Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection

Lannan Luo, Jiang Ming, Dinghao Wu,* *Member, IEEE,* Peng Liu, *Member, IEEE,* and Sencun Zhu

Abstract—Existing code similarity comparison methods, whether source or binary code based, are mostly not resilient to obfuscations. Identifying similar or identical code fragments among programs is very important in some applications. For example, one application is to detect illegal code reuse. In the code theft cases, emerging obfuscation techniques have made automated detection increasingly difficult. Another application is to identify cryptographic algorithms which are widely employed by modern malware to circumvent detection, hide network communications, and protect payloads among other purposes. Due to diverse coding styles and high programming flexibility, different implementation of the same algorithm may appear very distinct, causing automatic detection to be very hard, let alone code obfuscations are sometimes applied. In this paper, we propose a binary-oriented, obfuscation-resilient binary code similarity comparison method based on a new concept, *longest common subsequence of semantically equivalent basic blocks*, which combines rigorous program semantics with longest common subsequence based fuzzy matching. We model the semantics of a basic block by a set of symbolic formulas representing the input-output relations of the block. This way, the semantic equivalence (and similarity) of two blocks can be checked by a theorem prover. We then model the semantic similarity of two paths using the longest common subsequence with basic blocks as elements. This novel combination has resulted in strong resiliency to code obfuscation. We have developed a prototype. The experimental results show that our method can be applied to software plagiarism and algorithm detection, and is effective and practical to analyze real-world software.

Index Terms—Software plagiarism detection, algorithm detection, binary code similarity comparison, obfuscation, symbolic execution, theorem proving.

1 Introduction

Identifying similar or identical code fragments among programs has many important applications. For example, one application is to detect illegal code reuse. With the rapid growth of open-source projects, software plagiarism has become a serious threat to maintaining a healthy and trustworthy environment in the software industry. In 2005 there was an intellectual property lawsuit filed by Compuware against IBM [19]. As a result, IBM paid $140 million in fines to license Compuware's software and an additional $260 million to purchase Compuware's services. In the case of software plagiarism, determining the sameness of two code fragments is faced with an increasing challenge caused by emerging, readily available code obfuscation techniques [17], [18], by which a software plagiarist transforms the stolen code in various ways to hide its appearance and logic, not to mention that often the plaintiff is not allowed to access the source code of the suspicious program.

Another application is to identify a given algorithm present in a binary program [33]. For instance, when an algorithm is protected by patent right, the owner of the algorithm needs to defend their proprietary status by determining the presence of this algorithm in other products [78]. For another example, cryptographic algorithms are widely employed by modern malware to circumvent detection, hide network communications, and protect payloads, among many other purposes [14], which need to be identified for further security investigation. However, due to diverse coding styles and high programming flexibility, different implementations of the same algorithms may appear very distinct (e.g., many implementations of cryptographic algorithms adopt loop unwinding for optimization and yet change the code syntax), causing automatic detection to be very hard, let alone code obfuscations are sometimes applied.

The basic research problem for code similarity measurement techniques is to detect whether a component in one program is similar to a component in another program and quantitatively measure their similarity. A component can be a set of functions or a whole program. Existing code similarity measurement methods include clone detection, binary similarity detection, and software plagiarism detection. While these approaches have been proven to be very useful, each of them has its shortcomings. Clone detection (e.g., MOSS [55]) assumes the availability of source code and minimal code obfuscation. Binary similarity detection (e.g., Bdiff [9]) is binary code-based, but it does not consider

* *Corresponding author. Email: dwu@ist.psu.edu*

- *L. Luo is with the College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802.*
- *J. Ming is with the College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802.*
- *D. Wu is with the College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802.*
- *P. Liu is with the College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802.*
- *S. Zhu is with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802.*

A preliminary version of this paper [48] appeared in the Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '14), *Hong Kong, China, November 16–22, 2014.*

obfuscation in general and hence is not obfuscation resilient. Software plagiarism detection approaches based on dynamic system call birthmarks [72] have also been proposed, but in practice, they incur false negatives when system calls are insufficient in number or when system call replacement obfuscation is applied [75]. Another approach based on core value analysis [39] requires the plaintiff and suspicious programs be fed with the same inputs, which is often infeasible. Consequently, most of the existing methods are not effective in the presence of obfuscation techniques.

In this paper, we propose a binary-oriented, obfuscation-resilient method named *CoP*. CoP is based on a new concept, *longest common subsequence of semantically equivalent basic blocks*, which combines rigorous program semantics with longest common subsequence based fuzzy matching. Specifically, we model program semantics at three different levels: basic block, path, and whole program. To model the semantics of a basic block, we adopt the symbolic execution technique to obtain a set of symbolic formulas that represent the input-output relations of the basic block in consideration. To compare the similarity or equivalence of two basic blocks, we check via a theorem prover the pair-wise equivalence of the symbolic formulas representing the output variables, or registers and memory cells. We then calculate the percentage of the output variables of the plaintiff block that have a semantically equivalent counterpart in the suspicious block. We set a threshold for this percentage to allow some noises to be injected into the suspicious block. At the path level, we utilize the Longest Common Subsequence (LCS) algorithm to compare the semantic similarity of two paths, one from the plaintiff and the other from the suspicious, constructed based on the LCS dynamic programming algorithm, with basic blocks as the sequence elements. By trying more than one path, we use the path similarity scores from LCS collectively to model program semantic similarity. Note that LCS is different from the longest common substring. Because LCS allows skipping non-matching nodes, it naturally tolerates noises inserted by obfuscation techniques. *This novel combination of rigorous program semantics with longest common subsequence based fuzzy matching results in strong resiliency to obfuscation.*

We have developed a prototype of CoP using the above method. We evaluated CoP with several different experiments to measure its obfuscation resiliency, precision, and scalability. Benchmark programs, ranging from small to large real-world production software, were applied with different code obfuscation techniques and semantics-preserving transformations, including different compilers and compiler optimization levels. We also compared our results with four state-of-the-art detection systems, MOSS [55], JPLag [61], Bdiff [9] and DarunGrim2 [24], where MOSS and JPLag are source code based, and Bdiff and DarunGrim2 are binary code based. Our experimental results show that CoP has stronger resiliency to the latest code obfuscation techniques as well as other semantics-preserving transformations; it can be applied to software plagiarism and algorithm detection, and is effective and practical to analyze real-world software.

In summary, we make the following contributions.

- We propose CoP, a binary-oriented, obfuscation-resilient binary code similarity comparison method, which can be applied to software plagiarism and algorithm detection.
- We propose a novel combination of rigorous program semantics with the flexible longest common subsequence resulting in strong resiliency to code obfuscation. We call this new concept the *Longest Common Subsequence of Semantically Equivalent Basic Blocks*.
- Our basic block semantic similarity comparison is new in the sense that it can tolerate certain noise injection or obfuscation, which is in sharp contrast to the rigorous verification condition or weakest precondition equivalence that does not permit any errors.

The rest of the paper is organized as follows. Section 2 presents an overview of our method and the system architecture. Section 3 introduces our basic block semantic similarity and equivalence comparison method. Section 4 presents how we explore paths in the plaintiff and suspicious programs and calculate the LCS scores between corresponding paths. We introduce the function and program similarity comparison method in Section 5. The implementation is presented in Section 6. Section 7 and Section 8 present the application to software plagiarism detection and algorithm detection, respectively. We analyze the obfuscation resiliency and discuss the limitations in Section 9. The related work is discussed in Section 10 and the conclusion follows in Section 11.

2 Overview

2.1 Methodology

Given a plaintiff program (or component) and a suspicious program, we are interested in detecting components in the suspicious program that are similar to the plaintiff with respect to program behavior. Program behavior can be modeled at different levels using different methods. For example, one can model program behavior as program syntax [55]. Obviously, if two programs have identical or similar syntax, they behave similarly, but not vice verse. As program syntax can be easily made different with semantics preserved, this syntax-based modeling is not robust in the presence of code obfuscation. Another example to model program behavior uses system call dependency graphs and then measure program behavior similarity with subgraph isomorphism [72]. This method is also not very robust against obfuscations as an adversary can replace system calls.

Instead, we propose to use formal program semantics to capture program behavior. If two programs have the same semantics, they behave similarly. However, formal semantics is rigorous, represented as formulas or judgements in formal logic, with little room to accommodate *similarity* instead of the *equivalence* relation. If two programs, or components, have the same formal semantics, their logical representations are equivalent. If they are similar in terms of behavior, their formal semantics in a logical representation may be nonequivalent. That is, it is hard to judge similarity of two logical representations of program semantics.

To address this problem, we combine two techniques and model code semantics at four different levels: basic block, path, function, and program. The first technique is to model semantics formally at the binary code basic block level. We

not only model basic block semantic equivalence, but also model similarity semantically. The second one is to model semantics at the path level. Based on the basic block semantic equivalence or similarity, we compute the *longest common subsequence of semantically equivalent basic blocks* between two paths, one from the plaintiff (component) and the other from the suspicious.[1] The length of this common subsequence is then compared to the length of the plaintiff path. The ratio calculated indicates the semantic similarity of the plaintiff path as embedded in the suspicious path. Note that the common subsequence is not compared to the suspicious path since noise could be easily injected by an adversary. By trying more than one path, we can collectively calculate a similarity score that indicates the semantics from the plaintiff (component) embedded in the suspicious, potentially with code obfuscation or other semantics preserving program transformations applied.

In other words, the program semantics is modeled collectively as path semantics based on basic block semantics, and we compute a ratio of path semantic similarity between the plaintiff (component) and the suspicious. Note that we are actually not intended to discover what the semantics are of the plaintiff and suspicious programs, but rather to use the semantics to measure the basic block and path similarity, and thus report a similarity score indicating the likelihood that the plaintiff code is reused, with obfuscation or not, legally or illegally.

2.2 Architecture

The architecture of CoP is shown in Fig. 1. The inputs are the binary code of the plaintiff (component) and suspicious program. CoP consists of the following four components: *front-end*, *basic block similarity comparison*, *path similarity comparison*, as well as *function and program similarity comparison*.

To begin, the front-end disassembles the binary code, builds an intermediate representation, and constructs control-flow graphs and call graphs. The path similarity comparison component computes the longest common subsequence (LCS) of semantically equivalent basic blocks (SEBB) between two paths (one from the plaintiff and another from the suspicious). It explores multiple path pairs to collectively calculate a similarity score, indicating the likelihood of the plaintiff code being reused in the suspicious program. To compute the LCS of SEBB of two given paths, we must be able to compute the semantic equivalence of two basic blocks. The basic block similarity computation component in Fig. 1 is for this purpose. It relies on symbolic execution to get symbolic formulas representing the input-output relation of a basic block. Specifically, it computes a symbolic function for each output variable (a register or memory cell) based on the input variables (registers and memory cells). As a result, a basic block is represented as a set of symbolic formulas. The semantic equivalence of two basic blocks are then checked by a theorem prover on their corresponding sets of symbolic formulas. Since obfuscations or noise injection can cause deviations on semantics leading to nonequivalent formulas, we accommodate small deviations by collectively checking whether an output formula in one basic block has an equivalent one in the other, with possible permutations of input variables. We then set a threshold to indicate how semantically similar two basic blocks are. When the score is above the threshold, we regard them as "equivalent" during the LCS calculation. The details of basic block semantic similarity and equivalence computation are presented in the next section.

1. Here we refer semantically "equivalent" basic blocks to the blocks that are semantically similar with a score above a threshold which will be presented in the next section.

3 Block Similarity Comparison

Given two basic blocks, we want to find how semantically similar they are. We do not compute their strict semantic equivalence since noise can be injected by an adversary.

3.1 Strictly Semantic Equivalence

Here we describe how to compute the strictly semantic equivalence of two basic blocks. Take the following code snippet as an example:

```
p = a+b;  |  s = x+y;
q = a-b;  |  t = x-y;
```

For the sake of presentation, we do not use assembly code. At the binary code level, these variables are represented as registers and memory cells. These two code segments are semantically equivalent. The only difference is the variable names. At binary code level, different registers can be used.

Via symbolic execution, we get two symbolic formulas representing the input-output relations of the left "basic block."

$$p = f_1(a, b) = a + b$$
$$q = f_2(a, b) = a - b$$

Similarly, for the right "basic block" we have

$$s = f_3(x, y) = x + y$$
$$t = f_4(x, y) = x - y$$

We then check their equivalence by pair-wise comparison via a theorem prover and find that

$$a = x \wedge b = y \implies p = s$$

and similarly for q and t.

Strictly semantic equivalence checking asserts that there are equal number of input and output variables of two code segments and that there is a permutation of input and output variables that makes all the output formulas equivalent pair-wise between two segments. That is, when one of the following formulas is true, the two code segments are regarded as semantically equivalent.

$$a = x \wedge b = y \implies p = s \wedge q = t$$
$$a = x \wedge b = y \implies p = t \wedge q = s$$
$$a = y \wedge b = x \implies p = s \wedge q = t$$
$$a = y \wedge b = x \implies p = t \wedge q = s$$

A similar method is used in BinHunt [30] to find code similarity between two revisions of the same program. This can handle some semantics-preserving transformations. For example, the following code segment can be detected as semantically equivalent to the above ones.

