Value-Based Program Characterization and Its Application to Software Plagiarism Detection

Yoon-Chan Jhi[1] Xinran Wang[1] Xiaoqi Jia[2] Sencun Zhu[1] Peng Liu[1] Dinghao Wu[1]
[1]Penn State University, University Park, PA 16802
{jhi, xinrwang, szhu}@cse.psu.edu, {pliu, dwu}@ist.psu.edu
[2]State Key Laboratory of Information Security, Institute of Software, Chinese Academy of Sciences
xjia@is.iscas.ac.cn

ABSTRACT

Identifying similar or identical code fragments becomes much more challenging in code theft cases where plagiarizers can use various *automated* code transformation techniques to hide stolen code from being detected. Previous works in this field are largely limited in that (1) most of them cannot handle advanced obfuscation techniques; (2) the methods based on source code analysis are less practical since the source code of suspicious programs is typically not available until strong evidences are collected; and (3) those depending on the features of specific operating systems or programming languages have limited applicability.

Based on an observation that some critical runtime values are hard to be replaced or eliminated by semantics-preserving transformation techniques, we introduce a novel approach to dynamic characterization of executable programs. Leveraging such invariant values, our technique is resilient to various control and data obfuscation techniques. We show how the values can be extracted and refined to expose the critical values and how we can apply this runtime property to help solve problems in software plagiarism detection. We have implemented a prototype with a dynamic taint analyzer atop a generic processor emulator. Our experimental results show that the value-based method successfully discriminates 34 plagiarisms obfuscated by *SandMark*, plagiarisms heavily obfuscated by *KlassMaster*, programs obfuscated by Thicket, and executables obfuscated by Loco/Diablo.

Categories and Subject Descriptors

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

General Terms

Security

Keywords

Dynamic code identification, software plagiarism detection

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA
Copyright 2011 ACM 978-1-4503-0445-0/11/05 ...$10.00

1. INTRODUCTION

Identifying same or similar code fragments among different programs or in the same program is very important in some applications. For example, duplicated codes found in the same program may degrade efficiency in both development phase (*e.g.*, they can confuse programmers and lead to potential errors) and execution phase (*e.g.*, duplicated code can degrade cache performance). In this case, code identification techniques such as clone detection [1, 3, 18, 19, 16, 12, 15, 14] can be used to discover and refactor the identical code fragments to improve the program. For another example, same or similar code found in different programs may lead us to even more serious issues. If those programs have been individually developed by different programmers, and if they do not embed any public domain code in common, duplicated code can be an indication of *software plagiarism* or *code theft*. In code theft cases, determining the sameness of two code fragments becomes much more difficult since plagiarizers can use various code transformation techniques including code obfuscation techniques [8, 9, 37] to hide stolen code from detection. In order to handle such cases, code characterization and identification techniques must be able to detect the identical code (*i.e.*, two code fragments belonging to the same lineage) without being easily circumvented by code transformation techniques.

Previous works are largely insufficient in meeting all of the following three highly desired requirements: (R1) Resiliency to *automated* semantics-preserving obfuscation tools [7, 21, 32, 40]; (R2) Ability to directly work on binary executables of suspected programs since, in some applications such as code theft cases, the source code of suspect software products often cannot be obtained until some strong evidences are collected;(R3) Platform independence, *e.g.*, independent from operating systems and programming languages. As we can see in the related work section, the existing schemes can be broken down into four classes to see their limitations with respect to the aforementioned three requirements: (C1) static source code comparison methods [20, 33, 39, 17, 36, 28, 29, 13]; (C2) static executable code comparison methods [23]; (C3) dynamic control flow based methods [24]; (C4) dynamic API based methods [30, 34, 35]. First, Class C1, C2 and C3 do not satisfy requirement R1 because they are vulnerable to semantics-preserving obfuscation techniques such as outlining and ordering transformation. Second, C1 does not meet R2 because it has to access source code. Third, the existing C3 and C4 schemes do not satisfy R3 because they rely on features of Windows or Java.

To address the above issues, we introduce a novel approach to dynamic characterization of executable programs. After we examined various runtime properties of executable programs, we found an interesting observation that some runtime values of a program are hard to be replaced or eliminated by semantics-preserving transformation techniques such as optimization techniques, obfuscation techniques, different compilers, etc. We call such values *core-values.*

To investigate the resilience of core values (to semantics-preserving code transformation), we generated $e_{1..5}$, five different versions of executable files of test program p written in C, by compiling p with each of the five optimization switches of GCC (`-O0`, `-O1`, `-O2`, `-O3`, and `-Os`). From each of $e_{1..5}$ given the same test input, we extracted a *value sequence*, a sequence of values (4-bit, 8-bit, 16-bit, or 32-bit) written as computation results of arithmetic instructions and bit-wise instructions in the execution path. As a way of retaining (in the value sequence) only the values derived from input, we implemented a dynamic taint analyzer.[1] When we analyzed the value sequences of $e_{1..5}$, we found that some values survived all of the five optimization switches. Moreover, the sequence of the values surviving all of the five optimization switches was enclosed almost perfectly by the value sequences of executables generated by compiling p with different compilers (we tested *Tiny C Compiler* [4] and *Open Watcom C Compiler* [26]). This indicates that core-values do exist and we can use them to check whether two code fragments belong to the same lineage.

In this paper, we show (1) how we extract the values revealing core-values; and (2) how we can apply this runtime property to solve problems in software plagiarism detection. We implemented a value extractor with a specific dynamic taint analyzer and value refinement techniques atop a generic processor emulator, as part of our value-based program characterization method. As a machine code analyzer which directly works on binary executables, our technique satisfies R2. Because our technique analyzes generic characteristics of machine instructions, it satisfies R3. Regarding R1, we implemented a value-based software plagiarism detection method (VaPD) that uses similarity measuring algorithms based on sequences constructed from the extracted values. We evaluated it through a set of real world obfuscators including two commercial products, Zelix Pty Ltd.'s *KlassMaster* [40] and Semantic Designs Inc.'s *Thicket* [32]. Our experimental results indicate that the VaPD successfully discriminated 34 plagiarisms obfuscated by *SandMark* [7] (totally 39 obfuscators, but 5 of them failed to obfuscate our test programs); plagiarisms heavily obfuscated by *KlassMaster*,[2] programs obfuscated by the Thicket C obfuscator, and executables obfuscated by Control Flow Flattening implemented in the Loco/Diablo link-time optimizer [21].

