A First Step Towards Algorithm Plagiarism Detection

Fangfang Zhang
Department of Computer Science and Engineering
Penn State University
University Park, PA, USA
fuz104@cse.psu.edu

Yoon-Chan Jhi
Network & Client Lab.
SW R&D Center
Samsung SDS Co., Ltd.
Korea
yoonchan.jhi@samsung.com

Dinghao Wu
College of Information Sciences and Technology
Penn State University
University Park, PA, USA
dwu@ist.psu.edu

Peng Liu
College of Information Sciences and Technology
Penn State University
University Park, PA, USA
pliu@ist.psu.edu

Sencun Zhu
Department of Computer Science and Engineering
Penn State University
University Park, PA, USA
szhu@cse.psu.edu

ABSTRACT

In this work, we address the problem of *algorithm plagiarism*, which occurs when a plagiarist, violating intellectual property rights, steals others' algorithms and covertly implements them. In contrast to software plagiarism, which has been extensively studied, limited attention has been paid to algorithm plagiarism. In this paper, we propose two dynamic value-based approaches, namely *N-version* and *annotation*, for algorithm plagiarism detection. Our approaches are motivated by the observation that there exist some critical runtime values which are irreplaceable and uneliminatable for all implementations of the same algorithm. The N-version approach extracts such values by filtering out non-core values. The annotation approach leverages auxiliary information to flag important variables which contain core values. We also propose a value dependence graph based similarity metric in addition to the longest common subsequence based one, in order to address the potential value reordering attack. We have implemented a prototype and evaluated the proposed schemes on various algorithms. The results show that our approaches to algorithm plagiarism detection are practical, effective and resilient to many automatic obfuscation techniques.

Categories and Subject Descriptors

K.4.1 [**Computers and Society**]: Public Policy Issues—*Intellectual property rights*; D.2.8 [**Software Engineering**]: Metrics

General Terms

Security, Design, Experimentation, Legal Aspects

ISSTA '12, July 15-20, 2012, Minneapolis, MN, USA

Keywords

Algorithm plagiarism detection, software plagiarism detection, program slicing, N-version programming

1. INTRODUCTION

In recent years, plagiarism has raised great concern over intellectual property protection. Plagiarists violate intellectual property rights either by copying source/binary code or by stealing and covertly implementing protected algorithms. The first case is also known as software plagiarism, which has been thoroughly discussed in many literatures [17, 23, 24, 26, 34, 37, 32, 36]. However, very little attention has been paid to the second case, namely *algorithm plagiarism*.

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

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

In fact, algorithm plagiarism detection is more challenging than software plagiarism detection. A major reason is that an algorithm can be independently implemented in different ways by different programmers in different programming languages. These implementation processes involve human intelligence, coding style and creativity, which generate a lot of diversities in the resulted code. These diversities are hard to be described formally and can cause two programs implementing the same algorithm to appear dramatically different from each other. As a result, how to "peel off" these diversities and to capture the essential code-level characteristics of an algorithm remains a big challenge. In contrast, the diversities caused by software plagiarism assisted by automatic code obfuscation tools can be filtered out through birthmarks and structural features [17, 23, 24, 26, 34, 37, 36, 32]. In other words, the gap between essential characteristics of an algorithm and the (static/dynamic) exhibition of the algorithm implementations is much larger than the gap between the (static/dynamic) exhibition of a program and that of the obfuscated versions of the program.

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

In real-world, the verdict of an algorithm plagiarism case is often algorithm specific, therefore we believe it is more meaningful to provide an algorithm-level similarity score than to draw a simple yes/no conclusion in algorithm plagiarism detection. We also realize that no universal detection threshold can fit all algorithm plagiarism cases because the potential threshold for each case may vary due to the algorithm specific factors, e.g., how complex the algorithm is, how specific it is described, etc.

The main contributions of this paper are as follows: (1) to the best of our knowledge, this work is the first one on algorithm level similarity assessment. (2) We innovatively apply the idea of N-version programming in plagiarism detection. (3) We also propose a novel approach that leverages auxiliary information to extract core values, namely the annotation approach. We can do both manual annotation and automatic annotation. Manual annotation is more accurate while automatic annotation is more efficient. (4) Neither of our two approaches requires the source code of a suspicious program. (5) Besides the LCS similarity metric, we propose to use VDG to measure algorithm level similarity as well. VDG can effectively defend against value reordering attacks. The evaluation results show that our approaches to algorithm plagiarism detection are practical, effective and resilient to many automatic obfuscation techniques.

