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

Lannan Luo Jiang Ming Dinghao Wu Peng Liu Sencun Zhu
The Pennsylvania State University
University Park, PA 16802, USA
{{lzl144,jum310,dwu,pliu}@ist,szhu@cse}.psu.edu

ABSTRACT

Existing code similarity comparison methods, whether source or binary code based, are mostly not resilient to obfuscations. In the case of software plagiarism, emerging obfuscation techniques have made automated detection increasingly difficult. In this paper, we propose a binary-oriented, obfuscation-resilient 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 semantics equivalence (and similarity) of two blocks can be checked by a theorem prover. We then model the semantics 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 and our experimental results show that our method is effective and practical when applied to real-world software.

Categories and Subject Descriptors

D.2.m [**Miscellaneous**]: Software protection

General Terms

Security, Verification

Keywords

Software plagiarism detection, binary code similarity comparison, obfuscation, symbolic execution, theorem proving

1. INTRODUCTION

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 [15]. As a result, IBM paid $140 million in fines to license Compuware's software and an additional $260 million to purchase Compuware's services. Examples such as this point to a critical need for computer-aided, automated software plagiarism detection techniques that are capable of measuring code similarity.

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. Detecting code similarity is faced with an increasing challenge caused by emerging, readily available code obfuscation techniques, 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.

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 [38]) assumes the availability of source code and minimal code obfuscation. Binary similarity detection (e.g., Bdiff [8]) is binary code-based, but it does not consider obfuscation in general and hence is not obfuscation resilient. Software plagiarism detection approaches based on dynamic system call birthmarks [53, 54] 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 [56]. Another approach based on core value analysis [29] 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 semantics 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 [38], JPLag [44], Bdiff [8] and DarunGrim2 [19], 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, and can be applied to real-world software to detect code reuse or software plagiarism.

In summary, we make the following contributions.

- We propose CoP, a binary-oriented, obfuscation-resilient method for software plagiarism or code reuse detection, with strong obfuscation resiliency.
- 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 multiple paths in the plaintiff and suspicious programs and calculate the LCS scores between corresponding paths. The implementation and experimental results are presented in Section 6. We analyze the obfuscation resiliency and discuss the limitations in Section 7. The related work is discussed in Section 8 and the conclusion follows in Section 9.

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. Obviously, if two programs have identical or similar syntax, they behave similarly, but not vice versa. 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. This method is also not very robust against obfuscations as an adversary can replace system calls.

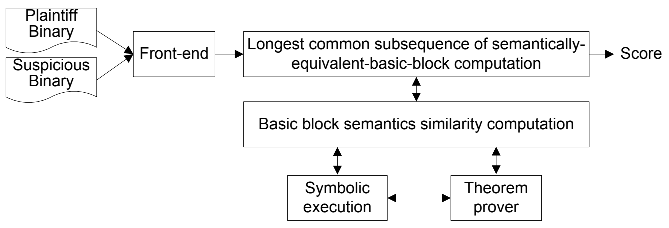

Plaintiff Binary
Suspicious Binary
Front-end
Longest common subsequence of semantically-equivalent-basic-block computation
Score
Basic block semantics similarity computation
Symbolic execution
Theorem prover

Figure 1: Architecture

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 judgments in formal logics, 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. The first is to model semantics formally at the binary code basic block level. We not only model basic block semantics equivalence, but also model similarity semantically. The second is the longest common subsequence. Based on the basic block semantics 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 semantics 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 semantics similarity between the plaintiff (component) and the suspicious. The semantics similarity is modeled at two levels, one at the binary code basic block level and one at the path level. 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 component is reused, with obfuscation or not, legally or illegally.

2.2 Architecture

The architecture of CoP is shown in Figure 1. The inputs are the binary code of the plaintiff (component) and suspi-

[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.

