Program Logic Based Software Plagiarism Detection

Fangfang Zhang* Dinghao Wu†
*Department of Computer Science and Engineering
Penn State University
University Park, PA, USA
{fuz104, szhu}@cse.psu.edu

Peng Liu† Sencun Zhu*†
†College of Information Sciences and Technology
Penn State University
University Park, PA, USA
{dwu, pliu}@ist.psu.edu

Abstract—**Software plagiarism, an act of illegally copying others' code, has become a serious concern for honest software companies and the open source community. In this paper, we propose LoPD, a program logic based approach to software 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 likely a plagiarism case. We leverage symbolic execution and weakest precondition reasoning to capture the semantics of execution paths and to find path dissimilarities. LoPD is more resilient to current automatic obfuscation techniques, compared to the existing detection mechanisms. In addition, since LoPD is a formal program semantics-based method, it can provide a guarantee of resilience against many known obfuscation attacks. Our evaluation results indicate that LoPD is both effective and efficient in detecting software plagiarism.**

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], SandMark [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.

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			✓		✓	✓

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 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 novel software plagiarism detection approach, called LoPD, which does not need the source code of tested programs. In addition, it is more resilient to automatic obfuscation techniques, compared to existing approaches. 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.

Based on our design philosophy, LoPD tries to rule out dissimilar programs by finding an input that will cause these two programs to behave differently, either with different output states or with different computation paths. 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 leverage symbolic execution [14] and weakest precondition [15], [16] to systemically find such path deviations. Besides, we also develop a *path equivalence checker* to make sure that a path deviation is really a semantics deviation, not caused by code obfuscation.

Since the symbolic formula and weakest precondition can capture complete semantics and constraint of an execution path of the tested program, LoPD will detect the semantics equivalence or difference of the execution paths. Therefore, LoPD is resilient to most known automatic obfuscation techniques, as shown in Table I. In addition, we can provide an assurance of its resilience against above obfuscation attacks, as discussed in Section V. Moreover, LoPD provides theoretical guarantees of the high detection accuracy, subject to the limitations of the current symbolic execution tools and constraint solvers.[1]

Scope of our paper: We focus on the detection of plagiarized PC programs that are generated by semantics preserving obfuscation tools. That is, LoPD will provide a Yes/No answer to the question: *are two programs semantically equivalent?* We will discuss the solution to measure the similarity between two semantically different programs in Section VII. 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 symbolic execution and weakest precondition to find dissimilarities between two programs in order to rule out semantically different programs. (2) LoPD is resilient to most current code obfuscation techniques. In addition, LoPD can provide an assurance of resilience against obfuscation attacks in Table I. (3) LoPD theoretically guarantees high detection accuracy.

[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.

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 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 [17] 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 [18] proposed a code-reuse evaluation framework which leverages k-grams of opcode sequences and feature hashing. It is also vulnerable to noise injection. ViewDroid [19] applied a user interface based birthmark, which is designed for user interaction intensive and event dominated programs, to detect smartphone application plagiarism.

Most above static analysis methods to detect traditional software plagiarism require the source code of the analyzed programs. This limits their practicability since the source code of a suspicious program is not always available. 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], [13] proposed to use core values as birthmark to detect software plagiarism. This approach has no theoretical guarantee, since core value is hard to define. Lu et al. [20] 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. [21], [22] introduced system call based birthmarks. Their approaches are not suitable for programs that invoke few system calls.

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 leverages symbolic execution to obtain path constraints, but 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 [23], Tree-based [24], [25], Token-based [26], [7], [27] and PDG-based [28], [29], [30]. Sæbjørnsen et al. [31] proposed a tree-based clone detection in binary code. 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. [32] tried to find the differences by comparing the input-output mapping. Symdiff [33] converted source code to intermediate verification language and then identified semantic differences. Person et al. [34] used incomplete symbolic summaries to compare two programs. 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.