The rest of the paper is organized as follows. We state the problem in Section 2. Section 3 introduces the signature selection. Section 4 describes the proposed approaches. Section 5 addresses the recording problem. Section 6 presents the implementation and the evaluation, followed by discussion in Section 7. Finally, related work and conclusions are presented in Section 8 and 9.

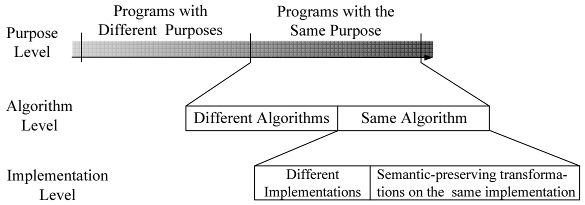

Purpose Level
Programs with Different Purposes
Programs with the Same Purpose
Algorithm Level
Different Algorithms
Same Algorithm
Implementation Level
Different Implementations
Semantic-preserving transformations on the same implementation

Figure 1: The spectrum of program similarity

2. PROBLEM STATEMENT

The similarity between programs can be reflected at different abstraction levels, including purpose level, algorithm level and implementation level, as shown in a similarity spectrum in Figure 1. On the implementation level, software plagiarism detection, which has been well studied in the literature, focuses on separating semantic-preserving transformations/obfuscations (of a program) from independent programming. Whenever software plagiarism exists, the plaintiff program and the plagiarist's program use the same algorithm. Hence, software plagiarism is a problem within "the same algorithm" scope. Beyond this scope, algorithm plagiarism detection aims to see whether two programs implement different algorithms or not. To the best of our knowledge, there was no previous work discussing similarity assessment on the algorithm level.

The goal of our work is to automatically detect algorithm plagiarism, i.e., given one (or more) implementation(s) of a plaintiff algorithm and one suspicious program, the proposed methods can automatically assess their algorithm-level similarity. As mentioned previously, there is no universal similarity threshold for all algorithm plagiarism cases. Therefore, instead of giving a yes/no answer, our approach provides users with similarity scores between programs and lets users make their own decisions.

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

3. SIGNATURE SELECTION

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

3.1 Signature Candidates

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

System call sequence/graph is an essential characteristic of a program that invokes many system calls. However, an examination of the algorithms listed in the "*Algorithm Design*" book [20] indicates that few of these algorithms involve system calls. This indicates that system call sequence/graph is not suitable to characterize an algorithm.

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

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

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

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

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

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

3.2 Core Values

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

Jhi et al. [17] have demonstrated that core values exist at implementation level. Our experiments in Section 6 demonstrate the existence of core values at algorithm level. The approach used to extract program's core values by Jhi et al. [17] is not suitable to obtain core values at algorithm level, since some of program's core values are not core values of the algorithm behind this program. Consider two programs independently implementing the same algorithm, the core values of the programs may be different, but the core values of the algorithm should remain the same.

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

4. OUR APPROACHES

In Section 3, we show that core values are a signature of an algorithm implemented in a program. The next challenge is how to extract core values from a program.

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

4.1 N-version Approach

The *N-version* approach is inspired by N-version programming [9, 4]. We use this approach to filter out the diversities in independent implementations of the same algorithm while keeping the persistent runtime values.

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

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

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

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

Value sequence extractor extracts refined value sequence of programs. The same extractor is also used in

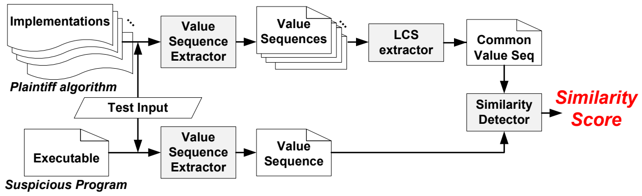


Figure 2: The design of N-version approach

our previous work [17]. This component first leverages the dynamic taint analysis technique [28] to only preserve the runtime values derived from input. We run a program in a virtual machine environment with the input as the taint seed. The dynamic taint analyzer monitors the taint propagation from the taint seed to registers and memory cells. Registers and memory cells are tainted if they appear in destination operands of any instructions that take values from tainted registers or tainted memory locations as input. The output values in these tainted destination operands are appended into a value sequence.