cious program. The front-end then disassembles the binary code, builds an intermediate representation, and constructs control-flow graphs and call graphs. We then compute the longest common subsequence (LCS) of semantically equivalent basic blocks (SEBB). We explore 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 Figure 1 is for this purpose. We rely on symbolic execution to get symbolic formulas representing the input-output relation of a basic block. Specifically, we compute 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 small 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 semantics similarity and equivalence computation are 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 [24] 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.

```
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

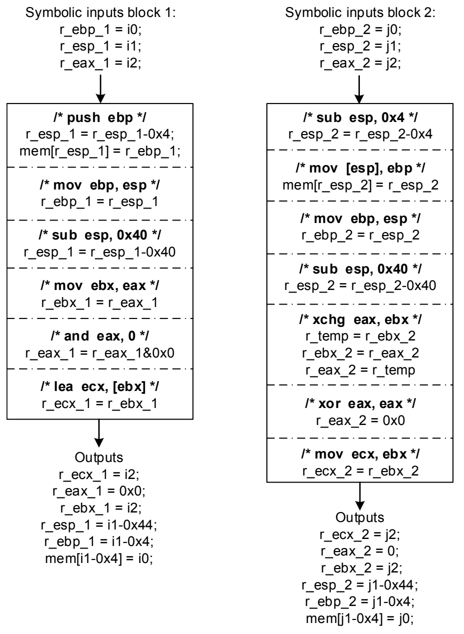
Since we do not know in general which input variable in one block corresponds to which input variable in the other block, we need to try different combinations of input variables. We define a pair-wise equivalence formula for the input variable combinations as follows.

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.



```
Symbolic inputs block 1:
r_ebp_1 = i0;
r_esp_1 = i1;
r_eax_1 = i2;

/* push ebp */
r_esp_1 = r_esp_1-0x4;
mem[r_esp_1] = r_ebp_1;
/* mov ebp, esp */
r_ebp_1 = r_esp_1
/* sub esp, 0x40 */
r_esp_1 = r_esp_1-0x40
/* mov ebx, eax */
r_ebx_1 = r_eax_1
/* and eax, 0 */
r_eax_1 = r_eax_1&0x0
/* lea ecx, [ebx] */
r_ecx_1 = r_ebx_1

Outputs
r_ecx_1 = i2;
r_eax_1 = 0x0;
r_ebx_1 = i2;
r_esp_1 = i1-0x44;
r_ebp_1 = i1-0x4;
mem[i1-0x4] = i0;
```

```
Symbolic inputs block 2:
r_ebp_2 = j0;
r_esp_2 = j1;
r_eax_2 = j2;

/* sub esp, 0x4 */
r_esp_2 = r_esp_2-0x4
/* mov [esp], ebp */
mem[r_esp_2] = r_ebp_2
/* mov ebp, esp */
r_ebp_2 = r_esp_2
/* sub esp, 0x40 */
r_esp_2 = r_esp_2-0x40
/* xchg eax, ebx */
r_temp = r_ebx_2
r_ebx_2 = r_eax_2
r_eax_2 = r_temp
/* xor eax, eax */
r_eax_2 = 0x0
/* mov ecx, ebx */
r_ecx_2 = r_ebx_2