D. Path Deviation Detection

Brumley et al. [35] first proposed the path deviation idea and used it to find protocol errors in different implementations. DARWIN [36] applied similar ideas to identify program bugs. We adopted their path deviation idea and applied it to a new context of software plagiarism detection. Brumley et al. [35] only compares the output of executions. DARWIN [36] compares two paths 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

Our approach relies on test input generation. There are vast amount of researches on test input generation such as fuzz testing [37]. We rely on random input generation for the initial input seed. We then leverage symbolic execution [14] and automatic test case generation using systematically white-box exploration (also called, concolic testing, directed systematic path exploration, etc.) [38], [39], [40], [41], [42], [43] for the subsequent path deviation computation. Path constraints are collected and manipulated to cover different paths, and a constraint solver [44], [45] 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], SandMark [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 [35]. It is motivated by the fact that if one program is an automatic semantics equivalent transformation of another program, these two programs would have *one-to-one (1:1) path correspondence*, as defined in Definition 1. That is, given the same input, the execution of each program follows a certain path, respectively, and when given a different input, the programs should either both follow their original path or both execute new paths. Note that there is one exception: when an execution path of one program is split into two semantically equivalent paths for the obfuscation purpose, there would be no one-to-one path correspondence, but it is still a software plagiarism case. We will therefore also handle this semantically equivalent path splitting problem in our detection system.

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

If we can find two inputs which 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**.

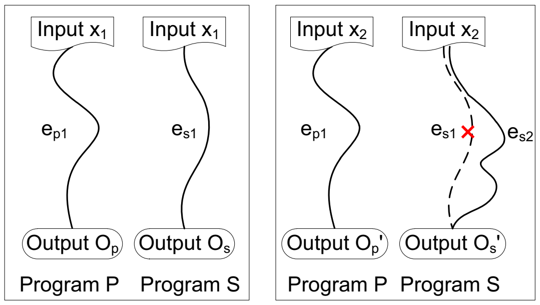

Fig. 1. Path deviation example

Figure 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 Figure 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, weakest precondition calculation and constraint solving (e.g., STP [44], [45]) 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

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

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 Figure 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.