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

- *Value-updating instructions only.* A value-updating instruction is a machine instruction that does not preserve input in its output. For example, `add` is a value-updating instruction, while `mov` is not. The value sequence should only contain the output of value-updating instructions.
- *Sequential reduction.* If the value of a register or memory cell is sequentially updated, the intermediate results, which are never read, will not be added into the value sequence.
- *Optimization-based refinement* is only applied on plaintiff programs. It is used to filter out the values that vary because of different compiler options. We use several different optimized executables of the same program to generate value sequences. Then, we calculate the longest common subsequence of all these value sequences.
- *Address removal.* Memory addresses are not core values, because they may be changed by binary transformation techniques, such as word alignment and local variable reordering. Hence, we refine the value sequence by removing addresses.

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

Similarity detector compares the LCS of the N plaintiff implementations with the refined value sequence of the suspicious program. It measures their similarity and calculates a similarity score. The similarity metric is described in Section 4.3.

4.2 Annotation Approach

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

The scheme is shown in Figure 3. The *code annotator* adds annotations to the source code either automatically or based on knowledge of domain experts. These annotations identify which variables in which statements will contain algorithm-level core values during execution. The *core value extractor* executes the annotated source code with a specific input and records all runtime values flagged by annotations. It also tracks the relations (e.g., the order of presence) among core values in runtime. These values are core values, the sequence of which is the signature of the plaintiff algorithm. Meanwhile, the *value sequence extractor* generates a refined value sequence during the execution of a suspicious program given the same input as the plaintiff algorithm. After execution, the core value sequence of a plaintiff algorithm and the refined value sequence of a suspicious program are compared by the *similarity detector* which provides a similarity score. Note that in annotation approach the value sequence extractor and similarity detector are the same as in the N-version approach.

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

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

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

Given the drawback of manual annotation, we propose an automatic annotation method. It combines the techniques of static backward slicing and static forward slicing [38, 16]. Backward slicing starts from the output variables and ends at the beginning of the implementation. The result of backward slicing is the set of statements which affect the output. In the opposite, forward slicing initiates from the input variables and terminates at the end of the implementation. The

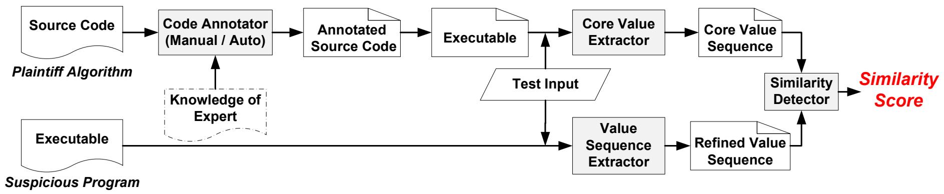
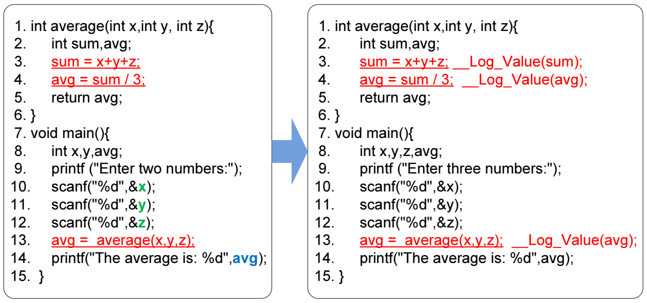


Figure 3: The design of annotation approach



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

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

Figure 4: Forward slicing and backward slicing annotation example

result of forward slicing is the set of statements affected by the input. The intersection of these two sets is the statements that derive the output from the input. After finding these important statements, we add an annotation to the result variable in each statement.

