Deviation-Based Obfuscation-Resilient Program Equivalence Checking with Application to Software Plagiarism Detection

Jiang Ming, Fangfang Zhang, Dinghao Wu, *Member, IEEE,* Peng Liu, *Member, IEEE,* and Sencun Zhu, *Member, IEEE*

***Abstract*—Software plagiarism, an act of illegally copying others' code, has become a serious concern for honest software companies and the open source community. Considerable research efforts have been dedicated to searching the evidence of software plagiarism. In this paper, we continue this line of research and propose LoPD, a deviation based program equivalence checking approach, which is an ideal fit for the whole-program plagiarism detection. 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; but if we cannot find any dissimilarity, it is more likely a plagiarism case. We leverage dynamic symbolic execution to capture the semantics of execution paths and to find path deviations. Compared to the existing detection approaches, LoPD's formal program semantics-based method is more resilient to automatic obfuscation schemes. Our evaluation results indicate that LoPD is effective in detecting whole-program plagiarism. Furthermore, we demonstrate that LoPD can be applied to partial software plagiarism detection as well. The encouraging experiment results show that LoPD is an appealing complement to existing software plagiarism detection approaches.**

***Index Terms*—Software plagiarism, program logic, semantical difference, symbolic execution, path deviation.**

ACRONYMS AND ABBREVIATIONS

LoPD	Logic Based Software Plagiarism Detection
API	Application Programming Interface
PDG	Program Dependence Graph
WPP	Whole Program Path
SSA	Static Single Assignment

NOTATIONS

I_P	The input space for program P
O_p	The output state of program P
e_{p1}	The execution paths of P with input x_1
$F_p^O(I)$	The path formula with the output state O_p
wp	The weakest precondition

Jiang Ming, Dinghao Wu, and Peng Liu are with the College of Information Sciences and Technology, Penn State University, University Park, PA 16802, USA, e-mail: {jum310, dwu, pliu}@ist.psu.edu. Fangfang Zhang and Sencun Zhu are with the Department of Computer Science and Engineering, Penn State University, University Park, PA 16802, USA, e-mail: {fuz104, szhu}@cse.psu.edu.

I. Introduction

Software plagiarism is an act of stealing others' software by illegally copying their code, applying code obfuscation techniques to make the code look different and then claiming that it is one's own program in a way violating the terms of original license. In recent years, software plagiarism has become a serious concern for honest software companies and open source communities. It violates the intellectual property of software developers and has been a severe problem, ranging from open source code reuse, software product stealing to smartphone application repackaging. The stolen code can be used by plagiarists to reduce the cost of their software development. The popular smartphone applications may be repackaged and injected with malicious payload to accelerate the propagation of malware. According to a recent study [1], it was found that 1083 (or 86.0%) of 1260 malicious app samples were repackaged versions of legitimate apps with malicious payloads. Moreover, the booming of software industry gives plagiarists more opportunities to steal others' code. The burst of open source projects (e.g., SourceForge.net has more than $430,000$ registered open source projects with 3.7 million developers and more than 4.8 million downloads a day [2]) provides plenty of easy targets for software thieves, since source code is easier to understand and modify than executable binaries. The existing automatic code obfuscation tools (e.g., Loco [3] and Obfuscator-LLVM [4]) can change the syntax of a program while preserving its semantics and therefore will help plagiarists to evade detection. Therefore, automated software plagiarism detection is greatly desired.

However, automated software 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 source code analysis. Besides, code obfuscation is also a huge obstacle to automatic software 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 the above two facts, there are two necessary requirements for a good software plagiarism detection scheme [5]: (R1) Capability to work on suspicious executables without the source code; (R2) Resiliency to code obfuscation techniques.

The existing approaches to software plagiarism detection can be divided into the following categories: (C1) static source

TABLE I
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			✓		✓	✓

This table presents some common obfuscation methods. In Section V, we will discuss more obfuscation attacks in detail, including binary packing and virtualization obfuscation.

code comparison methods [6], [7], [8]; (C2) static executable code comparison methods [9]; (C3) dynamic control flow based methods [10]; (C4) dynamic API based methods [11], [12]; (C5) dynamic value based approach [5], [13]. 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 I.

In this paper, we propose a new angle for software plagiarism detection. Our approach, called LoPD, is a deviation based program semantic equivalence checking method. Compared to existing approaches, LoPD does not require the source code of tested programs and is more resilient to automatic obfuscation techniques. Instead of directly measuring the similarity between two programs, LoPD is based on an opposite philosophy: *search for any dissimilarity between two programs*. As long as we can find one dissimilarity, the programs are semantically different; but if we cannot find any dissimilarity, it is likely a plagiarism case. Due to our design philosophy, LoPD is an ideal fit for the whole-program plagiarism detection. Furthermore, with a few engineering efforts, LoPD's formal program semantics-based method can be applied to partial software plagiarism detection as well.

More specifically, 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. The output states can be directly compared, but the comparison of computation paths is challenging. Our idea is to find *path deviation*, i.e., given two different inputs, one program will follow the same execution path, whereas the other 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 path deviation, we can claim the two programs in consideration are not semantically the same. We calculate the weakest precondition [14], [15] for each path condition and represent the path deviation conditions as formulas, which will be solved later by a constraint solver. Also, we perform another round of *path equivalence* checking to make sure that a path deviation is really a semantics deviation, not caused by code obfuscation. Since the symbolic formulas capture complete semantics and constraint of an execution path, LoPD can detect the semantics equivalence or difference of the execution paths.

We have developed the idea of LoPD on top of the BitBlaze platform [16], [17] performed the empirical study with a set of widely used applications and their obfuscated versions, including the advanced virtualization obfuscation. The experimental results show LoPD is effective in detecting whole-program plagiarism[1]. In addition, the library module reuse detection evaluation demonstrates LoPD's capability in identifying partial software plagiarism.

Scope of our paper: Currently, we focus on the detection of whole plagiarized PC programs that can be generated by semantics-preserving obfuscation tools. That is, LoPD will provide a Yes/No answer to the question: *are the suspicious program is semantically equivalent to the plaintiff program?* However, LoPD's formal program semantics-based method can be extended to detect partial software plagiarism as well. We will discuss the solution to partial software plagiarism in Section VI-C. The detection of smartphone app repackaging is also discussed in Section VII.

Contributions: (1) We present a novel logic-based software plagiarism detection approach—LoPD. LoPD relies on dynamic symbolic execution and theorem proving techniques to find dissimilarities between two programs in order to rule out semantically different programs. (2) LoPD is resilient to most of the known code obfuscation techniques that impede static analysis. (3) We have extended our preliminary conference paper [18] in several ways by discussing and evaluating the possible application to partial plagiarism detection, adding more testing cases of whole-program plagiarism detection, testing with the state-of-the-art obfuscators such as Obfuscator-LLVM [4] and Code Virtualizer [19], providing the related work with the latest publications, and updated the other related sections. In total, the new materials constitute over 30% of the manuscript.

The rest of the paper is organized as follows. We introduce the background and related work in Section II. Section III describes the basic idea of LoPD. Section IV presents the details of the system design. Section V discusses the resilience of LoPD to possible counterattacks. The implementation and the evaluation are presented in Section VI, followed by discussion in Section VII and conclusion in Section VIII.

