to meet the scalability requirement of Android app repackaging detection, feature view graph is generated by static analysis of the dissembled installation file of an Android app (i.e., the apk file).

Definition 3 gives the definition of *view*. Figure 5.1 is an example of a view, which has a corresponding main activity. The main activity is launched after users pressing the app launch icons on the phone. Four physical components are included in this view. A user can choose from one of the three levels to play with the game by pressing the corresponding buttons. Each button will bring the user to another view. Also, a user can press the white button at the bottom to get more puzzles online. This button will bring up the view of a browser. Figure 5.2 is an example of view navigation, where the app changes views based on users' interactive behavior. All the user interaction semantics for a view can be constructed by analyzing the corresponding Activity class that implements the view.

Definition 3. ***(View)*** *A view is a user interface that is displayed to users for interaction with the mobile app. Each view has a corresponding activity class that defines the view's functionality. A view contains one or more visible components (e.g. buttons, trackball) on the screen. When touched, the components might trigger other activities or services.*

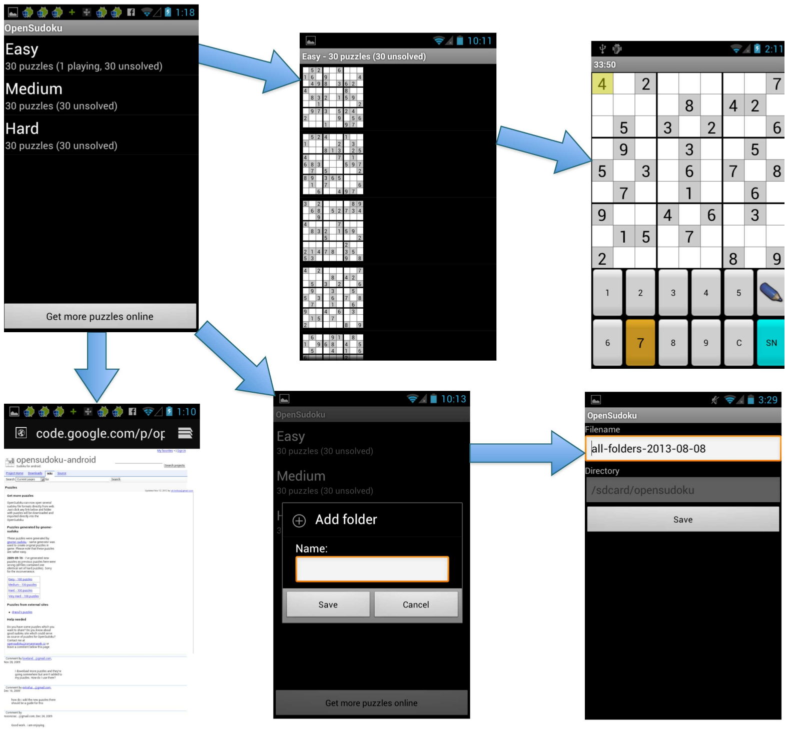


Figure 5.2. An example of view navigation

In order to describe the user interaction semantics for an Android app, we define *view graph* in Definition 4.

Definition 4. ***(View Graph)*** *A view graph of a mobile app is a directed graph* $G(V, E)$*, where* V *is a set of nodes, each of which represents a user interface view.* E *is a set of edges* $< a, b >$ *such that* $a \in V$*,* $b \in V$ *and the smartphone display can switch from view* a *to view* b *by user interaction or other triggers.*

By adding features to each view and each edge, a view graph can represent an app more accurately and also improve the efficiency of app similarity measurement at the later stage. The features of a view could be the number, types or layout of the visible components (e.g., buttons, menus), or a set of Android platform specific APIs invoked in this view's activity. However, the former one (e.g., layout) is much easier for an attacker to manipulate because it does not represent the fundamental semantics of an app. The latter one (i.e., Android specific APIs) is much more stable and can reflect app's semantics; as a result, we only consider that as view features.

In a feature view graph, the feature of an edge is the event listener function (e.g., `onClick()`, `onLongClick()`, `onTouch()`, etc.) that is directly triggered by user generated events. Generally, there are two types of events in Android platform, user-generated events and system-generated events. We only focus on user-generated events in our birthmark creation. This is because these events are highly associated with the functionality of an individual app and the corresponding user interaction with the app. For instance, the `onClick()` method of a registered listener is triggered when the corresponding button is pressed by a user, so we consider it as an edge feature. An example of system-generated events is when the system sends a short-message-received event, which triggers the `onReceive()` method registered for the SMS_RECEIVED intent in the manifest file. Clearly, this `onReceive()` method is not triggered by direct user interaction, so it is not considered as an edge feature. The *feature view graph* is defined in Definition 5.

Definition 5. ***(Feature View Graph)*** *A feature view graph of a smartphone app is based on its view graph, $G(V, E)$, where certain features are selected and attached to V and E.*

After creating the feature view graph of a plaintiff app and a suspicious app, ViewDroid measures the similarity between the two graphs by an applying subgraph isomorphism algorithm. Since it is an NP-complete problem, we need to improve the performance of graph matching. To this end, we apply a pre-filter to eliminate those more obvious non-matching pairs in advance.

5.3.2 System Architecture

Figure 5.3 shows the system architecture of ViewDroid, which has three primary components. Given two Android apps in *.apk* format, the *Code Extractor* will extract and parse the smali code (the dissembled version of Dalvik bytecode), human-readable Android Manifest file and layout XML files from each app's installation package. After that the *View Graph Constructor* performs some static analysis to generate a feature view graph for each app. A pre-filter is applied to remove app pairs that are not likely to be similar. Then the *Graph Similarity*

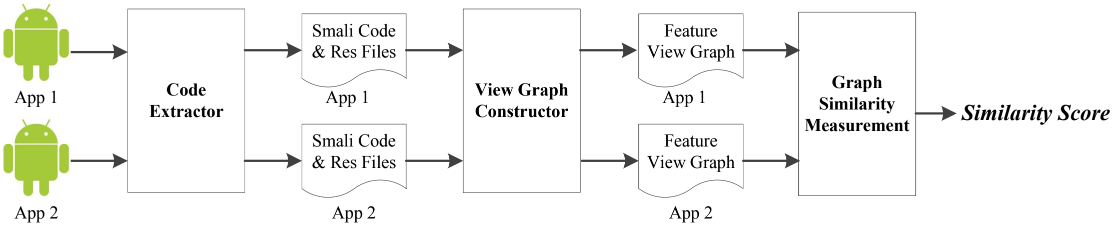


Figure 5.3. The ViewDroid system architecture.

Checker compares two feature view graphs and calculates a similarity score.

Code Extractor. The user interface layouts of an Android app are usually defined in XML files in the res/layout/ directory. The activity of a view is implemented in the *classes.dex*, which is compiled as Dalvik bytecode. Instead of focusing on the view layout that can be easily modified, we conduct the analysis on the activity class, which defines the functionality of a view and indicates the navigation between views. We choose to perform the static analysis directly on the smali code, which is an intermediate representation of Dalvik bytecode. This is because smali code is the direct dissembled version of Dalvik binary with rich annotation information. Our static analysis also uses some information from the Android manifest files and the layout component files. Existing tools can be leveraged to extract smali code and human-readable XML files from android app packages (e.g. baksmali [22], AXMLPrinter2 [91], apktool [21]).

View Graph Constructor. As discussed in Section 5.1, the layout of an app view is usually defined in XML resource files and loaded by activity code during the execution to be presented to users. Most activities load a view by invoking the `setContentView()` function with an XML file name as the parameter, in its `onCreate()` function. A few special views are loaded by other functions, e.g., the Settings view is loaded by the `addPreferencesFromResource()` function. View navigation is implemented by activity switching. When an activity calls another activity, an instance of the callee activity is created, a new view associated with the callee will be loaded and put on the top of the system's view stack to be presented to users. An activity switches to another activity by invoking function `startActivity()` or `startActivityForResult()` with an `Intent` object as the parameter. As a result, we can construct the view graph by statically analyzing these function invocations.