Contributions: (1) We present a novel code characterization method based on runtime values. To our best knowledge, our work is the first one exploring the existence of the *core-values.* (2) By exploiting runtime values that can hardly be changed or replaced, our code characterization technique is resilient to various control and data obfuscation techniques. (3) Our plagiarism detection method (VaPD) does not require access to source code of suspicious programs, thus it could greatly reduce plaintiff's risks through providing strong evidences before filing a lawsuit related to intellectual property.

2. STATE OF THE ART

We roughly group the literature into the following three categories.

Code Obfuscation Techniques: Code obfuscation is a semantics-preserving transformation to hinder figuring out the original form of the resulting code. A generic code obfuscation technique is not as simple as adding x before computation and subtracting x after the computation. Collberg *et al.* [8] provided an extensive discussion on automated code obfuscation techniques. They classify code obfuscation techniques in the following categories depending on the feature that each technique targets: data obfuscation, control obfuscation, layout obfuscation, and preventive transformations. Collberg *et al.* also introduced Opaque Predicates [9] to thwart static disassembly. Other techniques such as indirect branches, control-flow flattening, and function-pointer aliasing were introduced by Wang [37].

Several code obfuscation tools are available. SandMark [7] is one of such tools implementing 39 obfuscators applicable to Java bytecode. Array representation and orientation, functions, in-memory representation of variables, order of instructions, and control and data dependence are just a small set of the features that SandMark can alter. Another Java obfuscator is Zelix KlassMaster [40]. It implements comprehensive flow obfuscation techniques, making it a heavy duty obfuscator. Semantics is the only characteristic guaranteed to be preserved across the obfuscation.

Static Analysis Based Plagiarism Detection: The existing static analysis techniques except for the birthmark-based techniques are closely related to the clone detection [1, 3, 18, 19, 16, 12, 15, 14, 31]. While possessing common interests with the clone detection, the plagiarism detection is different in that (1) we must deal with code obfuscation techniques which are often employed with a malicious intention; (2) source code analysis of the suspicious program is not possible in most cases. Static analysis techniques for software plagiarism detection can be classified into five categories: string-based [1], AST-based [39, 17, 36], token-based [28, 29, 13], PDG-based [20], and birthmark-based [23, 33]. *String-based:* Each line of source code is considered as a string. A code fragment is labeled as plagiarism if the corresponding sequence of strings matches certain code fragment from original program. *AST-based:* The abstract syntax trees (AST) are constructed from two programs. If the two ASTs have common subtrees, plagiarism may exist. *Token-based:* A program is first parsed to a sequence of tokens. The sequences of tokens are then compared to find plagiarism. *PDG-based:* A program dependency graph (PDG) represents the control flow and data flow relations between the statements in a program procedure. To find plagiarism, two PDGs are constructed and compared to find a relaxed subgraph isomorphism. *Birthmark-based:* A software birthmark is a unique characteristic of a program that can be used to determine the program's identity. Two birthmarks are extracted from two programs and compared.

None of the above techniques is resilient to code obfuscation. String-based schemes are vulnerable even to sim-

[1] We also have noticed that there are studies on identifying and overcoming limitations of dynamic taint analysis. Please note that dealing with those limitations is out of our scope.

[2] Since SandMark and KlassMaster work on Java bytecode, we use GCJ, GNU ahead-of-time compiler for Java, to convert obfuscated programs to x86 native executables.

Table 1: Proportion of refined value sequences of GCC compiled executables that overlap value sequences of TCC and WCC compiled executables.

Compiler	Optimization switches tested	bzip2	gzip	oggenc
TCC	NA	100%	100%	92%
WCC	20 switches	100%	100%	> 91% (avg. 95%)

ple *identifier renaming.* AST-based schemes are resilient to identifier renaming, but weak against statement reordering and control replacement. Token-based schemes are weak against junk code insertion and statement reordering. Because PDGs contain semantic information of programs, PDG-based schemes are more robust than the other three types of the existing schemes. However, the PDG-based methods are still vulnerable to many semantics-preserving transformations such as inlining/outlining functions and opaque predicates. The existing birthmark-based schemes are vulnerable to either obfuscation techniques mentioned in [23] or some well-known obfuscation such as statement reordering and junk instruction insertion. Moreover, all existing techniques except for [23, 31] need to access source code.

Dynamic Analysis Based Plagiarism Detection: Myles and Collberg [24] proposed a whole program path (WPP) based dynamic birthmark. WPP was originally used to represent the dynamic control flow of a program. WPP birthmarks are robust to some control flow obfuscation such as opaque predicates insertion, but are still vulnerable to many semantics-preserving transformations such as flattening and loop unwinding. Tamada *et al.* [34, 35] also introduced two types of dynamic birthmarks for Windows applications: Sequence of API Function Calls Birthmark (EXESEQ) and Frequency of API Function Calls Birthmark (EXEFREQ). In EXESEQ, the sequence of Windows API calls are recorded during the execution of a program. These sequences are directly compared to find the similarity. In EXEFREQ, the frequency of each Windows API call is recorded during the execution of a program. The frequency distribution is used as a birthmark. Schuler *et al.* [30] proposed a dynamic birthmark for Java. The call sequences to Java standard API are recorded and the short sequences at object level are used as a birthmark. Their experiments showed that their API birthmarks are more robust to obfuscation than WPP birthmarks. These birthmarks, however, can only identify the same source code compiled by different compilers with different options, and the performance against real obfuscation techniques is questionable. For example, attackers may simply embed some of API implementations into their program so that fewer API calls will be observed. Wang *et al.* [38] proposed a system call based birthmark, addressing the problems with API based techniques. However, the proposed technique cannot be applied to computation oriented softwares containing few system calls, and is sill vulnerable to injecting transparent system calls in the middle of an edge on the system call dependence graph.

3. CORE VALUES

The *runtime values* of a program are defined as values from the output operands of the machine instructions executed. While examining the runtime values of executable programs, we observed that some runtime values of a program could not be changed through automated semantics-preserving transformation techniques such as optimization, obfuscation, different compilers, etc. We call such invariant values *core-values.*

Core-values of a program are constructed from runtime values that are pivotal for the program to transform its input to desired output. We can practically eliminate *non-core values* from the runtime values to retain core-values. To identify non-core values, we leverage taint analysis and easily accessible semantics-preserving transformation techniques such as optimization techniques implemented in compilers. Let v_P be a runtime value of program P taking I as input, and f be a semantics-preserving transformation. Then, the non-core values have the following properties: (1) If v_P is not derived from I, v_P is not a core-value of P; (2) If v_P is not in the set of runtime values of $f(P)$, v_P is not a core-value of P.