II. Related Work

A. Software Plagiarism Detection

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

Static birthmark based software plagiarism detection: Liu et al. [6] proposed a program dependence graph (PDG) based approach, which is vulnerable to obfuscation techniques such as opaque predicates and loop unwinding. Myles et al. [9] statically analyzed executables and used K-gram techniques to measure the similarity. This approach is vulnerable to instruction reordering and junk instruction insertion. There

[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.

are also several work focusing on detecting code plagiarism of smartphone applications. DroidMOSS [1] adopted fuzzy hashing to detect application plagiarism. It can only tolerate small local changes in code. Simple obfuscation, such as noise injection, can evade the detection of DroidMOSS. DNADroid [20] proposed a data dependence graph based detection approach. The data dependence of a program is easy to change by inserting intermediate variable assignment instruction into the code. Juxtapp [21] proposed a code-reuse evaluation framework which leverages k-grams of opcode sequences and feature hashing. It is also vulnerable to noise injection. Huang et al. [22] proposed a framework for evaluating mobile app repackaging detection algorithms. ViewDroid [23] applied a user interface based birthmark, which is designed for user interaction intensive and event dominated programs, to detect smartphone application plagiarism. Luo et al. [24] proposed a repackage-proofing method to protect Android apps.

Most above static analysis methods to detect traditional software plagiarism either require the source code of the analyzed programs or cannot work on obfuscated binary code. This limits their practicability since the source code of a suspicious program is not always available. Furthermore, to hide the plagiarism intent, the suspicious program is typically obfuscated. The detection methods of smartphone application plagiarism are either easy to be bypassed by applying obfuscation techniques or not suitable for normal PC programs.

Dynamic birthmark based software plagiarism detection: Jhi et al. [5], [25] proposed to use core values as birthmark to detect software plagiarism. Zhang et al. [13] used a similar method for algorithm detection. This approach has no theoretical guarantee, since core value is hard to define. Lu et al. [26] presented a dynamic opcode n-gram birthmark, which is vulnerable to instruction reordering and irrelevant instructions insertion. Myles et al. [10] developed a whole program path (WPP) birthmark, which is robust to some control flow obfuscations such as opaque prediction, but is vulnerable to many semantics-preserving transformations such as loop unwinding. Tamada et al. [11] 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. Wang et al. [27], [28] introduced system call based birthmarks. Their approaches are not suitable for programs that invoke few system calls. The latest dynamic birthmark work, DYKIS [29], measures the similarity of key instructions (e.g., value-updating instructions and input-correlated instructions in the execution traces, achieving good resilience to obfuscation schemes.

In contrast, LoPD is based on formal logic that captures program semantics. This makes LoPD resilient to most obfuscation techniques currently known in literature. In addition, LoPD relies on weakest precondition to capture path semantics, and thus connects to both dynamic and static techniques. This unique combination for path deviation detection and path equivalence checking and results in high detection accuracy for nontrivial programs.

B. Clone Detection

Clone detection is a technique to find duplicate code to decrease code size and facilitate maintenance. Existing clone detection techniques include String-based [30], Tree-based [31], [32], Token-based [33], [7], [34] and PDG-based [35], [36], [37]. Sæbjørnsen et al. [38] proposed a tree-based clone detection in binary code. Their approach first extracted a set of features from so-called "code regions" and represented them with a feature vector, then applied locality-sensitive hashing to locate the nearest neighbor vectors. BinClone [39] improved Sæbjørnsen et al.'s approach by improved inexact clone detection and flexible normalization procedure. Since most clone detection techniques do not take code obfuscation into consideration, when being applied to detect software plagiarism, they can be easily evaded by attackers.

C. Semantic Differential Detection

There are some researches focusing on find semantic differences between two programs. Jackson et al. [40] tried to find the differences by comparing the input-output mapping. Symdiff [41] converted source code to intermediate verification language and then identified semantic differences. Person et al. [42] used incomplete symbolic summaries to compare two programs. UCKLEE [43] synthesized inputs to the two C functions and used bit-accurate symbolic execution to verify that these two functions produce semantically equivalent outputs. All the above approaches use static analysis on source code and do not consider code obfuscation. As a result, they are not suitable for plagiarism detection. Another line of work detected semantic differences between executables. To match the basic blocks with the same semantics, BinHunt [44] and its successor iBinHunt [45] performed symbolic execution in the scope of basic blocks and verified the equivalence of the input-output relationship formulas with theorem proving techniques. BinJuice [46] represented abstraction of semantics of basic blocks as "semantic juice" and matched malware variants with such semantic juice. Exposé [47] combined function-level syntactic heuristics with semantics detection. Luo et al. [48] detected software plagiarism by matching longest common subsequence of semantically equivalent basic blocks. However, these tools suffered from the "block-centric" limitation [46]; that is, they were insufficient to capture similarities or differences across basic blocks. In contrast, LoPD is trace-oriented and is therefore able to find similarities and differences beyond the scope of basic blocks to a great extent.

D. Path Deviation Detection

Brumley et al. [49] first proposed the path deviation idea and used it to find protocol errors in different implementations of the same specification. Their approach [49] first built symbolic formulas from execution traces and then solved the formulas created from two different implementations. DARWIN [50] applied similar ideas to compare a stable version and a modified version to identify program bugs exposed by a difference in control flow. We adopted their path deviation idea and applied it to a new context of software plagiarism detection. Brumley et al. [49] only compared the output of executions. DARWIN [50] compared two paths (trace alignment) only after it has identified paths generating different concrete output. 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.

E. Software Testing and Symbolic Execution

LoPD relies on automatical test input generation to explore more paths. There have been vast amount of work on test input generation, and therefore our approach benefits from the active research in this field. We rely on random input generation for the initial input seed. We then leverage symbolic execution [51] and automatic test case generation using systematical white-box exploration (also called, concolic testing, directed systematic path exploration, etc.) [52], [53], [54], [55], [56], [57] for the subsequent path deviation computation. Path constraints are collected and manipulated to cover different paths, and a constraint solver (e.g., STP [58] and Z3 [59]) 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.

III. Overview

A. 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 [3] and Obfuscator-LLVM [4]) 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 probably 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.

In this work we have two assumptions: (1) we have pre-knowledge 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.

B. Basic Idea

Our basic idea is to search for any difference between the plaintiff program and the suspicious program. If differences are found, these two programs are not semantically equivalent thus it is not a software plagiarism case; otherwise, it is likely a software plagiarism case.

At high level, three things characterize program behavior—input, output, and the computation used to achieve the input-output mapping. Hence, we aim to find inputs 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 software plagiarism via automated code obfuscation.

Given an input, the comparison between output states is relatively straightforward: since the plaintiff has the pre-knowledge of his own software, he can specify which output variables and states are semantics-relevant 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 [49]. 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,

TABLE II
THE TABULAR REPRESENTATION OF RELATIONS BETWEEN THE REALITY AND THE DETECTION RESULTS.

			Reality	
			a.Software Plagiarism	b.Not Software Plagiarism
Detection Result	Case I. Same Output	I.1. Path Deviation	FN	TN
		I.2. No Path Deviation	TP	FP
	Case II. Diff Output	N/A	-	TN

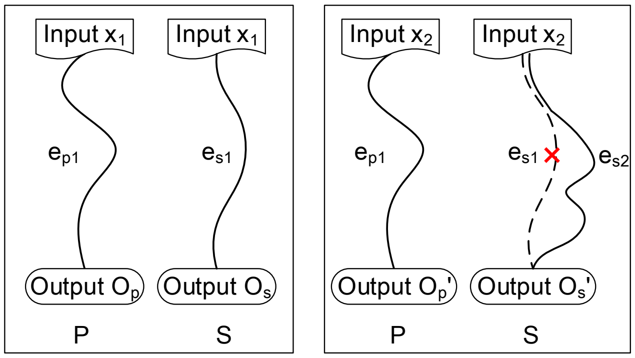


Fig. 1. Path deviation example.