The detailed steps of view graph construction are as follows:

1. **Generate view nodes**: We need to collect all the activities that are associated with potential UI views, each of which is usually a separate smali file loading a view layout in its `onCreate()` function. In each activity, we parse and grep the view loading function, such as `setContentView()` and `addPreferencesFromResource()` in the `onCreate()`function. The parameters of these view loading functions are the names of the XML resource files. After parsing all this relevant information, every view node and its relation to the corresponding activity class is generated.

2. **Extract view node features**: For the features of the view nodes, we only focus on the Android framework specific APIs. Since the Android platform use Java APIs that are built on a subset of the Apache Harmony Java implementation, we consider this set of APIs are more vulnerable to renaming attacks. Attackers can easily find semantics similar or equivalent APIs from other sources. However, the set of Android specific APIs, e.g., API methods from the *android.security.KeyChain* or *android.nfc.NfcManager* classes, are very hard to be replaced. In order to interact with the Android platform, an app has to register certain permissions in the manifest files and use the relevant APIs to perform tasks. Based on this observation, we build the feature for each view node accordingly. We first analyze each activity class file associated with a view node to extract a set of invocations of the Android specific APIs. Then we can build a *invocation vector* for each view node. In such invocations vector, instead of making a counter for each API, we only flag 1 whenever an API is invoked in the activity. This can protect ViewDroid from dummy code insertion attack and can also improve the efficiency on the invocations vector pattern matching.

3. **Generate edges**: Edges in the feature view graph represents the activity switch relationship among the set of views. The source view is associated with the caller activity of the `startActivity()` or `startActivityForResult()`

functions. The target view is associated with the activity declared by the `Intent` object. There are six kinds of `Intent` constructors [92]:

(1) `Intent()`

(2) `Intent(Intent o)`

(3) `Intent(String action)`

(4) `Intent(String action, Uri uri)`

(5) `Intent(Context packageContext, Class<?> cls)`

(6) `Intent(String action, Uri uri, Context packageContext, Class<?> cls)`

As described in [41], constructors (5) and (6) specify the target activity in an explicit way with a particular class name. We can perform analysis to trace back to this hard-coded class name. Constructors (3) and (4) initialize an implicit Intent object by an action name, with or without a URI. The associated target activity, which could be within the same app or in another app, is selected by matching *intent filters* in the Android manifest files. The external target is undecidable without knowing other apps installed in a smartphone. In ViewDroid, we create a general destination node `external_activity` to represent all external targets and add an edge from the source activity to this node. Constructor (1) initializes an empty intent, which is surrounded by `setClass()`, `setComponent()` or `setAction()`. Hence, the identification of the target activity is the same as constructors (3)-(6). Constructor (2) copies another Intent object *o*. In this case, our analysis needs to trace back to the activity, which is specified by the constructor of the object *o*.

In order to figure out all the possible switching relationships among views, static analysis is performed. By analyzing all `startActivity()` and `startActivityForResult()` functions, we can stitch the caller activity and callee activity and therefore create an edge from the view of the caller activity to the view of the callee activity. Our view switching-based invocation graph is more robust to code obfuscation than the traditional call graph, because it does not rely on the exact call sequence starting from one view node and ending at another view node. Whenever there is a view switching relationship, an edge is built to link the two views. It captures the user's real experience of view switching. Even though there might be several method invocations between an actual view switching, we ignore all the intermediary method calls, but just stitch the source and end view nodes for the corresponding activity classes. As long as attackers want to keep most functionality of the original app, the view switching relationship cannot be changed.

4. **Extract edge features**: In order to minimize false matches and improve the efficiency of similarity measurement in a latter stage, we add a feature to each edge. It is the user-generated event that triggers the view switch. During the static analysis, we can locate the `startActivity()` or `startActivityForResult()` functions and analyze which function call actually triggers the view switching. The trigger could be library provided event listener, such as `onClick()`, `onTouch()`, `OnItemSelected()` etc, or app developer self-defined functions. We consider these triggers to be the features of edges. Note that because the names of self-defined functions can be easily modified by an attacker, so we label all the developer self-defined trigger functions with the same name `self_defined_trigger` and consider them potentially matched with each other.

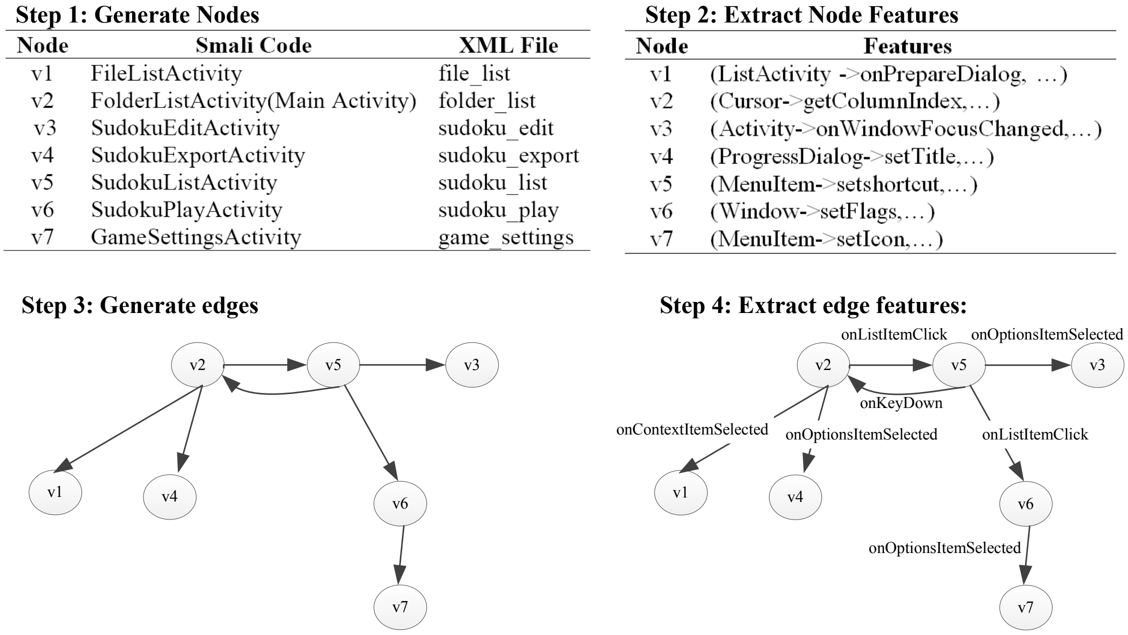
Figure 5.4 illustrates the steps of view graph construction for a simple Sudoku app. Figure 5.5 shows the feature view graph of an app that repackages the original app in Figure 5.4. The repackaged app copies the original app, adds an AdActivity (node $v8$) and additional social network functions (nodes $v9$, $v10$, $v11$, $v12$). Figure 5.6 shows the feature view graph of an independent app. Note that to make the graph clear in these figures, we omit the node features.

Graph Similarity Checker. We apply the VF2 [93] subgraph isomorphism algorithm to measure the similarity between two feature view graphs.