Outputs
r_ecx_2 = j2;
r_eax_2 = 0;
r_ebx_2 = j2;
r_esp_2 = j1-0x44;
r_ebp_2 = j1-0x4;
mem[j1-0x4] = j0;
```

Figure 2: Basic block symbolic execution

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*

$$\forall y_1 \in Y_1.\ \exists y_2 \in Y_2, p(X_1, \pi(X_2)).$$
$$p(X_1, \pi(X_2)) \implies f_1(X_1) = f_2(X_2).$$

where $f_1(X_1)$ and $f_2(X_2)$ are the symbolic formulas of y_1 and y_2 obtained by symbolic execution, respectively.

Each output equivalence formula is checked by a theorem prover. 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. Assume there are n output variables in the plaintiff block, and we find p of them have semantically equivalent counterparts in the suspicious block. The semantics embedding of the plaintiff into the suspicious is calculated as p/n.

Another method is to consider all the input variable pairwise permutations and check output variable equivalence simultaneously. This method leads to more than exponential number of theorem prover invocations instead of quadraticon the number of output variables, for each input variable combination, which is another drawback of the aforementioned strictly equivalence checking method.

3.4 Example

Our tool works on binary code. Figure 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 [19] and Bdiff [8], 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 semantics equivalence checking (i.e., similarity score above a threshold), we calculate a path embedding score for each linearly independent path [36, 55] of the plaintiff against the suspicious program using the LCS of semantically equivalent basic blocks.

4.1 Starting Blocks

The LCS of semantically equivalent basic blocks computation is based on the modified longest path algorithm to explore paths from starting points. We present how to identify the starting points from the plaintiff and suspicious programs, and with two starting points how to explore linearly independent paths to compute the highest LCS score (i.e., the LCS score of the longest path). It is important to choose the starting point so that the path exploration is not misled to irrelevant code parts in the plaintiff and suspicious programs. We first look for the starting block inside a function of the plaintiff program. This function can be randomly chosen or picked by an investigator with pre-knowledge of the plaintiff code. 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 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 program, and the process is repeated.

4.2 Linearly Independent Paths

At the path level, we select a set of *linearly independent paths* [36, 55] from the plaintiff program. For each selected path, we compare it against the suspicious program to calculate a path embedding score by computing the LCS of semantically equivalent basic blocks. Once the starting block of the plaintiff program and several candidate starting blocks of the suspicious program are identified, the next is to explore paths to calculate a path embedding score. For the plaintiff program, we select a set of linearly independent paths, a concept developed for path testing, from the starting block. A linearly independent path is a path that introduces at least one new node that is not included in any other linearly independent paths. We first unroll each loop in the plaintiff program once, and then adopt the Depth First Search algorithm to find a set of linearly independent paths from the plaintiff program.

4.3 Longest Common Subsequence of Semantically Equivalent Basic Blocks

The longest common subsequence between two sequences can be computed with dynamic programming algorithms [17]. However, we need to find the highest LCS score between a plaintiff 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

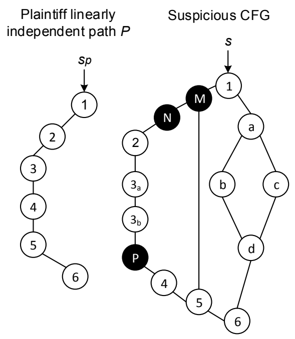
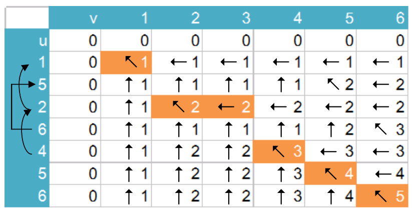


Figure 3: An example for path similarity calculation. The black blocks are inserted bogus blocks. There is an opaque predicate inserted in M that always evaluates to true at runtime which makes the direct flow to the node 5 infeasible.



	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

Figure 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.

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 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.

Algorithm 1 shows the pseudo-code for the Longest Common Subsequence of Semantically Equivalent Basic Blocks computation. The inputs are a linearly independent path P of the plaintiff program, the suspicious program G, and a starting point s 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 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 [17, p. 395, Fig. 15.8]. The

Algorithm 1 Longest Common Subsequence of Semantically Equivalent Basic Blocks

```
    $\delta$: the LCS dynamic programming memoization table
    $r$: the current row of the $\delta$ table
    $\gamma$: the direction table for the LCS search
    $\sigma$: the array stores the intermediate LCS scores
    $n$: the length of the plaintiff path P

