Program Characterization Using Runtime Values and Its Application to Software Plagiarism Detection

Yoon-Chan Jhi, Xiaoqi Jia,† *Member, IEEE,* Xinran Wang, Sencun Zhu, *Member, IEEE,* Peng Liu, *Member, IEEE,* and Dinghao Wu, *Member, IEEE*

Abstract—Illegal code reuse has become a serious threat to the software community. Identifying similar or identical code fragments becomes much more challenging in code theft cases where plagiarizers can use various *automated* code transformation or obfuscation techniques to hide stolen code from being detected. Previous works in this field are largely limited in that (i) most of them cannot handle advanced obfuscation techniques, and (ii) the methods based on source code analysis are not practical since the source code of suspicious programs typically cannot be obtained until strong evidences have been collected. Based on the observation that some critical runtime values of a program 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 value-based plagiarism detection method (VaPD) uses the longest common subsequence based similarity measuring algorithms to check whether two code fragments belong to the same lineage. We evaluate our proposed method through a set of real-world automated obfuscators. 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.

Keywords—Software plagiarism detection, dynamic code identification.

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 can be used to discover and refactor the identical code fragments to improve the program [1], [2], [3], [4], [5], [6], [7], [8]. 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 to hide stolen code from detection [9], [10], [11]. In order to handle such cases, code characterization and identification techniques must be able to detect semantically equivalent 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 the following two highly desired requirements: (R1) Resiliency to the *automated* semantics-preserving obfuscation tools that can easily transform most of the syntactic features such as strings [9], [12], [13], [14], [15]; and (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 have been collected.

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 [16], [17], [18], [19], [20], [21], [22], [23]; (C2) static executable code comparison methods

- *Y.C. Jhi is with Samsung SDS R&D Center, Korea. E-mail: yoonchan.jhi@samsung.com*
- *X. Jia is with Institute of Information Engineering, Chinese Academy of Sciences, China. E-mail: jiaxiaoqi@iie.ac.cn*
- *X. Wang is with Shape Security, Mountain View, CA 94040.*
- *S. Zhu is with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802.*
- *P. Liu is with the College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802.*
- *D. Wu is with the College of Information Sciences and Technology, Pennsylvania State University, University Park, PA 16802.*

A preliminary version of this paper has been published in the Proceedings of the ACM/IEEE 33rd International Conference on Software Engineering (ICSE 2011), Software Engineering in Practice Track, *Honolulu, Hawaii, USA, May 21–28, 2011.*

† *Corresponding author.*

[24]; (C3) dynamic control flow based methods [25]; (C4) dynamic API based methods [26], [27], [28]. We may briefly summarize their limitations as follows. 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.

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 (or computation results of some machine instructions) 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*. Note core values are values computed at runtime from program execution, not the static constants embedded in the executables such as strings, which can be easily obfuscated.

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* [29] and *Open Watcom C Compiler* [30]). 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 apply this runtime property to solve problems in software plagiarism detection. We have 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. Regarding the requirement R1, we have implemented a value-based software plagiarism detection method (VaPD) that uses similarity measuring algorithms based on sequences and dependence graphs constructed from the extracted values. We have evaluated it through a set of real world obfuscators including two commercial products, Zelix Pty Ltd.'s *KlassMaster* [15] and Semantic Designs Inc.'s *Thicket* [14]. Our experimental results indicate that the VaPD successfully discriminated 34 plagiarisms obfuscated by *SandMark* [12] (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 [13].

Contributions: In summary, we make the following 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.
4) We evaluate VaPD through a set of real world programs.

This paper is organized as follows. In the next section, we briefly discuss related works. In Section 3, we discuss the existence of core-values implied by our experimental results. In Section 4 and 5, we evaluate our value-based code characterization method by applying it to the problems of software plagiarism detection. In Section 6, we address reordering attacks and evaluate our dependence graph based method. Finally, the limitations, some potential counterattacks, and future work are discussed in Section 7.

2 State of the Art

We roughly group the literature into four categories: code obfuscation techniques, static analysis based plagiarism detection, dynamic analysis based plagiarism detection, and smartphone app repackaging detection.

2.1 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.* provided an extensive discussion on automated code obfuscation techniques [9]. They classify code obfuscation techniques

1. We also have noticed that there are studies on identifying and overcoming limitations of dynamic taint analysis. 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.

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 to thwart static disassembly [10]. Other techniques such as indirect branches, control-flow flattening, and function-pointer aliasing were introduced by Wang [11].

Several code obfuscation tools are available. SandMark is one of such tools implementing 39 obfuscators applicable to Java bytecode [12]. 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 [15]. It implements comprehensive flow obfuscation techniques, making it a heavy duty obfuscator. Semantics is the only characteristic guaranteed to be preserved across the obfuscation.