A pre-filter is leveraged to reduce the graph pairs that need to be compared. If one of the following three criteria meets, we will consider that they are not repackaging cases:

(1) If the size of two view graphs differs a lot (specifically, the size of the bigger graph is at least 3 times of the smaller graph);

(2) If the node features (i.e., those Android specific APIs considered in feature view



Step 1: Generate Nodes

Node	Smali Code	XML File
v1	FileListActivity	file_list
v2	FolderListActivity(Main Activity)	folder_list
v3	SudokuEditActivity	sudoku_edit
v4	SudokuExportActivity	sudoku_export
v5	SudokuListActivity	sudoku_list
v6	SudokuPlayActivity	sudoku_play
v7	GameSettingsActivity	game_settings

Step 2: Extract Node Features

Node	Features
v1	(ListActivity ->onPrepareDialog, …)
v2	(Cursor->getColumnIndex,…)
v3	(Activity->onWindowFocusChanged,…)
v4	(ProgressDialog->setTitle,…)
v5	(MenuItem->setshortcut,…)
v6	(Window->setFlags,…)
v7	(MenuItem->setIcon,…)

Figure 5.4. An example of view graph construction

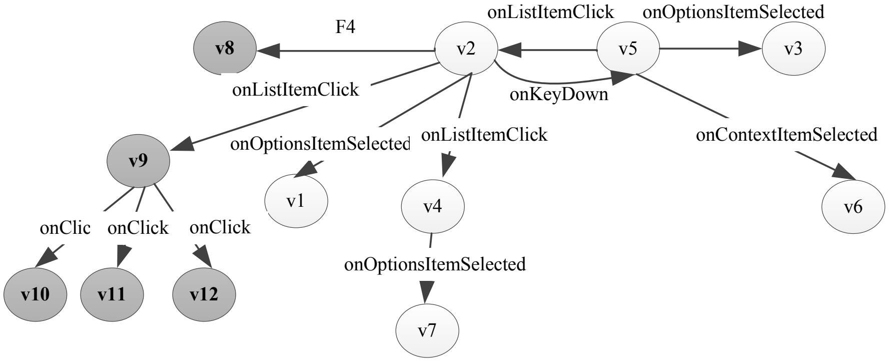


Figure 5.5. The feature view graph of a repackaging app.

graphs) in two view graphs have limited overlap (i.e., the number of overlapped features is below 1/3 of the size of the smaller graph)

(3) If the sets of edge features in two view graphs have limited overlap (i.e., the number of overlapped edge features is below 1/3 of the edge number in the smaller graph)

When two graphs are compared by the subgraph isomorphism algorithm, only

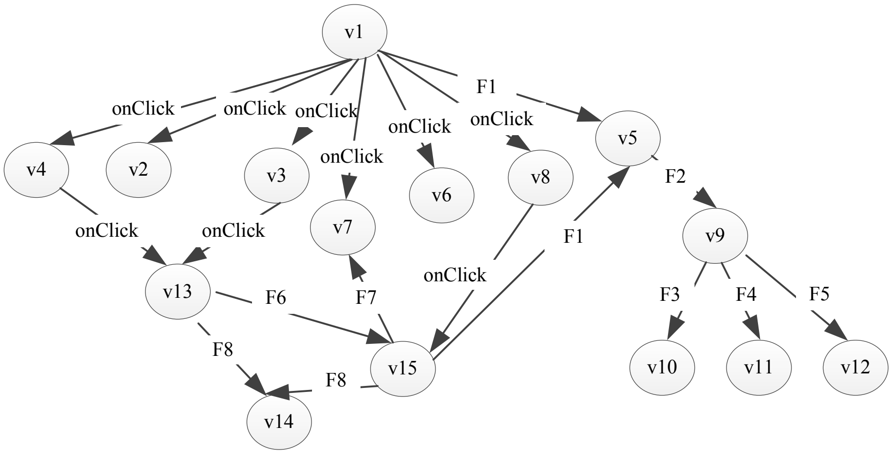


Figure 5.6. The feature view graph of an independent app.

nodes and edges with similar features can be matched. We consider two view nodes are similar when their API invocation vectors have the Jaccard distance below 0.5. The Jaccard distance between two sets A and B is calculated with Formula 5.1. Edges with the same event listener are considered as a matched pair. Not only can this feature pre-comparison reduce false matches of nodes and edges, thus decreasing the false positives caused by simple view graphs, but it can also improve the efficiency of subgraph isomorphism computation.

$$J_d(A, B) = 1 - \frac{A \cap B}{A \cup B} \tag{5.1}$$

If apps A and B have m matched nodes, with n_A and n_B nodes in their feature view graphs, respectively, their similarity score is calculated as:

$$\text{similarity score} = \frac{m}{min(n_A, n_B)}$$

5.4 Evaluation

ViewDroid is implemented in Python and Shell-script. The whole system is built with 2400 lines of Python code and 400 lines of Shell-scripts. Our experiment was

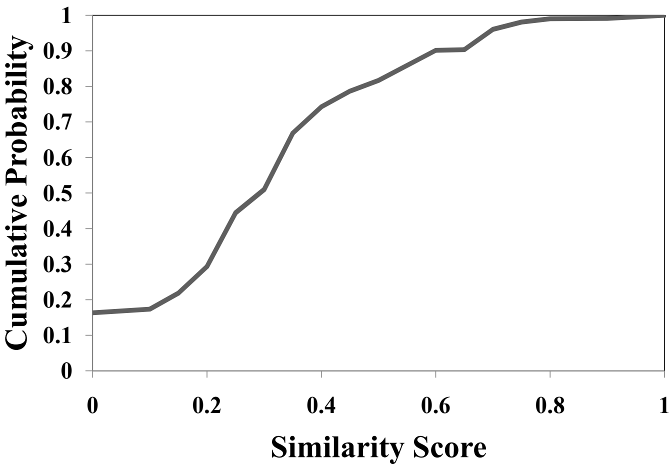


Figure 5.7. The cumulative distribution function (CDF) of similarity scores.

conducted on a commodity machine with 1.6 GHz Intel Core i5 processor and 4 GB memory.

We have two sets of experiments. First, we conduct evaluation on a large set of real-world apps to measure the effectiveness and efficiency of ViewDroid. We also test the percentage of the repackaged malware cases. Second, we evaluate the obfuscation resilience of ViewDroid by applying different obfuscation techniques on existing apps and using ViewDroid to detect their similarities.

5.4.1 Real-world Large-scale Experiment

5.4.1.1 False Positive and Efficiency

We crawl totally 10, 311 top Android apps from Google Play. These apps belong to 20 categories. We randomly choose 100 samples from each category and compare them with apps in the same category in a pairwise way. Totally 573, 872 app pairs are compared.

Figure 5.7 shows the cumulative probability of similar scores of these 573, 872 app pairs. Most pairs (90%+) have similarity score less than 0.6. There is a gap between similarity scores 0.6 and 0.7. As a result, we set the similarity score threshold at 0.7 in the evaluation.

After applying ViewDroid to detect the repackaged apps, we manually check the detected pairs to measure false positives. The manual checking has two criteria: (1) We execute the app on a smartphone to check the similarity of their functionality; (2) We check the code, including smali files, layout files and the permissions. Only when both criteria are similar, we consider them as the real repackaging cases. We find 129 false matched pairs in total in 11 categories. Most (112 out of 129) of the false matches are caused by the invocations of ad libraries. When two apps share the same ad libraries and one app's graph size is relatively small, the matched nodes related to the common ad libraries will result in a high similarity score. These false matches can be eliminated by whitelisting known ad libraries. That is, we can simply ignore views that are generated by whitelisted libraries. The other 17 false matches are due to that one of the apps in each pair is very simple. For their view features, no special API is invoked and therefore nodes are not distinguishable. Moreover, their view graphs are small and easy to find matchable (sub)graphs. Our detection results, after adding a whitelist to rule out the known ad libraries, are shown in Table 5.1. The percentage column is the proportion of apps, which either repackage other apps or are repackaged by others, in all apps in each category. On average 4.7% among tested apps are found to be the real repackaging cases. The book and comic categories have more repackaging cases than other categories, because in both categories, there are existing products that can convert an ebook into an Android app. The apps generated by the same converting product are detected as repackaging pairs by ViewDroid. They are true positives since they share the same code base and the same views.