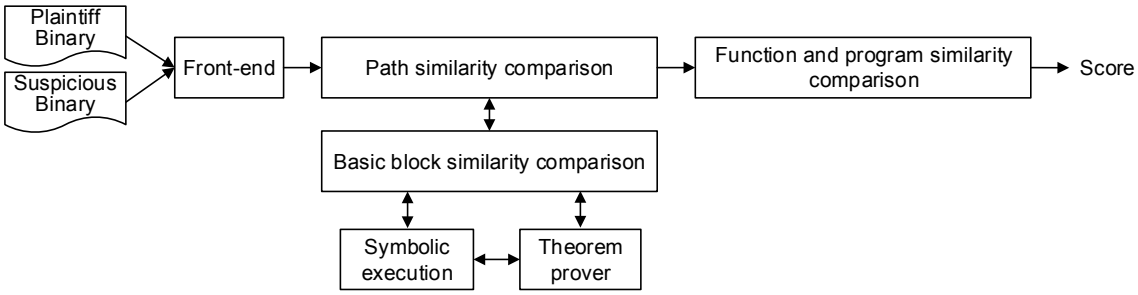

Fig. 1. Architecture

```
s = x+10;
t = y-10;
s = s+t;
t = x-y;
```

However, this method is not effective in general when code obfuscation can be applied, for example, noise can be easily injected to make two code segments not strictly semantic equivalent.

3.2 Semantic Similarity

Instead of the above black-or-white method, we try to accommodate noise, but still detect semantic equivalence of basic blocks. Consider the following block of code.

```
u = x+10;
v = y-10;
s = u+v;
t = x-y;
r = x+1;
```

Two temporary variables `u` and `v`, and a noise output variable `r` are injected. Strictly checking semantic equivalence will fail to detect its equivalence to the other block.

Instead, we check *each* output variable of the *plaintiff* block independently to find whether it has an equivalent counterpart in the suspicious block. In this way, we get

$$a = x \wedge b = y \implies p = s$$
$$a = x \wedge b = y \implies q = t$$

which are valid, and conclude a similarity score of 100% since there are only two output variables in the plaintiff block and both of them are asserted to be equivalent to some output variables in the suspicious block.

3.3 Formalization

Since we do not know in general which input variable in one block corresponds to which input variable in the other block, we need to try different combinations of input variables. We define a pair-wise equivalence formula for the input variable combinations as follows.

Definition 1. (Pair-wise Equivalence Formulas of Input Variables) Given two lists of variables: $X = [x_0, \ldots, x_n]$, and $Y = [y_0, \ldots, y_m]$, $n \leq m$. Let $\pi(Y)$ be a permutation of the variables in Y. A pair-wise equivalence formula on X and Y is defined as

$$p(X, \pi(Y)) = \bigwedge_{i=0}^{n} (X_i = \pi_i(Y))$$

where X_i and $\pi_i(Y)$ are the ith variables in X and the permutation $\pi(Y)$, respectively.

For each output variable in the plaintiff block, we check whether there exists an equivalent output variable in the suspicious block with some combination of input variables pair-wise equivalence.

Definition 2. (Output Equivalence) Given two basic blocks, let X_1 and X_2 be the lists of inputs and Y_1 and Y_2 be the lists of output variables, respectively; if $|X_1| \leq |X_2|$. Assume the first block is the plaintiff block and the second the suspicious block. Formally, we check

$$\forall y_1 \in Y_1.\ \exists y_2 \in Y_2, p(X_1, \pi(X_2)).$$
$$p(X_1, \pi(X_2)) \implies y_1 = y_2.$$

Each output equivalence formula is checked by a theorem prover.[2] Based on whether there is an equivalent output variable in the suspicious block for each plaintiff output variable, we compute a semantic similarity score that indicates how much semantics of the plaintiff block has been manifested in the suspicious block.

3.4 Basic Block Similarity Score

Definition 3 describes how to compute a basic block similarity score.

Definition 3. (Basic Block Similarity Score) Given a plaintiff bock B_1 and a suspicious block B_2, let n and m be the number of output variables of B_1 and B_2, respectively. Assume there are k output variables in B_1 that have semantically equivalent counterparts in B_2; then the basic block similarity score between B_1 and B_2 is defined as

$$\psi(B_1, B_2) = \frac{k}{n}.$$

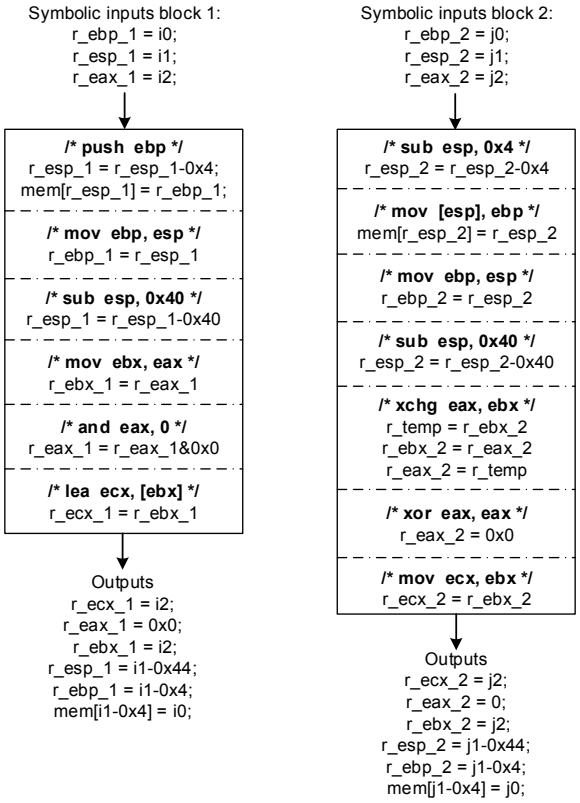

Fig. 2. Basic block symbolic execution

3.5 Example

Our tool works on binary code. Fig. 2 shows the binary code and its symbolic execution of two semantically equivalent basic blocks: the left one is the original basic block; the right one is the corresponding obfuscated basic block. The assembly instructions are in bold font.

Due to the noise from syntax differences caused by code obfuscation, most state-of-the-art binary diffing tools, such as DarunGrim2 [24] and Bdiff [9], are unable to identify whether or not the two basic blocks are semantically equivalent. Based on our basic block comparison method, CoP is able to detect that the semantics of the original block has been mostly embedded in the obfuscated block. In addition, it identifies different instructions that have the same semantics. For example, *and eax, 0* is semantically the same as *xor eax, eax*, and *lea ecx, [ebx]* is semantically the same as *mov ecx, ebx*.

4 Path Similarity Comparison

Based on the basic block semantic equivalence checking (i.e., similarity score above a threshold), we calculate a path embedding score for each linearly independent path [52] of the plaintiff against the suspicious program using the longest common subsequence of semantically equivalent basic blocks. How to find the linearly independent paths is presented in Section 5.2.

4.1 Longest Common Subsequence of Semantically Equivalent Basic Blocks

The longest common subsequence between two sequences can be computed with dynamic programming algorithms [21]. However, we need to find the highest LCS score between a plaintiff linearly independent path and many paths from the suspicious. Our *longest common subsequence of semantically equivalent basic blocks* computation is essentially the longest path problem, with the incremental LCS scores as the edge weights. The longest path problem is NP-complete. As we have removed all back edges in order to only consider linearly independent paths, the plaintiff and suspicious programs are represented as directed acyclic graphs (DAGs). In such case, the longest path problem of a DAG G can be converted to the shortest path problem of $-G$, derived from G by changing every weight to its negation. Since the resulted weight graphs contain "negative" weights, the Dijkstra's shortest path algorithm is not applicable, but still the problem is tractable as other shortest path algorithms such as Bellman-Ford [8] can be applied.

Instead, we adopt breadth-first search, with interactive deepening, combined with the LCS dynamic programming to compute the highest score of longest common subsequence of semantically equivalent basic blocks. For each step in the breath-first dynamic programming algorithm, the LCS is kept as the "longest path" computed so far for a basic block in the plaintiff program.

4.2 Algoroithm

Algorithm 1 shows the pseudo-code for the Longest Common Subsequence of Semantically Equivalent Basic Blocks computation. The inputs are a linearly independent path ρ of the plaintiff program, the suspicious program G, and a starting point s (presented in Section 5.1) of the suspicious program. Our algorithm uses the breadth-first dynamic programming LCS to explore the suspicious program. The intermediate LCS scores are stored in a memoization table δ. An intermediate LCS score of a node n is the length of the longest common subsequence of semantically equivalent basic blocks between ρ and the suspicious path segment beginning at s and ending at the node n. An index r points to the current row of the memoization table. The table δ is different from the conventional dynamic programming memoization table in that δ is a dynamic table. Each time, we encounter a new node, or a node with higher LCS scores, a new row is created in the table. The table γ is used to store the directions of the computed LCS [21, p. 395, Fig. 15.8]. The vector σ is used to store the intermediate highest LCS scores for each node.

The inputs of the function LCS() are a node μ of the suspicious program and the linearly independent path ρ. Function LCS() calculates the LCS of two paths, where the first path is denoted by a node in the suspicious program. The LCS path computed so far at its parent node (current node in the algorithm) is augmented with this node to form the suspicious path. The function SEBB() tells whether

2. Assume there are X_1 and X_2 input variables, and Y_1 and Y_2 output variables in the plaintiff and suspicious basic blocks, respectively. Let k and l be the smaller and bigger of X_1 and X_2, and s and t be the smaller and bigger of Y_1 and Y_2. respectively. There are $k!P_l^k$ number of pair-wise equivalence formulas as defined in Definition 1. For the output variables, there are Y_1Y_2 number of output equivalence formulas as defined in Definition 2. Thus, the total number of formulas is $Y_1Y_2k!P_l^k$. The computational complexity is quite high, but in practice there are usually small numbers of input and output variables. For cases there are large number of variables, we simple time out with negative answer.

Algorithm 1 Longest Common Subsequence of Semantically Equivalent Basic Blocks

δ: the LCS dynamic programming memoization table
r: the current row of the δ table
γ: the direction table for the LCS search
σ: the array stores the intermediate LCS scores
n: the length of the plaintiff path ρ

```
1: function PathSimilarityComparison(ρ,G,s)
2:     enq(s,Q) // Insert s into queue Q
3:     Initialize the LCS table δ
4:     Initialize the σ array to all zero
5:     r ← 0 // set the current row of table δ
6:     while Q is not empty do
7:         currNode ← deq(Q)
8:         for each neighbor μ of currNode do
9:             LCS(μ,ρ)
10:        end for
11:    end while
12:    ħ = max_{i=0}^{r}(δ(i,n)) // get the the highest score
13:    if ħ > θ then // higher than the threshold
14:        RefineLCS()
15:        ħ = max_{i=0}^{r}(δ(i,n))
16:    end if
17:    return ħ
18: end function

19: function LCS(μ,ρ)
20:    δ(μ,0) = 0
21:    for each node ν of ρ do
22:        if SEBB(μ,ν) then // semantically eq. blocks
23:            δ(μ,ν) = δ(parent(u), parent(ν)) + 1
24:            γ(μ,ν) = ↖
25:            if σ(μ) < δ(r,ν) then
26:                r++
27:            end if
28:        else
29:            δ(μ,ν) = max(δ(parent(μ),ν), δ(μ,parent(ν)))
30:            γ(μ,ν) = ← or ↑
31:        end if
32:        if σ(μ) < δ(r,ν) then
33:            σ(μ) = δ(r,ν)
34:            enq(μ,Q)
35:        end if
36:    end for
37: end function
```

two basic blocks are semantically equivalent or not. The RefineLCS() function refines the computed LCS so far by merging potential split or obfuscated basic blocks (see the next subsection for the LCS refinement).

The detailed process works as follows, using Fig. 3 as a running example. The intermediate LCS scores (stored in δ) and the directions of the computed LCS (stored in γ) are showed in Fig. 4. We first set s as the current node and insert it into the working queue Q (Line 2); then we initialize δ with one row in which each element equals to 0 (the first row in Fig. 4), and initialize the scores (stored in σ) of all nodes in the suspicious to 0 (Line 4). For its neighbor node 1, we found a node of ρ semantically equivalent to it and its new score calculated by the function LCS (Line 22 and Line 23) is higher than its original one; thus, a new row is created in δ (Line 26; the second row in Fig. 4). Next, we update the score of node 1 and insert it into the working queue (Line 33 and Line 34). During the second iteration (Line 6), for node

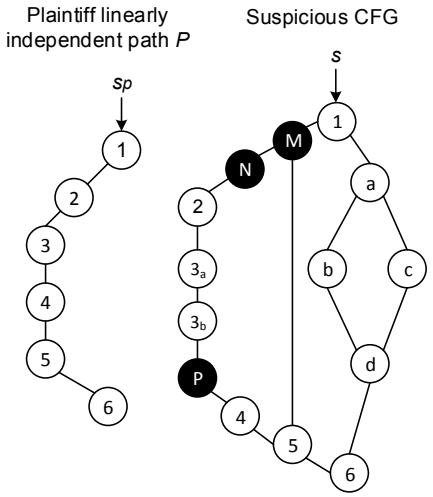

Fig. 3. An example for path similarity calculation. The black blocks are inserted bogus blocks. There is an opaque predicate inserted in M that always evaluates to true at runtime which makes the direct flow to the node 5 infeasible.

	v	1	2	3	4	5	6
u	0	0	0	0	0	0	0
1	0	↖1	←1	←1	←1	←1	←1
5	0	↑1	↑1	↑1	↑1	↖2	←2
2	0	↑1	↖2	←2	←2	←2	←2
6	0	↑1	↑1	↑1	↑1	↑2	↖3
4	0	↑1	↑2	↑2	↖3	←3	←3
5	0	↑1	↑2	↑2	↑3	↖4	←4
6	0	↑1	↑2	↑2	↑3	↑4	↖5

Fig. 4. The δ and γ tables store the intermediate LCS scores and the directions of the computed LCS, respectively. The three arrows on the left indicate the parent-child relationship between two nodes in the suspicious program during the LCS computation. For example, in the computed LCS, the parent node of node 2 is node 1, instead of node 5.