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 input $x$ by 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 input
            $x'$ cause $P$ and $S$ path deviated. 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"
```

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

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 [37]. However, random input generation might not be desired. We adopt symbolic execution [14] and automatic test case generation using systematically white-box exploration

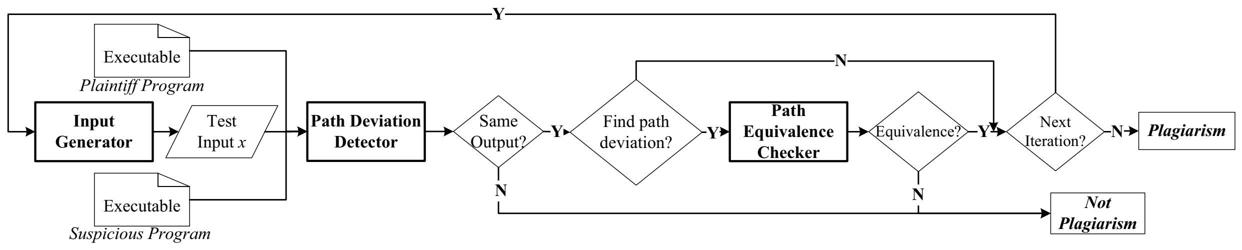

Fig. 2. LoPD system design

(also called, concolic testing and directed systematic path exploration) [38], [39], [40], [41], [42]. 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 program 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 and weakest precondition to find such x'. The design of path deviation detector is shown in Figure 3.

The symbolic executor performs a mixed concrete and symbolic execution [41], [46] for each tested program with x as input. In other words, the tested program is first concretely executed with the input x in the executor, which is a monitored environment with taint analysis. The input is the taint seed. The whole execution path is logged, including the executed instructions, the taint information and the output states.

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

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

$F_p^O(i)$ is true *iff* e_p is feasible on input i and ends with output O.

Given any input that satisfies the formula, the execution of the program will follow the original path, while given any input that does not satisfy the formula, the execution will follow a different path. As a result, to find a path deviation of plaintiff program and suspicious program, we need to find an input x', which makes the execution path of one program remain the same as its execution path with input x, and the execution path of the other program be different from its path with input x. As a result, we check the satisfiability of Formula (1), as used by Brumley et. al. [35], via a constraint solver STP [44], [45].

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

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 Figure 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:

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

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

[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.

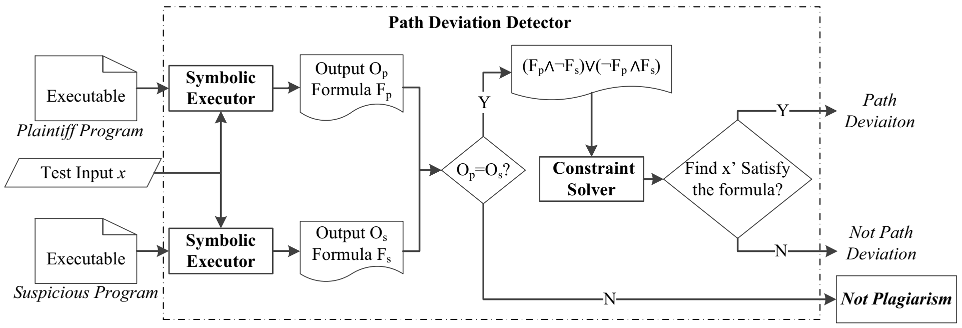

Fig. 3. Path deviation detector

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

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 Figure 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 Figure 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 Figure 2. Otherwise, we can conclude that P and S are not software plagiarism and call such path deviation as a *true path deviation*.

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

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

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

We check the satisfiability of Formula (2) via a constraint solver STP [44], [45]. 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

Since our logic based method captures path semantics by symbolic execution and weakest precondition, in theory it is resilient to most known obfuscation attacks, including but not limited to the following ones. In practical implementation, we need to take into consideration the limitations of symbolic execution, theorem proving and weakest precondition calculation.

Noise instruction/data injection: Suppose an irrelevant statement S_1 is inserted right after statement S_0. Given a postcondition R, the weakest precondition for the original program is $wp(S_0, R)$, while the weakest precondition for the

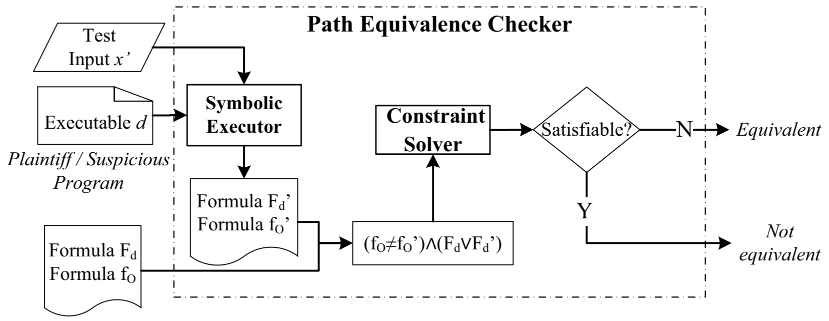

Fig. 4. Path equivalence checker

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 $f = g(f_1, f_2)$. 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 statements $S_1, ..., S_N$ 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.

In summary, LoPD provides guarantee of resilience against above obfuscation attacks.

VI. Implementation and Evaluation

We implement a prototype system. The symbolic executor is built atop Bitblaze infrastructure [46], [47]: we leverage their whole-system emulator, TEMU, to concretely execute the tested programs and record the whole execution path; we use vine, the static analysis component, to analyze the execution paths and extract their symbolic formulas. We apply STP [44], [45] as the constraint solver to solve path deviation Formula 1 and path equivalence Formula 2. STP is a decision procedure whose output indicates whether the formula is satisfiable or not. If so, it also provides an assignment to variables that satisfies the input formula. We integrate all the above components and implement an automatic software plagiarism detection system in C and Python.

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

A. Case Study I: Same Programs

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

For each program, we generate different semantics-preserving executable files by compiling the source code using gcc/g++ (with different optimization options: -O0, -O1, -O2, -O3 and -Os) and tcc. Besides, we apply Diablo, a link-time optimizer [48] and Loco [3], an obfuscation tool based on Diablo to generate two additional executables. Different