Among all 542 repackaging pairs, 262 of them belong to lazy attacks. The malware cases are analyzed in Section 5.4.1.3. The other pairs belong to the amateur attacks. Note that ViewDroid only measures the similarity between two apps. It does not identify which one is the original app and which one is the repackaged one.

The average execution time of ViewDroid for each testing is listed in Table 5.2. It is about 11s per pair. In rare cases, the graph construction time and graph comparison time may take minutes. Only 0.6% apps take more than 1 minute to construct view graph and 0.18% pairs need more than 1 minute to conduct graph

Table 5.1. The repackaging apps detected by ViewDroid

Category	Pair#	App#	Repackaged Pair	Repackaged App	%
Books	34,550	495	81	55	11.1%
Business	23,882	455	10	13	2.9%
Comics	40,850	558	110	75	13.4%
Communication	20,582	487	0	0	0.0%
Education	40,950	559	7	11	2.0%
Entertainment	25,758	512	10	16	3.1%
Finance	37,650	526	9	13	2.5%
Game arcade	30,496	543	64	37	6.8%
Game cards	27,329	545	11	13	2.4%
Game casual	20,662	509	12	18	3.5%
Health	36,550	515	13	20	3.9%
Lifestyle	20,538	509	10	13	2.6%
Media	39,150	541	56	35	6.5%
Medical	38,650	536	14	21	3.9%
Music	19,655	496	21	20	4.0%
News	10,466	495	21	24	4.8%
Personality	37,050	520	31	25	4.8%
Photography	23,914	518	17	22	4.2%
Shopping	28,185	495	23	23	4.6%
Social	17,005	497	22	26	5.2%
Total	573,872	10,311	542	480	4.7%

Table 5.2. The execution time of ViewDroid (in second)

	Code Extraction	Graph Construction	Graph Comparison
Max	15	146	590
Avg	4	6	1

comparison. In addition, when applying ViewDroid to check a large number of apps, code extraction and view graph construction for each app is only performed once.

5.4.1.2 False Negative

In this section, we use a set of repackaged apps provided by a research group to measure the false negative rate of ViewDroid. These apps were collected from multiple Android markets. The app dataset includes totally 901 pairs of apps, whose view graphs have more than 3 view nodes. By setting the similarity score threshold at 0.7, as in Section 5.4.1.1, ViewDroid detects 868 pairs as repackaging cases. Among 659 of them, each pair of apps have the similarity score 1.0.

We then manually check the 33 pairs that are not detected by ViewDroid. They can be divided into three different categories. (1) For 11 pairs, two apps of each pair do not share or share very little common code. They do not have common functionalities or views either. As a result, not reporting them is the correct detection result for these 11 pairs. They were falsely included in the app dataset. (2) Another 10 pairs are not real repackaging cases either, although they do share some code between each other. The shared code is not related to the functionalities or the views of these apps, but is used as malicious payload to create ad shortcuts or to send out messages without users' awareness. That is, attackers use different apps to propagate the same malicious payload. Therefore, ViewDroid is correct again not reporting them. (3) The other 12 pairs are false negatives of ViewDroid at detection threshold 0.7. Here, each pair of apps have repackaged code related to their major functionalities, but have different code that implements "add-on" functions. These add-on components are relative large and complex compared to their carrier code. For example, two apps both implement a Ninja game. The matched view nodes detected by ViewDroid are the game itself, while the unmatched view nodes represent different social network functions. It is very likely that these two apps both repackaged another benign app by inserting their own customized social network library, which targets a specific market. The similarity scores of false negative cases are all between 0.5 and 0.7. It indicates that ViewDroid is able to find their common views. The false negative rate of ViewDroid at detection threshold 0.7 is 1.3%.

5.4.1.3 Malware

We use VirusTotal [94], an online malware detection service, to scan all the repackaged pairs detected in Section 5.4.1. Among the 480 apps identified as involved in repackaging cases (either the original ones or the repackaged ones) in our previous experiment, we detect 93 malware, which is 19.3% of repackaged apps. The malware types are listed in Table 5.3. They mainly belong to two different cat-

Table 5.3. The malware attacks detected by ViewDroid

Type	Number
Trojan.FakeApp/FakeFlash	25
Adware.Airpush	14
Adware.Plankton	17
Adware.LeadBolt	26
Other Adware	10
Virus	1

egories: Adware and Trojan horse. Adwares aggressively show advertisements to smartphone users.Trojan horses usually pretend to be legitimate apps, but steal sensitive information covertly. There is one virus detected. It is labeled as `Virus:BAT/Rbtg.gen`.

5.4.1.4 Category-based Evaluation

Next we illustrate a different kind of evaluation on real-world apps. We first search by some keyword in Google Play, then download the returned apps and pairwisely compare their similarities. While our previous large-scale experiment randomly chooses pairs in the app market to evaluate the effectiveness and scalability of ViewDroid, this experiment is more interesting to individual app developers and app users to understand how repackaging may affect them.

We list two examples here. The first keyword is sudoku and we download 20 sudoku game apps. Based on the similarity scores, we cluster these apps as shown in Figure 5.8. An edge indicates two apps have a similarity score higher than 0.7. The largest cluster has 9 apps. The app with dashed circle is similar to all the other 8 apps in the cluster. The other two clusters both have 3 apps. Our manual checking verifies that the result has no false positives and false negatives. Further analysis indicates that there are 3 pairs belonging to lazy attack, where plagiarists only repackage the original apps without changing their functionality. The similarity scores of these pairs are 1.0. ViewDroid also discoveries one malware case, the red big node in Figure 5.8. VirusTotal identifies it as the Airpush Adware, which aggressively shows ads in the Android notification bar. This app inserts Airpush ads module into the original app and slightly modifies the functionality by removing a strategy help view. The other repackaging pairs are all amateur attacks, where

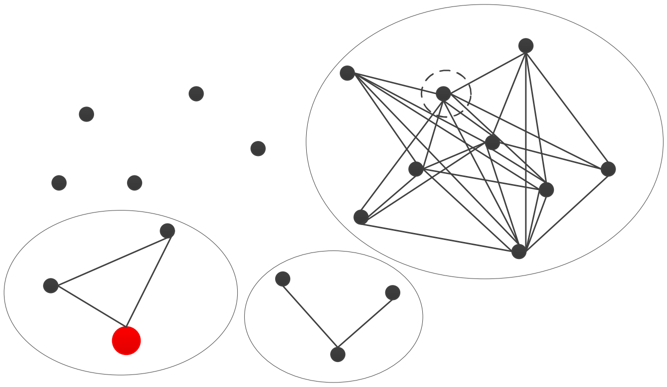


Figure 5.8. The cluster of sudoku apps based on the similarity scores.

functionalities are added or removed, such as social network modules, help view, strategy hint views and advertisements.

In the second example we search by the keyword flashlight and download 29 apps. We find 15 pairs with similarity scores higher than 0.7. Our manual checking indicates that 3 pairs are false positive cases. They are all caused by one app that has 4 views, only one of which relates to its functionality whereas the other three are generated by an ad library. When compared to apps that share the same library with it, the three ad views are matched and the similarity scores are 0.75. Again, such false positives can be eliminated by whitelisting the ad libraries. The similarity cluster is shown in Figure 5.9. Four clusters have more than one app. Among all the 12 repackaged pairs, 2 belong to the lazy attacks, and 9 belong to the amateur attack where views are added or removed (e.g., the "about" view, "setting" view). One malware attack, shown as a big red node in Figure 5.9, is found. It is reported as a trojan horse by VirusTotal (there is indeed another malware in the 29 downloaded apps, but it is not the app repackaging case. It is identified as Plankton [95]).