1, we cannot find a node in ρ that is semantically equivalent to either of its neighbors node M or node a; no new row is added to δ. Both their new scores are higher than their original ones; hence, their scores are updated and both them are inserted into the working queue (Line 33 and Line 34). The third iteration have two current nodes: node M and node a. For node M, its neighbor node N does not have a semantically equivalent node in ρ; hence, no new row is added to δ and node N is inserted into the working queue after its score is updated. Another neighbor node 5 has a semantically equivalent node in ρ; hence, a new row is added to δ (Line 26; see the third row in Fig. 4) and it is inserted into the working queue after updating its score. For node a, we cannot find a node in ρ that is semantically equivalent to either of its neighbors node b or node c; hence, no new row is added to δ, and the scores of both node b and node c are updated and both nodes are inserted into the working queue. During the forth iteration, node N has a neighbor node 2 which has a semantically equivalent node in ρ and it gets a new score higher than its original one; thus, a new row is added into δ (see the forth row in Fig. 4). To calculate its new score, the function LCS needs first to find its parent

node which is node 1 and uses the cell value of the row with respect to nodes 1 to calculate each cell value of a new row (shown in Fig. 4). Then the right-most cell value of this new row is the new score for node 2. The process repeats until the working queue is empty. When the working queue is empty, we obtain the highest score from the right-most column of δ (Line 12), and compare it with a threshold (Line 13). If it is higher than the threshold, the RefineLCS() will update the δ table (see the next subsection), and a new highest score will be obtained (Line 15); otherwise, the LCS computation is completed.

Here we use the example in Fig. 3 to illustrate a few interesting points. The first is how to deal with opaque predicate insertion. The node M is such an example. Since our path exploration considers both branches, we do not need to solve the opaque predicate statically. Our approach does not need to consider path feasibility, but focuses on shared semantically equivalent basic blocks. The second interesting scenario is when some basic blocks are obfuscated. For example, node 3 in ρ is split into two blocks and embedded into G as node 3_a and node 3_b. In this case, the basic block similarity comparison method determines neither node 3_a nor node 3_b is semantically equivalent to node 3 in ρ. To address this, the LCS refinement which tentatively merges unmatched blocks has been developed.

4.3 Refinement

Here we discuss some optimization techniques we developed to improve the obfuscation resiliency, which are implemented in the LCS Refinement.

Basic block splitting and merging. The LCS and basic block similarity comparison algorithms we presented so far cannot handle basic block splitting and merging in general. We solve this problem by the LCS refinement. First, CoP finds the consecutive basic block sequences which do not have semantically equivalent counterparts from both the suspicious and plaintiff paths, through backtracking on the LCS dynamic programming memoization table and the direction table (an example of them is showed in Fig. 4); then for each such sequence or list, CoP merges all basic blocks into one code trunk. After that, it adopts a method similar to the basic block comparison method to determine whether or not two corresponding merged code trunks (one from the plaintiff and the other from the suspicious) are semantically equivalent or similar. If the two merged code segments are semantically equivalent, the current longest common subsequence of semantically equivalent basic blocks is extended with the code segment in consideration. This method can handle basic block reordering and noise injection as well. Note that as we consider both memory cells and registers as input and output variables of basic blocks to detect the semantically equivalence between two merged blocks, the intermediate effects stored on the stack are not missed.

Consider the following complex example. An linearly independent path contains three basic blocks: $A \rightarrow B \rightarrow C$. In the suspicious program, they are first split in half, and filled up with bogus operations; then an arbitrary amount of bogus blocks are inserted between each split block. The modified path will be: $A_1 \rightarrow X \rightarrow A_2 \rightarrow B_1 \rightarrow Y \rightarrow Z \rightarrow B_2 \rightarrow C \rightarrow W \rightarrow C_2$; the similarity between A_1 and A (as well as between A_2 and A, B_1 and B, etc.) is about 0.5. In this case, CoP merges $A \rightarrow B \rightarrow C$ into one block, and $A_1 \rightarrow X \rightarrow A_2 \rightarrow B_1 \rightarrow Y \rightarrow Z \rightarrow B_2 \rightarrow C \rightarrow W \rightarrow C_2$ into another block. CoP then compares the two merged blocks and determine that they are semantically equivalent.

Conditional obfuscation. Conditionals are specified by the flags state in the FLAGS registers (i.e., CF, PF, AF, ZF, SF, and OF). These flags are part of the output state of a basic block. However, they can be obfuscated. We handle this by merging blocks during our LCS computation. No matter what obfuscation is applied, eventually a semantically equivalent path condition must be followed to execute a semantically equivalent path. Thus, when obfuscated blocks are combined, we will be able to detect the similarity.

4.4 Path Similarity Score

After exploring the suspicious program and computing the longest common subsequence of semantically equivalent basic blocks, we can calculate a path similarity score indicating the semantics of a plaintiff linearly independent path as manifested in the suspicious program. Definition 4 gives a high-level description of a path similarity score.

Definition 4. (Path Similarity Score) Given a plaintiff linearly independent path ρ, and a suspicious program G. Let $\Gamma = \{\gamma_1, \ldots, \gamma_n\}$ be all of the linearly independent paths of G, and $|\mathrm{LCS}(\rho, \gamma_i)|$ be the length of the longest common subsequence of semantically equivalent basic blocks between ρ and γ_i, $\gamma_i \in \Gamma$. Then, the path similarity score for ρ is defined as

$$\psi(\rho, G) = \frac{\max_{\gamma_i \in \Gamma} |\mathrm{LCS}(\rho, \gamma_i)|}{|\rho|}.$$

5 Function and Program Similarity Comparison

This section presents how to compare a plaintiff function to the suspicious program, and how to calculate the function and program similarity scores.

Given a plaintiff program (or component) and a suspicious program, our goal is to detect components in the suspicious program that are similar to the plaintiff. To this end, if an investigator has pre-knowledge on the plaintiff program, he can select a set of functions purposefully; if not, a set of functions can be randomly picked, or all of the functions can be simply chose for testing. CoP then tests each of them to find similar code in the suspicious program.

Because we will build the inter-procedural control flow graph for analysis (see Section 5.2), it may cause redundant checking of some functions. For example, if a function A calls another function B and both them are selected, then B will be checked twice. However, this can be avoided by leveraging the call graph. For instance, we can carefully select those functions that do not have the same descendant within the maximum depth (in our experiments, the maximum depth for inlining functions is set to 3, which is configurable), or those that cannot be reached by any other selected functions within the maximum depth.

5.1 Starting Blocks

The LCS of semantically equivalent basic blocks computation is based on the modified longest path algorithm to explore paths beginning at the starting points from the plaintiff function and suspicious program. It is important to choose the starting points so that the path exploration is not misled to irrelevant code parts of the plaintiff function and suspicious program.

We first look for the starting block inside the plaintiff function. To avoid routine code such as calling convention code inserted by compilers, we pick the first branching basic block (a block ends with a conditional jump instruction) as the starting block. We simply use this heuristic to skip the calling convention code at the beginning of the function. We then check whether we can find a semantically equivalent basic block from the suspicious program. This process can take as long as the size of the suspicious in terms of the number of basic blocks. If we find one or several semantically equivalent basic blocks, we proceed with the longest common subsequence of semantically equivalent basic blocks calculation. Otherwise, we choose another block as the starting block from the plaintiff function, and the process is repeated until the last block of the plaintiff function is checked.

5.2 Linearly Independent Paths

Once the starting blocks are identified, we select a set of *linearly independent paths* beginning at the starting block from the inter-procedural control flow graph of the plaintiff function. A linearly independent path is a path that introduces at least one new edge or node that is not included by other linearly independent paths [52]. We then adopt the Depth First Search algorithm to find a set of linearly independent paths from the plaintiff function. Because our goal is to measure the similarity between the plaintiff and suspicious programs, each basic block is examined only once.

For each selected path, we compare it against the suspicious program beginning at the starting block to calculate a path embedding score by computing the LCS of semantically equivalent basic blocks (see Section 4). Note that sometimes we may find several starting blocks in the suspicious program; in that case, we will test each of them and iterate the whole process.

5.3 Function and Program Similarity Scores

Once we have computed the path similarity scores (the lengths of the resulted LCS) for each selected plaintiff linearly independent path, we calculate the function similarity score for the plaintiff function. We assign a weight to each calculated LCS according to the plaintiff path length, and the function similarity score is the weighted average score. For each selected function in the plaintiff program, we compare it to a set of function in the suspicious program identified by the potential starting blocks, and the similarity score of this function is the highest one among those. We define the function similarity score in Definition 5.

Definition 5. (Function Similarity Score) Given a plaintiff function F, and a set of functions in the suspicious program identified by its starting blocks, $\Lambda = \{D_0, \ldots, D_m\}$.

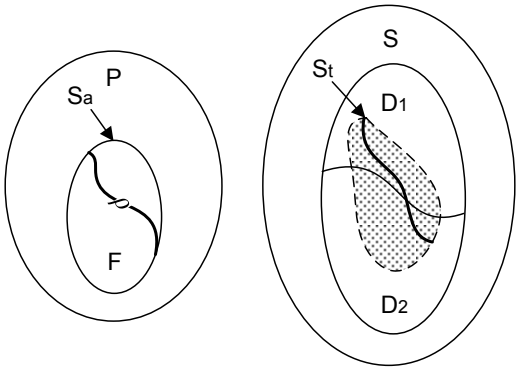

Fig. 5. An example of function similarity calculation. P and S are the plaintiff and suspicious programs, respectively. P has a function F, semantically equivalent to the shadow area of S; the shadow area is contained in two functions, D_1 and D_2, of S.

Let $\Omega = \{\rho_0, \ldots, \rho_n\}$ be a set of linearly independent paths of F. Given a function $D_t \in \Lambda$, the similarity score between F and D_t is defined as

$$\psi(F, D_t) = \sum_{i=0}^{n} \frac{w_i \psi(\rho_i, D_t)}{n},$$

where $w_i = |\rho_i| / \sum_{\rho_j \in \Omega} |\rho_j|$ is the weight for $\rho_i \in \Omega$, and $\psi(\rho_i, D_t)$ is the path similarity score as defined in Definition 4.

The function similarity score for F is then defined as

$$\psi(F) = \max_{D_t \in \Lambda} \psi(F, D_t).$$

After we calculate the similarity scores for each selected plaintiff function, we output their weighted average score as the similarity score of the plaintiff and suspicious programs. The weights are assigned according to the corresponding plaintiff function size.

Definition 6. (Program Similarity Score) Given a plaintiff program P, and a suspicious program S. Let $\zeta = \{F_0, \ldots, F_m\}$ be a set of functions of P. Then the program similarity score between P and S is defined as

$$\psi(P, S) = \sum_{i=0}^{m} \frac{w_i \psi(F_i)}{m},$$

where $w_i = |F_i| / \sum_{F_j \in \zeta} |F_j|$ is the weight for $F_i \in \zeta$, and $\psi(F_i)$ is the function similarity score as defined in Definition 5.

5.4 Resiliency

Because the plaintiff code can be transformed in various ways to hide the appearance and logic, the transformed (suspicious) code may have different function abstractions and different path segments. Therefore, our approach should be able to address this. Here, we show how our approach handles this using Fig. 5 as an example.

In Fig. 5, we want to detect the similarity between the plaintiff program P and the suspicious program S. To achieve it, we test each function of P to measure its similarity with S; all of these function similarities can collectively detect the program similarity. Consider a function F of P as an

example. To measure its similarity with S, we first find the starting blocks S_a and S_t, in P and S, respectively. Note that it is possible that S_t is inside of a function (D_1) of S. We then select a set of linearly independent paths from F based on the inter-procedural control flow graph of P; thus, these paths include the paths from the callee functions of F. We compare each linearly independent path against S to measure how much semantics of each path is manifested in S, using the path similarity computation technique (see Section 4). Assume we want to compare ρ (one of the linearly independent paths) against S. We explore S (beginning at S_t) based on its inter-procedural control flow graph, to compute the longest common subsequence of semantically equivalent basic blocks. The exploration not only compares the function D_1 identified by the starting block S_t, but also compares its callee function D_2. Because of semantic transformation techniques, the transformed (suspicious) code may contain different path segments that have the same functionality with the plaintiff. Detection of the semantics between these path segments is handled by our path similarity computation, which is based on the LCS computation and the merging basic blocks (see Section 4). Then, we can identify a path from S (the bold path) that is semantically equivalent to ρ (or has similar functionality as ρ). After testing all of the linearly independent paths of F, we can collectively detect the semantics of F as manifested in S. Once all functions of P are compared against S, we are able to determine the similarity between P and S.