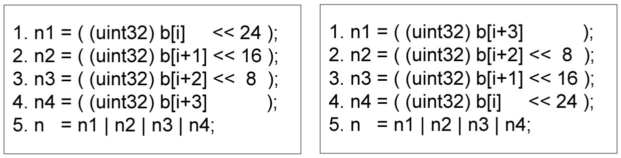
Figure 4 shows an example. The left part is the original source code. First, we specify input variables and output variables by preknowledge. x, y, z are the input variables and avg is the output variable. The second step is to apply static forward slicing and backward slicing based on the input and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

Although automatic annotation method is not as accurate as the manual method in identifying core values, it is often effective enough to detect algorithm plagiarisms while much more efficient and scalable.

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

4.3 Similarity Metric

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



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

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

Figure 5: Reordering problem example

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

5. ADDRESS REORDERING PROBLEMS

Although the LCS metric is efficient, it is sensitive to value reordering. For example, an adversary can reduce the length of the LCS by exchanging the order of independent instructions or independent basic blocks. As shown in Figure 5, two code segments are semantically equivalent, but the length of their LCS is only 2. To defend this attack, we propose a technique to organize a value sequence into subsequences showing unchangeable partial ordering of the values. To get such reordering-intolerant subsequences, we build dynamic value dependence graphs (VDGs) of the core values. Then we use a novel path comparison technique to check whether the reordering-intolerant subsequences of the plaintiff program are similar to any paths in the VDG of the suspect program.

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

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

5.1 VDG Comparison

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

We calculate the longest matched path in VDG(S) of p following Formula (1) and (2), where p_i is the ith node in p, n_j is a node in VDG(S), v_{p_i} and v_{n_j} represent the values of p_i and n_j, respectively. The computational complexity is $O(|p||V_S|^2)$, where $|p|$ is the length of path p and $|V_S|$ is the number of nodes in VDG(S).

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

$$\begin{aligned} \mathrm{LCS}(p, \mathrm{VDG}(S)) = &\max_{n_l}(\mathrm{LCS}(p_{|p|}, n_l)), \\ & n_l \in \{\text{leaf node of } \mathrm{VDG}(S)\} \end{aligned} \tag{2}$$

5.2 VDG Reduction

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

5.3 VDG Similarity Metric

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

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

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

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

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

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

Table 1: The similarity scores in MD5 experiment with various inputs

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

Table 2: The similarity scores in AES experiment with various inputs

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

Table 3: The similarity scores in max flow experiment with various inputs

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

6. IMPLEMENTATION AND EXPERIMENT

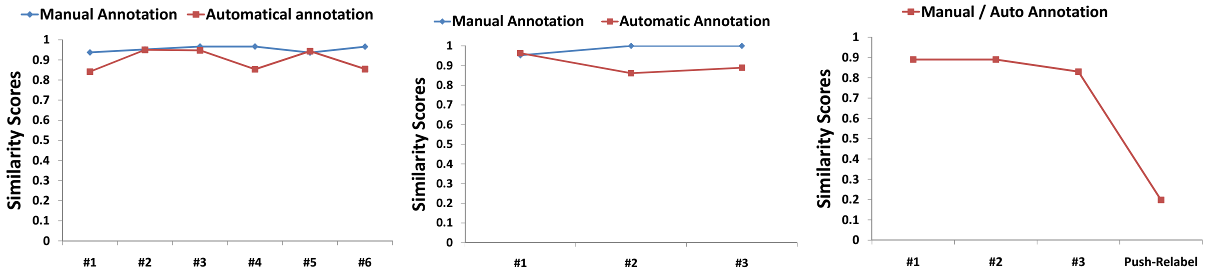
We implement the value sequence extractor inside QEMU 0.9.1 [7] by adding dynamic taint analyzer. The static backward slicing and forward slicing utilize the CodeSurfer 2.1 API [15]. Core value extractor is implemented by adding a new system call which is dedicated to flag core values.

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

6.1 Effectiveness of the N-version Approach

In this part of the evaluation, we use three algorithms: MD5, AES and network flow. We obtain multiple implementations of each plaintiff algorithm. All implementations are from open source libraries. To assure that these implementations are independent, we use MOSS, an online software plagiarism detection service [2], and VaPD, a dynamic value-based software plagiarism detection system [17], to measure their pair-wise similarities. A low pair-wise similarity would suggest independent. Therefore, we filter out the implementations with high similarity scores.