2.2 Static Analysis Based Plagiarism Detection

The existing static analysis techniques except for the birthmark-based techniques are closely related to the clone detection [1], [2], [3], [4], [5], [6], [7], [8], [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 [18], [19], [20], token-based [21], [22], [23], PDG-based [4], [16], and birthmark-based [17], [24]. *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 simple *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 [24] or some well-known obfuscation such as statement reordering and junk instruction insertion. Moreover, all existing techniques except for [24], [31] need to access source code. Luo *et al.* [32] proposed the concept of *Longest Common Subsequences (LCS) of Semantically Equivalent Basic Blocks*, which combines the flexible LCS with rigorous program semantics for code equivalence checking at the binary level. This method is quite obfuscation-resilient; however, the computational overhead is quite high due to the use of advanced techniques such as symbolic execution and constraint solving.

2.3 Dynamic Analysis Based Plagiarism Detection

Myles and Collberg proposed a whole program path (WPP) based dynamic birthmark [25]. 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.* 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) [27], [28]. In EXESEQ, the sequence of Windows API calls is 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.* proposed a dynamic birthmark for Java [26]. 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.* [33] 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. More recently, Zhang *et al.* [34] proposed a dynamic approach to the *algorithm* plagiarism detection, a different but related research problem. Zhang *et al.* [35] proposed a method to search for path deviations for plagiarism detection

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%)

using symbolic execution and constraint solving on the execution traces.

2.4 Smartphone App Repackaging Detection

App repackaging, a form of software plagiarism, has become a common phenomenon in the mobile app markets like Google Play and Apple iTunes. Dishonest users may repackage others' apps under their own names or embed different advertisements, and then republish it to the app market to earn monetary profit. Furthermore, to leverage the popularity of mobile apps to increase the propagation of their malware, malware writers may modify popular apps to insert malicious payloads into the original apps.

Prior work in this area includes the following methods: Opcode-based approach: DroidMOSS [36] and Juxtapp [37], AST based approach: [38], and PDG based approach: DNADroid [39]. A common drawback is that most of them are not obfuscation-resilient. Our research is obfuscation-resilient and can be potentially applied to the smartphone app repackaging detection. More recently, Huang *et al.* [40] developed a repackaging detection evaluation framework so that different methods can be systematically evaluated and compared, with obfuscations applied. ViewDroid [41] applied a user interface based birthmark, which is designed for user interaction intensive and event dominated programs, to detect smartphone application plagiarism.

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 such as standard C library or C startup code, we compare the refined value sequences against 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.

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*

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%

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* [11] implemented in *Loco* based on *Diablo* link-time optimizer [13]. 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

With the rapid development of software industry and the burst of open source projects (*e.g.*, SourceForge.net is hosting over 430,000 open source projects as of March 2014), 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 [12], [13], [14], [15], the existing code characterization techniques may face an impediment to finding sameness of plagiarized code and the original. In this section, we discuss in detail how we apply our technique to software plagiarism detection. Later, we evaluate our value-based code characterization 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 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 and various automated obfuscation techniques have been applied to the suspect program. Our proposed technique, VaPD, can also be a useful tool to solve many partial, or core-part, 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, it is important to limit the types of values to be included in a value sequence. 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 program's semantics. In the following, we discuss the rationale for these requirements.

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. For example, `add` is a value-updating instruction, but `mov` is not. 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

TABLE 3
Applicability of value sequence refinement techniques.

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

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 [42]. 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-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*. Fig. 1 shows a partial list of instructions compiled by GCC for "`a=1; a=(a+1)*11;`." The registers `%eax` and `%edx` are initialized to value 1. 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 '$(\texttt{a} + 1) \times 11$'. 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 $i_{k,v}$ denote k-th instruction updating variable (register or memory) v. Then, we can skip logging output of $i_{k,v}$ if v is immediately killed in the next instruction $i_{k+1,v}$). Repeat the same process until the first instruction that reads n and updates a variable ($\neq n$) is executed. Note that we do not implement reduction to remove all intermediate values. We simply only remove those that are sequentially killed immediately because it effective and can be implemented with little cost.

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 Fig. 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 test bed dynamically monitors the changes of memory pages

	Assembly code	IN	OUT	Output value	Note
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 $0xb,%eax`	`eax`	`eax`	22	
006:	`mov %eax,-8($ebp)`				

Fig. 1. Sequential refinement example (EAX is initially tainted)

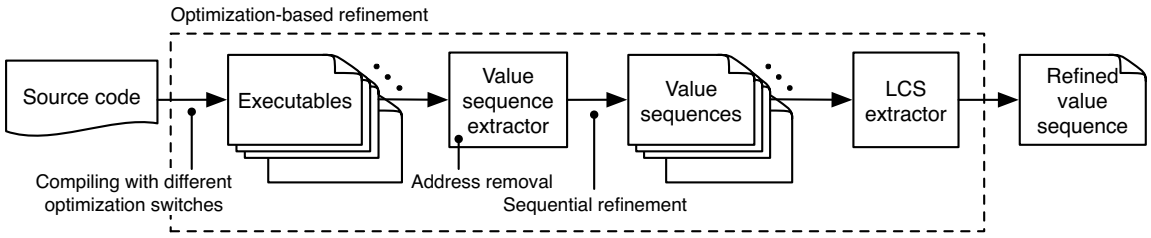

Fig. 2. Optimization-based refinement on plaintiff programs.