To examine the existence of core-values, we perform a dynamic analysis on three test programs gzip, bzip2, and oggenc: Gzip and bzip2 are well-known compression utilities, and oggenc is a OggVorbis audio format encoder. For the dataset to be used as the input to the programs, we generate ten wav audio files (seven 16KB files, two 24KB files, and one 8KB file), cropped from a 43.5MB wav file containing an 8'37"-long speech. In each set of experiments, we use these ten inputs, and take the average outcome as the final result. With each of the three programs, we generate five different versions of executable files by compiling it with each of the following optimization switches of GCC: `-O0`, `-O1`, `-O2`, `-O3`, and `-Os`. From each of the executables given the same input, we extract a *value sequence*, a sequence of values (4-bit, 8-bit, 16-bit, or 32-bit) that are the computation results of arithmetic and bit-wise instructions in the execution path. We also implement refinement techniques (Section 4.1 and 4.2) including a dynamic taint analyzer to retain only the values derived from input in the sequence. Then, we refine the value sequences by computing their longest common subsequence, which contains the runtime values that survive all of the five optimization switches.

To verify that the refined value sequences are not from compiler-specific common routines, we compare the refined value sequences against the value sequences extracted from the same programs compiled by different compilers, Tiny C Compiler (TCC) and Open Watcom C Compiler (WCC). Compared to GCC, TCC uses different compiler components such as parser and optimizer, and support library (libtcc.a), however the code it produces borrows GCC's runtime libraries (libc.so). WCC is a self-contained development suite implementing its own C libraries. Therefore, the code it produces does not need to use GCC's runtime libraries. Also, WCC provides plenty of optimization options, and we test all the 20 optimization switches to examine the refined value sequences. As shown in Table 1, the longest common subsequence of the five sequences are enclosed almost completely by the value sequences of executables generated by compiling the same test program with TCC and WCC. Although 92% and 95% matches shown in the cases of oggenc indicate that the refined value sequences still contain some non-core values, these are much higher scores than those between irrelevant programs: as we will show shortly, the scores between irrelevant programs range from 0% to 11% in our experiments.

Table 2: Proportion of refined value sequences that overlap value sequences of executables obfuscated by Thicket and control flow flattening

Obfuscator	bzip2	gzip	oggenc
Thicket C Obfuscator	100%	100%	95%
Control Flow Flattening	100%	100%	100%

We further investigate the core-values through real obfuscation tools. For a source code obfuscation tool, we use Semantic Designs, Inc.'s *Thicket C obfuscator* that implements abstract syntax tree (AST) based code transformation. Its features include, but not limited to, identifier scrambling, format scrambling, loop rewriting, and if-then-else rewriting. As a more advanced obfuscation technique, we use *control flow flattening* [37] implemented in *Loco* based on *Diablo* link-time optimizer [21]. Control flow flattening can transform statements '`s1; s2;`' into '`i=1; while(i) {switch(i) {case 1: s1; i++; break; case 2: s2; i=0; break;}}`' of which the control flow graph is hugely different from the original. As shown in Table 2, again our refined value sequences are almost completely enclosed by the value sequences of obfuscated executables.

To see overlapping portion of value sequences of different programs, we compare the refined value sequences of bzip2, gzip, and oggenc against irrelevant pairs (*i.e.*, the refined value sequence of bzip2 to value sequence of oggenc optimized with -O1). In 30 comparison cases (three test programs, each of which has two irrelevant peers, five optimization switches), the value sequences of each program contain only 0% to 11% of the refined value sequences of different programs. This indicates that the core-values do exist and we can use them to identify the sameness of codes.

4. DESIGN

Software theft has become a very serious concern to software companies and open source communities. In the presence of automated semantics-preserving code transformation tools [40, 21, 7, 32], the existing code characterization techniques may face an impediment to finding sameness of plagiarized code and the original. In this section, we discuss how we apply our technique to software plagiarism detection. Later, we evaluate our method against such code obfuscation tools in the context of software plagiarism detection.

Scope of Our Work: We consider the following types of software plagiarisms in the presence of *automated* obfuscators: *whole-program plagiarism*, where the plagiarizer copies the whole or majority of the plaintiff program and wraps it in a modified interface, and *core-part plagiarism*, where the plagiarizer copies only a part such as a module or an engine of the plaintiff program. Our main purpose of VaPD is to develop a practical solution to real-world problems of the whole-program software plagiarism detection, in which *no source code* of the suspect program is available. VaPD can also be a useful tool to solve many partial plagiarism cases where the plaintiff can provide the information about which part of his program is likely to be plagiarized. We present applicability of our technique to core-part plagiarism detection in the discussion section. We note that if the plagiarized code is very small or functionally trivial, VaPD would not be an appropriate tool.

4.1 Value Sequence Extraction

Since not all values associated with the execution of a program are core-values, we establish the following requirements for a value to be added into a value sequence: The value should be output of a *value-updating instruction* and be closely related to the program's semantics.

Informally, a computer is a state machine that makes state transition based on input and a sequence of machine instructions. After every single execution of a machine instruction, the state is updated with the outcome of the instruction. Because the sequence of state updates reflects how the program computes, the sequence of state-updating values is closely related to the program's semantics. As such, in value-based characterization, we are interested only in the state transitions made by value-updating instructions. More formally, we can conceptualize the *state-update* as the change of data stored in devices such as RAM and registers after each instruction is performed, and we call the changed data a *state-updating value*. We further define a *value-updating instruction* as a machine instruction that does not always preserve input in its output. Being an output of a value-updating instruction is a sufficient condition to be a state-updating value. Therefore, we exclude output values of non-value-updating instructions from a value sequence. In our x86 implementation, the value-updating instructions are the standard mathematical operations (`add`, `sub`, etc.), the logical operators (`and`, `or`, etc.), bitshift arithmetic and logical (`shl`, `shr`, etc.), and rotate operations (`ror`, `rcl`, etc.).