MD5. We have 6 independent implementations of MD5. First, to verify the existence of algorithm level core values that present in all implementations, we obtain the common value sequence of the first n ($1 \leq n \leq 6$) implementations, respectively (when $n = 1$, the common value sequence is



(a) MD5 (b) AES (c) Max Flow

Figure 7: The similarity scores of the annotation approach

itself). As shown in Figure 6, the length of common value sequence converges quickly as N increases and eventually becomes stable. Similar results are observed for the other two algorithms. This indicates that the irreplaceable and uneliminable core values of an algorithm do exist. Detailed results cannot be included here due to space limitation.

To test the effectiveness of N-version approach, we randomly choose N ($2 \leq N \leq 5$) implementations as plaintiff programs, while the rest are suspicious. Some statistics of similarity scores are shown in Table 1. These results demonstrate that as N increases, the similarity scores between plaintiff algorithm and plagiarized program increase as well. In other words, the ability to detect algorithm plagiarism is improved. When $N = 3$, the minimum similarity is 0.720, which is enough to identify algorithm plagiarism. Figure 6 also indicates that as N increases, similarity scores increase and converge to be stable.

AES. We use 3 implementations as plaintiff programs and the other one as the suspicious program. The statistics of similarity scores are shown in Table 2. Some similarity scores are not high enough to distinguish the same algorithm from different algorithms. The reason is that in AES, a lot of intermediate values are independent, so they could be in any order. Figure 5 shows an example. It may result a false negative. In Section 6.3, we will show that by using a VDG-based metric we are able to eliminate this false negative.

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

Maximum flow algorithms. We have four implementations of the Ford-Fulkerson algorithm and another implementation of the push-relabel algorithm. The result is shown in Table 3. We are able to distinguish the case of the same algorithm from that of different algorithms with the same purpose. For different algorithms, the similarity scores are all very low, irrespective of N.

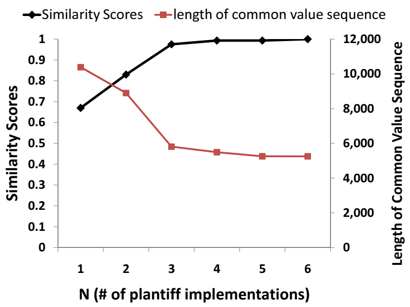


Figure 6: The similarity scores and lengthes of common value sequences for MD5

Conclusion. Results in Table 1, 2 and 3 demonstrate that as N increases, similarity scores of implementations of the same algorithm increase, while similarity scores of implementations of different algorithms are not affected. This indicates that applying multiple implementations can significantly reduce noises in core value extraction. The results also show that based on the LCS metric, false negative exists due to the value reordering problem.

6.2 Effectiveness of the Annotation Approach

6.2.1 *Manual Annotation Approach*

We perform proof-of-concept experiments on algorithms of MD5, AES and network maximum flow. For each algorithm, we randomly choose one implementation as plaintiff and manually annotate its source code. The rest of implementations are treated as suspicious programs. The results are shown in Figure 7. All similarity scores are higher than 0.85 when the plaintiff program and suspicious program implement the same algorithm. The similarity scores between implementations of different maximum flow algorithms are all around 0.25. The results indicate that the manual annotation approach can distinguish the case of the same algorithm from the case of different algorithms.

6.2.2 *Automatic Annotation Approach*

First, we conduct the same experiments as for the manual annotation approach, except that the plaintiff programs are automatically annotated through static forward slicing and backward slicing. The results are also shown in Figure 7. For both MD5 and AES algorithms, the similarity scores between implementations of the same algorithm are slightly lower than those measured by the manual annotation approach, but are still higher than 0.80. For the max flow algorithms, automatic annotation annotates the same variables in the plaintiff program as the manual annotation does, since max flow algorithms only contain calculation operations in very few statements. The automatic annotation approach is effective for all above three applications.

Besides the above three applications, we also conduct experiments on 6 other applications. The list of all applications and their algorithms is shown in Table 4.