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 $|\text{LCS}(s_1, s_2)|$ 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:

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

4.4 Design Overview

Fig. 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 $\text{Sim}(v_P, v_S)$, 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.

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

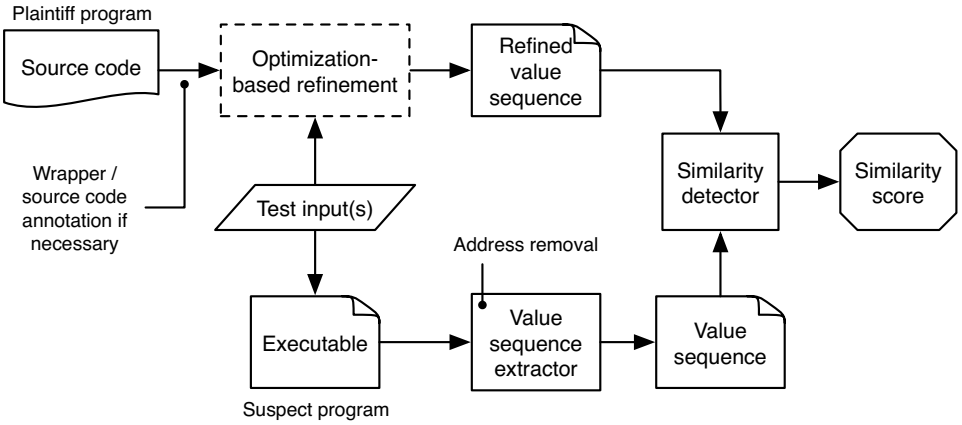

Fig. 3. Plagiarism detection process

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 implement Value Sequence Extractor (VSE) inside QEMU 0.9.1 [43]. QEMU improves execution speed mainly through *block translation* and caching the translated blocks. QEMU translates each basic block on-the-fly into instrumented machine instructions and directly executes the translated code. *Translation block cache*, the cache of once translated blocks prevents QEMU from re-translating the same code many times. For a rapid prototyping, we tweak QEMU's block translation into instruction translation and implement VSE within the translation. In addition, we disable the translation cache to force QEMU to invoke VSE for every single instruction that it executes. Our measurement indicates that the QEMU without block translation and translation block cache is observed to be 56 time slower than the original QEMU, and the performance overhead attribute to our taint analyzer is relatively insignificant. We can improve the performance if we enable the block translation and translation block cache features by embedding taint updating code in all translated code blocks.

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 [12], Zelix Pty Ltd.'s KlassMaster [15], and Semantic Designs Inc.'s Thicket C obfuscator [14]. 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 [44]. 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 [45], 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

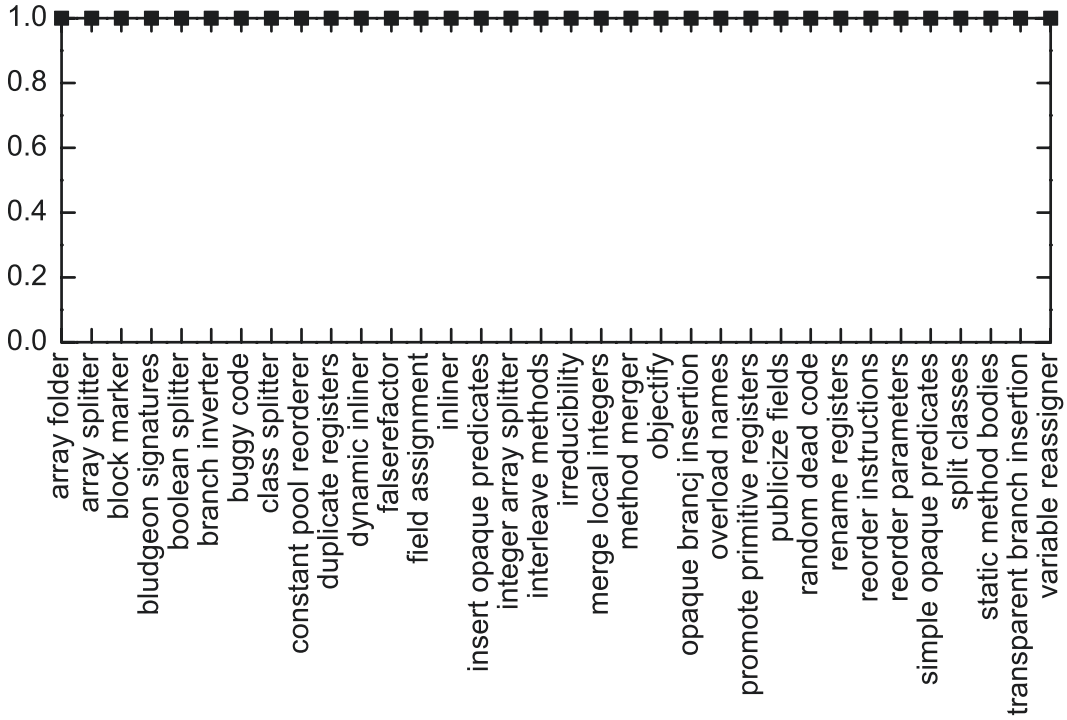

Fig. 4. Similarity scores (y-axis) of original JLex to obfuscated ones (x-axis)