The above technique helps dramatically reduce the size of a value sequence; however, in practice it is still challenging to analyze all values produced by all the value-updating instructions. Therefore, we must apply further restrictions to refine value sequences. There are two classes of values computed by value-updating instructions: *Class-1* includes those derived from input of the program, and *Class-2* consists of those that are not. For example, when program P is processing input I in environment E, some instructions take values derived from input I as their input, but some others take input from environment E such as program load location, stack pointer, size of stack frame, etc. Since the semantics is a formal representation of the way that a program processes the input, it is obvious that the values in *Class-1* are more closely related with the semantics of a program. So, we include only the values of *Class-1* in a value sequence. To identify the values included in *Class-1*, we run a program in a virtual machine environment and perform a dynamic taint analysis [25]. We start with tainting the input, and then our analyzer in the virtual machine propagates the taint to every byte in registers, memory cells, and files derived from the input. Registers and memory cells appearing in destination operands of all the instructions that take input from tainted registers or tainted memory locations are also tainted, and the output values of value-updating instructions are appended into the value sequence. In the example of JLex used as a case study in this paper, the value sequences contain less than 7,000 values after applying taint analysis, which is significantly shorter, approximately $\frac{1}{250}$ of the original sequences.

4.2 Value Sequence Refinement

In this section, we discuss heuristics to refine value sequences. An initial value sequence constructed through the dynamic taint analysis may still contain a number of non-

Table 3: Applicability of value sequence refinement techniques.

Refinement technique	Plaintiff program	Suspect program
Sequential refinement	√	
Optimization-based refinement	√	
Address removal	√	√

```
     Assembly code       IN         OUT   Output value
001: shl $0x2,%eax       eax        eax        4      ⎫
002: add %edx,%eax       edx,eax    eax        5      ⎬ Invisible
003: add %eax,%eax       eax        eax        10     ⎪ at line 6
004: add %edx,%eax       edx,eax    eax        11     ⎭
005: add $0xb,%eax       eax        eax        22
006: ...
```

Figure 1: Sequential refinement example (EAX is initially tainted)

core values produced by intermediate or insubstantial computational steps. We need to eliminate those values to make the value sequence (1) as close to core-values as possible; and (2) capable of characterizing larger programs. We believe a number of heuristics such as control/data flow dependence analysis and abnormal code pattern detection can be adopted to achieve these goals, and below we introduce some of them. One principle that we consider here is that we have to be conservative in processing value sequences of suspect programs. Since some heuristics may be abused by sophisticated plagiarizers, we summarize applicability of each heuristic that we will introduce in Table 3.

4.2.1 Sequential Refinement

Inside the value sequence extractor, we implement a refinement technique named *sequential refinement*. Figure 1 shows how GCC compiles "`a=1; a=(a+1)*11;`." When variable `a` is initially tainted, our taint analysis extracts value sequence $s = \{4, 5, 10, 11, 22\}$. Note that sequence $s_{1:4} = \{4, 5, 10, 11\}$, a subsequence of s is generated by intermediate steps computing '$(\mathtt{a}+1) \times 11$'. All the values in $s_{1:4}$ are overwritten in register `eax` without affecting any other memory locations until line 005. Since instructions after line 005 would never read (or be affected by) the values in $s_{1:4}$, we can remove $s_{1:4}$ from s and retain only $\{22\}$. We formalize this heuristic in the following rule:

Sequential Reduction Rule: Let $i_{m,n}$ denote m-th instruction updating variable (register or memory) n. Then, we can skip logging output of $i_{m,n}$ if n is never read within range $(i_{m,n}, i_{m+1,n})$. Repeat the same process until the first instruction that reads n and updates a variable ($\neq n$) is executed.

Through out our experiments presented in this paper, average reduction rate by the sequential refinement is 16%, and the maximum is 34%. Note that the sequential refinement only applies to plaintiff programs because, in obfuscated programs, original values could appear as the results of the intermediate computational steps.

4.2.2 Optimization-Based Refinement

Only for plaintiff programs, we perform *optimization-based refinement* as shown in Figure 2. One of the easiest way to obtain different executable files that are semantically identical is to compile the same source code with the same compiler with different optimization switches enabled. Motivated by this idea, we use several optimized executables of the same program to sift non-core values out. With GCC and its five selected optimization flags (`-O0`, `-O1`, `-O2`, `-O3`, and `-Os`), we can extract five *optimized value sequences* from the plaintiff program. Each optimized value sequence has been processed with the sequential refinement while it is extracted. Then, we compute a longest common subsequence of all the optimized value sequences to retain only the common values in the resulting value sequence. As we do not assume we have access to the source code of suspect programs, this refinement heuristic is only applicable to plaintiff programs.

4.2.3 Address Removal

Memory addresses or pointer values stored in registers or memory locations are transient. For example, some binary transformation techniques such as word alignment and local variable reordering can change pointers to local variables or offsets in stack; and heap pointers may not be the same next time the program is executed even with the same input. Therefore, we do not include pointer values in a refined value sequence.

In our VaPD prototype, we implement a range checking based heuristic to detect addresses. Our testbed dynamically monitors the changes of memory pages allocated to the program being analyzed, and it maintains a list of ranges of all the allocated pages with write permission enabled. If a runtime value is found to be within the ranges in the list, VaPD discards the value, regarding the value as an address. Although this heuristic may also delete some non-pointer values, it can remove pointers to stack and to heap with no exception. Address removal heuristic is applicable to both plaintiff and suspect programs.

4.3 Similarity Metric

In the literature, there are many metrics for measuring the degree of similarity of two sequences. In our prototype, we define it based on the longest common subsequence (LCS). It should be noted that the definition of the LCS does not require every subsequence to be a continuous segment of the mother sequence. For example, both {1, 6, 120} and {2, 24} are valid subsequences of value sequence {1, 2, 6, 24, 120}. Let $|\mathrm{LCS}(s_1, s_2)|$ denote the length of the LCS of sequence s_1 and s_2. Given v_P, a fully refined value sequence of a plaintiff program and v_S, a value sequence of a suspect program, similarity score of the suspect program over the plaintiff program is intuitively defined as:

$$\mathrm{Sim}(v_P, v_S) = \frac{|\mathrm{LCS}(v_P, v_S)|}{|v_P|}$$

4.4 Design Overview

Figure 3 shows overall design of VaPD. Here, provided with executable files of plaintiff program P and suspect program S, and common test input I, *Value Sequence Extractor*(VSE) extracts v_P and v_S, the value sequences of P and S. After refining v_P and v_S, *Similarity Detector* computes $\mathrm{Sim}(v_P, v_S)$, the similarity score of v_P and v_S. VaPD repeats this process with different inputs (say, 10 or 20 inputs), and claims plagiarism if the average of the scores shows a significant similarity.