Algorithms with the same purpose. We first compare the similarities in two cases: (1) the same algorithm

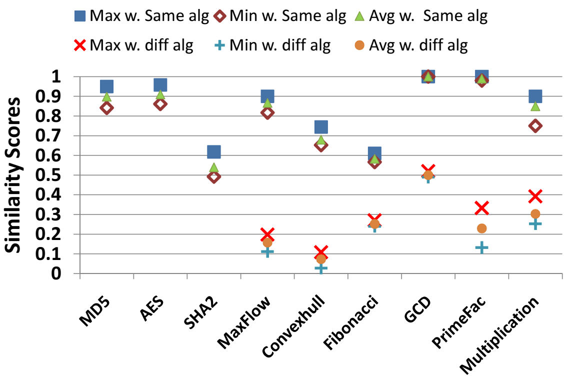


Figure 8: The similarity scores of automatic annotation with the LCS Metric

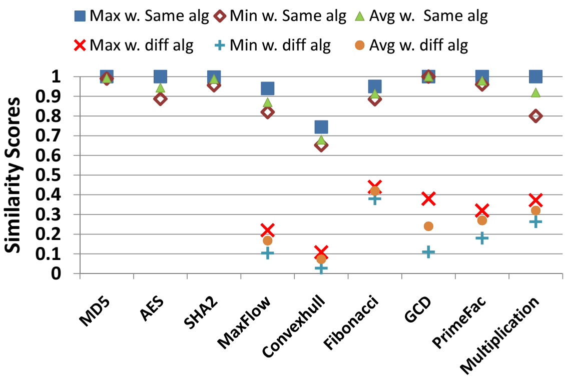


Figure 9: The similarity scores of automatic annotation with the VDG metric

and (2) different algorithms with the same purpose. For each application, we choose one plaintiff implementation and two suspicious implementations, one of which implements the same algorithm with the plaintiff implementation and the other does not. We measure their similarity scores by giving 10 randomly generated inputs to each of them. Figure 8 shows the experiment results. It indicates that for each application, there is a significant gap between similarity scores of the same algorithm and those of different algorithms, although no universal threshold can be applied for all applications. The low similarity scores (< 0.7) for the same algorithm in the SHA2 and Fibonacci applications are caused by the value reordering problem, which will be solved in Section 6.3. The high similarity score of different algorithms for greatest common divisor (GCD) application is caused by the reason that the brute force algorithm goes through every integer until the GCD is found. As a result, all integers between the GCD and the smaller integer are in its value sequence, therefore there are false matches, which will be eliminated by the VDG metric. The same algorithm for the convex hull application does not achieve high similarity scores (around 0.65) because the suspicious program optimizes the algorithm and does less calculations. Even though, the differences between similarity scores of the same algorithm and those of different algorithms are still large enough to distinguish them from each other. As a result, the automatic annotation approach is effective to distinguish the same algorithm from different algorithms with the same application.

Table 4: The List of Applications and their algorithms

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

Algorithms with different purposes. We evaluate the similarities between algorithms of different applications in Table 4. Since our approach is input sensitive, we choose 28 pairs of different algorithms, each pair of which can accept the same input. The results are quite positive: 20 pairs have the similarity scores lower than 1%. The other 8 pairs have the similarity scores between 1% and 30%. These higher similarity scores are all caused by the reason that the plaintiff algorithm is simple and has short core value sequence, while the suspicious one is complicated with much longer core value sequence, which increases the chance of false matches. As a result, for simple plaintiff algorithms, the plaintiff can use manual annotation to assure the accuracy of core value annotation and to reduce false matches. Even applying automatic annotation, all the similarity scores between programs implementing algorithms with different purposes are low enough to be distinguished from those between programs implementing the same algorithm.

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

6.3 VDG Based Metric

Both the N-version and the annotation approach can adopt VDG as a metric to measure algorithm-level similarities. We perform the same experiments on the VDG metric as in Section 6.1 and 6.2. Due to the limited space, we only show the results of automatic annotation using the VDG metric in Figure 9. Since both SHA2 and Fibobacci implementations suffer from the value reordering problem, their similarity scores are significantly increased. For the brute force algorithm of the GCD application, false matches are eliminated because its VDG is wide with all paths shorter than 3, which are not matched to the paths in the plaintiff VDG. The other results are similar to the results in Figure 8. Thus automatic annotation approach using VDG as the metric is effective in detecting algorithm similarity.

In addition, previously we have one false negative in Section 6.1, when the N-version approach is applied to detect similarity of the AES implementations with the LCS metric. The false negative is eliminated when we adopt the VDG metric. Its similarity scores are significantly increased (minimum score = 0.757, when $N = 3$).