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 may cause one program to execute the same path, while causing the other program to execute two different paths with these two inputs, we can rule out the 1:1 path correspondence case; that is, the suspicious program will not be considered as a plagiarized one. We call these 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. If $\exists x_1, x_2 \in I_P \cup I_S$, 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, such that $(e_{p1} = e_{p2} \wedge e_{s1} \neq e_{s2}) \vee (e_{p1} \neq e_{p2} \wedge e_{s1} = e_{s2})$, P and S have **path deviation**.

Fig. 1 illustrates this path deviation idea. Given the same input x_1, programs 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 software 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 software 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 Fig. 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. We leverage the techniques of logic-based execution path characterization including symbolic execution and constraint solving (e.g., STP [60], [58]) 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. Note that we ignore the limitations of current symbolic execution tools and constraint solvers during the analysis. The relations between the reality and the detection results are shown in Table II.

- **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 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.

As a result, 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. This path deviation based approach is more efficient than directly comparing two programs' execution paths, because the former can find semantically different paths within fewer iterations. In each iteration, the latter compares only one pair of execution paths, whereas LoPD not only compares such pair of paths but also can detect differences in other execution paths that share some parts with the current tested paths.

IV. Design

A. Architecture

The overview of the system design is shown in Fig. 2. We tackle the problem by three phases: Input Generation, Path Deviation Detection and Path Equivalence Checking. In the first phase, the *input generator* generates a test input. Then the *path deviation detector* checks whether there exists any path deviation between the plaintiff and suspicious programs. If there is a path deviation, the path equivalence checker decides whether the deviated path is a semantically equivalent path split from the original one, if yes, this is likely generated 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 case. If we cannot find a path deviation or the path deviation is caused by path-splitting, we repeat the iteration with a new input. This process is repeated until a true path deviation is found or the number of iterations reaches a threshold. If no true path deviation is found, LoPD concludes that this is a plagiarism case, because we believe it is impossible that two nontrivial independent programs have 1:1 path correspondence.

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

Algorithm 1 Path Deviation based Software Plagiarism Detection

```
Input: plaintiff program $p$, suspicious program $s$
Output: plagiarism / not plagiarism.
1:  for $i = 1$ to max_iteration do
2:      Generate an input $x$ by the input generator
3:      $p$, $s$ 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 $e_s$
4:      if $o_p = o_s$ then
5:          if the path deviation detector can find another devi-
            ation input $x'$ that produces different paths for $p$ and
            $s$ then
6:              the execution paths of $p$ and $s$ with input $x'$ are
                $e'_p$ and $e'_s$
7:              $d \Leftarrow p$ or $s$, the one executes different paths with
                $x, x'$
8:              The path equivalence checker checks the sematic
                equivalence of $e_d$ and $e'_d$
9:              if $e_d$ and $e'_d$ are semantically equivalent then
10:                 continue
11:             else
12:                 return ”not plagiarism”
13:             end if
14:         else
15:             continue
16:         end if
17:     else
18:         return “not plagiarism”
19:     end if
20: end for
21: return “plagiarism”
```

B. 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 may not provide high path coverage [53]. Therefore, we adopt automatic test case generation using systematical white-box exploration (also called, concolic testing and directed systematic path exploration) [52], [53], [54], [55], [56]. 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.

C. Path Deviation Detector

The path deviation detector is used to detect if two tested programs have path deviation. Generally speaking, given an input x, we are trying to find another input x' that causes one of the programs to 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 to find such x'. The design of the path deviation detector is shown in Fig. 3.

The symbolic executor performs a mixed concrete and symbolic execution [55], [16] 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

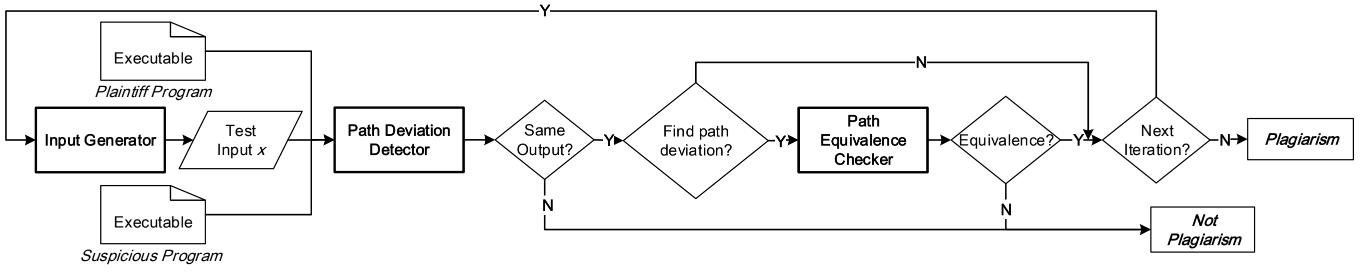


Fig. 2. LoPD system design.

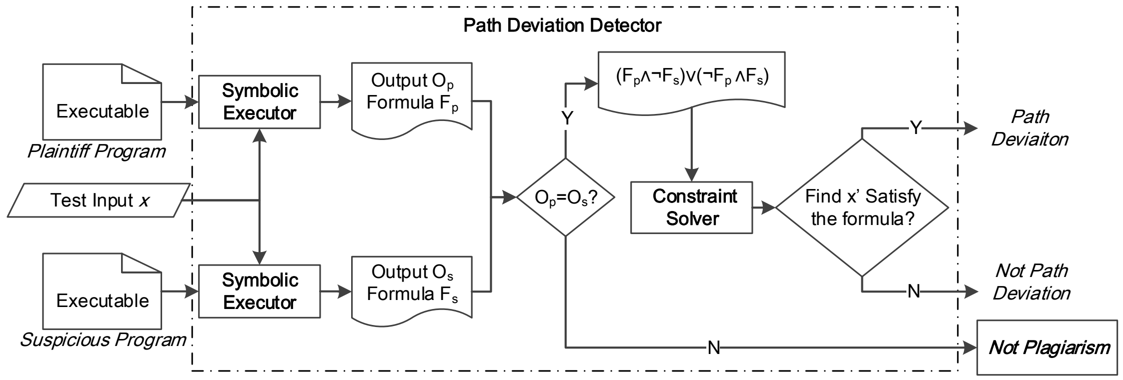


Fig. 3. Path deviation detector.

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 ($O = O_p = O_s$). Since weakest precondition based algorithm can generate more succinct symbolic formulas than forward symbolic execution [62], 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: $F_p^O(i)$ is true *iff* e_p is feasible on input i and ends with output O.

Specifically, let's assume the execution path e_s we collected contains a sequence of instructions $\langle i_1, i_2, ..., i_n\rangle$. Our weakest precondition (WP) calculation takes (e_s) as input, and the output state O_s as the postcondition (P). Inductively, we first calculate $wp(i_n, P) = P_{n-1}$, then $wp(i_{n-1}, P_{n-1}) = P_{n-2}$ and until $wp(i_1, P_1) = P_0$. The weakest precondition, denoted as $wp(e_s, P) = F_s^O(I) = P_0$, is a boolean formula over the inputs that follow the same execution path e_s and force the execution to reach the given point satisfying P. The calculation of $F_p^O(I)$ is similar.

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 (1), as used by Brumley et. al. [49], via a constraint solver STP [60], [58].

$$(F_p^O(I) \wedge \neg F_s^O(I)) \vee (\neg F_p^O(I) \wedge F_s^O(I)) \qquad (1)$$

If Formula (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 Fig. 1. That is, two programs behave differently on input x and x', unless paths e_{s1} and e'_{s2} are semantically equivalent. If Formula (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:

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

[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.