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

19: function LCS(u,P)
20:     $\delta(u,0) = 0$
21:     for each node v of P do
22:         if SEBB(u,v) then // semantically eq. blocks
23:             $\delta(u,v) = \delta(\text{parent}(u), \text{parent}(v)) + 1$
24:             $\gamma(u,v) = \nwarrow$
25:             if $\sigma(u) < \delta(r,v)$ then
26:                 r++
27:             end if
28:         else
29:             $\delta(u,v) = \max(\delta(\text{parent}(u),v), \delta(u,\text{parent}(v)))$
30:             $\gamma(u,v) = \leftarrow$ or $\uparrow$
31:         end if
32:         if $\sigma(u) < \delta(r,v)$ then
33:             $\sigma(u) = \delta(r,v)$
34:             enq(u,Q)
35:         end if
36:     end for
37: end function
```

vector σ is used to store the intermediate highest LCS scores for each node.

The inputs of the function LCS() are a node u of the suspicious program and the linearly independent path P. 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 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 Figure 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 Figure 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 Figure 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 P 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 Figure 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 1, we cannot find a node in P 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 P; 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 P; hence, a new row is added to δ (Line 26; see the third row in Figure 4) and it is inserted into the working queue after updating its score. For node a, we cannot find a node in P 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 P and it gets a new score higher than its original one; thus, a new row is added into δ (see the forth row in Figure 4). To calculate its new sore, 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 Figure 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 Figure 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 P 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 P. To address this, the LCS refinement which tentatively merges unmatched blocks has been developed.

4.4 Refinement

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

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.

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 in the suspicious through backtracking. Then for each such sequence or list, CoP merges all basic blocks, as well as two basic blocks which are right before and after the sequence, into one code trunk. Finally, 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. One may wonder when blocks are merged the intermediate effects stored on the stack may be missed. However, since we consider both memory cells and registers as input and output variables of basic blocks, the intermediate effects are not missed.

5. FUNCTION AND PROGRAM SIMILARITY COMPARISON

Once we have computed the path similarity scores (the lengths of the resulted LCS), we calculate the similarity score between the two functions. 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 (see Section 4.1), and the similarity score of this function is the highest one among those. After we calculate the similarity scores of the selected plaintiff functions, 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.

6. IMPLEMENTATION AND EVALUATION

6.1 Implementation

Our prototype implementation consists of 4,312 lines of C++ code measured with CLOC [13]. The front-end of CoP disassembles the plaintiff and suspicious binary code based on IDA Pro. The assembly code is then passed to BAP [7] to build an intermediate representation the same as that used in BinHunt [24], and to construct CFGs and call graphs. 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 [23] for the equivalence checking of symbolic formulas representing the basic block semantics.

6.2 Experimental Settings

We evaluated our tool on a set of benchmark programs to measure its obfuscation resiliency and scalability. We conducted experiments on 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 [38], JPLag [44], Bdiff [8] and DarunGrim2 [19], 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 [45] 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 [19] 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 require the selected linearly independent paths cover at least 80% of the plaintiff program.

6.3 Thttpd

The purpose of the experiments of *thttpd*, *openssl* (see Section 6.4), and *gzip* (see Section 6.5) is to measure the obfuscation resiliency of our tool. In our first experiment, we evaluated *thttpd-2.25b* [43] and *sthttpd-2.26.4* [48], 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.*

In all of our experiments, we select 10% of the functions in the plaintiff program (or component) 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).

6.3.1 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 [19], a state-of-the-art patch analysis and binary diffing tool. Figure 5 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 Figure 5, we can see our tool CoP is quite effective compared to DarunGrim2 [19]. Both CoP and DarunGrim2 have good results when the same level of optimizations is applied (see the left half of Figure 5). 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 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

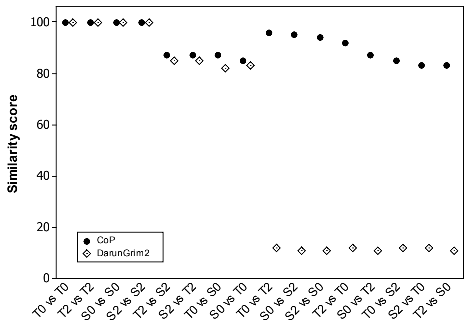


Figure 5: Code similarity scores resulting from different compiler optimization levels. Higher is better since these two programs share codebase. (Legend: T*i* and S*i* stand for *thttpd* and *sthttpd* compiled with -O*i*, respectively.)

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. 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. In the second scenario, where the inlined/outlined function is a dynamically linked library function (e.g., *strspn()*), CoP is not able to "inline" the callee function, which may result in a lower similarity score. 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.