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

6.4 Resiliency to Automatic Obfuscation

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

We apply 3 different automatic obfuscation tools: Semantic Designs Inc's C obfuscator [3], Diablo link-time optimizer based obfuscator (Loco) [25] and binobf [1]. The first one is source code based while the latter two are binary based. The features of Semantic Designs Inc's C obfuscator include, but are not limited to, identifier scrambling, format scrambling, loop rewriting, and if-then-else rewriting. Loco can obfuscate binaries by control flow flattening and opaque predicate. Binobf performs junk insertion, opaque predicate, jump table spoofing, etc.

We obfuscate the suspicious program of each application by these three tools and repeat the experiments in Section 6.2. The core value sequences of obfuscated suspicious programs are almost the same as those of original programs. The only differences are caused by several value reordering cases. As a result, the similarity scores are nearly the same as in Section 6.2. After we applied VDG as similarity metric to eliminate value reordering problem, the similarity scores become the same as in Section 6.3. Due to the space limitation, we omit the detailed results in this paper. The results show our approach is resilient to automatic obfuscation techniques.

7. DISCUSSION

7.1 Counterattacks

Software obfuscation techniques. An attacker may apply obfuscation techniques to evade algorithm plagiarism detection. Since our approaches apply core values as signature, we mainly focus on obfuscation methods that manipulate runtime values. These methods include noise injection, irrelevant instruction reordering and data transformation [12].

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

Our VDG metric is resilient to irrelevant instruction reordering as discussed in Section 5.

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

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

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

7.2 Partial Plagiarism

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

7.3 Limitations and Future Work

We discuss a few limitations of our methods in this section. First, our detection results rely on the selection of a similarity score threshold to decide whether or not an algorithm is plagiarized. However, there is no such universal threshold for all algorithms, because the threshold may vary for each algorithm depending on how complex it is, how specific it is described, etc. To this end, instead of giving a yes/no answer, our approach provides users with similarity scores between programs as initial evidences. Based on these evidences, users can take further investigations, which often involve non-technical actions. Second, our value-based methods are input sensitive. This means it is possible that different algorithms handle certain input in the same way. This may cause false positives. Nevertheless, since we choose multiple inputs randomly, this risk would be effectively mitigated in practice. Third, our approaches leverage dynamic taint analysis to extract values derived from input. It suffers from the common limitations of dynamic taint analysis techniques [8]. A plagiarist could use anti-taint-analysis techniques to hide core values. Solutions to this issue still remain an open question. Forth, we conduct experiments on well-known algorithms that are representative in different areas. Compared to large commercial/open source software products, the code sizes of collected implementations are relatively small. The reason why our experiments are not conducted on large software is that it is hard, if not impossible, to find a large software implementing one single algorithm. Nevertheless, we will consider extracting individual algorithms from these large software products and evaluating our scheme on extracted algorithms as one of our future works. Finally, our value based approaches are not applicable to all algorithms, since they rely on extracting runtime values from tainted value-updating instructions. Some algorithms e.g., sorting algorithms and finding minimum/maximum value in an array, contain very few of such instructions.

8. RELATED WORK

8.1 Software Plagiarism Detection

The most related work to algorithm plagiarism detection is software plagiarism detection.

Static birthmark based plagiarism detection: Liu et al. [23] proposed a program dependence graph (PDG) based approach, which is vulnerable to some obfuscation techniques such as control flow flattening and opaque predicates. Tamada et al. [33] proposed four static birthmarks for Java programs: Constant Values in Field Variables (CVFV), Sequence of Method Calls (SMC), Inheritance Structure (IS) and Used Classes (UC). These birthmarks can be changed by some obfuscation techniques, such as method call reordering. Myles et al. [27] statically analyzed executables to obtain opcode sequences of length k. Then they used K-gram techniques to measure the similarity. This approach is vulnerable to instruction reordering and junk instruction insertion.