6 Implementation

Our prototype implementation consists of 4,312 lines of C++ code measured with CLOC [16]. The front-end of CoP disassembles the plaintiff and suspicious binary code based on IDA Pro. The assembly code is then passed to BAP [6] to build an intermediate representation the same as that used in BinHunt [30], and to construct CFGs and call graphs, which are used to produce the inter-procedural control flow graph. The symbolic execution of each basic block and the LCS algorithm with path exploration are implemented in the BAP framework. We use the constraint solver STP [29] for the equivalence checking of symbolic formulas representing the basic block semantics.

7 Software Plagiarism Detection

7.1 Experimental Settings

We first evaluated our tool on its application to software plagiarism detection. The evaluation on its application to algorithm detection is presented in Section 8. We evaluated CoP on a set of benchmark programs to measure its obfuscation resiliency and scalability. We conducted experiments on basic block semantics comparison, small programs as well as large real-world production software. We compared the detection effectiveness and resiliency between our tools and four existing detection systems, MOSS [55], JPLag [61], Bdiff [9] and DarunGrim2 [24], where MOSS and JPLag are source code based, while Bdiff and DarunGrim2 are binary code based. Moss is a system for determining the similarity of programs based on the winnowing algorithm [63] for document fingerprinting. JPlag is a system that finds similarities among multiple sets of source code files. These two systems are mainly syntax-based and the main purpose has been to detect plagiarism in programming classes. Bdiff is a binary diffing tool similar to diff for text. DarunGrim2 [24] is a state-of-the-art patch analysis and binary diffing tool. Our experiments were performed on a Linux machine with a Core2 Duo CPU and 4GB RAM. In our experiments, we set the basic block similarity threshold to 0.7 and required the selected linearly independent paths cover at least 80% of the plaintiff program. There is a trade-off between the basic block similarity threshold, efficiency, and accuracy. Based on our experiments, a 0.7 threshold achieved both good efficiency and accuracy. In practice, this can also be adjusted based on different scenarios.

TABLE 1
Examples of semantically equivalent basic blocks with very different instructions

Pair 1		Pair 2	
lea ecx, [eax]	lea edx, [ebx]	mov eax, [ebp+8]	mov eax, [ebp+8]
mov edx, 1	add edx, edx	shl dword ptr [eax], 1	mov eax, [eax]
add ecx, edx	lea ecx, [eax]	mov eax, [eax+4]	lea edx,[eax+eax]
mov edx, ebx	mov eax, 1	mov [ebp+8], eax	mov eax, [ebp+8]
add edx, edx	add ecx, eax		mov [eax], edx
cmp ecx, edx	cmp ecx, edx		mov eax, [ebp+8]
jnz target	jnz target		mov eax, [eax+4]
			mov [ebp+8], eax
Pair 3		**Pair 4**	
setg al	mov eax, 0	add eax, 0x2	add eax, 0x1
movzx eax, al	cmovg esi, eax	sub eax, 0x1	and eax, 0x7fffffff
dec eax		shl eax, 0x1	
and esi, eax		shr eax, 0x1	

7.2 Basic Block Similarity Comparison

The basic block semantic similarity measure is a crucial part of our work. We evaluated the basic block semantic similarity measure on a set of basic block pairs in which each consists of an original basic block and a semantically equivalent version with very different instructions. We generate equivalent code sequences using the methods proposed in *superdiversifier* [37] including: (F1) substitution; (F2) instruction reordering; (F3) operand reordering; (F4) junk code injection; and (F5) register renaming.

In total, the set has 17 pairs of basic blocks, including 10 pairs created using the *superdiversifier* methods [37] and 7 pairs found in real world. We are limited to 17 pairs because we do not have access to tools such as *superdiversifier* [37]. Although the set is small, the pairs included are quite different with respect to semantic equivalence comparison. Note that basic block semantic equivalence will also be evaluated through our software plagiarism and code reuse experiments in the later sections.

Table 1 shows four pairs of semantically equivalent basic blocks with very different instructions. The first pair shows the effect of F1 and F5; the second shows the effect of F1, F2, and F5; the third pair shows the effect of substitution F1; and the fourth pair shows the effect of F1. Our tool reported the similarity scores higher than 0.9 for all 17 pairs, more than half of which have score 1.0. Overall, our tool is effective in measuring basic block semantic similarity.

7.3 Thttpd

The purpose of the experiments of *thttpd*, *openssl* (see Section 7.4), and *gzip* (see Section 7.5) is to measure the obfuscation resiliency of our tool. In our first experiment, we evaluated *thttpd-2.25b* and *sthttpd-2.26.4*, where *sthttpd* is forked from *thttpd* for maintenance. Thus, their codebases are similar, with many patches and new building systems added to *sthttpd*. To measure false positives, we tested our tool on several independent program, some of which have similar functionalities. These programs include *thttpd-2.25b*, *atphttpd-0.4b*, *boa-0.94.13* and *lighttpd-1.4.30*. We summarize the sizes (only include C/C++ source files, C/C++ header files, and assembly code) of the benchmark programs below, counted by CLOC.

Program	Size (LOC)
atphttpd-0.4b	641
boa-0.94.13	5,116
thttpd-2.25b	8,137
sthttpd-2.26.4	8,392
lighttpd-1.4.30	39,939

In all of our experiments, we select 10% of the functions in the plaintiff program randomly, and test each of them to find similar code in the suspicious program. For each function selected, we identify the starting blocks both in the plaintiff function and the suspicious program (see Section 4). We selected 13 functions from *thttpd-2.25b*.

7.3.1 Resilience to Transformation via Different Compiler Optimization Levels

Different compiler optimizations may result in different binary code from the same source code, but preserve the program semantics. We generated different executables of *thttpd-2.25b* and *sthttpd-2.26.4* by compiling the source code using GCC/G++ with different optimization options (-O0, -O1, -O2, -O3, and -Os). This has produced 10 executables. We cross checked each pair of the 10 executables on code reuse detection and compared the results with DarunGrim2 [24], a state-of-the-art patch analysis and binary diffing tool. Fig. 6 shows the results with respect to four of the ten executables compiled by optimization levels -O0 and -O2. Results with other optimization levels are similar and we do not include them here due to the space limitation.

From Fig. 6, we can see our tool CoP is quite effective compared to DarunGrim2. Both CoP and DarunGrim2 have good results when the same level of optimizations is applied (see the left half of Fig. 6). However, when different optimization levels (-O0 and -O2) are applied, the average similarity score from DarunGrim2 is only about 13%, while CoP is able to achieve an average score of 88%.

To understand the factors that caused the differences, we examined the assembly codes of the executables, and found these differences were mainly caused by different register allocation, instruction replacement, basic block splitting and combination, and function inline and outline. Due to the syntax differences caused by different register allocation and instruction replacement, DarunGrim2 is unable to determine the semantic equivalence of these basic blocks; while CoP is able to identify these blocks as very similar or identical.

Two interesting cases are worth mentioning. The first case is the basic block splitting and combination. One example

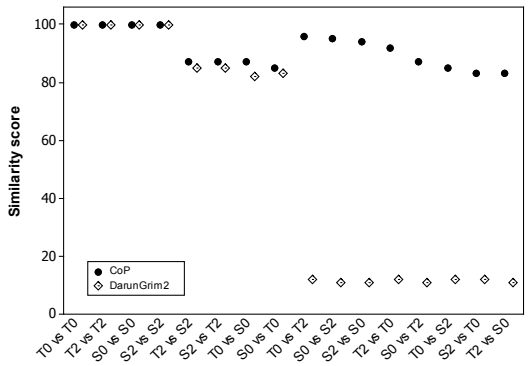

Fig. 6. Code similarity scores resulting from different compiler optimization levels. Higher is better since these two programs share codebase. (Legend: T*i* and S*i* stand for *thttpd* and *sthttpd* compiled with -O*i*, respectively.)

is the use of conditional move instruction (e.g., `cmovs`). We found that when *thttpd-2.25b* was compiled with -O2, there was only one basic block using the `cmovs` instruction; when it was compiled with -O0, there were two basic blocks. CoP addresses this by merging neighboring blocks through the LCS Refinement. As a result, CoP found the two basic blocks compiled by -O0, when merged, were semantically equivalent to the one block compiled by -O2.

Another interesting case is function inline and outline. There are two basic scenarios. One is that the inlined/outlined function is a user-defined or statically-linked library function; another is that the inlined/outlined function is from a dynamically linked library function or other unresolved function. Let us take `de_dotdot()`, a user-defined function in *thttpd-2.25b*, as an example. The function is inlined in `httpd_parse_request()` when it is compiled with -O2, but not inlined with -O0. It is similar for *sthttpd-2.26.4*. CoP handles this by "inlining" the callee function, since its code is available, during the LCS computation. Note that the maximum depth for inlining functions is set to 3, which is configurable. In the second scenario, where the inlined/outlined function is a dynamically linked library function (e.g., `strspn()`), CoP will simply not inline the function. As we handle the plaintiff and suspicious programs in the same way, we believe it won't cause serious issues. However, inlining some function here and there does not significantly affect the overall detection result of CoP since we can test all the functions. One may wonder whether an adversary can hide stolen code in a dynamically linked library. Our assumption is that the source or binary code of the plaintiff program and at least binary code of the suspicious program is available for analysis. Although CoP, relying on static analysis, has some difficulty to resolve dynamically linked library calls, it is able to analyze the dynamically linked library as long as it is available, and identify the stolen code if exists.

7.3.2 Resilience to Transformation via Different Compilers

We also tested CoP on code compiled with different compilers and compared with DarunGrim2. We generated different executables of *thttpd-2.25b* and *sthttpd-2.26.4* using different compilers, GCC and ICC, with the same optimization option (-O2). Fig. 7 shows the detection results. With different

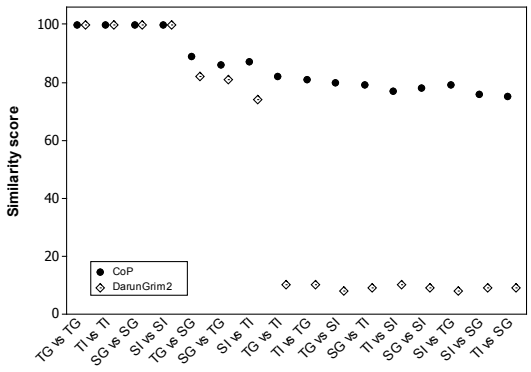

Fig. 7. Code similarity scores resulting from different compilers. Higher is better since these two programs share codebase. (Legend: P*c* stands for program P compiled with compiler c, where P is either T for *thttpd* or S for *sthttpd*, and c is either G for GCC or I for ICC, respectively.)

compilers, the differences between the resulted code are not only caused by different compilation and optimization algorithms, but also by using different C libraries. GCC uses *glibc*, while ICC uses its own implementation. The evaluation results show that CoP still reports good similarity scores (although a little bit lower than those of using the same compiler), but DarunGrim2 failed to recognize the similarity.

7.3.3 Resilience to Code Obfuscations

To evaluate the obfuscation resiliency, we used two commercial products, Semantic Designs Inc.'s C obfuscator [65] and Stunnix's CXX-obfuscator [67], as the source code obfuscation tools, and two open-source products, Diablo [26] and Loco [50], as the binary code obfuscation tools. We also utilized CIL [58] as another source code obfuscation tool. CIL possesses many useful source code transformation techniques, such as converting multiple returns to one return, changing `switch/case` to `if/else`, and replacing logical operators (&&, ?:, etc.) with `if/else`.

Component vs. Suspicious. In the previous tests, we evaluated the similarity between two programs. In this test, we evaluated whether a component from the plaintiff program is reused by a suspicious program. The experiments we conducted with different compilers and compiler optimizations between *thttpd* and *sthttpd* can be viewed as a special case of the component vs. suspicious scheme. The motivation is that, for software plagiarism or code reuse scenarios, the original software developers often have insights on the plaintiff program and can point to the critical component. Therefore, we can test critical components to see whether they are reused in the suspicious program. In this experiment, we test on a small component, function `httpd_parse_request()` vs. *thttpd* and MD5 vs. *openssl*. In our subsequent experiment, we test in large program components: the Gecko rendering engine vs. the Firefox browser.

The obfuscation techniques can be divided into three categories: layout, control-flow, and data-flow obfuscation [17]. Each category contains different obfuscation transformations. We chose 13 typical obfuscation transformations [17] from all the three categories to obfuscate *thttpd*, and then compiled the obfuscated code to generate the executables. We compared the detection results of CoP with those of four state-of-the-art plagiarism detection systems including MOSS [55], JPLag [61], DarunGrim2 [24] and Bdiff [9]. We evaluated on code with a single and multiple obfuscations applied. The single and multiple obfuscation results are shown in Table 2 and Table 3, respectively.