5.4.2 Obfuscation Resilience

To test the obfuscation resilience of ViewDroid, we try to obfuscate the existing apps and malware with different obfuscators, and then check with ViewDroid the similarity score between each original app and its corresponding obfuscated one.

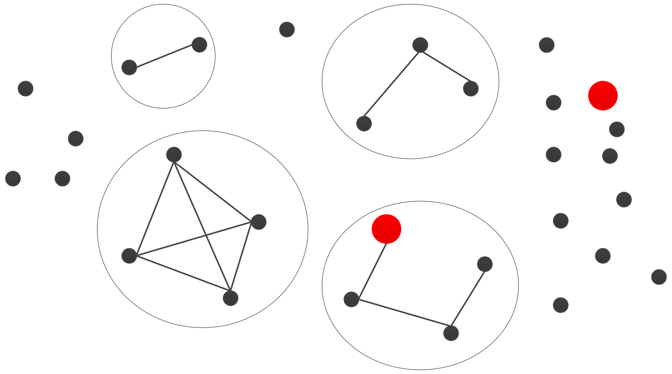


Figure 5.9. The cluster of flashlight apps based on the similarity scores.

Most existing popular obfuscation tools (e.g. ProGuard [96] and DexGuard [97]) work on Java source code level and their obfuscators are limited to method renaming, string encryption and class name encryption, etc. Therefore, we choose to use an obfuscation resilience evaluation tool developed by Huang et al. [30]. This evaluation framework can obfuscate and repackage apps by using one or multiple obfuscators from different Java bytecode obfuscation platforms (e.g., Sandmarks [28]). It directly targets the Dalvik bytecode. This actually mimics the real world scenarios where a plagiarist or repackager who only has access to the compiled Dalvik bytecode but not the high-level Java source code and is eager to use various obfuscation techniques to evade detection. In our current obfuscation resilience test, we equip the framework to perform 39 obfuscators from both SandMarks [28] and KlassMaster [29]. To our knowledge, this is the broadest obfuscation resilience evaluation on Android app repackaging detection.

First of all, we generate pairs of apks from the obfuscation resilience evaluation tool. Then, we use ViewDroid to measure the similarity pairwisely between the obfuscated apk and the original apk. The higher similarity scores our ViewDroid returns for each specific obfuscator, the better resilience against that particular obfuscation.

We choose 50 apps from the Android app market based on different categories and 50 malwares from the malware Gnome project based on different families [98]. With this 100 Android app set, we perform *broadness* analysis and *depth* analysis to evaluate the obfuscation resilience aspect of ViewDroid provided by the evaluation framework. The broadness analysis result shows the general weakness and strength of ViewDroid against a broad range of obfuscation techniques. In this analysis, each obfuscator is applied individually. On the other hand, the depth analysis result evaluates the overall obfuscation resilience of ViewDroid against deep code manipulation by serializing a set of obfuscators. In this analysis, ViewDroid is evaluated against repackaged apps that have been obfuscated by multiple obfuscators. For example, an app may be obfuscated by variable renaming, followed by noise injection and/or control-flow flattening. With depth analysis, we can test the robustness of our detection scheme against more sophisticated obfuscation attacks.

5.4.2.1 Applying Single Obfuscation Algorithm

In our current evaluation setup, the *broadness* analysis is based on 39 obfuscation algorithms from *SandMarks* and *KlassMaster*. In Table 5.4, the *Obfuscation Algorithm* columns indicate the names of the obfuscation algorithms applied in our framework. The *ViewDroid* columns list an average similar score for each obfuscation case. Specifically, in each obfuscation case, ViewDroid computes a similarity score for each original app (among totally 100 apps) and its obfuscated version and finally reports the average over 100 apps. The *AndroGuard* columns are the results reported by Huang et al. in [30], and we also compute three average similarity scores for AndroGuard based on three obfuscators from KlassMaster, which were not provided in the previous case study. All these three obfuscators have a K-tag at the beginning of the obfuscators' names in Table 5.4.

Based on the classification by Collberg et al. [88], all the single obfuscators can be categorized as *layout obfuscation*, *control-based obfuscation* and *data-based obfuscation*, which are tagged L, C and D after each obfuscator. The detailed explanation of the difference between these categories can be found in Huang et al. [30]. Overall, ViewDroid has better obfuscation resilience than AndroGuard. This is because in ViewDroid, repackaging detection is performed based on the similarity of the high level semantics of the app using the created featured view graph, while ignoring the detailed control/data dependency or data structure. From the result, we can see that only 4 out of 39 obfuscators have an effect on ViewDroid, and the average similarity scores of all the other 35 obfuscators tested against ViewDroid are all 1.00.

Table 5.4. Average Similarity Score by ViewDroid compared with AndroGuard for each Obfuscator from Broadness Analysis

Obfuscation Algorithm	ViewDroid	AndroGuard	Obfuscation Algorithm	ViewDroid	AndroGuard
Const Pool Reorder (L)	1.00	.92	Node Spliter (D)	.94	.94
Static Method Bodies (C)	1.00	.88	Class Encrypter (D)	.00	.03
Method Merger (C)	1.00	.65	Reorder Parameters (D)	1.00	.92
Interleave Methods (C)	1.00	.56	Promote Prim Register (D)	1.00	.92
Opaque Pred Insert (C)	1.00	.92	Promote Prim Types (D)	1.00	.93
Branch Inverter (C)	1.00	.77	Bludgeon Signatures (D)	1.00	.96
Rand Dead Code (C)	1.00	.92	Objectify (D)	1.00	.83
Class Splitter (C)	.97	.87	Publicize Fields (D)	1.00	.91
Method Madness (C)	.92	.43	Field Assignment (D)	1.00	.86
Simple Opaque Pred (C)	1.00	.92	Variable Reassign (D)	1.00	.85
Reorder Instructions (C)	1.00	.89	Parameter Alias (D)	1.00	.92
Buggy Code (C)	1.00	.67	Boolean Splitter (D)	1.00	.85
Inliner (C)	1.00	.89	String Encoder (D)	1.00	.87
Branch Insert (C)	1.00	.87	Overload Names (D)	1.00	.91
Dynamic Inliner (C)	1.00	.84	Duplicate Registers (D)	1.00	.89
Irreducibility (C)	1.00	.86	Rename Registers (D)	1.00	.96
Opaque Branch Insert (C)	1.00	.85	False Refactor (D)	1.00	.95
Exception Branch (C)	1.00	.81	Merge Local Int (D)	1.00	.94
K-Flow Obfuscation (C)	1.00	.77	K-Name Obfuscation (D)	1.00	.89
			K-String literals Encrypter (D)	1.00	.91

The *Class Encrypter* obfuscator reduces the similarity score to 0, which is the only obfuscator that ViewDroid returns a lower score than AndroGuard. However, the score for AndroGuard is .03, which is very close to zero. This indicates that static analysis based detection schemes are not well-suited for encryption based obfuscation. By encrypting class files and decrypting them at runtime, Class Encrypter can completely hide the static structure of the program. However, certain heuristic can be built to preprocess these extreme encryption cases. For instance, whenever decryption or decoding is used in the program very intensively or is identified for a very large portion of the code, it can be flagged as suspicious. Usually, dynamic analysis-based detection is needed in this situation, which is, however, lack of scalability. Overall, handling the heavy encryption and encoding-based obfuscation is an interesting topic to explore in the future.