6.3.2 Resilience to Transformation via Different Compilers

We also tested CoP on code compiled with different compilers and compared with DarunGrim2 [19]. 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). Figure 6 shows the detection results. With different 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.

6.3.3 Resilience to Code Obfuscations

To evaluate the obfuscation resiliency, we used two commercial products, Semantic Designs Inc.'s C obfuscator [47]

Table 1: Detection results (resilience to single code obfuscation)

Obfuscation		Similarity score (%)				
Category	Transformation	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 2: 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

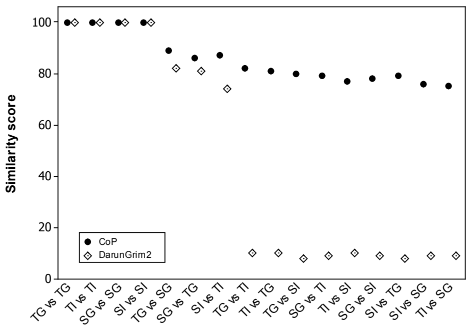


Figure 6: Code similarity scores resulting from different compilers. Higher is better since these two programs share codebase. (Legend: P*c* stands for program *P* compiled with compiler *c*, where *P* is either *T* for *thttpd* or *S* for *sthttpd*, and *c* is either *G* for GCC or *I* for ICC, respectively.)

and Stunnix's CXX-obfuscator [49], as the source code obfuscation tools, and two open-source products, Diablo [20] and Loco [35], as the binary code obfuscation tools. We also utilized CIL [39] 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 [14]. Each category contains different obfuscation transformations. We chose 13 typical obfuscation transformations [14] 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 [38], JPLag [44], DarunGrim2 [19] and Bdiff [8]. We evaluated on code with a single and multiple obfuscations applied. The single and multiple obfuscation results are shown in Table 1 and Table 2, 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 [14], 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.

6.3.4 *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.

6.4 Openssl

This experiment also aims to measure the obfuscation resiliency. We first evaluated *openssl-1.0.1f* [41], *openssh-6.5p1* [40], *cyrus-sasl-2.1.26* [18], and *libgcrypt-1.6.1* [33], where *openssh-6.5p1*, *cyrus-sasl-2.1.26*, and *libgcrypt-1.6.1* use the library *libcrypto.a* from *openssl-1.0.1f*. We tested our tool on completely irrelevant programs to measure false positives. These programs include *attr-2.4.47* [3] and *acl-2.2.52* [1].

In this experiment, we test whether the suspicious programs contain an MD5 component from the plaintiff program *openssl-1.0.1f*. 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*. 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 87% 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 2%) were reported.

6.5 Gzip

In our third experiment, we first evaluated our tool on *gzip-1.6* [27] 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* [9] and *advanceCOMP-1.18* [2], to measure false positives.

The plaintiff program *gzip-1.6* was compiled by GCC with the -O2 optimization level. To measure obfuscation resiliency, we conducted the similar experiments as on *thttpd* and *openssl*. 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. Moreover, when various obfuscation were applied, the average similarity score only reduced around 9%, 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 2%) were reported. Because we have presented a detailed analysis of 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.

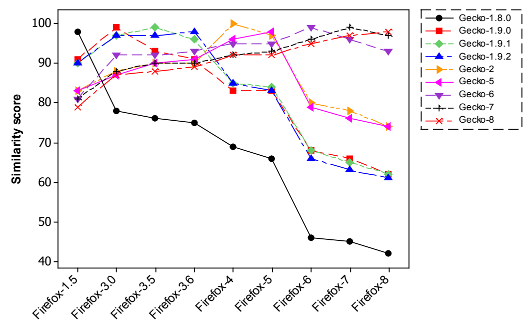


Figure 7: Gecko vs. Firefox (%)

6.6 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 select 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. We cross checked each pair of them and the results are shown in Figure 7. 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 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.

7. DISCUSSION

7.1 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 [14].

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.

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. *Rewriting a loop* including reversing a loop and unrolling a loop is another possible counterattack.

However, automatic loop rewriting is very difficult because they could result in semantically different programs. So far, we are not aware of such tools to our best knowledge. One might manually reverse or unroll 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 compromises the initial purpose of plagiarism.

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 blocks. Since our basic block semantics 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. In the other scenario, obfuscation is applied in a inter-block manner. An example is splitting structure objects 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.

7.2 Limitations

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, 6]. In addition, a plagiarist can pack or encrypt the original code (e.g., VMProtect [52] and Code Virtualizer [42]); our current tool does not handle such cases.

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 [16, 31]), 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.

8. RELATED WORK

There is a substantial amount of work on the problem of finding similarity and differences of two files whether 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.

8.1 Code Similarity Detection

SymDiff [32] 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. [28] compared execution traces using LCS. 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 [21, 19, 24]. 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. iBinHunt [37] 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.

8.2 Software Plagiarism Detection

Mainly motivated by obfuscation, our work compares better with software plagiarism detection work. We discuss here the more relevant research along several axes.