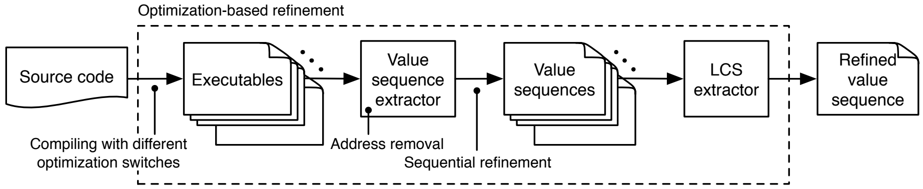


Figure 2: Optimization-based refinement on plaintiff programs.

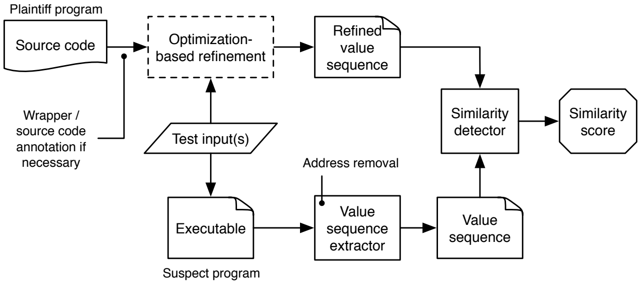


Figure 3: Plagiarism detection process

By default, VaPD uses value sequences v_P and v_S extracted through the entire execution of P and S respectively. However, when it deals with the cases where only part of P is reused in S, VaPD can extract partial value sequence from only the suspicious part of P. To extract partial value sequences, we insert special system calls into the source code of P (note that we do not assume access to the source code of S) to notify VSE when to start (or resume) and when to stop (or pause) extracting the value sequence. Provided by the plaintiff with the intelligence about which part of his program is likely to be plagiarized, we can annotate plaintiff's source code and capture the sequence from the part that is believed to be stolen.

VSE is a virtual machine that executes given program instruction by instruction. We implement two operation modes in VSE: *normal mode* and *partial extraction mode*. In *the normal mode*, VSE operates as follows. After fetching an instruction, Taint Analyzer taints the destination operands if any of the source operands is tainted. After the instruction is executed by the virtual machine, VSE checks whether the instruction is a value-updating instruction and whether its output is tainted; if this is true, the output of the instruction is added to the value sequence. VSE then fetches and decodes the next instruction and repeats the same process until the program is finished. When the program terminates, VSE stops extracting values and passes completed value sequence to VaPD. Note that VSE also performs the address removal refinement. In *the partial extraction mode*, VSE intercepts two special system calls `START_EXTRACT()` and `STOP_EXTRACT()` (system call numbers are 0xFFFFFFFF and 0xFFFFFFF0 respectively) requested by the test program. When VSE starts in the partial extraction mode, value sequence recording is initially turned off. It starts (or resumes) recording values when the test program requests `START_EXTRACT()` system call, and it stops (or pauses) storing values when the program calls `STOP_EXTRACT()` system call. Using the partial extraction mode, we can extract value sequences from part of plaintiff programs.Note that the partial extraction mode is to extract partial value sequence of plaintiff programs. Malicious plagiarizers will not be able to prevent this mode from excluding plagiarized part in value sequence extraction process.

To reduce the number of values added into the value sequence, VSE does not extract values from dynamic linked libraries or shared libraries by default. However, if necessary, we can enable VSE to include specific shared libraries in the value sequence extraction because the virtual machine knows which libraries are loaded and where they are.

5. EXPERIMENT

We implemented Value Sequence Extractor inside QEMU 0.9.1. During our evaluation of the prototype, we answer three questions. First, how *resilient* is VaPD to obfuscation techniques? Second, how likely will it make a *false accusation*? Finally, how *credible* is VaPD when tested with very similar programs independently implemented to meet the same specification? We thoroughly test obfuscation resiliency of VaPD using the obfuscators implemented in SandMark [7], Zelix Pty Ltd.'s KlassMaster [40], and Semantic Designs Inc.'s Thicket C obfuscator [32]. SandMark and KlassMaster are Java bytecode obfuscators: The latest SandMark includes 15 application obfuscations, 7 class obfuscations, and 17 method obfuscations; Zelix Pty Ltd. claims KlassMaster is a heavy duty obfuscator implementing name obfuscation, comprehensive flow obfuscation techniques, and string encryption. The Thicket C obfuscator is a C source code rewriting tool based on abstract syntax tree. It performs several obfuscation techniques including identifier scrambling, format scrambling, replacing/simplifying statements, loop rewriting, and rewriting if-then-else conditionals [2]. Because VaPD analyzes x86 machine code, we convert Java byte code (used in SandMark and KlassMaster experiments) to x86 executable using GCJ 4.1.2, the GNU ahead-of-time Compiler for Java. As a front-end of GCC, GCJ benefits from GCC's optimization features. We also examine VaPD's credibility by deliberately using programs that are similar to but disparate from each other. Experiments are performed on a Linux machine equipped with an Intel Quad-Core 2.00 GHz CPU and 4GB RAM.

5.1 Case Study I: Obfuscation Tools

We evaluated resiliency of VaPD against advanced obfuscation techniques of SandMark and KlassMaster. Since SandMark and KlassMaster are Java bytecode obfuscators, we selected JLex [5], a lexical analyzer generator written in Java, as the subject of out tests. In this case study, we set up two cases of experiments: a single-obfuscation experiment, where only one obfuscation technique is applied at a time, and a multiple-obfuscation experiment, where multiple obfuscators are applied to one program at once. As a dynamic analysis based solution, VaPD may not reliably

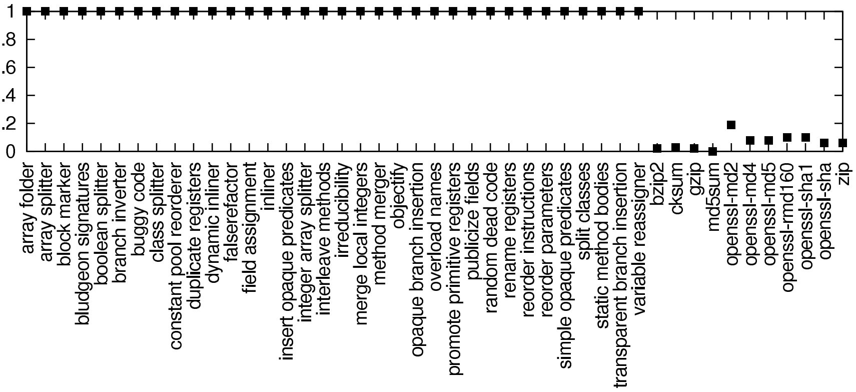


Figure 4: Similarity scores (y-axis) of original JLex to obfuscated ones and other programs (x-axis)

Table 4: Names of obfuscation techniques applied to JLex to generate multiply obfuscated versions

Control obfuscation	Data obfuscation
Transparent Branch Insertion, Simple Opaque Predicates, Inliner, Insert Opaque Predicates, Dynamic Inliner, Interleave Methods, Method Merger, Reorder Instructions	Array Folder, Integer Array Splitter, Promote Primitive Registers, Variable Reassigner, Duplicate Registers, Boolean Splitter, Merge Local Integers

identify (non-)plagiarism based on a single high similarity score. Hence, in this experiment, we used 20 different inputs and compute the average similarity scores.