The other obfuscation algorithms that have some influence on ViewDroid are *Node Spliter*, *Method Madness* and *Class Splitter*. After further analysis of the feature view graph pairs computed from the 100 apps' obfuscated versions, we find that some graph nodes are split by obfuscators Node Spliter and Class Splitter, and the names of the methods that trigger view switching are replaced by some random names by Method Madness, which can potentially modify the feature of our view graph. However, from the overall similarity scores of these four obfuscators, we can see that these types of obfuscation cannot be performed frequently, as certain conditions have to be satisfied before these obfuscators make the actual manipulations. For instance, some class inheritance relationship has to be met in order to perform Node Spliter or Class Splitter, and also relevant specification in the Android manifest file should be changed accordingly. Furthermore, the obfuscation of method randomization in Method Madness cannot be performed directly on the Android framework APIs, tedious method rewriting work has to be performed before replacing the invocation of the Android APIs. For instance, simply changing the invocation *Landroid/app/Activity.dispatchTouchEvent (Landroid/view/MotionEvent;)Z* into *Landroid/app/Activity. M103456d(Landroid/view/MotionEvent;)Z* does not work. As a result, we find that most apks become non-executable after the Method Madness obfuscation.

5.4.2.2 Serializing Multiple Obfuscation Algorithms

Practically, especially when detection algorithms become more powerful, it is very possible that an attacker will try a combination of various obfuscation algorithms. Hence, besides the *broadness* analysis performed on ViewDroid, for *depth* analysis we also apply multiple obfuscators by serializing the top-three obfuscators reported from our broadness analysis, excluding the *Class Encrypter*. Due to the conflicts among various obfuscators, not all the obfuscated apks are complete. We test various permutation cases with these three obfuscators and find two of all the permutations can be performed more successfully for the testing apps. One can output 99 out of 100 obfuscated apks and the other outputs 96 out of 100 for all the tested ones. These two interesting permutations are shown as follows:

1. *[Node Spliter ⇒ Method Madness ⇒ Class Splitter]*
   Average Similarity Score of 99 apps : 0.915;
2. *[Class Splitter ⇒ Method Madness ⇒ Node Spliter]*
   Average Similarity Score of 96 apps : 0.906;

Both cases have the same three obfuscators but at a different serializing order. Although they can slightly reduce the average similarity scores by ViewDroid compared to the solely applying the obfuscator *Method Madness* case (with score 0.92), these scores are both above .90, sufficiently large for the obfuscated apps to be detected. *Case* 1 reduces the average score from 0.92 to 0.915, which shows that applying serialized multiple obfuscators has only slightly higher influence on ViewDroid than applying a single obfuscator. For *Case* 2, the average similarity score is a little bit lower than *Case* 1. However, there are four apps that cannot finish the whole serialized obfuscations. This indicates that although serialized obfuscation is slightly more powerful, the attacker has to take the risk of ending up with incomplete obfuscation. We encountered more failures when performing other orders of serialization. Overall, our evaluation demonstrates that multiple obfuscations are hard to be serialized, and even if successfully performed, they have little impact on ViewDroid's detection capability. Huang et al. [30] also reported that, in some scenarios, applying multiple obfuscations can lower the similarity scores reported by tools such as *AndroGuard*. Our experiment shows that ViewDroid's high-level abstracted birthmark is not affected much by the low-level (multiple) code obfuscation.

5.5 Discussion

5.5.1 Attack Analysis on ViewDroid

As discussed in Section 5.2, based on different repackaging purposes, ViewDroid might face various types of attacks.

- **Lazy attack:** In this attack, the attacker does not change the functionality of original apps but applies automatic code obfuscation tools to repackage an app. As a result, a lazy attack does not change the view navigation relations of an app. In addition, code obfuscation has little impact on the feature view graph generation, as demonstrated by evaluation in Section 5.4.2. Therefore, ViewDroid can effectively detect such attacks.

- **Amateur attack:** An attacker not only applies automatic code obfuscation but also makes small modifications on the functionalities. The feature view graph could be changed slightly. However, because we use the subgraph isomorphism algorithm to compare graphs, small changes of the view graph may reduce the similarity score a little but will not affect the overall detection result much. As a result, ViewDroid can tolerate small changes on app functionalities and views.

- **Malware:** An attacker inserts some malicious payload into the original program while trying to make the repackaged app look the same or similar to the original one in order to leverage the popularity of the original program for wide propagation. Clearly, their feature view graphs would also be very similar. Therefore, ViewDroid can effectively detect such repackaging.

Other Potential Professional attacks: An attacker, who knows ViewDroid, may attempt to change feature view graphs to evade detection. Attackers may (1) insert a dummy view into the path of two directly connected views; (2) split one view node into two view nodes; (3) self implement or obfuscate the invocation of `startActivity()` and `startActivityForResult()` functions. Since we use the subgraph isomorphism algorithm with a certain matching threshold (e.g., 0.7), in order to affect the detection result, attackers need to modify many views of the original apps. On one hand, it will significantly increase the workload of repackaging an app. On the other hand, the dummy nodes, edges and self-implemented functions will increase the code size and decrease the performance of apps. We have not seen such attacks in the real world yet.

5.5.2 Limitations

ViewDroid can detect the repackaging of non-trivial apps effectively, but for the detection of apps with few views, more false positives can be reported. Even so, the API vector node features can significantly reduce such false positives, because only nodes with very similar API vectors can be matched.

ViewDroid can effectively detect the following three types of mobile app repackaging attacks: lazy attacks, amateur attacks and malware. However, some professional attacks can potentially change view graphs, regardless of the workload of attackers and the performance overhead of the repackaged apps. Dummy view insertion may be defeated by examining the trigger function of the view switches. If a switch is not triggered by user behavior, we can merge the target view with its predecessor/successor in the feature view graph. A similar strategy has been used by Chen et al. [42] to check for malicious behavior. Current ViewDroid can effectively raise the bar of app repackaging.

As shown in our evaluation, ViewDroid has false negatives when the encrypter obfuscation is used. This is because encryption changes the code completely and hides all the static characteristics of an app. This is also a common problem of all static analysis-based detection. To defeat against such attacks, dynamic analysis may be applied. However, dynamic analysis is not efficient enough to be used as a large-scale detection approach. This is the fundamental tradeoff between accuracy and performance. How to build a hybrid approach to leverage both dynamic and static analysis for encrypter obfuscation is also a very interesting and important topic.

5.6 Summary

In this chapter, we proposed a user interface-based Android app repackaging detection method, ViewDroid. The evaluation results show that ViewDroid can effectively detect Android app repackaging with the presence of various obfuscation techniques. ViewDroid is also efficient enough for performing large-scale experiments.

Chapter 6

Conclusion

This dissertation focuses on plagiarism detection. It proposes three components: LoPD, ValPD and ViewDroid to solve different types of plagiarism, i.e., software plagiarism, algorithm plagiarism and Android app repackaging, respectively. LoPD is a program logic-based software plagiarism method. It applies symbolic execution and weakest pre-condition reasoning to find dissimilarity between two programs to rule out non-plagiarism cases. ValPD is a value-based algorithm plagiarism detection approach. It uses core values as algorithm signature and proposes the N-version and the annotation method to extra core values. A value dependence graph-based similarity metric is proposed to solve value reordering problem. ViewDroid is a user interface-based Android app repackaging detection scheme. It leverages view graph to characterize an app. View graph is a higher level birthmark than code-level birthmarks and therefore is more resilient to code obfuscation. All three components are obfuscation resilient and none of them need the source code of suspicious program.

Each method is proposed, implemented and evaluated on real-world programs or apps. The evaluation results demonstrate their effectiveness, even with the presence of various automatic code obfuscation techniques. Besides, ViewDroid is efficient and scalable enough to perform evaluation on a large scale app set.

This work not only provides more obfuscation-resilient approaches to software plagiarism detection and mobile app repackaging detection, but also lays a foundation as a first step towards a practical solution for algorithm plagiarism detection. The proposed techniques can be used to collect plagiarism evidences for lawsuits, as well as to maintain the health of open source community and the app markets. They will help deter the violation of intellectual property .

In the future, we will study how to apply our schemes to detect partial plagiarism. For the smartphone app repackaging problem, we are going to investigate how to deal with professional attacks.