We analyzed how CoP addresses these obfuscation techniques. The layout obfuscations do not affect binary code, but impair the source code based detection systems. The data obfuscations also do not affect CoP because its basic block comparison method is capable of addressing noise input and output, and is insensitive to data layout changes.

Control-flow obfuscations reduce quite a bit the scores reported by MOSS, JPLag, DarunGrim2, and Bdiff, but have little impact on CoP. We analyzed the obfuscation that changes the `switch/case` statements to `if/else` statements. This obfuscation is done by CIL as source-to-source transformation in our experiment. GCC applied an optimization on the `switch/case` statements. It generated either a balanced binary search tree or a jump table depending on the number of the case branches. We then conducted further experiments on this case. When GCC generated a balanced binary search tree code, the similarity scores between two code segments (one contains `switch/case` statements and the other contains the corresponding `if/else` statements) reported by MOSS, JPLag, DarunGrim2, Bdiff, and CoP are 0%, 34%, 38%, 36%, and 90%, respectively. When GCC generated a jump table, the similarity scores are 0%, 31%, 19%, 16%, and 92%, respectively. The result shows our method is quite resilient to advanced code obfuscations.

We especially note that existing tools are not resilient to the control flow flattening obfuscation [17], which transforms the original control flow with a dispatch code that jumps to other code blocks. Control flow flattening has been used in real world for software software protection. For example, Apple's FairPlay code has been obfuscated with control flow flattening. Clearly this defeats syntax-based methods. CoP is able to get good score against control flow flattening because of our symbolic execution based path exploration and basic block semantics LCS similarity calculation method.

7.3.4 Independent Programs

To measure false positives, we also tested CoP against four independently developed programs: *thttpd-2.25b*, *atphttpd-0.4b*, *boa-0.94.13*, and *lighttpd-1.4.30*. Very low similarity scores (below 2%) were reported.

Because some basic blocks in the plaintiff program may happen to be semantically equivalent to some blocks in the suspicious program, there may be a low similarity score between them. For example, both programs may contain a code snippet that has the same functionality of summing up a sequence of numbers using a for-loop. Then the loop bodies from the two programs happen to be semantically equivalent. However, considering the low similarity, they should not be regarded as the case of software plagiarism.

7.4 Openssl

This experiment also aims to measure the obfuscation resiliency. We first evaluated *openssl-1.0.1f*, *openssh-6.5p1*, *cyrus-sasl-2.1.26*, and *libgcrypt-1.6.1*, where *openssh-6.5p1*, *cyrus-sasl-2.1.26*, and *libgcrypt-1.6.1* use the library *libcrypto.a* from

TABLE 2
Detection results (resilience to single code obfuscation)

Obfuscation		Similarity score (%)				
	Source code based	Binary code based				
		MOSS	JPLag	DarunGrim2	Bdiff	CoP
Layout	Remove comments, space, and tabs	47	62	100	100	100
	Replace symbol names, number, and strings	22	90	100	100	100
Control	Insert opaque predicates	–	–	47	43	95
	Inline method	–	–	32	34	91
	Outline method	–	–	38	33	90
	Interleave method	45	40	32	19	89
	Convert multiple returns to one return	75	91	98	86	97
	Control-flow flattening	–	–	5	3	86
	Swap `if/else` bodies	72	78	81	73	98
	Change `switch/case` to `if/else`	74	51	69	51	94
	Replace logical operators (&&, ?:, etc.) with `if/else`	79	95	97	88	96
Data	Split structure object	83	87	93	82	100
	Insert bogus variables	93	88	86	75	100

TABLE 3
Detection results (resilience to multiple code obfuscation)

Obfuscation	Similarity score (%)				
	Source code based		Binary code based		
	MOSS	JPLag	DarunGrim2	Bdiff	CoP
Insert opaque predicates, convert multiple returns to one return	–	–	33	29	89
Inline method, outline method	–	–	25	20	87
Interleave method, insert bogus variables	29	27	30	15	88
Swap `if/else` bodies; Split structure object	38	51	53	39	91

openssl-1.0.1f. We tested our tool on completely irrelevant programs to measure false positives. These programs include *attr-2.4.47* and *acl-2.2.52*. We summarize the sizes (only include C/C++ source files, C/C++ header files, and assembly code) of the benchmark programs below, counted by CLOC.

Program	Size (LOC)
attr-2.4.47	2,360
acl-2.2.52	5,524
cyrus-sasl-2.1.26	57,851
openssh-6.5p1	81,997
libgcrypt-1.6.1	89,383
openssl-1.0.1f	278,612

In this experiment, we test whether the suspicious programs contain an MD5 component from the plaintiff program *openssl-1.0.1f*. There are four functions in the MD5 component from *openssl-1.0.1f*: `MD5_Init`, `MD5_Update`, `MD5_Transform`, and `MD5_Final`. The MD5 plaintiff component was compiled by GCC with the -O2 optimization level. To measure obfuscation resiliency, we conducted the similar experiments as on *thttpd*. Fig. 8 shows the similarity scores when different kinds of obfuscations were applied. The detection results showed that *openssh-6.5p1* which is based on *openssl-1.0.1f* contains the MD5 component of *openssl-1.0.1f*. Their similarity scores were between 82% and 100%, with obfuscations applied. We especially noted that the scores for *openssl-1.0.1f* vs. *libgcrypt-1.6.1* and *cyrus-sasl-2.1.26*, with obfuscations applied, are between 12% and 30%. With further investigation, we confirmed that although both *libgcrypt-1.6.1* and *cyrus-sasl-2.1.26* are based on *openssl-1.0.1f*, their MD5 components are re-implemented independently. For the completely irrelevant programs *acl-2.2.52* and *attr-2.4.47*, very low similarity scores (below 4%) were reported.

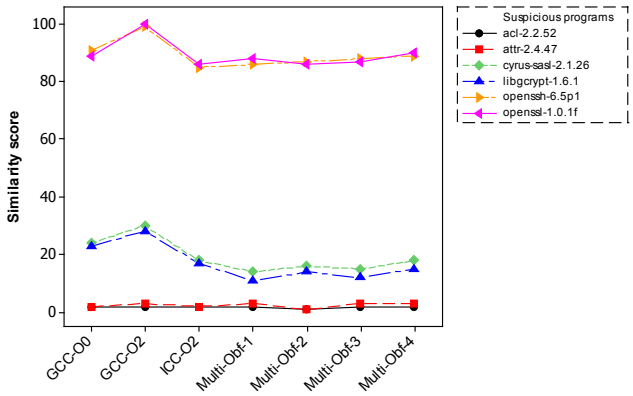

Fig. 8. Code similarity scores when different kinds of obfuscations were applied. The plaintiff component is the MD5 component from `openssl-1.0.1f`. (Legend: GCC-O*i* and ICC-O*i* stand for GCC and ICC compiled with -O*i*, respectively. Multi-Obf-*j* stands for the *j*th multiple code obfuscation in Table 3.)

7.5 Gzip

In our third experiment, we first evaluated our tool on *gzip-1.6* against its different versions including *gzip-1.5*, *gzip-1.4*, *gzip-1.3.13*, and *gzip-1.2.4*. We also tested *gzip-1.6* against two independent programs with some similar functionalities, *bzip2-1.0.6* and *advanceCOMP-1.18*, to measure false positives. The following is a summary of their sizes (only include C/C++ source files and C/C++ header files).

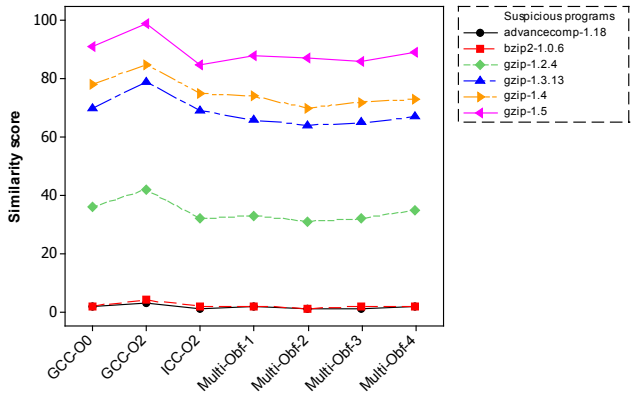

Fig. 9. Code similarity scores when different kinds of obfuscations were applied. The plaintiff program is `gzip-1.6`. (Legend: GCC-O*i* and ICC-O*i* stand for GCC and ICC compiled with -O*i*, respectively. Multi-Obf-*j* stands for the *j*th multiple code obfuscation in Table 3.)

Program	Size (LOC)
bzip2-1.0.6	5,823
advanceCOMP-1.18	17,087
gzip-1.2.4	5,809
gzip-1.3.13	21,265
gzip-1.4	24,262
gzip-1.5	31,957
gzip-1.6	32,079

The plaintiff program *gzip-1.6* was compiled by GCC with the -O2 optimization level. We selected 10% functions in *gzip-1.6*, which were 17 functions. To measure obfuscation resiliency, we conducted the similar experiments as on *thttpd* and *openssl*, and show the results in Fig. 9. The results showed that the similarity scores between *gzip-1.6* and *gzip-1.5*, *gzip-1.6* and *gzip-1.4*, *gzip-1.6* and *gzip-1.3.13*, and *gzip-1.6* and *gzip-1.2.4*, are 99%, 86%, 79%, and 42%, respectively, when compiled by GCC with -O2. Moreover, when various obfuscation were applied, the average similarity score only reduced around 12%, indicating that our tool is resilient to obfuscation. From the results, we can see that the closer the versions, the higher the similarity scores. For the independent programs *bzip2-1.0.6* and *advanceCOMP-1.18*, very low similarity scores (below 3%) were reported. As we have presented a detailed analysis on how our tool addresses various obfuscation techniques for the experiment of *thttpd*, due to the space limit, we do not include the similar analysis here.

7.6 Gecko

To measure the scalability of detecting large real-world production software, we chose the Gecko layout engine as the plaintiff component, and evaluated it against the Firefox web browser. We selected 8 versions of Firefox, each of which includes a different version of Gecko. Thus, we have 8 plaintiff components (8 Gecko versions) and 8 suspicious programs. For each Gecko version, we selected 10% functions for analysis, which were from 3726 to 6193 functions. We cross checked each pair of Gecko and Firefox, and the results are shown in Fig. 10. The line graph contains 8 lines. The results showed that *the closer two versions are the more similar their code is*. Especially. the highest score of each line is the

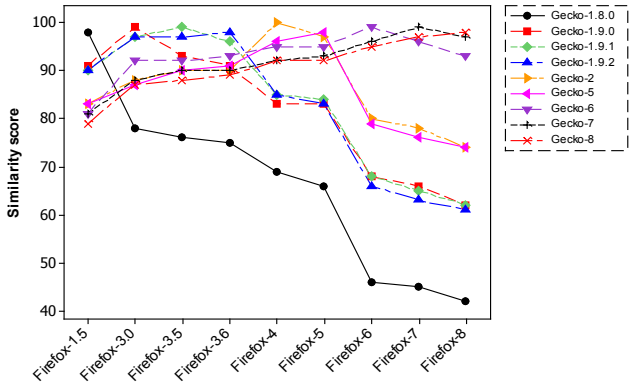

Fig. 10. Gecko vs. Firefox (%)

case where the Gecko version is included in that Firefox version. To measure false positives, we also checked CoP on Gecko vs. 4 versions of Opera (11.00, 11.50, 12.00, and 12.16) and 3 versions of Google Chrome (28.0, 29.0, and 30.0), which do not use Gecko as layout engine. CoP reported scores below 3% for all cases.

8 Algorithm Detection

8.1 Experimental Settings

We evaluated the effectiveness of CoP on six cryptographic algorithms, two sort algorithms, and one search algorithm against a set of benchmark programs ranging from small to large real-world software.

To detect an algorithm present in a given program, we chose a representative set of programs as the reference implementations (a special situation in which only one reference implementation is chosen) to represent the algorithm. We conducted the following two categories of experiments: (1) pair-wise comparison of the reference implementations, and (2) cross-checking the reference implementations against the benchmark programs. In addition, different compilers, different compiler optimization levels, as well as code obfuscation techniques were applied. For the second category, we compared each reference implementation of the algorithm to the suspicious program, to compute their similarity scores, and output the highest one among those as the algorithm similarity score, detecting whether the algorithm is present in the suspicious program.

Definition 7. (Algorithm Similarity Score) Given a set of reference implementations of an algorithm, $\mathbb{A} = \{A_0, \ldots, A_n\}$, and a suspicious program S. The algorithm similarity score for $\mathbb{A}$ is defined as

$$\psi(\mathbb{A}, S) = \max_{A_i \in \mathbb{A}} \psi(A_i, S),$$

where $\psi(A_i, S)$ is the program similarity score between A_i and S as defined in Definition 6.