5.1.1 Impact of Single Obfuscation

In these experiments, original JLex is compared to obfuscated versions of itself. Also, we compare JLex to 11 additional programs (bzip2, cksum, gzip, md5sum, zip, and openssl computing MD2, MD4, MD5, RMD160, SHA1, and SHA) totally different from JLex while processing the same input. The result is shown in Figure 4, where the x-axis shows suspect program names (JLex's obfuscated versions[3] and other programs), and the y-axis is the similarity scores.

We observed that in all cases of comparing original JLex to its obfuscated versions (totally 680 comparisons, given by 34 obfuscators and 20 inputs), the similarity scores mark 1.0. In contrast, the similarity scores between JLex and 11 other programs mark very low scores with average of 0.07. Only one case mark 0.19, which is still very low considering that the similarity score for a real plagiarism is 1.0.

Therefore, the results shown in Figure 4 provide us with clear answers to the questions we raised earlier: Regardless of obfuscation techniques, VaPD computed noticeably high similarity scores between the original and obfuscated programs, and discernably lower similarity scores between different programs. In all cases, VaPD can identify the identical programs with no false accusation with an appropriate threshold (say 0.90).

5.1.2 Impact of Multiple Obfuscation

We also notice that a plagiarist may attempt to hide plagiarism by heavily transforming a plagiarized program through a series of obfuscators. Therefore, evaluating resiliency of VaPD against multiple obfuscation techniques applied to single program is necessary.

Although it is theoretically possible for a series of multiple obfuscators to transform a program, applying many obfuscators to a single program could raise practical issues of correctness of the target program and efficiency. For example, we attempted to apply all the 39 obfuscation techniques of SandMark to JLex, but after trying several obfuscation orders, only some of them could be successfully applied. To address this problem, we selected two groups of obfuscation techniques, following the classification of Collberg *et al.* [8]: data obfuscation and control obfuscation. By transforming JLex through each group of obfuscators, we created two multiply obfuscated programs $\text{JLex}_{\text{control}}$ and $\text{JLex}_{\text{data}}$. In summary, we could apply 8 control obfuscators and 7 data obfuscators to JLex as shown in Table 4. We also generated JLex_{zkm} by transforming JLex through KlassMaster with the most aggressive configuration options enabled.

We compared each of $\text{JLex}_{\text{control}}$, $\text{JLex}_{\text{data}}$, and JLex_{zkm} to original JLex. In all three groups of comparisons between heavily obfuscated JLex and original JLex, we observe similarity score of 1.00. This shows that VaPD is effective in detecting plagiarisms obfuscated heavily.

5.2 Case Study II: Similar Programs

To investigate the credibility of VaPD on analyzing highly similar but disparate programs, we cross analyze five individual XML parsers: RXP, used by the LT XML toolkit and the Festival speech synthesis system; Expat XML parser, the underlying XML parser for the open source Mozilla project and Perl's XML::Parser; Libxml2, the XML C parser and toolkit of Gnome; Xerces-C++ supported by Apache XML project; and Parsifal XML parser C library. For each of the five XML parsers, we wrote a simple test program that parses test input and prints the parser's internal information to the terminal. We cross-compared the refined value sequences of plaintiff programs to the value sequences of suspect programs through 375 distinct comparison cases given by five programs, five different optimization switches (`O0-3` and `Os`), and three test inputs.

To our best knowledge, these five XML parsers do not share code. Since they are all individually developed projects, it would be a false accusation if VaPD computes a higher similarity score (say, greater than 0.9) for any of them. Av-

[3] We could not test all 39 obfuscators because some of them failed in transforming JLex.

Table 5: Similarity scores of five XML parsers cross compared. (P=Plaintiff, S=Suspect)

S P	expat	libxml2	parsifal	rxp	xercesc
expat		0.12	0.09	0.17	0.03
libxml2	0		0.01	0	0.01
parsifal	0.02	0.1		0.04	0.23
rxp	0.08	0.09	0.08		0.02
xercesc	0	0.02	0.01	0.02	

erage and standard deviation of similarity scores of 75 cases comparing same programs are 1.0 and 0 respectively. Average and standard deviation of 300 cases comparing different programs are both 0.06. Table 5 summarizes the results (We show only average of similarity scores per each program pair for brevity). In all cases comparing different programs, except one case, we observe similarity scores lower than 0.17. Only one comparison case shows a similarity score of 0.23, which is still very low. Therefore, it is safe to say VaPD claims no false accusation in this case study.

5.3 Case Study III: Different Programs

Previously, at the end of Section 3, we presented preliminary results on the likelihood of VaPD raising false accusations by cross-comparing bzip2, gzip, and oggenc. In this section, we investigate even further by comparing each of bzip2, gzip, and oggenc against 9 of 11 programs used in Section 5.1.1—two are excluded because they overlap bzip2 and gzip. Bzip2, gzip, and oggenc used in this experiment are compiled from self-contained, single compilation-unit C programs [22], therefore they need no external libraries other than the standard C library.

From 270 distinct comparisons given by three plaintiff programs (bzip2, gzip, and oggenc), 9 suspect programs (cksum; md5sum; openssl computing MD2, MD4, MD5, RMD160, SHA1, and SHA; and zip), and 10 input files, we observe similarity scores between 0 and 0.27 except the cases of zip and gzip pairs in which all the similarity scores are 1.0. According to the documentations of zip and gzip projects, we found that zip and gzip are based on the same compression algorithm *deflate* which is also implemented in the zLib library. Our source code analysis confirms that the gzip used in this experiment contains code from zLib 1.1.4 in itself, and the zip is dynamically linked to the system-wide zLib 1.2.3. Therefore, high similarity scores of zip and gzip pairs give more credential to VaPD's detection. In addition, zip scored very low similarity scores (0.01 to 0.03) against bzip2. This result is also correct because bzip2 uses a different compression algorithm called *block sorting*.