Bibliography

[1] "Compuware, IBM settle lawsuit," `http://www.zdnet.com/news/compuware-ibm-settle-lawsuit/141925`.

[2] Zhou, W., Y. Zhou, X. Jiang, and P. Ning (2012) "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, pp. 317–326.

[3] Jhi, Y.-C., X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu (2011) "Value-Based Program Characterization and Its Application to Software Plagiarism Detection," in *33rd International Conference on Software Engineering (ICSE 2011), the SEIP track*.

[4] Liu, C., C. Chen, J. Han, and P. S. Yu (2006) "GPLAG: detection of software plagiarism by program dependence graph analysis," in *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 872–881.

[5] Schleimer, S., D. S. Wilkerson, and A. Aiken (2003) "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pp. 76–85.

[6] Yang, W. (1991) "Identifying syntactic differences between two programs," *Softw. Pract. Exper.*, **21**(7), pp. 739–755.

[7] Myles, G. and C. Collberg (2005) "K-gram based software birthmarks," in *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pp. 314–318.

[8] Park, H., S. Choi, H. il Lim, and T. Han (2008) "Detecting Java Theft Based on Static API Trace Birthmark," *Advances in Information and Computer Security*, pp. 121–135.

[9] Myles, G. and C. S. Collberg (2004) "Detecting Software Theft via Whole Program Path Birthmarks," in *ISC*, pp. 404–415.

[10] Tamada, H., K. Okamoto, M. Nakamura, A. Monden, and K. ichi Matsumoto (2004) "Dynamic software birthmarks to detect the theft of Windows applications," in *Int. Symp. on Future Software Technology.*

[11] Schuler, D., V. Dallmeier, and C. Lindig (2007) "A dynamic birthmark for Java," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pp. 274–283.

[12] Zhang, F., Y. Jhi, D. Wu, P. Liu, and S. Zhu (2012) "A first step towards algorithm plagiarism detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ACM, pp. 111–121.

[13] King, J. C. (1976) "Symbolic execution and program testing," *Commun. ACM*, **19**.

[14] Dijkstra, E. W. (1976) *A Discipline of Programming*, Prentice Hall, Inc.

[15] Hoare, C. A. R. (1969) "An axiomatic basis for computer programming," *Commun. ACM*, **12**(10).

[16] Lu, B., F. Liu, X. Ge, B. Liu, and X. Luo (2007) "A Software Birthmark Based on Dynamic Opcode N-gram," *International Conference on Semantic Computing*, pp. 37–44.

[17] Wang, X., Y.-C. Jhi, S. Zhu, and P. Liu (2009) "Detecting Software Theft via System Call Based Birthmarks," in *ACSAC*, vol. 0, pp. 149–158.

[18] ——— (2009) "Behavior based software theft detection," in *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pp. 280–290.

[19] "Number of avaliable Android applications," `http://www.appbrain.com/stats/number-of-android-apps`.

[20] Gibler, C., R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi (2013) "AdRob: Examining the Landscape and Impact of Android Application Plagiarism," in *Proceedings of 11th International Conference on Mobile Systems, Applications and Services.*

[21] "Android-Apktool: A tool for reverse engineering Android apk files," `http://code.google.com/p/android-apktool/`.

[22] "Smali: An assembler/disassembler for Android's dex format," `http://code.google.com/p/smali/`.

[23] Crussell, J., C. Gibler, and H. Chen (2013) "Scalable Semantics-Based Detection of Similar Android Applications." in *ESORICS*.

[24] ——— (2012) "Attack of the Clones: Detecting Cloned Applications on Android Markets," in *ESORICS*, pp. 37–54.

[25] Hanna, S., L. Huang, E. Wu, S. Li, C. Chen, and D. Song (2012) "Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications," in *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment.*

[26] Zhou, W., Y. Zhou, M. Grace, X. Jiang, and S. Zou (2013) "Fast, scalable detection of Piggybacked mobile applications," in *Proceedings of the third ACM conference on Data and application security and privacy*, ACM, pp. 185–196.

[27] Desnos, A. and G. Gueguen. "Android: From Reversing to Decompilation," in *Black hat 2011, Abu Dhabi.*

[28] Collberg, C., G. Myles, and A. Huntwork (2003) "Sandmarks - a tool for software protection research," in *IEEE Security and Privacy, vol. 1, no. 4.*

[29] "KlassMaster," `http://www.zelix.com/klassmaster/docs/index.html`.

[30] Huang, H., S. Zhu, P. Liu, and D. Wu. (2013) "A Framework for Evaluating Mobile App Repackaging Detection Algorithms," in *Proceedings of the 6th International Conference on Trust & Trustworthy Computing.*

[31] Lim, H.-i., H. Park, S. Choi, and T. Han (2008) "Detecting Theft of Java Applications via a Static Birthmark Based on Weighted Stack Patterns," *IEICE - Trans. Inf. Syst.*, **E91-D**(9), pp. 2323–2332.

[32] Tamada, H., M. Nakamura, A. Monden, and K. ichi Matsumoto (2004) "Design and Evaluation of Birthmarks for Detecting Theft of Java Programs," in *Proc. IASTED International Conference on Software Engineering (IASTED SE 2004)*, pp. 569–575, innsbruck, Austria.

[33] Krinke, J. (2001) "Identifying Similar Code with Program Dependence Graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01.

[34] Tamada, H., K. Okamoto, M. Nakamura, A. Monden, and K. ichi Matsumoto (2007) *Design and Evaluation of Dynamic Software Birthmarks Based on API Calls*, *Information Science Technical Report NAIST-IS-TR2007011, ISSN 0919-9527*, Graduate School of Information Science, Nara Institute of Science and Technology.

[35] Potharaju, R., A. Newell, C. Nita-Rotaru, and X. Zhang (2012) "Plagiarizing Smartphone Applications: Attack Strategies and Defense Techniques," in *Engineering Secure Software and Systems, Lecture Notes in Computer Science*, pp. 106–120.

[36] Baker, B. S. (1995) "On finding duplication and near-duplication in large software systems," in *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95.

[37] Kamiya, T., S. Kusumoto, and K. Inoue (2002) "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, **28**, pp. 654–670.

[38] Jiang, L., G. Misherghi, Z. Su, and S. Glondu (2007) "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," in *Proceedings of the ICSE '07*, pp. 96–105.

[39] Prechelt, L., G. Malpohl, and M. Phlippsen (2000) *JPlag: Finding plagiarisms among a set of programs*, *Tech. rep.*, `http://page.mi.fu-berlin.de/prechelt/Biblio/jplagTR.pdf`.

[40] Chen, K., P. Liu, and Y. Zhang (2014) "Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets," in *36th International Conference on Software Engineering (ICSE).*

[41] Zheng, C., S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou (2012) "SmartDroid: an automatic system for revealing UI-based trigger conditions in Android applications," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, ACM, pp. 93–104.

[42] Chen, K. Z., N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. Song (2013) "Contextual Policy Enforcement in Android Applications with Permission Event Graphs," in *NDSS'13*, San Diego, USA.

[43] Grace, M. C., W. Zhou, X. Jiang, and A.-R. Sadeghi (2012) "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, ACM, pp. 101–112.

[44] Anand, S., M. Naik, M. J. Harrold, and H. Yang (2012) "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ACM, p. 59.

[45] Baxter, I. D., A. Yahin, L. Moura, M. Sant'Anna, and L. Bier (1998) "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the International Conference on Software Maintenance*, ICSM '98.

[46] Komondoor, R. and S. Horwitz (2001) "Using Slicing to Identify Duplication in Source Code," in *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pp. 40–56.

[47] Gabel, M., L. Jiang, and Z. Su (2008) "Scalable detection of semantic clones," in *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pp. 321–330.