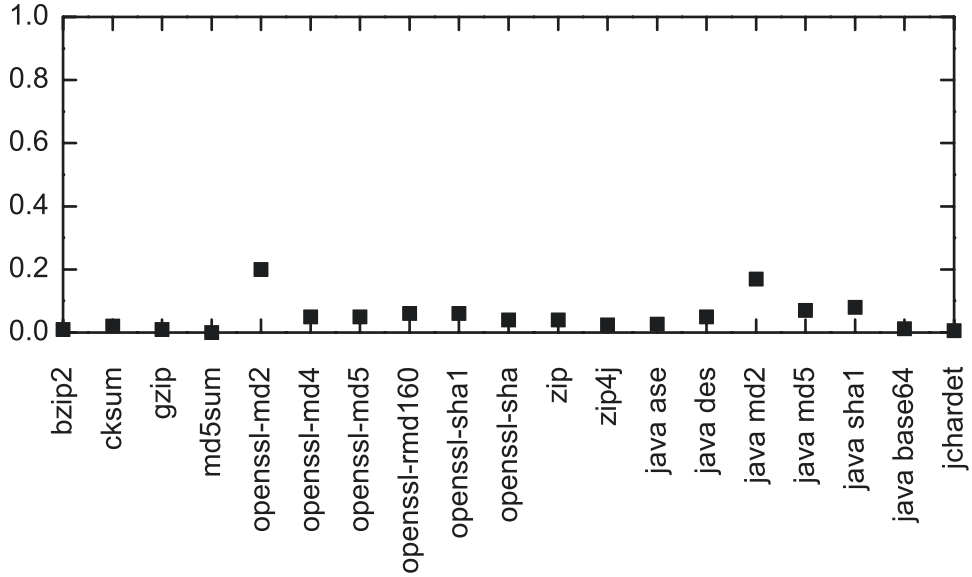

Fig. 5. Similarity scores (y-axis) of original JLex to other programs written in Java and C (x-axis)

based solution, VaPD may not reliably 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 single-obfuscation experiments, original JLex is compared to obfuscated versions of itself. Also, we compare JLex to 19 additional programs, 8 of which are Java programs (zip4j, ase, des, md2, md5, sha1, base64, and jchardet) and 11 of which are C 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 results are shown in Fig. 4 and Fig. 5, where the x-axis shows suspect program names (JLex's obfuscated versions[3] and other programs), and the y-axis shows the similarity scores.

We observed that in all cases of comparing original JLex to its obfuscated versions (totally 680 comparisons,

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

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

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

given by 34 obfuscators and 20 inputs), the similarity scores mark 1.0. In contrast, the similarity scores between JLex and 19 other programs mark very low scores with average of 0.064. The maximal is only 0.19, which is still very low considering that the similarity score for a real plagiarism is 1.0.

Therefore, the results shown in Fig. 4 and Fig. 5 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.* [9]: 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.

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

P \ S	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	

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. Average 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 [46], therefore they need no external libraries other than the standard C library.

The results are shown in Fig. 6. From 270 distinct comparisons given by three plaintiff programs (bzip2,

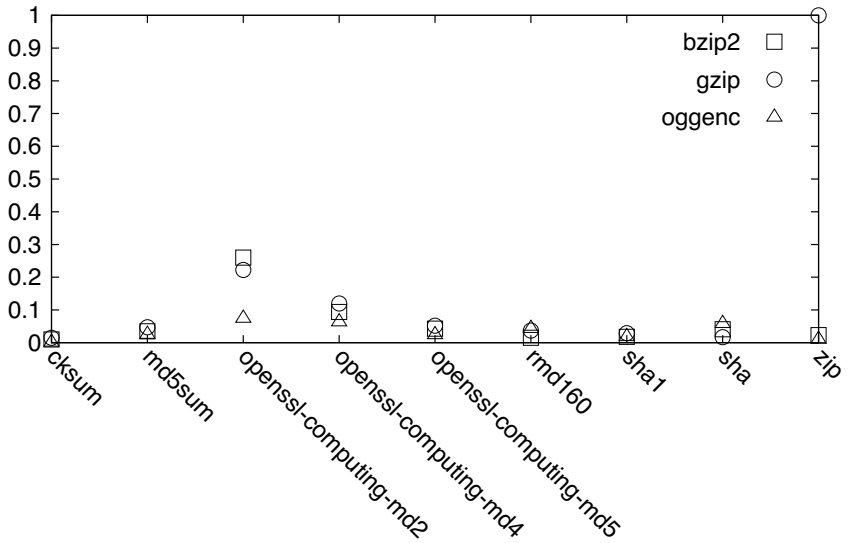

Fig. 6. Similarity scores (y-axis) between different programs (x-axis)