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. In case the constraint solver could not find a path deviation, we continue with white-box symbolic exploration [53], [56] to generate a new input for the next round. This process repeats until the new generated input 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 follow a different path.

D. 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 Fig. 4. The new test input x' is a satisfiable assignment of Formula (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 Fig. 1 as an example, $d = S$. In other words, in the path deviation detection step, if the first part of the disjunction of Formula (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 Fig. 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:

$$\begin{aligned} \text{Path Equivalent} \quad &\Leftrightarrow \quad (F_d \vee F'_d) \rightarrow (f_O = f'_O) \\ &\Leftrightarrow \quad (f_O = f'_O) \vee \neg(F_d \vee F'_d) \\ \text{Not Path Equivalent} \quad &\Leftrightarrow \quad \neg((f_O = f'_O) \vee \neg(F_d \vee F'_d)) \\ &\Leftrightarrow \quad (f_O \neq f'_O) \wedge (F_d \vee F'_d) \end{aligned} \tag{2}$$

We check the satisfiability of Formula (2) via a constraint solver STP [60], [58]. 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 (2) is $(n \neq n) \wedge (n > 0 \vee \neg(n > 0))$, which is not satisfiable. As a result, the two paths are equivalent.

V. COUNTERATTACK ANALYSIS

A benefit of LoPD logic based detection method is that it is resilient to most of the known attacks that obfuscate binary static analysis. We will discuss LoPD's resilience to the following obfuscation schemes in detail. Note that in the practical implementation, we have to take into consideration the limitations of symbolic execution on binary code and the capability of the constraint solving (see Section VII).

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.

Value splitting and aggregation: In a program, a value v is either initialized from some constant values or other variables. Without loss of generality, assume $v = f(u_1, ..., u_k)$, where u_i is a variable on which v depends. When v is split into two values v_1 and v_2, where $v_1 = f_1(u_1, ..., u_k)$ and $v_2 = f_2(u_1, ..., u_k)$. Assume $v = g(v_1, v_2)$, we have

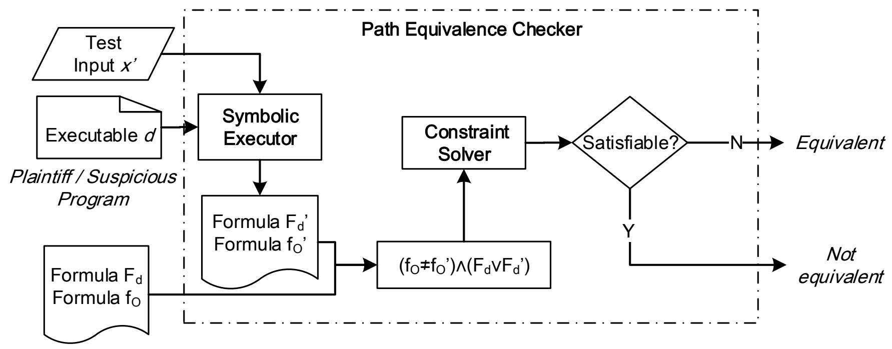

Path Equivalence Checker
Test Input x'
Executable d
Plaintiff / Suspicious Program
Formula F_d Formula f_O
Symbolic Executor
Formula F_d' Formula f_O'
$(f_O \neq f_O') \wedge (F_d \vee F_d')$
Constraint Solver
Satisfiable?
N
Y
Equivalent
Not equivalent

Fig. 4. Path equivalence checker.

$f = g(f_1, f_2)$. For simple value splitting obfuscation, usually constraint solvers are able to prove this relationship and thus the resulted formulas will be considered equivalently.

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.

Control flow flattening: It has little effect on LoPD due to two reasons: (1) LoPD identifies paths dynamically by emulating program execution with concrete inputs, so the paths inspected are valid feasible paths. (2) Since we obtained output formulas by symbolic execution, the semantics from flattened paths are captured faithfully.

Loop unwinding: Paths being considered by LoPD are dynamic execution traces, so loop unwinding has little effect since execution paths with loops are unwound anyway.

API implementation embedding: Assume extracting s-tatements $S_1, ..., S_N$ as API function F, during dynamic execution, these instructions will be executed exactly as the original order. Therefore, $wp(S_1, ...S_N, R) = wp(F, R)$.

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

Binary packing: Binary packing is a widely used method to obfuscate binary code and hinder static analysis [63]. The effect is the original code is stored as the compressed or encrypted data. When a packed binary starts running, the unpacking routine will first restore the original code. To address this issue, we rely on Temu's generic runtime unpacking tool, Renovo [64]. We activate our trace logging tool when the newly generated code is identified. Currently, Renovo can extract hidden code from most common binary packers, such as UPX, ASPack, ASProtect, FSG, WinUPack, and YodaProtector.

Virtualization obfuscation: The virtualization-based obfuscation is currently the state of the art in binary obfuscation [65]. The obfuscators such as Code Virtualizer [19] translates the binary code to a new kind of bytecode. At run time, an attached interpreter will simulate each bytecode through a decode-dispatch loop [66]. As a result, the collected path conditions will contain a large number of redundant conditions related to the decode-dispatch loop. The path formula explosion will impede the constraint solving. To remove these unnecessary conditions, we need extra efforts and fine-grained taint analysis. In Section VI-A, we will evaluate LoPD's resilience to virtualization obfuscation and discuss possible countermeasures.

VI. Implementation and Evaluation

We have implemented the idea of LoPD on top of BitBlaze [16], [17], a binary analysis platform. We concretely execute the tested programs in BitBlaze's whole-system emulator, Temu. Also, we leverage Temu's generic unpacking plug-in, Renovo [64], to start the trace logging after the binary unpacking. Our trace logging tool instruments each instructions executed to record the types of x86 opcode, values of operands and each instruction's address. All the information is formatted as raw data to limit the trace size. In general, not all of the instructions are of interest (e.g., the code generated by program start up and binary unpacking routine code), so LoPD can optionally record instructions which need to be further analyzed. We use Vine, the static analysis component, to lift x86 instructions to Vine IL, a RISC-like intermediate language with static single assignment (SSA) format. The symbolic execution is performed by interpreting each Vine IL instructions. LoPD's path equivalence checker is an extension of Vine's symbolic execution module. To solve the path deviation formula 1 and the path equivalence formula 2, we apply STP [60], [58] as the constraint solver. We integrate all the above components and implement an automatic software plagiarism detection system in C and Python.

Our evaluations consist of three case studies: software plagiarism case, different program case and partial plagiarism case. The evaluation is performed on a Linux machine with Intel Centrino duo 1.83GHz CPU and 2 GB RAM.

A. Case Study I: Same Programs

In this experiment, 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 10 tested programs: thttpd, mini_httpd, 7-Zip, gzip, exiv2, faad2, tar, scp, Ford-Fulkerson maximum flow implementation and tcc. The input variables of thttpd and mini_httpd are the

TABLE III
The tested programs and their running time per iteration for the same program case.

Name	Type	Execution Time (seconds)							
		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
7-Zip	File compression	12.68	48.53	28.39	12.96	43.73	30.46	18.72	107.47
gzip	File compression	4.89	13.87	2.53	5.07	14.83	3.69	7.02	30.36
exiv2	Image metadata manipulation	6.04	12.16	3.34	5.67	13.54	4.18	5.84	31.30
faad2	MPEG-4 Decoder	8.32	23.10	7.89	6.64	23.58	8.45	7.02	58.42
tar	File archive	1.89	6.86	4.36	2.24	6.78	4.28	2.52	22.83
scp	Secure copy	2.57	10.16	5.35	3.82	11.12	5.32	3.80	31.12
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 (1) extraction [6] STP slover [7] Formula (2) extraction

TABLE IV
Different optimization or obfuscation options.