Dynamic birthmark based plagiarism detection: Jhi et al. [17] proposed to use core values as birthmark to detect software plagiarism. Lu et al. [24] presented a dynamic opcode n-gram birthmark, which is vulnerable to instruction reordering and irrelevant instructions insertion. Myles et al. [26] developed a whole program path (WPP) birthmark, which is robust to opaque prediction, but is still vulnerable to some other obfuscations such as loop unwinding. Tamada et al. [34] used dynamic API birthmark for windows applications. Their approach relied on the sequence and the frequency of API invocations. Schuler et al. [32] proposed a dynamic API-based birthmark for Java. Both Java and Windows API-based birthmarks are platform dependent. Wang et al. [37, 36] introduced a system call based birthmark. Their approach is not suitable for programs that invoke few system calls.

8.2 Clone Detection

Clone detection is a technique to find duplicate code. Besides being used to decrease code size and facilitate maintenance, clone detection can also be used to detect software plagiarism. Existing source code clone detection techniques include String-based [5], Tree-based [6, 18], Token-based [19, 31, 29] and PDG-based [21, 14, 22]. Sæbjørnsen et al. [30] proposed a tree-based clone detection in binary code. Most clone detection techniques do not take code obfuscation into consideration. As a result, they are not robust to some obfuscation techniques.

As discussed in Section 1, neither software plagiarism detection nor clone detection tools can be applied to detect algorithm plagiarism. To the best of our knowledge, there is no other work on algorithm plagiarism detection. Our approaches do not require source code of suspicious programs and are applicable to any programming language. Moreover, our approaches are resilient to many kinds of attacks discussed in the previous section.

8.3 Software Watermarking

There exists a large volume of literatures on software watermarking (e.g. [11, 13, 35, 10]). Software watermarking embeds secret information into a program to protect intellectual property. The embedded information can be used to identify the ownership of the program. However, since software watermarks are embedded into implementations, they cannot be used to protect algorithms.

9. CONCLUSION

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

10. ACKNOWLEDGMENTS

The work of Zhang and Zhu was supported by NSF CAREER 0643906 and the work of Liu was supported by AFOSR FA9550-07-1-0527 (MURI), ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, and ARO W911NF1210055.

11. REFERENCES

[1] binobf: Binary obfuscation software. http://www.cs.arizona.edu/~debray/binary-obfuscation/.

[2] MOSS - a system for detecting software plagiarism. http://theory.stanford.edu/~aiken/moss/.

[3] Semantic designs inc. C source code obfuscator. http://www.semdesigns.com/products/obfuscators/CObfuscator.html.

[4] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, 11, December 1985.

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

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

[7] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[8] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *DIMVA' 08*, 2008.

[9] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *IEEE 8th International Symposium on Fault Tolerant Computing (FTCS-8)*, 1978.

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

[11] C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, Jan. 1999.

[12] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Jul 1997.

[13] P. Cousot and R. Cousot. An abstract interpretation-based framework for software watermarking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, 2004.

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

[15] GrammaTech. Codesurfer. `http://www.grammatech.com`.

[16] GrammaTech Inc. Dependence graphs and program slicing. Technical report. White Paper.

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

[18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the ICSE '07*, 2007.

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

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

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

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

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

[24] B. Lu, F. Liu, X. Ge, B. Liu, and X. Luo. A software birthmark based on dynamic opcode N-gram. *International Conference on Semantic Computing*, pages 37–44, 2007.

[25] M. Madou, L. Van Put, and K. De Bosschere. Loco: An interactive code (de)obfuscation tool. In *Proceedings of ACM SIGPLAN Workshop on PEPM '06*, 2006.

[26] G. Myles and C. Collberg. Detecting software theft via whole program path birthmarks. *Information Security*, 3225/2004:404–415, 2004.

[27] G. Myles and C. Collberg. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, 2005.

[28] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the NDSS 2005*, 2005.

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

[30] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, 2009.

[31] 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*, SIGMOD '03, 2003.

[32] D. Schuler, V. Dallmeier, and C. Lindig. A dynamic birthmark for Java. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, 2007.

[33] H. Tamada, M. Nakamura, and A. Monden. Design and evaluation of birthmarks for detecting theft of java programs. In *In Proc. IASTED International Conference on Software Engineering*, 2004.

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

[35] C. Thomborson, J. Nagra, R. Somaraju, and C. He. Tamper-proofing software watermarks. In *Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation - Volume 32*, ACSW Frontiers '04, 2004.

[36] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 280–290, New York, NY, USA, 2009. ACM.

[37] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *ACSAC*, 2009.

[38] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, 1981.