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 are not false positives. Rather, it gives more credential to VaPD's detection. In addition, zip scored very low similarity scores (0.01 to 0.03) against bzip2 which is also a compression utility. This result is also correct because bzip2 uses a different compression algorithm called *block sorting*.

6 Addressing Reordering Attacks

As we have seen in Section 5, our results show that the runtime value sequence based method is resilient to various obfuscation techniques. However, our metric LCS is sensitive to reordering. For example, an adversary can reduce the length of common sequences by exchanging the order of independent instructions or independent basic blocks. In this section, we present value dependence graphs (dynamic data dependence graphs of the runtime values) and an efficient algorithm, which is resilient to reordering attacks, to determine similarity of two programs based on the graphs.

In general, runtime value dependence graphs can be constructed and we can apply subgraph isomorphism to defend the reordering attacks. However, subgraph isomorphism algorithms are not practical for large graphs. Instead, we propose a technique to organize a value sequence into subsequences showing unchangeable partial ordering of the values. To get such subsequences that cannot be reordered, we build specific value dependence graphs (dynamic data dependence graphs of the runtime values) denoted by VDG. Then we use a novel path containment test technique to check whether the reordering-intolerant subsequences of the plaintiff program are contained in the VDG of the suspect program. Such path containment checks (and the checking results) may replace LCS metrics in measuring the similarity of two programs. Through a Lemma, we show that such containment checks are "immune" to reordering attacks.

The overall process of VDG based plagiarism detection is shown in Fig. 7. To avoid using expensive graph matching algorithms, we develop two techniques, path containment test and graph reduction. In the following sections, we discuss about how we compare paths of the plaintiff program against the VDG of the suspect program, how we select the paths to compare, and how we reduce the search space through the VDG reduction.

6.1 VDG Construction

Construction of VDG is done based on the dynamic data dependence extracted while VaPD performs a dynamic taint analysis to extract a value sequence of a program. We define the value dependence graph (VDG) as follows:

Definition 1 (Dynamic Instruction Sequence): Given a program P, a dynamic instruction sequence of P is a sequence of ι_n representing the machine instructions executed during an execution of P in a temporal order. For any two dynamic instructions ι_i and ι_j ($i \neq j$), ι_i

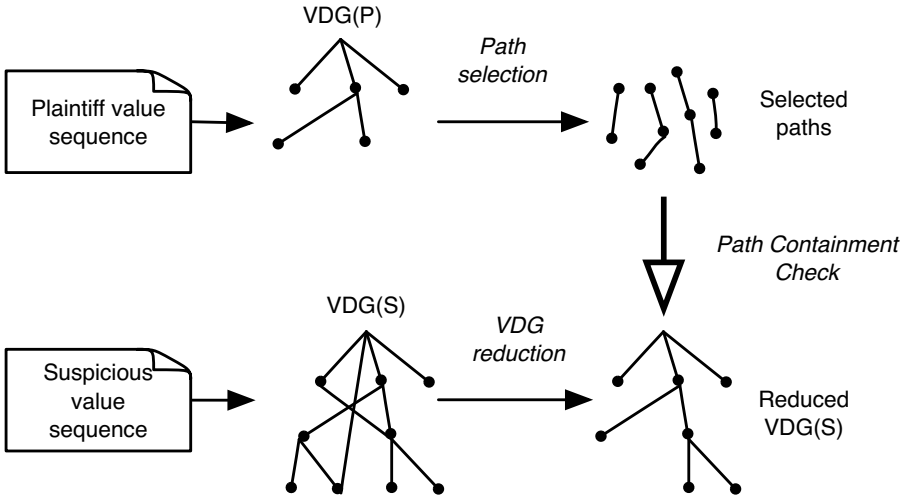

Fig. 7. VDG based detection overview

is executed prior to ι_j if and only if $i < j$. Multiple instances of one static instruction can appear in a dynamic instruction sequence if the static instruction is executed more than once due to backward branches, i.e., loops or subroutine calls.

Definition 2 (Direct Ancestors and Direct Descendants): Given two runtime values v_i and v_j in a value sequence, v_i is a direct ancestor of v_j and v_j is a direct descendant of v_i if and only if the dynamic instruction computing v_j takes v_i as an input parameter. A dynamic value may have multiple direct ancestors and direct descendants. The set of direct ancestors of v_j is given as $\Phi(v_j)$ where $v_i \in \Phi(v_j)$, and the set of direct descendants of v_i is given as $\Psi(v_i)$ where $v_j \in \Psi(v_i)$. When v_i is removed from P's value sequence during the value sequence refinement phase, for all $v_j \in \Psi(v_i)$, v_i is removed from $\Phi(v_j)$ and $\Phi(v_i)$ is migrated into $\Phi(v_j)$. If $v_i \in \Phi(v_j)$, we say v_j is *derived by* v_i.