Static plagiarism detection or clone detection. This type work includes string-based [4], AST-based [51, 57] token-based [30, 38, 44], and PDG-based [34, 22]. The methods based on source code are less applicable in practice since the source code of the suspicious program is often not available for analysis. 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. [50] proposed an API birthmark for Windows application. Schuler et al. [46] proposed a dynamic birthmark for Java. Wang et al. [53, 54] introduced two system call based birthmarks, which are suitable for programs invoking many different system calls. Jhi et al. [29] 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. Zhang et al. [58] further proposed the methods of n-version programming and annotation to extract the core-value birthmarks for detecting algorithm plagiarism. 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.

8.3 Others

Symbolic path exploration [11, 25, 26, 12, 10] 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 [36, 55] to cover more blocks with few paths.

9. CONCLUSION

In this paper, we introduce a binary-oriented, obfuscation-resilient software plagiarism detection 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. Our experimental results show that CoP is effective and practical when applied to real-world software.

10. ACKNOWLEDGMENTS

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

11. REFERENCES

[1] ACL. http://savannah.nongnu.org/projects/acl, 2013.

[2] AdvanceCOMP. http://advancemame.sourceforge.net/comp-download.html.

[3] Attr. http://savannah.nongnu.org/projects/attr, 2013.

[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] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 202–213, 2005.

[7] BAP: The Next-Generation Binary Analysis Platform. http://bap.ece.cmu.edu/, 2013.

[8] Binary Diff (bdiff). http://sourceforge.net/projects/bdiff/, 2013.

[9] Bzip2. http://bzip.org/index.html, 2010.

[10] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*, 2008.

[11] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN'05*, 2005.

[12] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *CCS'06*, 2006.

[13] CLOC—Count Lines of Code. http://cloc.sourceforge.net/, 2013.

[14] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, The Univ. Auckland, 1997.

[15] Compuware-IBM Lawsuit. http://news.zdnet.com/2100-3513_22-5629876.html, 2013.

[16] J. H. Conway. Unpredictable iterations. In *Proceedings of the Number Theory Conference*, pages 49–52. Univ. Colorado, Boulder, Colo., 1972.

[17] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, third edition, 2009.

[18] Cyrus. http://asg.web.cmu.edu/sasl/, 2013.

[19] DarunGrim: A patch analysis and binary diffing tool. http://www.darungrim.org/, 2013.

[20] Diablo: code obfuscator. http://diablo.elis.ugent.be/obfuscation, 2013.

[21] H. Flake. Structural comparison of executable objects. In *Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2004.

[22] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE'08*, pages 321–330, 2008.

[23] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV'07*, 2007.

[24] 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.

[25] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI'05*, 2005.

[26] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS'08*, 2008.

[27] Gzip. http://www.gzip.org/index.html, 2013.

[28] K. J. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. In *PLDI'09*, 2009.

[29] 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.

[30] J.-H. Ji, G. Woo, and H.-G. Cho. A source code linearization technique for detecting plagiarized programs. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE)*, pages 73–77, 2007.

[31] J. C. Lagarias. The ultimate challenge: The 3x+1 problem. *American Mathematical Soc.*, 2010.

[32] S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebelo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *CAV'12*, 2012.

[33] Libgcrypt. http://www.gnu.org/software/libgcrypt/, 2011.

[34] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *KDD*, 2006.

[35] M. Madou, L. V. Put, and K. D. Bosschere. LOCO: an interactive code (de)obfuscation tool. In *PERM'06*, pages 140–144, 2006.

[36] T. J. McCabe. Structured testing: A software testing methodology using the cyclomatic complexity metric. *NIST Special Publication 500-99*, 1982.

[37] 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.

[38] MOSS: A System for Detecting Software Plagiarism. http://theory.stanford.edu/~aiken/moss/, 2013.

[39] 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.

[40] OpenSSH. http://www.openssh.com/, 2014.

[41] OpenSSL. http://www.openssl.org/source/, 2014.

[42] Oreans Technologies. Code Virtualizer: Total obfuscation against reverse engineering. http://oreans.com/codevirtualizer.php, 2013.

[43] J. Poskanzer. thttpd, 2013. http://www.acme.com/software/thttpd/.

[44] L. Prechelt, G. Malpohl, and M. Phlippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, Univ. of Karlsruhe, 2000.

[45] 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.

[46] D. Schuler, V. Dallmeier, and C. Lindig. A dynamic birthmark for Java. In *ASE'07*, pages 274–283, 2007.

[47] Semantic Designs, Inc. http://www.semdesigns.com/, 2013.

[48] sthttpd. http://opensource.dyc.edu/sthttpd, 2013.

[49] Stunnix, Inc. http://www.stunnix.com/, 2013.

[50] 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.

[51] N. Truong, P. Roe, and P. Bancroft. Static analysis of students' Java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education*, pages 317–325, Darlinghurst, Australia, 1991.

[52] VMProtect Software. VMProtect software protection. `http://vmpsoft.com`, last reviewed, 02/20/2013.

[53] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. In *CCS'09*, pages 280–290, 2009.

[54] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *ACSAC'09*, pages 149–158, 2009.

[55] A. H. Watson and T. J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST Special Publication 500-235*, 1996.

[56] 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.

[57] W. Yang. Identifying syntactic differences between two programs. In *Software–Practice & Experience*, pages 739–755, New York, NY, USA, 1991.

[58] 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.