Compiler/Obfuscator	Options
gcc/g++	-O0, -O1, -O2, -O3 and -Os
Loco	-freorder-blocks (reorder basic blocks) -funroll-loops (unroll loops) -finline-small-functions (inline small function)
Obfuscator-LLVM	-mllvm -sub (instructions substitution) -mllvm -bcf (opaque predicate) -mllvm -fla (control flow flattening)

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 7-Zip and gzip, the input is a text file and the output is the compressed file. Exiv2's input is a jpg image and output is the image's metadata information. Faad2 is an open source MPEG-4 decoder; tar creates an archive for multiple files and scp transfers the tar archive file to a remote machine. These programs are chosen for two reasons. The first reason is that they are representative and widely used. Most of them have been in production for decades. Therefore, they are more likely to be the target of software plagiarism. Second, the input to these programs can be represented as a byte stream, which is well suited for symbolic execution. Also, the choice of two HTTP servers and file compression tools is for the evaluation in Section VI-B: different programs with the same purpose. The tested programs and their average processing time per iteration are shown in Table III.

For each program, we first generate different semantics-preserving executable files by compiling the source code using gcc/g++ (with different optimization options: -O0, -O1, -O2, -O3 and -Os). Also, we obfuscate the tested programs by applying two obfuscators: Loco [3] and Obfuscator-LLVM [4]. Loco [3] is an obfuscation tool based on Diablo [67], a link-time optimizer. Diablo rewrites the binaries during link-time. Loco can obfuscate binaries by control flow flattening and opaque predicate. Obfuscator-LLVM is a fork of the LLVM compilation suite [68] to provide code obfuscation options, which are implemented as an LLVM pass. The detailed optimization or obfuscation options are shown in Table IV. 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 [69].

By switching different optimization or obfuscation options, we generate 11 different executables for each program in total. To help readers understand the binary differences, we present three examples under different optimization options and control flow obfuscation methods, respectively. Fig. 5 shows the disassembly code of traversing link list function under gcc optimization option O0 and O2. Basic block loc_8000005 and loc_8000008 are corresponding to the loop body of function `traverse`. However, they contain completely different instructions in syntax. Under the optimization of O2, `eax` stores the value of `head` before entering the loop body. The gcc O2 optimization does a dataflow analysis to cache the iteration variable (`head`) in `eax`, so that the generated code does not need to load and store `head` at each iteration. An opaque predicate is a predicate that always evaluate the same value. Fig. 6 illustrates an example of opaque predicate, in which predicate $(x^3 - x = 0 \pmod 3)$ holds true for all integers x and the false condition leads to junk code.

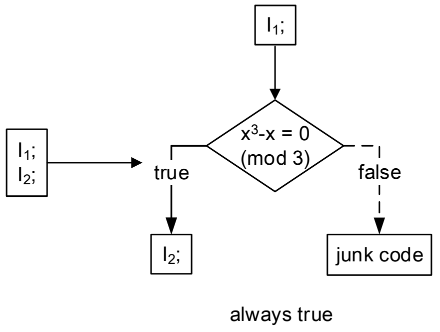
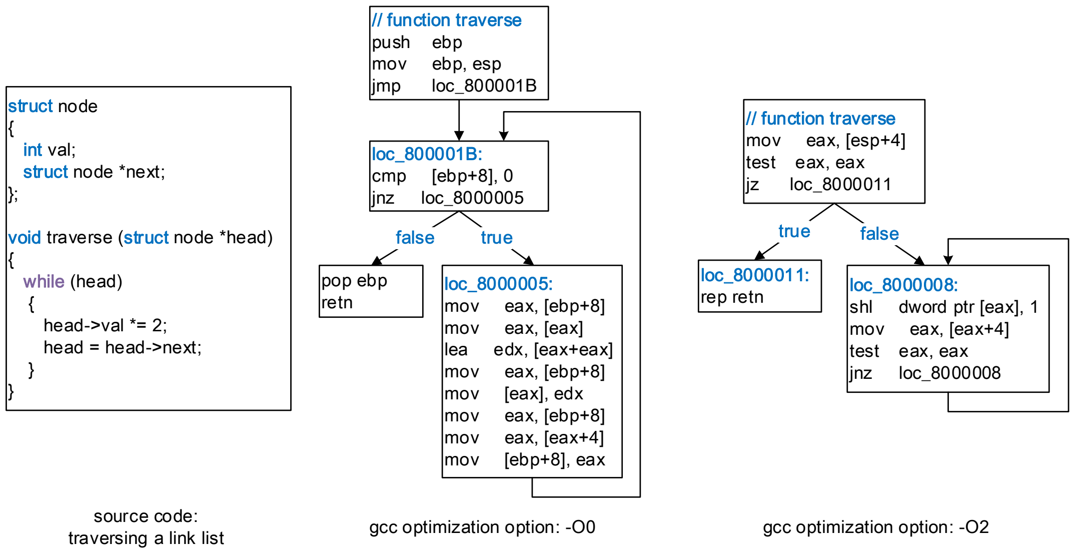


Fig. 6. Example: opaque predicate.

We use LoPD to perform pairwise comparison of the generated 11 executables for each program in Table III. Starting with an initial input seed, we automatically generate new inputs by negating the path formula and solve it with STP. We set the threshold of the maximum number of iterations to be 100. For all 550 tested pairs (55 comparison pairs for each of the 10 tested programs), LoPD does not find any true path deviation. That is, LoPD draws the right conclusion that they are software plagiarism cases. There is no false negative.



```
struct node
{
   int val;
   struct node *next;
};

void traverse (struct node *head)
{
   while (head)
   {
      head->val *= 2;
      head = head->next;
   }
}
```

```
// function traverse
push   ebp
mov    ebp, esp
jmp    loc_800001B

loc_800001B:
cmp    [ebp+8], 0
jnz    loc_8000005

pop ebp
retn

loc_8000005:
mov    eax, [ebp+8]
mov    eax, [eax]
lea    edx, [eax+eax]
mov    eax, [ebp+8]
mov    [eax], edx
mov    eax, [ebp+8]
mov    eax, [eax+4]
mov    [ebp+8], eax
```

```
// function traverse
mov    eax, [esp+4]
test   eax, eax
jz     loc_8000011

loc_8000011:
rep retn

loc_8000008:
shl    dword ptr [eax], 1
mov    eax, [eax+4]
test   eax, eax
jnz    loc_8000008
```

Fig. 5. Example: different compiler optimization options.

Virtualization obfuscation. The effect of virtualization obfuscation is that the original code path conditions mix with decode-dispatch loop logic conditions, which will greatly increase the size of path formulas. In this experiment, we evaluate LoPD with Code Virtualizer [19], a commercial software virtualization obfuscation tool. We test three different obfuscation levels, 10%, 20%, and 30%. Table V shows the experiment results. The "# PC" column shows the normalized path condition number. Our baseline is the clean version without virtualization obfuscation. The "SS " column displays the STP solving time (seconds) when performing path deviation detection. We set the limit of STP solving time as 600 seconds. As the performance of LoPD is linearly dependent on the size of trace formulas, it is evident that virtualization obfuscation can significantly delay and complicate LoPD's detection. In our test, when the obfuscation level exceeds certain degree, the STP solver will run out of time. On the other side, since one x86 instruction will be translated into several virtual bytecode that are emulated at run time, applying virtualization obfuscation on the whole program will introduce unacceptable time and space overheads. Typically, virtualization obfuscation is used to selectively protect the key component [70]. As shown in Table V, two cases fail at 10% obfuscation level and three test cases do not work when 30% code is obfuscated. Although the virtualization obfuscation may lead to the excessive resource usage of LoPD, we do not view it as a hard limit. The latest work on symbolic execution of obfuscated code has proposed using fine-grained bit-level taint analysis to mitigate this problem [71]. We plan to integrate the bit-level taint analysis to improve LoPD's robust against the virtualization obfuscation.