Definition 3 (Value Dependence Graph): Given a program P and an execution trace T_p, its value dependence graph VDG(T_p) is a directed acyclic graph $G(V_p, E_p)$. Here V_p is a set of vertices each of which represents a runtime value output from some dynamic instruction of ι. E_p is a set of edges between two distinct dynamic instructions a and b of V_p, and the runtime value represented by b is derived from the runtime value represented by a.

We construct a VDG from a program execution trace, i.e., runtime value sequences in our context and the dynamic instructions that have computed the runtime values. Given a runtime value sequence $T = \{v_0, v_1, \ldots, v_{n-1}\}$, for any two runtime values v_i and v_j, $i < j$, if $v_i \in \Phi(v_j)$, we add an edge (v_i, v_j) in the resulting VDG.

Note that a value dependence graph is constructed from a program execution trace which is a dynamic trace, not from the static program text. The execution trace in our context can be viewed as a list of the output values of a straight-line program, with loops and back-edges unfolded. For example, in a dynamic trace, a single instruction pointer executed many times due to some backward branches such as a loop yields as many output values as the instruction is revisited. Thus the value dependence graph resulted from such traces, or runtime value sequences, does not have back-edges. Thus, VDGs are directed acyclic graphs (DAG). We construct the value dependence graphs on the fly during the simulated execution in QEMU.

6.2 Path Selection

Since there could be large number of paths in a VDG, we pick representative paths to improve efficiency. Instead of enumerating all the paths of VDG(T_p), we pick one dynamic data dependence path per each group of paths that transform an initial input to a final output, since our focus is to capture the dependence among critical values monitored when an input is processed to an output.

Given that we have the plaintiff program P and its execution trace T_p, we construct the value dependence graph from VDG(T_p) the runtime values (refined to expose the core-values) of the plaintiff program from its execution trace. As we discussed in section 6.1, VDG(T_p) is a DAG. Then, with VDG(T_p), we chop the graph into paths starting from each leaf node to the root node. We extract one path per leaf node. Instead of enumerating all the paths of VDG(T_p), we pick one dynamic data dependence path per each group of paths that transform an initial input to a final output because our focus is to capture the dependence among critical values monitored when an input is processed to an output.

Algorithm 1 SELECT_PATH(g)

Require: Directed acyclic graph g.
Ensure: List of selected paths.
  Mark all nodes in g as not visited.
  for all $v \in$ leaf nodes of g **do**
    $w \leftarrow v$
    while w **do**
      Add w at the beginning of $path$
      for all $p \in$ the list of parent nodes of w **do**
        if p is not visited **then**
          Mark p as visited.
          $w \leftarrow p$
          Exit the for loop.
        end if
      end for
      if all p were visited **then**
        $w \leftarrow$ randomly pick a parent of w
      end if
    end while
    Add $path$ to $path_list$.
  end for
  return $path_list$

Our path selection algorithm is shown in Algorithm 1. Let us denote by p the path being constructed. For each leaf node of VDG(T_p), we mark the leaf node as visited, and add it to p. Then we pick an unvisited predecessor of the leaf node, mark it as visited, and add it to p. If all predecessors are visited, we randomly pick one. After repeating this procedure until the root of VDG is reached, p is ready to be added to the set of selected paths.

This simplification could increase the chances of false positives, but the impact would be insignificant since we perform the path containment test on hundreds of paths containing thousands of values in each.

6.3 Path Containment Test

All the values contained in a path of VDG have partial ordering dependence; therefore reordering techniques cannot change their orders. Given the paths of VDG(T_p) of the plaintiff, we check whether these paths are contained in VDG(T_s) to measure the similarity of S to P. This turns out to be a critical observation that has deeper implications which we formalize as follows in the form of a lemma. This lemma suggests that VDG-based similarity checking can be done efficiently using path containment tests.

Lemma 1 (Path Containment): Given two programs P and S, and their corresponding execution traces T_p and S_p, resulted from the same input, if the following conditions hold, then for any path $\mathcal{P}_p$ in the VDG(T_p), there always exists a path $\mathcal{P}_s$ in the VDG(T_s) such that all the values contained in $\mathcal{P}_p$ will be a continuous or non-continuous subsequence of $\mathcal{P}_s$.

1) S is obtained from P through semantics-preserving transformations.
2) The values in $\mathcal{P}_p$ are core-values.