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
7za	File archiver	12.68	48.53	28.39	12.96	43.73	30.46	18.72	107.47
gzip	File archiver	4.89	13.87	2.53	5.07	14.83	3.69	7.02	30.36
Ford-Fulkerson	Maximum flow	1.62	6.11	7.18	1.52	5.78	9.27	3.71	18.43
tcc	C compiler	2.89	58.91	27.25	3.30	62.91	32.83	5.36	112.57

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

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 [49]. Diablo rewrites the binaries during link-time. Loco can obfuscate binaries by control flow flattening and opaque predicate. Hence, we have 8 different executables for each program.

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

Path splitting resilience check. In order to test the resilience of LoPD to semantically-equivalent-path splitting/ merging attacks, we manually add 2 to 3 such split paths in the source code of each program in Table III. Briefly, we find a code segment s_1, s_2, ... s_n, (s_i could be any type of statement, e.g., assignment, declaration, conditional branch, etc). We obfuscate this segment by independent statement reordering, variable splitting/merging, opaque predicate, etc. Then we add the `if...else` statement, where if c is true, the original segment will be executed; otherwise, the obfuscated segment will be executed. As demonstrated in the following example, the left part is the original code and the right part is the code after path splitting. We compile the new code into executable and compare it with one of the original executables by LoPD. LoPD finds no dissimilarity between the obfuscated and original executables within 100 iterations. It indicates the two programs are software plagiarism, as expected.

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

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

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

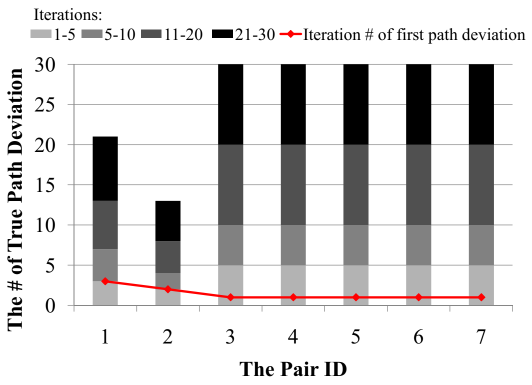

Fig. 5. The number of path deviations discovered within the first N iterations.

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 IV 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 Figure 5. The x-axis are different program pairs, whose IDs are the same as in Table IV. The bars indicate the count of true path deviations LoPD finds within N $(N = 5, 10, 20, 30)$ iterations. The red line shows the number of iterations when LoPD find the first true path deviation.

Thttpd and mini_httpd are two HTTP servers. If their settings are the same, both of them should give the same response when receiving the same request. The first path deviation happens in the $3rd$ iteration. We find total 21 true path deviations within 30 iterations. The deviations are caused because one of the programs does not follow the HTTP protocol specifications and has bugs in its implementation. A path deviation example is shown in Figure 6. When given request x, both of them normally response "`200 Ok`". Based

TABLE IV
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	7za	gzip	12.68	48.53	13.78	18.65	10.19	22.80	20.81	12.39	124.83
3	Ford-Fulkerson	Push-relabel	1.62	6.11	6.95	10.41	1.45	6.94	11.83	3.21	48.52
4	Ford-Fulkerson	Dijkstra shortest path	1.62	6.11	5.26	7.86	2.12	-	-	-	22.97
5	thttpd	gzip	1.08	6.21	13.87	7.27	1.32	-	-	-	29.75
6	tcc	gzip	2.89	58.91	13.87	17.49	5.12	-	-	-	98.28
7	Ford-Fulkerson	7za	1.62	6.11	48.53	20.90	13.21	-	-	-	90.37

[1] Input Generator [2] Path deviation detector [3] Path equivalence checker [4] Dynamic Running of P on TEMU [5] Dynamic Running of S on TEMU [6] Formula (1) extraction [7] STP slover [8] Dynamic Running of d ($d = S$ OR P) on TEMU [9] Formula (2) extraction

on x, LoPD finds another input x' that causes path deviation, where mini_httpd still returns "`200 Ok`", but thttpd returns "`400 Bad Request`".

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

Ford-Fulkerson and Push-relabel are two maximum flow implementations, using different algorithms. Given the same flow network, they should always calculate the same maximum flow. This is an example of the case that 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 IV are such program pairs. Since in most cases, two programs of such pairs cannot generate the same output regarding the same input, we can simply draw the conclusion that they are different programs by comparing the outputs. However, in order to evaluate how different the paths are in this case, we use LoPD to find path deviation regardless of their outputs. Similar to previous evaluation, we do not terminate the detection when we find a path deviation, although we have already gotten the right conclusion that they are different programs. LoPD continues until finishing 30 iterations.

The results are shown in Figure 5 with pair ID $4-7$. For each pair, LoPD can find true path deviation in all 30 iterations. The results are as expected, since two programs in each pair have different functionalities and it is not hard to image that both their path constraints and output states are different.

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

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

VII. DISCUSSION AND FUTURE WORK

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

First, LoPD is not suitable for small programs, because when the program logic and semantics are too simple, it is possible that two programs, such as bubble sort and quick sort, have the one-to-one path correspondence. However, for nontrivial software products, it is unlikely that two independent programs have such path correspondence. Therefore, in input x :