6. DISCUSSION

6.1 Obfuscation Transformations and Attacks

Since the value based approach leverages selected runtime values to characterize a code fragment, it can be affected by the data obfuscation techniques that can alter majority of the runtime values. We discuss about the impact of data obfuscation and potential attacks to VaPD in this section.

6.1.1 Data Transformation

Simple data transformations expose the core-values of the original program. Figure 5(a) is an example where the original values of x are transformed by *adding a constant*. Assuming that x is tainted and is 10 at the beginning, the value sequence of the transformed code is $\{11, ..., 10, ..., y, 10\}$. In this sequence, $\{..., 10, ..., y\}$ are the values captured from intermediate data for computing y, and this must appear in the value sequence of the original code as well. Let us look at a more complex example, *variable encoding transformation*. In general, variable encoding transforms variable v to $\alpha v + \beta$. In Figure 5(b), variable y at line 7 of the original code is transformed to be $y + x$. For this transformation, we apply the same procedure as Drape *et al.* used [10]. Assuming that y is tainted at line 2, the refined value sequence that VaPD extracts from the original code is $\{1, 2, 6, 24\}$, and the value sequence extracted from the transformed code is $\{2, 1, 1, 2, 4, 1, 2, 5, 8, 2, 6, 12, 16, 6, 24, 34, 39, 24, 120, 135, 120, 24\}$. Again we see some of the encoded values are restored to original data at some points during the execution. *Splitting a variable* and *merging two variables into one* also have similar characteristics. Those invariant values are very close to the core-values of the original program, and will be included in the values extracted by VaPD.

6.1.2 Inserting Arbitrary Instructions (Noise)

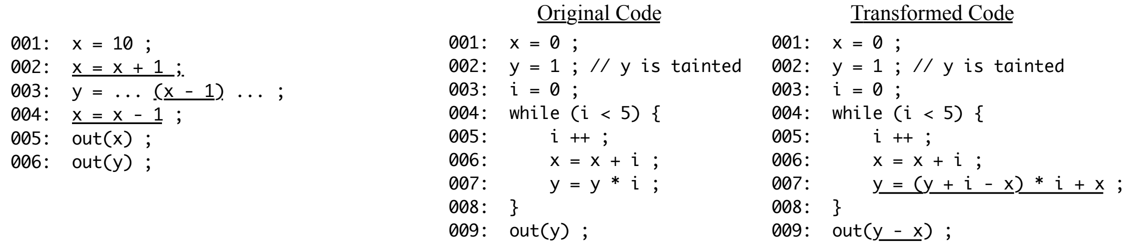
Under the LCS metrics, injection of a huge amount of noise might increase the similarity score. If a naive program happens to generate many noisy values, this will raise the chance of false accusation. However, for malicious programs that try to hide their plagiarisms, intentionally injected noise values will result in a higher chance of being accused. Therefore, if a plagiarist comes to know the mechanism of VaPD, he will never try to evade VaPD by injecting random noise. Moreover, automated noise injection is difficult because if the noise is not tainted, it will be filtered due to our dynamic taint analysis. However, if injected successfully, noise could dramatically increase the size of an extracted value sequence, thus slowing down the similarity score computation, consuming more memory space. We will consider sliding over a stream of values so that we may keep only a small portion of a value sequence in memory during runtime.

6.1.3 Loop Rewriting

Another possible counterattack is rewriting a loop in a reverse order. However, automatic loop reversing is very difficult because they could result in semantically different programs. So far, we are not aware of such tools to our best knowledge. Although some specific types of loops that are not tightly bound with the loop counters could theoretically be reversed, reversing the loop counter variable only will not affect the whole value sequence because we can eliminate values produced by loop counters (by dynamic taint analysis). One might manually reverse a loop, if at all possible, but its impact could be very limited in a large program.

6.2 Core-Part Plagiarism

Core-part plagiarism is a harder problem. In such case, only some part of a program is plagiarized. For example, a less ethical developer may steal code from some open source projects and fit the essential module into his project with obfuscation. Let I_{PM} and I_{SM} be the input to the plagiarized module and suspect module respectively, and $\mathcal{V}(x)$ be a value based characteristic such as a value sequence extracted from x, a program or a module. Then the value based method can be applied to a subproblem of the core-



```
x = 10 ;
x = x + 1 ;
y = ... (x - 1) ... ;
x = x - 1 ;
out(x) ;
out(y) ;
```

(a) Simple Data Transformation

Original Code

```
x = 0 ;
y = 1 ; // y is tainted
i = 0 ;
while (i < 5) {
    i ++ ;
    x = x + i ;
    y = y * i ;
}
out(y) ;
```

Transformed Code

```
x = 0 ;
y = 1 ; // y is tainted
i = 0 ;
while (i < 5) {
    i ++ ;
    x = x + i ;
    y = (y + i - x) * i + x ;
}
out(y - x) ;
```

(b) More Complex Data Transformation

Figure 5: Data Transformation Examples. Underlined codes are added by the transformation.

part plagiarism detection where $I_{\text{PM}} = I_{\text{SM}}$. In this case, we can directly search in $\mathcal{V}$(S) of the suspect program, for $\mathcal{V}$(PM) of the plaintiff module to check whether the module has been plagiarized. For example, in the case of web browser layout engine plagiarism, given an input URL I, we can first obtain $\mathcal{V}$(PM) from the plaintiff layout engine module; then, using the same input I we can obtain $\mathcal{V}$(S) from the suspect program. If the plaintiff program and the suspect program use the same layout engine, then $\mathcal{V}$(PM) and part of $\mathcal{V}$(S) (*i.e.*, $\mathcal{V}$(SM)) to bear significantly similar patterns. Therefore, we can search for $\mathcal{V}$(PM) in $\mathcal{V}$(S).

6.3 Limitations

Our technique bears the following limitations. First, VaPD provides the partial extraction mode in which it can extract value sequences from only a small part of the program. Based on this, we discuss about the feasibility of applying VaPD to the partial plagiarism detection problems in Section 6.2. However, we have not yet comprehensively evaluated this issue with real world test subjects.

Second, VaPD may not apply if the program implements a very simple algorithm. In such cases, the value sequences can be too short, which increases sensitivity to noises. Our metric is likely to cause false positives when a very short value sequence is compared to a much longer one.

Third, as a detection system, there exists a trade-off between false positives and false negatives. The detection result of our tool depends on the similarity score threshold. Unfortunately, without many real-world plagiarism samples, we are unable to show concrete results on such false rates. As such, rather than applying our tool to "prove" software plagiarisms, in practice one may use it to collect initial evidences before taking further investigations, which often involve nontechnical actions.