TABLE V
Normalized path condition number (times) and STP solving time (s) when comparing virtualization obfuscated program and the original clean version.

Program	10% obfuscated		20% obfuscated		30% obfuscated	
	# PC	SS	# PC	SS	# PC	SS
thttpd	8.7	28.1	22.6	240.5	52.2	∞
mini_httpd	7.5	18.3	16.3	108.3	35.6	326.6
7-Zip	11.3	55.2	32.0	∞	×	×
gzip	10.1	51.8	31.2	∞	×	×
exiv2	12.2	43.4	33.5	445.2	×	×
tar	7.1	31.5	21.2	201.5	47.9	∞
scp	9.0	36.6	23.4	324.6	×	×
Ford-Fulkerson	4.2	9.7	13.2	68.3	24.3	257.7

PC is short of path conditions and SS stands for STP solving time. × indicates that the tested program failed due to program crash. Both faad2 and tcc failed even at 10% obfuscation level. ∞ denotes that the STP solving time is beyond the time limit of 600 seconds.

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 III. Briefly, we find a code segment $s_1,\ s_2,\ \ldots\ s_n$, (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 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
    $obf(s_1;\ s_2;\ \ldots\ s_n;)$
end if
...
```

The execution time per iteration is also shown in Table III.

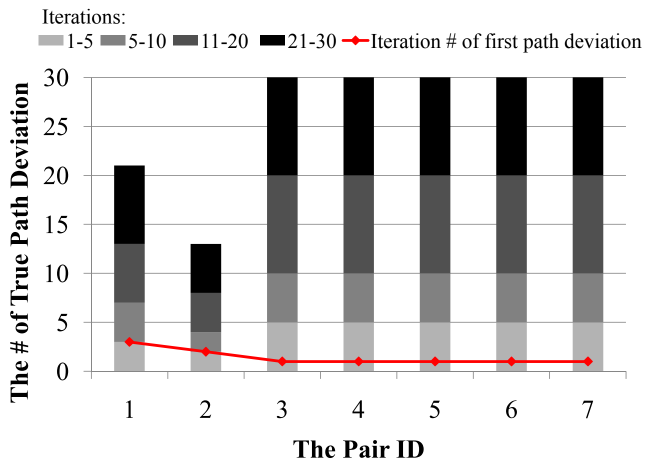


Fig. 7. The number of path deviations discovered within the first N iterations.

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 times in this line, because path equivalence checker is only needed when there is a path deviation. The total execution time of 100 iterations is within three hours, which is reasonable for offline detectors.

B. 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 VI are such program pairs.

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 Fig. 7. The x-axis are different program pairs, whose IDs are the same as in Table VI. 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 tiny HTTP servers developed by ACME Labs[3]. Both of them implement all the basic features of an HTTP server. If their settings are the same, thttpd and mini_httpd 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 Fig. 8. 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`". Fig. 9 presents another deviation example for thttpd and mini_httpd. The main difference comes from `HOST` field, which points to an executable file (/bin). In this case, thttpd rejects this dangerous request while mini_httpd accepts it.

[3]http://acme.com/software

The second program pair, 7-Zip and gzip, are two file compression tools. If given the particular parameters (e.g., no parameter for gzip and `a -tgzip` for the 7-Zip), 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 decompress correctly; LoPD generates a new input file x' based on x; both 7-Zip and gzip report a CRC-Failed upon x'. After that, gzip terminates without decompression, whereas 7-Zip 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 even if two programs always have the same input-output pairs, they can still be non-plagiarized different programs. Their computation steps are different, using different algorithms in this case. On one hand, Formula (1) guarantees that if two programs' execution paths have different path constraints, LoPD can detect the difference. On the other hand, it is very rare that different algorithms have the same path constraints for all execution paths. Therefore, LoPD can correctly detect them as non-plagiarism case. LoPD can find true path deviations in all 30 iterations, although they can always get the same output. For all examples in this part, within 3 iterations, LoPD draws the right conclusion that they are two different programs.

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 VI 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

TABLE VI
The tested programs and their running time per iteration for the different programs case.

ID	Program P	Program S	Execution Time (seconds)								
			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	7-Zip	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	7-Zip	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 (1) extraction [7] STP slover [8] Dynamic Running of d ($d = S$ OR P) on TEMU [9] Formula (2) extraction

Input x:

```
0x0000: 48 45 41 44 20 2F 69 6E 64 65 78 2E 68 74 6D 6C     HEAD /index.html
0x0010: 20 48 54 54 50 2F 31 2E 30 0A 0A 0A                   HTTP/1.0...
```

Input x':

```
0x0000: 48 45 41 44 20 2F 69 6E 64 65 78 2E 68 74 6D 6C     HEAD /index.html
0x0010: 20 48 01 01 10 FF FF 02 01 0A 0A 0A                  H........
```

Fig. 8. Path deviation example of thttpd *vs.* mini_httpd.

Candidate deviation input:

```
0x0000: 47 45 54 20 2F 69 6E 64 65 78 2E 68 74 6D 6C 20     GET /index.html
0x0010: 48 54 54 50 2F 31 2E 31 0D 0A 48 4F 53 54 3A 2F     HTTP/1.1..HOST:/
0x0020: 62 69 6E 0D 0A                                       bin..
```

Thttpd output:

```
HTTP/1.1 400 Bad Request
```

Mini_httpd output:

```
HTTP/1.1 200 OK
```

Fig. 9. Another deviation example for thttpd and mini_httpd.

results are shown in Fig. 7 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 image that both their path constraints and output states are different.

The execution time is shown in Table VI. 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.

C. Case Study III: Partial Plagiarism

In this experiment, we study the efficacy of LoPD against partial plagiarism, which steals code partially. Since it is quite challenging to identify the boundary of stolen code [28], in this paper, we focus on detecting the partial plagiarism case with a well-defined interface. To this end, we simulate the effect of partial software plagiarism by library function reuse. There are three major reasons behind our choice: 1) software plagiarism can be viewed as code reuse in disguise [6]; 2) in spite of multiple functionalities supplied by certain library, applications typically only reuse part of the library code; 3) the library code usually encapsulates each functionality as a module and offers a well-defined interface through standard function calls and returns. For example, to convert a `gif` image to a `png` image, we only need to call png write operation interface offered by `libpng`. We view our partial code reuse detection as the first step towards partial software plagiarism detection. As shown in Fig. 10, only part of plaintiff program code (labeled as “F1”) are reused by suspicious program. Note that unlike Fig. 1, the input passed to the “F1” part has been processed (x $\rightarrow$ x’), such as encoding or decoding, so that simply feeding the initial input (x) to the plaintiff program does not work. To this end, we leverage Temu to instrument the interface to the library code in the suspicious program and record the necessary parameters (e.g., context and environment variables) when the execution reaches the boundary of the library function code. Note that in practice, we do not need to enumerate all possible library functions. Many heuristics can help us quickly identify possible suspicious modules. For example, decryption, decompression, and hash functions reveal an excessive use of arithmetic and bitwise instructions [72]; the input and output formats to image transformation function may be different. Then we feed the same parameters to the plaintiff library code to force the execution to have the same starting point as the plaintiff program. We also capture the

TABLE VII
The tested programs and their running time per iteration for the same program case.

Program P	Program S	Execution Time (seconds)							
		IG [1]	PDD [2]			PEC [3]			Total
			DR [4]	FE [5]	SS [6]	DR [4]	FE [7]	SS [6]	
Libpng	png2html	5.37	13.42	3.26	3.49	13.85	3.06	3.15	34.46
Libpng	gif2png	4.55	12.30	3.24	2.16	11.69	2.98	2.82	30.37
Libpng	FishEye	5.09	14.84	3.59	3.68	15.74	4.21	3.98	38.34
Libtiff	tif2pdf	3.12	11.67	2.81	2.11	10.90	2.50	2.17	25.62
Libtiff	gif2tiff	2.51	12.58	2.85	2.50	12.59	2.66	2.43	25.28
Libtiff	thumbnail	2.86	12.35	2.52	2.46	14.04	2.59	2.28	29.32
Crypto++	TEA	5.77	23.64	10.31	9.13	24.17	11.64	8.59	45.60
Crypto++	XTEA	5.20	23.15	11.02	9.15	21.57	10.50	8.46	47.92
Crypto++	Blowfish	6.37	40.63	24.16	25.85	40.75	24.77	23.96	105.08
Crypto++	DES	7.20	30.42	23.70	26.28	31.79	25.56	27.84	97.24

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

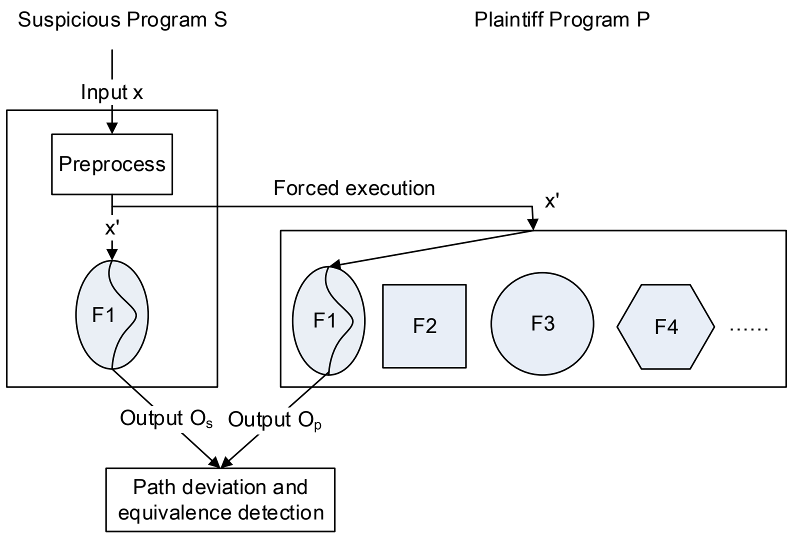


Fig. 10. Example: partial plagiarism.

output when the execution leaves the module (O_s and O_p in Fig. 10). After that, we follow the similar style with Fig. 1 to detect deviation.

We evaluate three widely used libraries: libpng (official png reference library), libtiff (tiff manipulation library and utilities), and Crypto++ (library of cryptographic schemes). For libpng and libtiff, we compare 3 different applications with each of them. Most of the applications are related to image format transformation (e.g., png2html, gif2png and gif2tiff), which only invokes part of the library code, such as `png_read_image` or `png_write_image`. For Crypto++, we test 4 block cipher encryption examples: TEA, XTEA, Blowfish and DES. The threshold of the maximum number of iterations are set to 100 and the reported evaluation data are similar to the previous experiments as well. The performance data are presented in Table VII and the total detection time ranges from 25.28 to 105.08 seconds. In summary, for all 10 library code partial reuse tests, LoPD is able to find no true path deviation within 100 iterations, which indicates a highly possible partial reuse case.

D. Summary

We evaluate the effectiveness and efficiency of LoPD with the same program, different program and library code reuse cases respectively in this section. The evaluation result demonstrates that LoPD can effectively and efficiently detect the whole program plagiarism and partial 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.

VII. Discussion and Future Work

In this section, we discuss the limitations and future work of LoPD.

First, 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, recent research [1], [73] found that 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. Our current partial plagiarism detection relies on the assumption of location of suspicious modules and well-defined interface (e.g., standard function calls and returns). Malicious authors could impede such assumptions by applying the obfuscation methods which violate calling-convention [63]. As a result, function boundary cannot be easily identified. In the future, we will continue to study the detection of partial plagiarism in the obfuscated binary code.

Second, LoPD is limited by the capability of the constraint solving and the limitation of current symbolic execution tools. 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 our tool can theoretically report false negative. In our experiments, we have not seen such false positives or false negatives. Besides, LoPD suffers from the common limitations of current symbolic execution tools, e.g. they cannot perform non-linear arithmetic operation or floating point calculation.

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 VI 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). Therefore, we believe setting the threshold at 100 is reasonable. We can also leverage the preknowledge of the plaintiff program to make the decision, 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 the future, in order to be resilient to manual modification on plaintiff programs, LoPD will 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 is not suitable for small programs, because when the program logic and semantics are too simple, it is possible that two programs have the one-to-one path correspondence (e.g., bubble sort and quick sort). 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.

Besides, we will extend our approach to detect smartphone app repackaging. Most current app repackaging detection methods focus on the detection scalability and cannot tolerate code obfuscation. Our approach will be a complementary solution that provides obfuscation-resilient detection. All the user interactions will be considered as input. The interactive information provided by the app, such as the display view and the message sent out through text message or the Internet, will be regarded as output. We are going to further investigate how to effectively generate test input and how to compare the output. We will also implement a new framework to perform symbolic execution of smartphone applications with dalvik virtual machine. Lastly, the complicated thread interleaving may have a negative effect on software dynamic birthmarks [74]. We leave the thread-aware software plagiarism detection as our future work.

VIII. Conclusion

In this paper, we propose LoPD, a deviation-based obfuscation-resilient program equivalence checking method, which is well suited for whole-program plagiarism detection. By leveraging dynamic symbolic execution to capture the semantics of execution paths, LoPD automatically builds symbolic formulas generated from two programs under investigation and solves the formulas to find deviations. A benefit of LoPD's formal program semantics-based method is that it is resilient to most of the known obfuscation attacks to static analysis. Our empirical study shows LoPD's efficacy in detecting whole-program plagiarism in obfuscated binaries. Furthermore, we demonstrate that LoPD can be extended to detect partial software plagiarism as well, with a few engineering efforts.

IX. Acknowledgments

We thank the anonymous reviewers for their informative feedback. This work was supported in part by the NSF Grants CCF-1320605 and CNS-1223710, and ONR Grants N00014-13-1-0175 and N00014-16-1-2265. P. Liu was also supported by ARO W911NF-09-1-0525, ARO W911NF-13-1-0421, NSF CNS-1422594, and NSF CNS-1505664.

References

[1] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the 2nd ACM conference on Data and Application Security and Privacy*, ser. CODASPY '12, 2012.

[2] [online]. Available: http://sourceforge.net/about March 2014.

[3] M. Madou, L. Van Put, and K. De Bosschere, "Loco: An interactive code (de)obfuscation tool," in *Proceedings of ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, 2006.

[4] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM - software protection for the masses," in *Proceedings of the 1st International Workshop on Software Protection (SPRO'15)*, 2015.

[5] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," in *33rd International Conference on Software Engineering (ICSE 2011), Software Engineering In Prictice (SEIP) track*, 2011.

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

[7] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003.

[8] W. Yang, "Identifying syntactic differences between two programs," *Softw. Pract. Exper.*, vol. 21, no. 7, pp. 739–755, Jun. 1991.

[9] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proceedings of the 2005 ACM symposium on Applied computing*, 2005.

[10] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," *Information Security*, vol. 3225/2004, 2004.

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