To measure the similarity between a reference implementation and the suspicious program, unlike the software plagiarism detection cases which randomly picked 10% of the functions from the plaintiff, we selected core functions from the reference implementation based on pre-knowledge; if we did not have access to the source code, we simply tested all

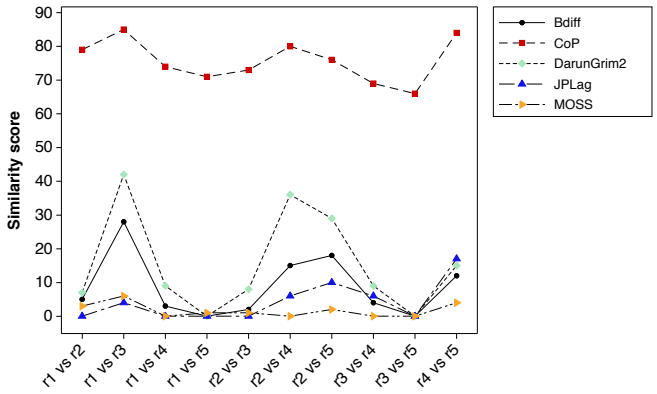

Fig. 11. Code similarity scores for the five implementations of MD5 (%). Higher is better since they are from the same algorithm. (Legend: r*i* stand for the *i*th implementation.)

the functions (there are usually limited functions in reference implementations). A core function refers to a function that implements the core functionality of an algorithm. For example, the functionality for MD5 includes initializing hash digests, compressing messages, and updating hash digests. Thus, we selected these core functions, and skiped other unrelated ones (e.g., a function obtaining inputs from keyboard or from a file).

We set the basic block similarity threshold to 0.7, and selected a set of linearly independent paths covering at least 80% of the algorithm reference implementations. The detection result threshold was set to 60%. In general, a 60% threshold is good enough to recognize an algorithm based on our experience.

8.2 MD5

Our first experiment evaluated MD5. MD5 is a cryptographic hash function that produces a 128-bit hash value. The input message is broken up into 512-bit chunks that are then processed in an iterative fashion.

8.2.1 Evaluation on Standalone Implementations

We found five independent implementations of MD5 from the open source code. To confirm their independence, we used four detection systems (MOSS, JPLag, Bdiff and DarunGrim2) to measure their pair-wise similarities, and showed the results in Fig. 11.[3] Note that for all the other experiments, we did the same confirmation. A low similarity would suggest independence. Fig. 11 shows that the similarity scores from MOSS and JPLag are very low (below 10%), suggesting the five implementations are independent. The similarity scores from CoP (between 66% and 86%) are high enough to identify MD5, even though the binary codes are quite different according to the results from Bdiff and DarunGrim2.

To understand the factors that caused the binary code differences, we examined the source code of the five implementations. All five are independently implemented in different ways by different programmers; this generates many variations in the resulting code, such as different function and variable names, different procedure (function) abstractions, conditional transformation, loops rolling and unrolling, etc.

Because CoP is based on symbolic execution, different function and variable names do not affect the result. With respect to different procedure (function) abstractions, CoP is able to address it. For example, some implementations inline the functions of `decode` (changes an array with the *char* type to an array with the *uint4* type) and `encode` (changes an array with the *uint4* type to an array with the *char* type), while others outline them as two separated functions. Because CoP inlines the callee functions during the LCS of semantically equivalent basic blocks calculation, CoP is able to handle this problem. However, in another scenario in which the inlined/outlined function is a dynamically linked library function, CoP cannot locate the callee function, which may result in a lower similarity score. For example, three of our five implementations call the library functions of `memcpy` and `memset`, while the other two do not; instead, they implement their own functions substituting for the library calls. Although CoP relying on static analysis, has some difficulty resolving this problem, it is able to analyze the dynamically linked library as long as it is available. In addition, an examination of the algorithms listed in the "*Algorithm Design*" book [42] reveals that few algorithms involve library calls, indicating this problem has little impact on algorithm detection. However, if some algorithm implementations do invoke library functions whose code is not available for analysis, a lower similarity score may incur as CoP is not able to inline the callee function.

CoP can also handle conditional transformation. For example, the following are two code segments (upper and lower) from two of the MD5 implementations, respectively.

```
padn=(last<56)?(56-last):(120-last);
pad=(bytes>=56)?(120-bytes):(56-bytes);
```

In the two code segments, the branching conditionals are implemented in opposite ways. It is handled by merging blocks during the LCS calculation. We consider the upper code segment from the reference implementation, and the lower one from the target program. We select a linearly independent path from the reference implementation containing the branch block (`last<56`) and the block (`56-last`) that follows the true conditional. Next we use the selected path to explore the target program, and find a block (`56-bytes`) semantically equivalent to (`56-last`) in the reference implementation, but we cannot find a block semantically equivalent to (`last<56`). However, after merging blocks, we are able to detect that the path segment of (`last<56`) and (`56-last`) in the reference program is semantically equivalent to the path segment of (`56-bytes`) and (`bytes>=56`) in the target program. Thus, no matter how a branch is implemented, as long as the corresponding blocks are merged, we are able to detect the similarity.

Resilience to transformation via semantic transformation techniques. We also tested CoP on its resilience to semantic transformation techniques, including different compiler, different compiler optimization levels, and code obfuscation techniques.

3. We use MOSS and JPLag to compare the source codes of the reference implementations, and use Bdiff and DarunGrim2 to compare the binary codes of the reference implementations.

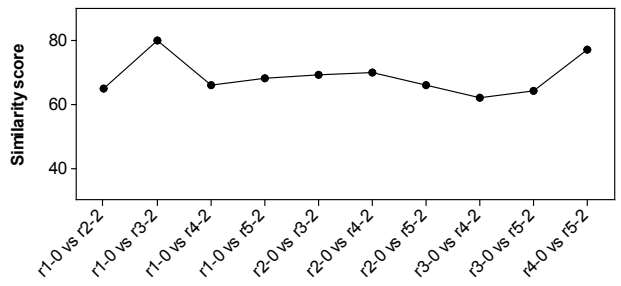

Fig. 12. Similarity scores for different compiler optimization levels (%). Higher is better since they are from the same algorithm. (Legend: r*i*-*j* stand for the *i*th implementation compiled with -O*j*.)

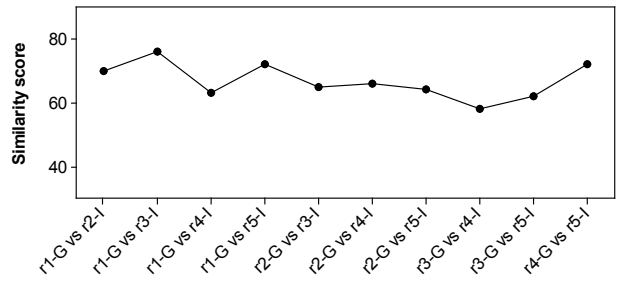

Fig. 13. Similarity scores for different compilers (%). Higher is better since they are from the same algorithm. (Legend: r*i*-*j* stands for the *i*th implementation compiled by j, where j is either G for GCC or I for ICC.)

We first generated different executables of each reference implementation using GCC/G++ with different optimization levels (-O0, -O1, -O2, -O3, and -Os). This produced 25 executables. We then pair-wise compared the executables and show the results in Fig. 12. Due to space limitation, we show only the results related to the executables compiled by -O0 and -O2; results with other optimization levels are similar. Next, we generated different executables using different compilers, GCC and ICC, with the same optimization level (-O2), and show the results in Fig. 13. Furthermore, we used code obfuscation techniques, such as insert opaque predicates, inline method, outline method, swap `if/else` bodies, and change `switch/case` to `if/else`, etc., to obfuscate each implementation and generate executables. The results indicate that CoP can successfully identify MD5.

8.2.2 Evaluation on Production Software

We tested CoP on a set of benchmark programs, including *cryptlib-3.4.2*, *openssl-1.0.1f*, *openssh-6.5p1*, *libgcrypt-1.6.1*, *truecrypt-7.1a*, *berkeley DB-6.0.30*, *MySQL-5.6.17*, *git-1.9.0*, *glibc-2.19*, *p7zip-9.20.1*, *cmake-2.8.12.2*,*thttpd-2.25b*, and *sthttpd-2.26.4*. We cross-checked all the reference implementations against each benchmark, and reported the highest similarity scores. The results show that *cryptlib-3.4.2*, *openssl-1.0.1f*, *openssh-6.5p1*, *libgcrypt-1.6.1*, *MySQL-5.6.17*, *glibc-2.19*, and *cmake-2.8.12.2* implement MD5 with the highest similarity scores between 74% and 83%; *thttpd-2.25b* and *sthttpd-2.26.4* do not implement MD5 with quite low similarity scores (around 1%). We investigated the source code and confirmed the results.

We especially noted that the highest similarity scores for *p7zip-9.20.1*, *truecrypt-7.1a*, *berkeley DB-6.0.30*, and *git-1.9.0* were between 19% and 24%. Upon further investigation, we found that the similarity was related to the implementation of SHA1. When we checked MD5, some parts of SHA1 were found to be similar to MD5. However, such a score is not high enough to identify an algorithm.

8.3 SHA1

Our second experiment tested SHA1. SHA1 produces a 160-bit hash value, typically rendered as a hex number with 40 digits long.

8.3.1 Evaluation on Standalone Implementations

We found six independent implementations. We used CoP to pair-wise compare them. Their similarity scores were between 46% and 87%.

To verify the results, we examined the source code of the six implementations, and found that three implementations use loops to implement the compression function, while the others unroll the corresponding loops. Moreover, the loops of the compression function are divided into four parts with four different loop body calculations, and none of the four parts is semantically equivalent to the unrolled block. Thus, the similarity score between two implementations (one with loop and another without loop) was a little lower. As we use the highest similarity score to identify an algorithm present in a program, this does not affect the detection results.

8.3.2 Evaluation on Production Software

We next tested CoP on the same set of the benchmark programs. We cross-checked the reference implementations and the benchmark programs, and found that except for *glibc-2.19*, *thttpd-2.25b*, and *sthttpd-2.26.4*, all other benchmarks got highest similarity scores between 77% and 89%, indicating that they have implemented SHA1. For *glibc-2.19*, the similarity score was 24%. From investigation, we found that the similarity was related to the implementation of MD5. However, such a score is not high enough to identify SHA1.

8.4 SHA2

Our third experiment tested SHA2. SHA2 is a set of cryptographic hash functions, including SHA224, SHA256, SHA384, and SHA512. SHA224 is simply a truncated version of SHA256, with the only difference being that they are computed with different initial values; hence, we consider them to be the same algorithm. The same is true for SHA384 and SHA512. Thus, SHA2 is represented as two algorithms: SHA256 (or SHA224) and SHA512 (or SHA384).

8.4.1 Evaluation on Standalone Implementations

We had five independent implementations of SHA256 and four independent implementations of SHA512. We used CoP to pair-wise compare them and show results in Table 4.

From Table 4, we can see that CoP can successfully identify SHA256 and SHA512 within their corresponding implementations with the average similarity score of 74% and 71%, respectively. However, the average cross-checking similarity score of SHA256 and SHA512 was only 16%,

TABLE 4
The similarity scores in SHA2 experiment

	SHA256 vs. SHA256	SHA512 vs. SHA512	SHA256 vs. SHA512
Min	0.64	0.62	0.15
Max	0.87	0.83	0.19
Avg	0.74	0.71	0.16

which surprised us as they are from the same cryptographic family SHA2. We analyzed their code and found two main reasons. First, they operate on different sizes of variables. SHA256 operates on 32-bit variables, while SHA512 on 64-bit variables. Because the implementations were compiled in 32-bit, each 64-bit variable was split into two parts stored in two different memory cells; these two parts were considered as the outputs of basic blocks. Second, SHA256 and SHA512 have different hashing computations (e.g., their right rotations are different), resulting in different input-output relations of basic blocks. Therefore, we verified that SHA256 and SHA512 are not similar and confirmed our results. We further applied semantic transformation techniques on each reference implementation. CoP also successfully identified and distinguished SHA256 and SHA512.

8.4.2 Evaluation on Production Software

We cross-checked the reference implementations and the benchmark and got the following results: *cryptlib-3.4.2*, *openssl-1.0.1f*, *openssh-6.5p1*, *libgcrypt-1.6.1*, *truecrypt-7.1a*, *MySQL-5.6.17*, *glibc-2.19*, and *cmake-2.8.12.2* contain both SHA256 (or SHA224) and SHA512 (or SHA384), while *p7zip-9.20.1* only implements SHA256. The other programs of *berkeley DB-6.0.30*, *git-1.9.0*, *thttpd-2.25b*, and *sthttpd-2.26.4* contain neither SHA256 nor SHA512. We then examined the source code and verified the results.

8.5 AES

Our fourth experiment was on AES. AES is a 16-byte block cipher with a key of either 128, 256, or 512 bits. It processes input data through a substitution-permutation network in which each iteration (round) employs a round key derived from the input key.