Fourth, built upon a dynamic taint analyzer, VaPD may generate much shorter value sequences if tainted data is used as an index into a translation table, or a plagiarist attempts anti-taint-analysis techniques [25]. Anti-taint attacks and some countermeasures (*e.g.*, tainting the program counter when a tainted conditional is tested [11]) are well summarized by Cavallaro *et al.* [6].

Fifth, regarding whether one could use reordering to evade VaPD, the answer is a coin with two sides: On one hand, the plagiarist cannot do arbitrary or random reordering, which will very likely distort the semantics of the program. Once the semantics is distorted, the plagiarist can no longer get the functionality he intends to steal. On the other hand, if the plagiarist can find a way to do semantics-preserving reordering, our LCS based metric in theory may be sensitive to the attempt. The existing tools we have experimented actually include some reordering transformations, and VaPD shows resilience to them. But, we think that (new) automatic reordering transformations could be created to evade VaPD. To address this threat, we have started looking at the dependencies between values, leveraging dependence graphs of the values and the subgraph similarity algorithms. To preserve the program semantics, these dependencies cannot be violated. So the orders required by these dependencies cannot be flipped.

6.4 Future Work

As our future work, we will examine the relationship between values. A better understanding of the logical connection among the values will enable us to further remove system noise or less significant values. We will also study the problems in handling partial plagiarism. We are particularly interested in the impact of inputs on value sequences. As a dynamic analysis, VaPD requires the programs being analyzed to be fed with the same input. This requirement sometimes is difficult to meet especially in the partial plagiarism cases. For example, a software plagiarist may illegally use a real time computer vision library as a part of their motion recognition software, whereas the original program uses the library for different purposes, say face recognition. In addition, we will study the impact of emulation-based obfuscators such as Themida and Code Virtualizer [27] on VaPD's performance. Such obfuscators encode original program into a specially designed bytecode instruction set and run the bytecode program in an emulator. We believe our detection method can handle such obfuscators.

7. CONCLUSION

Obfuscation resilient code characterization is important for many code analysis applications, including code theft detection. Motivated by an observation that some outcome values computed by machine instructions survive various semantics-preserving code transformations, we have proposed a technique that directly examines executable files and does not need to access the source code of suspicious programs. Our results show that the value-based method is effective in identifying software plagiarism.

8. ACKNOWLEDGMENTS

This work was supported by AFOSR FA9550-07-1-0527 (MURI), ARO W911NF-09-1-0525 (MURI), NSF CNS-0905131, NSF CNS-0916469, and AFRL FA8750-08-C-0137. Xiaoqi Jia was partly supported by NSFC:61073179.

9. REFERENCES

[1] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95*, page 86, 1995.

[2] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *ICSE'04*, pages 625–634, 2004.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Int. Conf. on Software Maintenance*, 1998.

[4] F. Bellard. Tiny C compiler. http://bellard.org/tcc/.

[5] E. J. Berk and C. S. Ananian. JLex: A lexical analyzer generator for Java. online. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

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

[7] C. Collberg, G. Myles, and A. Huntwork. Sandmark–a tool for software protection research. *IEEE Security and Privacy*, 1(4):40–49, 2003.

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

[9] C. S. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL '98*, 1998.

[10] S. Drape, A. Majumdar, and C. Thomborson. Slicing aided design of obfuscating transforms. *ICIS*, 00:1019–1024, 2007.

[11] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *USENIX Annual Technical Conference*, pages 233–246. USENIX, 2007.

[12] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *ICSE '08*, 2008.

[13] J.-H. Ji, G. Woo, and H.-G. Cho. A source code linearization technique for detecting plagiarized programs. In *ITiCSE '07*, 2007.

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

[15] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE '07*, 2007.

[16] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE TSE*, 28(7), 2002.

[17] Y.-C. Kim and J. Choi. A program plagiarism evaluation system. In *Information and Communication Technology Education Workshop*, 2005.

[18] K. Kontogiannis, M. Galler, and R. DeMori. Detecting code similarity using patterns. In *Working Notes of 3rd Workshop on AI and Software Engineering*, 1995.

[19] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE '01*, 2001.

[20] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *KDD '06*, 2006.

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

[22] S. McCamant. Large single compilation-unit C programs, Jan 2006. `http://people.csail.mit.edu/smcc/projects/single-file-programs/`.

[23] G. Myles and C. Collberg. K-gram based software birthmarks. In *SAC*, 2005.

[24] G. Myles and C. S. Collberg. Detecting software theft via whole program path birthmarks. In *ISC '04*, 2004.

[25] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS '05*, 2005.

[26] Open Watcom Contributors. Open Watcom. `http://www.openwatcom.org`.

[27] Oreans Technologies. Code Virtualizer. `http://www.oreans.com/codevirtualizer.php`.

[28] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPLAG. *Universal Computer Science*, 2000.

[29] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *ACM SIGMOD Int. Conf. on Management of Data*, 2003.

[30] D. Schuler, V. Dallmeier, and C. Lindig. A dynamic birthmark for Java. In *IEEE/ACM ASE '07*, 2007.

[31] A. Sebjornsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *ISSTA '09*, 2009.

[32] Semantic Designs, Inc. Thicket^{TM}. `http://www.semanticdesigns.com`.

[33] H. Tamada, M. Nakamura, A. Monden, and K. ichi Matsumoto. Design and evaluation of birthmarks for detecting theft of java programs. In *IASTED SE '04*, February 2004. Innsbruck, Austria.

[34] H. Tamada, K. Okamoto, M. Nakamura, and A. Monden. Dynamic software birthmarks to detect the theft of Windows applications. In *Int'l Symp. on Future Software Technology (ISFST)*, October 2004.

[35] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. ichi Matsumoto. Design and evaluation of dynamic software birthmarks based on API calls. Info. Science Technical Report NAIST-IS-TR2007011, ISSN 0919-9527, Nara Institute of Science and Technology, May 2007.

[36] N. Truong, P. Roe, and P. Bancroft. Static analysis of students' Java programs. In *ACE '04: Proc. of the 6th conf. on Australasian computing education*, 2004.

[37] C. Wang. *A security architecture for survivability mechanisms*. PhD thesis, Charlottesville, VA, USA, 2001. Adviser-John Knight.

[38] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. In *ACM CCS*, 2009.

[39] W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21(7):739–755, 1991.

[40] Zelix Pty Lt. Java obfuscator - Zelix KlassMaster. online. `http://www.zelix.com/klassmaster/features.html`.