```
00000000: 4845 4144 202F 696E 6465 782E 6874 6D6C 2048 5454 502F 312E 300A 0A0A HEAD /index.html HTTP/1.0...
```

input x' :

```
00000000: 4845 4144 202F 696E 6465 782E 6874 6D6C 2048 0101 10FF FF02 010A 0A0A HEAD /index.html H........
```

Fig. 6. Path deviation example of THTTPD vs. mini_httpd.

practice, we do not need to concern about these potential false positive cases.

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 functions.

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 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], [50] 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. In the future, we will study the detection of partial plagiarism. One possible solution is to leverage some characteristics of the plaintiff program to provide a hint about the location of suspicious modules, such as invoking special system calls or APIs and then use LoPD to detect if two program segments have differences.

Lastly, 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.

VIII. Conclusion

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

IX. Acknowledgments

This research was supported in part by the NSF Grant CCF-1320605, NSF Grant CNS-1223710, ARO W911NF-09-1-0525 (MURI), and ARO W911NF-13-1-0421 (MURI).

References

[1] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, 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] C. Collberg, G. Myles, and A. Huntwork, "Sandmark–a tool for software protection research," *IEEE Security and Privacy*, vol. 1, 2003.

[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), the 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 *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 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, 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] ——, "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.

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

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

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

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

[18] 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.

[19] 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 2014).*, 2014.

[20] 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.

[21] 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.

[22] ——, "Behavior based software theft detection," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

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

[24] 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.

[25] 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.

[26] 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.

[27] L. Prechelt, G. Malpohl, and M. Phlippsen, "JPlag: Finding plagiarisms among a set of programs," Tech. Rep., 2000. [Online]. Available: http://page.mi.fu-berlin.de/prechelt/Biblio/jplagTR.pdf

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

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

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

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

[46] 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. Keynote invited paper.*, 2008.

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

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

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

[50] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing smartphone applications: Attack strategies and defense techniques," in *Engineering Secure Software and Systems, Lecture Notes in Computer Science*, 2012.