8.5.1 Evaluation on Standalone Implementations.

We found three independent implementations. We used CoP to pair-wise compare them, and obtained the similarity scores between 14% and 88%. Upon further investigation, we found the reason for the lower similarity scores to be that one implementation uses a lookup table to optimize AES and, hence, the logic operations are implied (or hidden) in the lookup table. In this case, CoP has some difficulty in detecting the semantic similarity. However, this kind of optimization can be only applied to the algorithms that involve a sufficient number of matrix operations; thus, it has limited impact. Worth mentioning here is the splitting and merging of variables. Take the following two code segments from two implementations, respectively, as an example.

```
cipher[0]=(unsigned long)(x0&0xffff)|
        ((unsigned long)(x1&0xffff)≪16L);
cipher[1]=(unsigned long)(x2&0xffff)|
        ((unsigned long)(x3&0xffff)≪16L);
```

```
cipher[0]=(unsigned char)x10;
cipher[1]=(unsigned char)(x10≫8);
cipher[2]=(unsigned char)x32;
cipher[3]=(unsigned char)(x32≫8);
cipher[4]=(unsigned char)x54;
cipher[5]=(unsigned char)(x54≫8);
cipher[6]=(unsigned char)x76;
cipher[7]=(unsigned char)(x76≫8);
```

The upper code segment has two 32-bit outputs (`cipher[0]` and `cipher[1]`). The lower code segment has eight 8-bit outputs (`cipher[i]`, where $0 \leq i \leq 7$). Because the outputs have different lengths, CoP cannot compare them to detect whether they are equivalent given the corresponding inputs are the same, and hence determines that the two code segments are not semantically equivalent. However, in general, the splitting and merging of variables usually occurs at the beginning/end of encryption and decryption, unless programmers adopt complicated methods to convert all operations on variables of the original type to those on variables of the new type, which usually is not practical. Thus, this has limited impact on the detection results.

8.5.2 Evaluation on Production Software

We also compared the AES implementations against the benchmark programs, and found that *cryptlib-3.4.2*, *openssl-1.0.1f*, *openssh-6.5p1*, *libgcrypt-1.6.1*, *truecrypt-7.1a*, *berkeley DB-6.0.30*, and *MySQL-5.6.17* contain AES, and the others do not. In addition, their implementations of AES are very similar to that implementation utilizing a lookup table, with the highest similarity scores approximately 86%. To verify it, we checked the source code and obtained consistent results.

8.6 RC4, Blowfish, Bubble Sort, Binary Tree Sort, and Redblack Tree

We then evaluated RC4 and Blowfish. CoP precisely identified them in their corresponding implementations and the benchmark. We further evaluated two sort algorithms (Bubble sort and Binary tree sort), and one search algorithm (Redblack tree) on their corresponding reference implementations. CoP also successfully identified them with the average similarity score of 82%. Due to the space limit, we do not include the analysis here.

8.7 Performance

We reported the average execution time of CoP for detecting the six cryptographic algorithms from the thirteen benchmark programs in Table 5. The second row shows the size of these benchmarks (only include C/C++ source files, C/C++ header files, and assembly code). The third to the eighth rows show the execution time. In our experiments, most of the execution time was spent on finding the starting blocks for the algorithm reference implementation and the suspicious program. In our current prototype, we perform brute-force search in this step and have not optimized for performance yet. In practice, it is often possible to develop heuristics to locate starting blocks and improve the performance.

TABLE 5
Execution time (hr).

	thttpd	sthttpd	openssh	truecrypt	libgcrypt	p7zip	git	cryptlib	openssl	cmake	berkeley DB	glibc	MySQL
Size (LOC)	8,137	8,392	81,997	87,196	89,383	120,730	148,046	241,274	278,612	324,252	547,448	917,589	1,600,471
MD5	4.02	4.13	2.11	3.15	2.21	4.27	3.40	2.83	3.00	3.30	6.35	7.94	10.13
SHA1	4.60	4.23	1.98	2.01	2.16	2.37	2.39	2.72	2.89	3.64	4.83	7.24	10.00
SHA2	4.13	4.24	1.94	1.97	2.02	2.33	8.48	2.66	2.91	3.81	9.77	10.39	9.88
AES	4.83	5.26	2.46	2.49	2.57	10.84	11.47	3.62	4.37	10.13	6.21	13.32	11.48
RC4	1.13	1.16	1.38	4.00	4.12	5.36	4.62	2.43	2.84	5.54	8.93	7.24	9.40
Blowfish	3.48	3.73	1.93	2.12	2.39	7.82	8.14	3.32	3.46	9.87	10.98	10.37	9.44

8.8 Algorithm Detection Result Summary

We summarize the detection results for the six cryptographic algorithms in Table 6. We verified these detection results via investigating the source code of the benchmark programs. CoP reported no false positive or false negative.

9 Discussion

9.1 Obfuscation Resiliency Analysis

The combination of the rigorous program semantics and the flexibility in terms of noise tolerance of LCS is powerful. Here we briefly analyze its obfuscation resiliency. Obfuscation can be classified into three categories: layout, control-flow, and data-flow obfuscations [17].

9.1.1 Layout Obfuscation

Because CoP is a semantic-based plagiarism detection approach, layout obfuscation (e.g., comments, space and tabs removal, identifier replacement, etc.) does not have any effect.

9.1.2 Control-flow Obfuscation

CoP deals with *basic block splitting and combination* by merging blocks. *Basic block reordering* can also be handled by merging blocks. After merging, the order does not matter to the symbolic execution used in the basic block similarity comparison if there is no dependency between two blocks or instructions. *Instruction reordering* is also taken care by the symbolic execution. *Function inline and outline* is handled by inter-procedural path exploration in the LCS computation. It is virtually impossible to solve *opaque predicates*; however, CoP can tolerate unsolvable conditions since it explores multiple possible paths in the suspicious program for the LCS computation. CoP also has no difficulty on *control-flow flattening*, *branch swapping*, and *switch-case to if-else conversion obfuscation* since it is based on the path semantics modeling which naturally takes control-flow into consideration (these are also analyzed and illustrated in our evaluation section). It is similar for the obfuscation that *converts multiple returns to one return*. The obfuscation that *replaces logical operators* can be handled by symbolic execution and path semantics modeling with LCS.

Loop rolling and unrolling can be handled to some extent (see Section 8.2.1). *Loop reversing* is also a possible counterattack. However, automatic loop reversing is difficult as it may result in semantically different programs. So far, we are not aware of such tools to our best knowledge. One might manually reverse a loop, but its impact could be very limited in a large program; moreover, it requires a plagiarist understands the loop and involves a lot of manual work, which is laborious and error-prone and violates the initial purpose of plagiarism.

9.1.3 Data-flow Obfuscation

There are two scenarios. The first scenario is that the obfuscation is applied inside a basic block, e.g., bogus variables insertion in a basic block. Since our basic block semantic similarity comparison can tolerate variations in the suspicious block, it has no effect no matter how many bogus variables are inserted in the suspicious block except for increased computation cost. Note the block from the plaintiff is not obfuscated, and the base of block comparison is the plaintiff block. Another possible counterattack is *splitting and merging variables* (see Section 8.5.1). Because CoP does not combine variables during basic block similarity computation, it fails to detect these variables are equivalent. However, this usually occurs at the beginning/end of programs, unless developers adopt complicated methods to convert all operations on variables of the original type to those on variables of the new type, which is usually not practical; thus, it has little impact on detection results.

In the other scenario, obfuscation is applied in a inter-block manner. An example is splitting variables and dispersing each part into several basic blocks. Since we compare semantics at the machine code level and merge multiple block when it is necessary, this attack can be dealt with unless an object is split into two basic blocks far away enough that they are not merged in CoP.

9.1.4 Block Similarity Matching "Error"

To tolerate obfuscation, program transformation, and optimizations, exact matching of block semantics does lead to good result. Thus block matching "error" can be interpreted as a way to accommodate some flexibility in the process due to obfuscation or transformation. Although in theory "error" might accumulate, but in experiment we haven't encountered any problem. It is interesting to investigate in the future whether we can use taint analysis to verify to some extent if an "error" is relevant and filter it out if not.

9.2 Limitations

CoP bears the following limitations. First, CoP is static with symbolic execution and theorem proving. It bears the same limitations as static analysis in general. For example, static analysis has difficulty in handling indirect branches (also known as computed jumps, indirect jumps, and register-indirect jumps). This problem can be addressed to some extent by Value-Set Analysis (VSA) [5]. Moreover, CoP relies on the assumption that the binary code of the plaintiff and suspicious programs can be disassembled. If the binary

TABLE 6
Detection results for six cryptographic algorithms. √ indicates we detect the algorithm in the benchmark. × indicates we do not detect the algorithm in the benchmark. No false positive or false negative is reported.

	cryptlib	openssl	openssh	libgcrypt	truecrypt	berkeley DB	MySQL	git	glibc	p7zip	cmake	thttpd	sthttpd
MD5	√	√	√	√	×	×	√	×	√	×	√	×	×
SHA1	√	√	√	√	√	√	√	√	×	√	√	×	×
SHA2	√	√	√	√	√	×	√	×	√	√	√	×	×
AES	√	√	√	√	√	√	√	×	×	×	×	×	×
RC4	√	√	√	×	×	×	√	×	×	×	×	×	×
Blowfish	√	√	√	√	√	×	√	×	×	×	×	×	×

code cannot be disassembled, CoP cannot analyze them. In addition, a plagiarist can pack or encrypt the original code (e.g., VMProtect [71] and Code Virtualizer [59]); our current tool does not handle such cases. Some dynamic execution tool (e.g., Anubis [7], CWSandbox [74], Renovo [41], PolyUnpack [62], OmniUnpack [51], etc.) can be used to partially address this by capturing unpacked or decrypted code when dynamically executing programs.

Symbolic execution combined with automated theorem proving is powerful, but has its own limitations. For example, for theorem proving, it cannot solve the opaque predicates or unsolved conjectures (e.g., the Collatz conjecture [20], [43]), but the impact could be very limited in large programs. Also, its computational overhead is high. In our experiment with *thttpd* and *sthttpd*, it took as long as an hour to complete, and in our Gecko vs. Firefox experiment, it took half a day. Currently we perform brute-force search to find pairs of semantically equivalent basic blocks with which to start the path exploration and LCS computation. We plan to develop heuristics and optimizations to minimize the calls to the symbolic execution engine and theorem prover in the future.

The detection result of our tool depends on the basic block similarity threshold. There is a trade-off between the basic block similarity threshold, efficiency, and accuracy. It is likely to increase false negative with a higher threshold; in contrast, reducing the threshold may increase false positive. For the efficiency, it depends on the repeating time of the process of finding starting blocks in the suspicious program and the number of starting blocks found in the suspicious program (as we will explore the suspicious program based on each starting block). Unfortunately, without many real-world plagiarism samples, we are unable to show concrete results on the selection of threshold. As such, rather than applying our tool to "prove" software plagiarism, in practice one may use it to collect initial evidences before taking further investigations, which often involve nontechnical actions.

10 Related Work

There is a substantial amount of work on the problem of finding similarity and differences of two files either text or binary. The classic Unix *diff* and *diff3*, and its Windows derivation *Windiff*, compare text files. We discuss the work focusing on finding software semantic difference or similarity.

10.1 Code Similarity Detection

SymDiff [44] is a language-agnostic semantic diff tool for imperative programs. It presents differential symbolic execution that analyzes behavioral differences between different versions of a program. To facilitate regression analysis, Hoffman et al. compared execution traces using LCS [35]. Our work is mainly motivated with obfuscation in the context of plagiarism detection, while these works do not consider obfuscation. Our work is also different from binary diffing tools based mainly on syntax (e.g., bsdiff, bspatch, xdelta, JDiff, etc.). Purely syntax-based methods are not effective in the presence of obfuscation. Some latest binary diffing techniques locate semantic differences by comparing intra-procedural control flow structure [24], [27], [30]. Although such tools have the advantage of being more resistant to instruction obfuscation techniques, they rely heavily on function boundary information from the binary. As a result, binary diffing tools based on control flow structure can be attacked by simple function obfuscation. The tool iBinHunt [53] overcomes this problem by finding semantic differences in inter-procedural control flows. CoP adopts similar basic block similarity comparison method, but goes a step further in this direction by combining block comparison with LCS.

10.2 Software Plagiarism Detection

Static plagiarism detection or clone detection. The existing static analysis techniques except for the birthmark-based techniques are closely related to the clone detection [4], [28], [40], which is a technique to find duplicate code and then decrease code size and facilitate maintenance. While possessing common interests with the clone detection, the plagiarism detection is different in that (1) we must deal with code obfuscation techniques which are often employed with a malicious intention; (2) source code analysis of the suspicious program is less applicable in practice since the source code is often not available for analysis. Static analysis techniques for software plagiarism detection includes string-based [4], AST-based [76], token-based [55], [61], PDG-based [28], [46], and birthmark-based [56], [69]. The methods based on source code are not possible in most cases. In general, this category is not effective when obfuscation can be applied.

Dynamic birthmark based plagiarism detection. Several dynamic birthmarks can be used for plagiarism detection, including API birthmark, system call birthmark, function call birthmark, and core-value birthmark. Tamada et al. [70] proposed an API birthmark for Windows application. Schuler et al. [64] proposed a dynamic birthmark for Java. Wang et al. [72] introduced two system call based birthmarks, which are suitable for programs invoking many different system calls. Jhi et al. [38], [39] proposed a core-value based birthmark for detecting plagiarism. Core-values of a program are constructed from runtime values that are pivotal for the program to transform its input to desired output. A limitation

of core-value based birthmark is that it requires both the plaintiff program and suspicious program to be fed with the same inputs, which sometimes is difficult to meet, especially when only a component of a program is stolen. Thus, core-value approach is not applicable when only partial code is reused.