Algorithm 2 MATCH_PATH(p, g, i, n)

Require: Source path p, target graph g, i= current position in p, n= node to start path containment test with.
Ensure: The number of matched values.
  if n matches $p[i]$ **then**
    if i is the end of p **then**
      set $stop_flag$
      return 1
    end if
    $num_match \leftarrow$ MATCH_SUB(p, g, $i+1$, n) + 1
  else
    $num_match \leftarrow$MATCH_SUB(p, g, i, n)
  end if
  return num_match

Algorithm 3 MATCH_SUB(p, g, i, n)

Require: Source path p, target graph g, i= current position in p, n= node to start path containment test with.
Ensure: The number of matched values.
  for all $c \in$ children of n **do**
    $m \leftarrow$ MATCH_PATH(p, g, i, c)
    $num_match \leftarrow$ maximum of m
    if $stop_flag$ is set **then**
      exit the loop
    end if
  end for
  return num_match

Given a path from the plaintiff VDG, we test whether or not this path is embedded in the suspicious VDG. If yes, it indicates that the semantics of this path is copied or transformed into the suspicious program. Collectively, the path containment test indicates the semantics embedding from the plaintiff into the suspicious program. The overall process of the path-containment test based on Lemma 1 is shown in Algorithm 2 and Algorithm 3.

Algorithm 2 defines a function called MATCH_PATH, which takes as inputs a plaintiff path p from the plaintiff VDG, a target graph g (the suspicious VDG), an index i, and a node n. This function computes the maximal length of the matching subsequences between the source path p and each path from the suspicious VDG g, starting at the source path position $p[i]$ and the target graph node n. It first compares $p[i]$ and node n in the target graph, if it matches, increment the matching score by 1 and continue with the path matching to each of the child node of n at position $p[i+1]$. This is done by calling to the function MATCH_SUB in Algorithm 3. Otherwise, do not increment the matching score, but continue with the matching to each of the child nodes of n at position $p[i]$.

Algorithm 3 defines a function called MATCH_SUB, which is similar to MATCH_PATH defined in Algorithm 2. The difference is that MATCH_SUB starts the path matching at each child node of the input node n, and output the maximal length, while MATCH_PATH starts at the current node n. For each child node c, it simply calls MATCH_PATH to compute the maximal matching length, and output the maximal one among all children. MATCH_PATH and MATCH_SUB are mutually recursive. The return condition is either MATCH_PATH reaches the end of the source path p or MATCH_SUB tested all the children of node n.

At a high level, these two algorithms performs a heuristic matching of a source path against all path s in the VDG of a suspicious program. Since VDGs are directed acyclic graphs, the process is quite efficient.

6.4 VDG Reduction

Moreover, we further improve path matching performance through removing useless nodes and edges from VDG(T_s). When constructing VDG(T_s) from the runtime values of T_s, we remove the nodes whose values do not appear in VDG(T_p). Construction of VDG(T_s) is done within $O(E_s)$ time where E_s is the number of edges in VDG(T_s) since, by nature of dynamic trace, the nodes appearing in the dynamic dependence edges are created in a topologically sorted order. Given the hash table containing all the values from VDG(T_p), this reduction can be done on-the-fly during the construction of VDG(T_s) with little extra cost. Since the time complexity of the DFS search on a DAG is known to be $O(n^2)$, where n is the number of nodes, reducing the number of nodes dramatically improves the matching performance. In the cases shown in Table 6, we observed that VDG reduction eliminated 80.7% of the edges in VDG(S) on average.

6.5 VDG similarity metric

When p, a path of VDG(P) is compared to VDG(S), the per-path containment score PCS_{path} is computed as follows:

$$\text{PCS}_{\text{path}}(p, \text{VDG}(T_s)) = \frac{\text{\# of matching nodes in } p}{|p|}$$

Given a set of paths extracted from VDG(T_p), we use the weighted average of the per-path containment scores. The longer the path is, the lower the chance of false positives is. 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 to minimize the chance of false positives and false negatives caused by noise in the plaintiff paths. Therefore, we define the path containment score of VDG(T_p) and VDG(T_s) as follows,

$$\text{PCS}(\text{VDG}(T_p), \text{VDG}(T_s)) = \sum_{i=1}^{|\mathcal{P}|} \omega_i \text{PCS}_{\text{path}}(p_i, \text{VDG}(T_s)),$$

where $\mathcal{P}$ is the set of paths selected from VDG(T_p), $|\mathcal{P}|$ is the number of paths in $\mathcal{P}$, p_i is the i-th path of $\mathcal{P}$, and $|p_i|$ is the length of path p_i. ω_i, the weight of i-th path is defined as