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

[13] F. Zhang, Y. Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012.

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

[15] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, Oct. 1969.

[16] D. Song, 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*, 2008.

[17] "BitBlaze: Binary analysis for computer security," [online]. Available: http://bitblaze.cs.berkeley.edu/.

[18] F. Zhang, D. Wu, P. Liu, and S. Zhu, "Program logic based software plagiarism detection," in *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering (ISSRE'14)*, 2014.

[19] Oreans Technologies, "Code Virtualizer: Total obfuscation against reverse engineering," http://www.oreans.com/codevirtualizer.php.

[20] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," *Computer Security–ESORICS 2012*, 2012.

[21] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "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*, 2012.

[22] H. Huang, S. Zhu, P. Liu, and D. Wu, "A framework for evaluating mobile app repackaging detection algorithms," in *Proceedings of the 6th International Conference on Trust & Trustworthy Computing (TRUST'13)*, 2013.

[23] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'14)*, 2014.

[24] L. Luo, Y. Fu, D. Wu, S. Zhu, and P. Liu, "Repackage-proofing Android Apps," in *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'16)*, 2016.

[25] Y.-C. Jhi, X. Jia, X. Wang, S. Zhu, P. Liu, and D. Wu, "Program characterization using runtime values and its application to software plagiarism detection," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 925–943, 2015.

[26] B. Lu, F. Liu, X. Ge, B. Liu, and X. Luo, "A software birthmark based on dynamic opcode n-gram," *International Conference on Semantic Computing*, 2007.

[27] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Detecting software theft via system call based birthmarks," in *Computer Security Applications Conference, 2009. ACSAC'09.*, 2009.

[28] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[29] Z. Tian, Q. Zheng, T. Liu, M. Fan, E. Zhuang, and Z. Yang, "Software plagiarism detection with birthmarks based on dynamic key instruction sequences," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1217–1235, Dec 2015.

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

[31] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98, 1998.

[32] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07, 2007.

[33] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, July 2002.

[34] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with jplag," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016–1038, 2002.

[35] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proceedings of the 8th International Symposium on Static Analysis*, ser. SAS '01, 2001.

[36] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08, 2008.

[37] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, ser. WCRE '01, 2001.

[38] A. Sæbjϕrnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009.

[39] M. Farhadi, B. Fung, P. Charland, and M. Debbabi, "BinClone: Detecting code clones in malware," in *Proceedings of the 8th IEEE International Conference on Software Security and Reliability (SERE'14)*, 2014.

[40] D. Jackson and D. A. Ladd, "Semantic diff: A tool for summarizing the effects of modifications," in *Software Maintenance, 1994. Proceedings., International Conference on*, 1994.

[41] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: A language-agnostic semantic diff tool for imperative programs," in *Computer Aided Verification*, 2012, pp. 712–717.

[42] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Psreanu, "Differential symbolic execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008.

[43] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*, 2011.

[44] D. Gao, M. Reiter, and D. Song, "BinHunt: Automatically finding semantic differences in binary programs," in *Poceedings of the 10th International Conference on Information and Communications Security (ICICS'08)*, 2008.

[45] J. Ming, M. Pan, and D. Gao, "iBinHunt: Binary hunting with interprocedural control flow," in *Proceedings of the 15th Annual International Conference on Information Security and Cryptology (ICISC'12)*, 2012.

[46] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW'13)*, 2013.

[47] B. H. Ng and A. Prakash, "Exposé: Discovering potential binary code re-use," in *Proceedings of the 37th IEEE Annual Computer Software and Applications Conference (COMPSAC'13)*, 2013.

[48] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*, 2014.

[49] D. Brumley, J. Caballero, Z. Liang, N. James, and D. Song, "Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation," in *Proceedings of 16th USENIX Security Symposium*, 2007.

[50] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, "Darwin: An approach to debugging evolving programs," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 3, p. 19, 2012.

[51] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, July 1976.

[52] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, 1990.

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

[54] K. Sen, D. Marinov, and G. Agha, "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*, ser. ESEC/FSE-13, 2005.

[55] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05, 2005.

[56] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," in *Proceedings of the 13th ACM conference on Computer and communications security (CCS '06)*, 2006.

[57] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, vol. 8, 2008, pp. 209–224.

[58] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification (CAV '07)*, 2007.

[59] L. D. Moura and N. Bjørner, "Z3: an efficient smt solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[60] "STP Constraint Solver," [online]. Available: http://sites.google.com/site/stpfastprover/STP-Fast-Prover.

[61] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, December 1990.

[62] D. Brumley, H. Wang, S. Jha, and D. Song, "Creating vulnerability signatures using weakest preconditions," in *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF'07)*, 2007.

[63] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys*, vol. 46, no. 1, 2013.

[64] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: A hidden code extractor for packed executables," in *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM'07)*, 2007.

[65] F. Guo, P. Ferrie, and T. Chiueh, "A study of the packer problem and its solutions," in *Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08)*, 2008.

[66] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Automatic reverse engineering of malware emulators," in *Proceedings of the 2009 IEEE Symposium on Security and Privacy*, 2009.

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

[68] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[69] "Optimize options - using the gun compiler collection (GCC)," [online]. Available: http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[70] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[71] B. Yadegari and S. Debray, "Symbolic execution of obfuscated code," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*.

[72] F. Gröbert, C. Willems, and T. Holz, "Automated identification of cryptographic primitives in binary programs," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID'11)*, 2011.

[73] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing smartphone applications: Attack strategies and defense techniques," in *Proceedings of the 4th International Conference on Engineering Secure Software and Systems (ESSoS'12)*, 2012.

[74] Z. Tian, Q. Zheng, T. Liu, M. Fan, X. Zhang, and Z. Yang, "Plagiarism detection for multithreaded software based on thread-aware software birthmarks," in *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*.

Jiang Ming is currently a Ph.D. candidate in the College of Information Sciences and Technology at the Pennsylvania State University. His research interests include program analysis and verification for security issues, intrusion detection, malware analysis and detection.

Fangfang Zhang received the M.S. degrees from Peking University in 2008, and the Ph.D. degree in computer science from the Pennsylvania State University in 2013. She is currently a data scientist at FireEye, Inc. Her research mainly focuses on software plagiarism detection, algorithm plagiarism detection and malicious JavaScript code obfuscation detection.

Dinghao Wu is an assistant professor in the College of Information Sciences and Technology at the Pennsylvania State University. He received his Ph.D. in Computer Science from Princeton University in 2005. He does research on software systems, including software security, software protection, software analysis and verification, information and software assurance, software engineering, and programming languages. His research has been funded by National Science Foundation (NSF), Office of Naval Research (ONR), and U.S. Department of Energy (DOE).

Peng Liu is a professor of information sciences and technology in the College of Information Sciences and Technology, Pennsylvania State University. He is the research director of the Penn State Center for Information Assurance and the director of the Cyber Security Laboratory. His research interests are in all areas of computer and network security. He has published a book and more than 200 refereed technical papers. His research has been sponsored by DARPA, US NSF, AFOSR, US DOE, US DHS, ARO, NSA, CISCO, HP, Japan JSPS, and Penn State.

Sencun Zhu is an associate professor in Penn State University. He received the B.S. degree in precision instruments from Tsinghua University, Beijing, China, the M.S. degree in signal processing from the University of Science and Technology of China, Graduate School at Beijing, Hefei, China, and the Ph.D. degree in information technology from George Mason University, Fairfax, VA, USA, in 1996, 1999, and 2004, respectively. His research interests include network and systems security and software security. His research has been funded by National Science Foundation, National Security Agency, and Army Research Office/Lab.