[48] Sæbjørnsen, A., J. Willcock, T. Panas, D. Quinlan, and Z. Su (2009) "Detecting code clones in binary executables," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pp. 117–128.

[49] Collberg, C. and C. Thomborson (1999) "Software Watermarking: Models and Dynamic Embeddings," in *Principles of Programming Languages 1999, POPL'99*.

[50] Cousot, P. and R. Cousot (2004) "An abstract interpretation-based framework for software watermarking," in *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pp. 173–185.

[51] Thomborson, C., J. Nagra, R. Somaraju, and C. He (2004) "Tamper-proofing software watermarks," in *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation - Volume 32*, ACSW Frontiers '04, pp. 27–36.

[52] Collberg, C., E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp (2004) "Dynamic path-based software watermarking," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04.

[53] Arboit, G. (2002) "A method for watermarking Java programs via opaque predicates," in *Proc. Int. Conf. Electronic Commerce Research (ICECR-5).*

[54] Brumley, D., J. Caballero, Z. Liang, N. James, and D. Song (2007) "Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pp. 15:1–15:16.

[55] Chen, L. and A. Avizienis (1977) "On the Implementation of N-Version Programming for Software Fault-Tolerance During Execution," in *IEEE 1st Computer Software and Applications Conference(COMPSAC 77).*

[56] ——— (1978) "N-Version programming: A fault-tolerance approach to reliability of software operation," in *IEEE 8th International Symposium on Fault Tolerant Computing (FTCS-8).*

[57] Lyu, M. R. and Y. He (1993) "Improving the N-version programming process through the evolution of a design paradigm," in *IEEE Transactions on Reliability.*

[58] Avizienis, A. (1995) "The methodology of n-version programming," in *Software Fault Tolerance.*

[59] Nagy, L., R. Ford, and W. Allen (2006) "N-Version Programming for the Detection of Zero-day Exploits," in *IEEE Topical Conference on Cybersecurity.*

[60] Cox, B., D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser (2006) "N-Variant Systems A Secretless Framework for Security through Diversity," in *15th USENIX Security Symposium (SS06).*

[61] Miller, B. P., L. Fredriksen, and B. So (1990) "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, **33**, pp. 32–44.

[62] Forrester, J. E. and B. P. Miller (2000) "An empirical study of the robustness of Windows NT applications using random testing," in *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4.*

[63] Korel, B. (1990) "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, **16**, pp. 870–879.

[64] Godefroid, P., M. Levin, and D. Molnar (2008) "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS).*

[65] Sen, K., D. Marinov, and G. Agha (2005) "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pp. 263–272.

[66] Godefroid, P., N. Klarlund, and K. Sen (2005) "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pp. 213–223.

[67] Cadar, C., V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler (2006) "EXE: Automatically Generating Inputs of Death," in *Proceedings of the 13th ACM conference on Computer and communications security (CCS '06)*, pp. 322–335.

[68] "STP Constraint Solver," `http://sites.google.com/site/stpfastprover/STP-Fast-Prover`.

[69] Ganesh, V. and D. L. Dill (2007) "A Decision Procedure for Bit-Vectors and Arrays," in *Computer Aided Verification (CAV '07)*.

[70] Madou, M., L. Van Put, and K. De Bosschere (2006) "Loco: An Interactive Code (De)Obfuscation tool," in *Proceedings of ACM SIGPLAN Workshop on PEPM '06*.

[71] Collberg, C., G. Myles, and A. Huntwork (2003) "Sandmark–A Tool for Software Protection Research," *IEEE Security and Privacy*, **1**, pp. 40–49.

[72] Song, D., D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*

[73] "BitBlaze: Binary Analysis for Computer Security," `http://bitblaze.cs.berkeley.edu/`.

[74] Ford, D. R., L. R.and Fulkerson (1954) "Maximal flow through a network," *Canadian Journal of Mathematics*, pp. 399–404.

[75] "Diablo Is A Better Link-time Optimizer," `http://diablo.elis.ugent.be/`.

[76] "Optimize Options - Using the GUN Compiler Collection (GCC)," `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[77] Kleinberg, J. and E. Tardos (2005) *Algorithm Design*, Addison-Wesley Longman Publishing Co., Inc.

[78] Avizienis, A. (1985) "The N-Version Approach to Fault-Tolerant Software," *IEEE Trans. Softw. Eng.*, **11**, pp. 1491–1501.

[79] Newsome, J. and D. Song (2005) "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *Proceedings of the NDSS 2005*.

[80] Weiser, M. (1981) "Program slicing," in *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pp. 439–449.

[81] GrammaTech Inc. *Dependence Graphs and Program Slicing*, *Tech. rep.*, white Paper.

[82] Bellard, F. (2005) "QEMU, a fast and portable dynamic translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05.

[83] "QEMU: opensource processor emulator `http://wiki.qemu.org/Main_Page`," .

[84] GrammaTech, "CodeSurfer," `http://www.grammatech.com`.

[85] "MOSS - A System for Detecting Software Plagiarism. `http://theory.stanford.edu/~aiken/moss/`," .

[86] "Semantic Designs Inc. C Source Code Obfuscator." `http://www.semdesigns.com/products/obfuscators/CObfuscator.html`.

[87] "binobf: Binary Obfuscation Software." `http://www.cs.arizona.edu/~debray/binary-obfuscation/`.

[88] Collberg, C., C. Thomborson, and D. Low (1997) *A Taxonomy of Obfuscating Transformations*, *Tech. Rep. 148*, `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html`.

[89] Cavallaro, L., P. Saxena, and R. Sekar (2008) "On the limits of information flow techniques for malware analysis and containment," in *DIMVA' 08*.

[90] Ostrander, J. (2012) *Android UI Fundamentals: Develop and Design*, Peachpit Press.

[91] "Prints XML document from binary XML file," `http://code.google.com/p/android4me/downloads/detail?name=AXMLPrinter2.jar&can=2&q=`.

[92] "Intent Android Developers," `developer.android.com/reference/android/content/Intent.html`.

[93] Cordella, L. P., P. Foggia, C. Sansone, and M. Vento (2004) "A (sub) graph isomorphism algorithm for matching large graphs," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **26**(10), pp. 1367–1372.

[94] "VirusTotal," `https://www.virustotal.com/en/`.

[95] "Security Alert: New Stealthy Android Spyware - Plankton - Found in Official Android Market," `http://www.csc.ncsu.edu/faculty/jiang/Plankton/`.

[96] "ProGuard," `http://developer.android.com/tools/help/proguard.html/`.

[97] "DexGuard," `http://www.saikoa.com/dexguard`.

[98] Zhou, Y. and X. Jiang (2012) "Dissecting Android Malware: Characterization and Evolution," *Security and Privacy, IEEE Symposium on*, pp. 95–109.

Vita

Fangfang Zhang

Fangfang Zhang is a PhD candidate in the Department of Computer Science and Engineering at Pennsylvania State University since 2008. She received the B.S. degree and M.S. degree in Computer Science from Peking University, China in 2005 and 2008, respectively.

Publications

1. F. Zhang, Y. Jhi, D. Wu, P. Liu and S. Zhu. A First Step Towards Algorithm Plagiarism Detection. In Proceedings of the 2012 ACM International Symposium on Software Testing and Analysis (ISSTA 2012), 2012.
2. W. Xu, F. Zhang, and S. Zhu. "JStill: Mostly Static Detection of Obfuscated Malicious JavaScript Code" Proceedings of Third ACM Conference on Data and Application Security and Privacy (CODASPY), 2013.
3. W. Xu, F. Zhang, and S. Zhu. "The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study." Proceedings of MALWARE 2012.
4. W. Xu, F. Zhang, and S. Zhu. "Toward Worm Detection in Online Social Networks" Proceedings of 25th Annual Computer Security Applications Conference (ACSAC), 2010.