Dynamic whole program path based plagiarism detection. Myles and Collberg [57] proposed a whole program path (WPP) to detect plagiarism. Although WPP is robust to some semantic-preserving transformations, it is still vulnerable to many obfuscations, e.g., control flow flattening and loop unwinding. Zhang et al. [54], [79] presented a program logic based approach which searches for dissimilarity between two programs by finding path deviations. They used symbolic execution and theorem proving to find certain inputs that will cause the two programs in consideration to behave differently. Our work also captures path semantics, but can tolerate certain deviations caused by obfuscation, resulting more robust obfuscation resiliency.

10.3 Cryptographic Primitives Detection

Previous study on cryptographic primitives detection can be categorized as two approaches. Static detection approaches like DRACA [2], KANAL [1] and Signsrch [3], are based on recognition of syntactic signature (e.g., magic constants, instruction segments). The main drawback is that an attacker can use code obfuscation techniques to impede signature identification. Dynamic detection methods disclose cryptographic primitives by analyzing the execution traces. Lutz [49] proposed to detect cryptographic code by measuring three heuristics in execution traces—existence of loops, large number of bitwise arithmetic operations and entropy variation of tainted data when decrypting. ReFormat [73] relied on the observation that processing ciphertext and plaintext is different; thus it attempts to locate turning point between two phases of traces. However, ReFormat's idea may break down if cryptographic programs do not contain two easily distinguishable phases. Caballero et al. [10] applied similar detection heuristics to extract encrypted protocol messages. With the knowledge of the well known cryptographic algorithms, Gröbert [33] developed several refined search heuristics to improve the detection accuracy. Aligot [14] observed an essential evidence that input-output relationship of cryptographic functions holds even in the presence of obfuscated code. Aligot leveraged this fact to identify cryptographic functions and retrieve their parameters.

10.4 Algorithm Plagiarism Detection

Compared with software plagiarism detection which has been thoroughly discussed in open literature, very little attention has been applied to algorithm plagiarism detection. Building on the core-value based birthmark proposed by Jhi et al. [39], Zhang et al. [78] proposed the methods of n-version programming and annotation to extract the core-value birthmarks for detecting algorithm plagiarism. Zhang's approach is limited to specific algorithms. It relies on extracting runtime values from tainted value-updating instructions, and cannot be applied on some algorithms, e.g., sorting algorithms, algorithms that find a min/max value in an array.

10.5 Mobile App Repackaging Detection

Mobile app repackaging refers to the problem of reverse engineering others' app code and repackage it as a new app, often with injected malicious payload. DroidMOSS [81] detects app repackaging using fuzzy hash. DNADroid [22] uses program dependence graphs to detect repackaging. The efficiency of program dependence graph similarity comparison is further improved in AnDarwin [23]. Juxtapp [34] uses k-grams of opcode sequences to build hash features and then applies a sliding window to detect repackaging. Chen et al. [15] proposes a geometry encoding of control flow graph to detect mobile app clone. Zhou et al. [80] proposes a module decoupling method to partition app code into primary and non-primary modules to identify the malicious payloads reside in the benign apps. Androguard [25] uses Normal Compression Distance for repackaging detection. In general, these methods are not very resilient to code obfuscation techniques. ViewDroid [77] uses the unique mobile app UI features to detect app repacking and makes the method more resilient to general code obfuscation techniques. In addition, Huang et al. [36] proposes a framework for mobile app repackaging evaluation. A recent work by Luo et al. [47] proposes an app repackage-proofing method to protect apps from repackaging.

10.6 Others

Code Obfuscation. Collberg et al. [17] proposed function transformation obfuscation (e.g., function inline and outline) to prevent binary reverse engineering. Linn et al. [45] demonstrated static disassembly is difficult on the Intel x86 platform. This difficulty is further exacerbated by code packing and encryption [68]. Popov et al. [60] obfuscated disassembler by replacing control transfers with exceptions. Sharif et al. [66] encrypted equality conditions that are dependent on inputs with one-way hash functions. All these obfuscation methods improve resistance to syntactic and sometimes semantics based binary code similarity comparison.

Symbolic Path Exploration. Symbolic path exploration [11], [12], [13], [31], [32] combined with test generation and execution is powerful to find software bugs. In our LCS computation, the path exploration has a similar fashion, but we only discover linearly independent paths to cover more blocks with few paths.

11 Conclusion

Identifying similar or identical code fragments among programs is very important in some applications, such as code theft detection. Prior code similarity comparison techniques have limited applicability because they either require source code analysis or cannot handle automated obfuscation tools. In this paper, we introduce a binary-oriented, obfuscation-resilient binary code similarity comparison approach, named CoP, based on a new concept, *longest common subsequence of semantically equivalent basic blocks*, which combines the rigorous program semantics with the flexible longest common subsequence. This novel combination has resulted in more resiliency to code obfuscation. We have developed a prototype. We have analyzed a number of real world program, including *thttpd*, *openssl*, *gzip*, *Gecko*, *cryptlib*, *openssh*,

libgcrypt, *truecrypt*, *berkeley DB*, *MySQL*, *git*, *glibc*, *p7zip*, *cmake*, etc., along with a comprehensive set of obfuscation techniques Semantic Designs Inc.'s C obfuscator, Stunnix's CXX-obfuscator, Diablo, Loco, and CIL. Our experimental results show that CoP can be applied to software plagiarism detection and algorithm detection, and is effective and practical to analyze real-world software.

12 Acknowledgments

This work was supported in part by the NSF Grant CCF-1320605.

References

[1] KANAL—Krypto Analyzer for PEiD. ftp.lightspeedsystems.com/RyanS/utilities/PEiD/plugins/kanal.htm.
[2] DRACA—Draft Cryto Analyzer, 2014. www.literatecode.com/about.
[3] L. Auriemma. Signsrch tool, 2013. aluigi.altervista.org/mytoolz.htm.
[4] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE'95*, pages 86–95, 1995.
[5] G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):23:1–23:84, Aug. 2010.
[6] BAP: The Next-Generation Binary Analysis Platform. http://bap.ece.cmu.edu/, 2013.
[7] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, pages 67–77, 2006.
[8] Bellman–Ford Algorithm. https://en.wikipedia.org/wiki/Bellman-Ford_algorithm.
[9] Binary Diff (bdiff). http://sourceforge.net/projects/bdiff/, 2013.
[10] J. Caballero, P. Poosankam, and C. Kreibich. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *CCS*, 2009.
[11] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*, 2008.
[12] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN'05*, 2005.
[13] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *CCS'06*, 2006.
[14] J. Calvet, J. M. Fernandez, and J.-Y. Marion. Aligot: Cryptographic function identification in obfuscated binary programs. In *CCS*, 2012.
[15] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *36th International Conference on Software Engineering (ICSE)*, 2014.
[16] CLOC—Count Lines of Code. http://cloc.sourceforge.net/, 2013.
[17] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, The Univ. Auckland, 1997.
[18] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1998.
[19] Compuware-IBM Lawsuit. http://news.zdnet.com/2100-3513_22-5629876.html, 2013.
[20] J. H. Conway. Unpredictable iterations. In *Proceedings of the Number Theory Conference*, pages 49–52. Univ. Colorado, Boulder, Colo., 1972.
[21] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, third edition, 2009.
[22] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on Android markets. In *ESORICS*, pages 37–54, 2012.
[23] J. Crussell, C. Gibler, and H. Chen. Scalable semantics-based detection of similar Android applications. In *ESORICS*, 2013.
[24] DarunGrim: A patch analysis and binary diffing tool. http://www.darungrim.org/, 2013.
[25] A. Desnos and G. Gueguen. Android: From reversing to decompilation. In *Black hat, Abu Dhabi*, 2011.
[26] Diablo: code obfuscator. http://diablo.elis.ugent.be/obfuscation, 2013.
[27] H. Flake. Structural comparison of executable objects. In *Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2004.
[28] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE'08*, pages 321–330, 2008.
[29] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV'07*, 2007.
[30] 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)*, pages 238–255, 2008.
[31] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI'05*, 2005.
[32] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS'08*, 2008.
[33] F. Gröbert, C. Willems, and T. Holz. Automated identification of cryptographic primitives in binary programs. In *RAID*, 2012.
[34] 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.
[35] K. J. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. In *PLDI'09*, 2009.
[36] 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*, 2013.
[37] M. Jacob, M. H. Jakubowski, P. Naldurg, C. W. Saw, and R. Venkatesan. The superdiversifier: Peephole individualization for software protection. In *Proceedings of the 3rd International Workshop on Security (IWSEC)*, pages 100–120, Kagawa, Japan, 2008.
[38] 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*, 41(9):925–943, Sept 2015.
[39] 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 *ICSE'11*, pages 756–765, Waikiki, Honolulu, 2011.
[40] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, pages 654–670, 2002.
[41] M. 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)*, pages 46–53, 2007.
[42] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley, 2005.
[43] J. C. Lagarias. The ultimate challenge: The 3x+1 problem. *American Mathematical Soc.*, 2010.
[44] S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebelo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *CAV'12*, 2012.
[45] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS'03*, 2003.
[46] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *KDD*, 2006.
[47] 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.
[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 *FSE*, 2014.
[49] N. Lutz. Towards revealing attackers' intent by automatically decrypting network traffic. In *Master's thesis, ETH Zürich*, 2008.
[50] M. Madou, L. V. Put, and K. D. Bosschere. LOCO: an interactive code (de)obfuscation tool. In *PERM'06*, pages 140–144, 2006.
[51] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the 2007 Annual Computer Security Applications Conference(ACSAC)*, pages 431–441. IEEE Computer Society, 2007.
[52] T. J. McCabe. Structured testing: A software testing methodology using the cyclomatic complexity metric. *NIST Special Publication 500-99*, 1982.
[53] J. Ming, M. Pan, and D. Gao. iBinHunt: Binary hunting with inter-procedural control flow. In *Proc. of the 15th Annual Int'l Conf. on Information Security and Cryptology (ICISC)*, 2012.

[54] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu. Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection. *IEEE Transactions on Reliability*, PP(99):1–18, 2016.
[55] MOSS: A System for Detecting Software Plagiarism. http://theory.stanford.edu/~aiken/moss/, 2013.
[56] G. Myles and C. Collberg. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing*, 2005.
[57] G. Myles and C. S. Collberg. Detecting software theft via whole program path birthmarks. In *ISC'04*, pages 404–415, 2004.
[58] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC'02*, pages 213–228, 2002.
[59] Oreans Technologies. Code Virtualizer: Total obfuscation against reverse engineering. http://oreans.com/codevirtualizer.php, 2013.
[60] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *USENIX Security '07*, 2007.
[61] L. Prechelt, G. Malpohl, and M. Phlippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, Univ. of Karlsruhe, 2000.
[62] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of 2006 Annual Computer Security Applications Conference (ACSAC)*, pages 289–300. IEEE Computer Society, 2006.
[63] 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*, pages 76–85, 2003.
[64] D. Schuler, V. Dallmeier, and C. Lindig. A dynamic birthmark for Java. In *ASE'07*, pages 274–283, 2007.
[65] Semantic Designs, Inc. http://www.semdesigns.com/, 2013.
[66] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS '08*, 2008.
[67] Stunnix, Inc. http://www.stunnix.com/, 2013.
[68] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, February 2005.
[69] H. Tamada, M. Nakamura, A. Monden, and K. ichi Matsumoto. Design and evaluation of birthmarks for detecting theft of Java programs. In *Proc. IASTED International Conference on Software Engineering (IASTED SE 2004)*, pages 569–575, February 2004. Innsbruck, Austria.
[70] H. Tamada, K. Okamoto, M. Nakamura, and A. Monden. Dynamic software birthmarks to detect the theft of Windows applications. In *Int'l Sym. Future Software Technology (ISFST'04)*, 2004.
[71] VMProtect Software. VMProtect software protection. http://vmpsoft.com, last reviewed, 02/20/2013.
[72] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. In *CCS'09*, pages 280–290, 2009.
[73] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. ReFormat: Automatic reverse engineering of encrypted messages. In *European Symposium on Research in Computer Security (ESORICS)*, 2009.
[74] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. In *IEEE Security and Privacy, v.5 n.2, p.32-39*, 2007.
[75] Z. Xin, H. Chen, X. Wang, P. Liu, S. Zhu, B. Mao, and L. Xie. Replacement attacks on behavior based software birthmark. In *ISC'11*, pages 1–16, 2011.
[76] W. Yang. Identifying syntactic differences between two programs. In *Software–Practice & Experience*, pages 739–755, New York, NY, USA, 1991.
[77] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec '14, pages 25–36, 2014.
[78] F. Zhang, Y.-C. Jhi, D. Wu, P. Liu, and S. Zhu. A first step towards algorithm plagiarism detection. In *ISSTA'12*, pages 111–121, 2012.
[79] F. Zhang, D. Wu, P. Liu, and S. Zhu. LoPD: Logic-based software plagiarism detection. Submitted for publication, 2013.
[80] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 185–196. ACM, 2013.
[81] 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 (CODASPY '12)*, pages 317–326, 2012.