$$\omega_i = \frac{|p_i|}{\sum_{k=1}^{|\mathcal{P}|} |p_k|}.$$

6.6 Evaluation of the VDG Based Method

As we discussed in the previous subsections, VaPD can be extended to use the VDG based plagiarism detection. First, we evaluate the effectiveness of VDG-based VaPD in comparison to LCS-based VaPD. We performed the same experiments shown in Section 5.1 and 5.3 to compare the results of the VDG based method with the results of the LCS based method. The detailed results are shown in Fig. 8, 9 and 10. Under all 34 types of obfuscation from SandMark, VDG-based method reports similarity scores of 1.0. For different program comparison, VDG-based approach reports similarity scores which are (up to 24%) lower than those from the LCS-based approach, indicating that VDG is effective in filtering out noise values. Second, to see the impact of a large VDG, we evaluate the performance through running a test case where each VDG contains more than a million edges. We perform path containment test on VDGs constructed from three versions of GCC (3.4.6, 4.1.2, and 4.3.2) compiling a long C source file (xlfun2.c) from XLisp source tree. The results are shown in Table 6, where P is a plaintiff program, S is a suspect program, E_p and E_s are the number of edges in VDG(T_p) and VDG(T_s), respectively, E'_s is the number of edges in VDG(T_s) after VDG reduction, $|\mathcal{P}|$ is the number of paths in VDG(T_p), t is the elapsed time when testing all paths, and t_{10} is the elapsed time when testing the top 10 percent longest paths. Note that the only reason that we tested all the paths of VDG(T_p) generated from a long input is to see the impact on the performance. In this experiment, we got similar results by testing only the top 10% longest paths obtained from a smaller input file.

In addition, GCC 3.4.6 uses a Bison generated parser, and it was completely rewritten into a hand-written recursive descent parser since GCC 4.1.0 was released. Our path-containment scores confirm this fact: The containment scores of GCC 3.4.6 against GCC 4.1.2 and 4.3.2 are 0.68 and 0.72, respectively, and the containment score of GCC 4.1.2 against 4.3.2 is 0.92.

7 Discussion

7.1 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.

TABLE 6
Number of edges and elapsed time of the VDG based method comparing GCC versions.

P	S	E_p	E_s	E'_s	$\|\mathcal{P}\|$	t(min)	t_{10}(min)	PCS
3.4.6	4.1.2	1,725K	1,968K	303K	215K	276	90	0.68
3.4.6	4.3.2	1,725K	1,491K	298K	215K	273	130	0.72
4.1.2	4.3.2	1.928K	1,457K	330K	90K	92	55	0.92

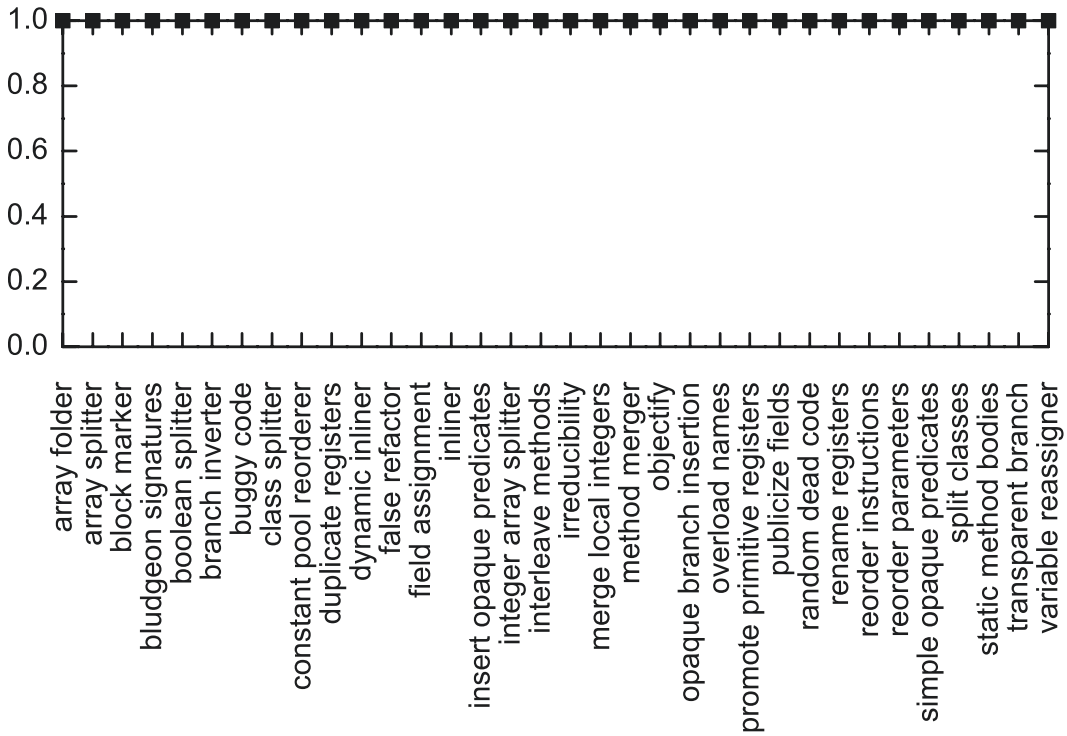

Fig. 8. VDG-based similarity scores (y-axis) of original JLex to obfuscated ones (x-axis)

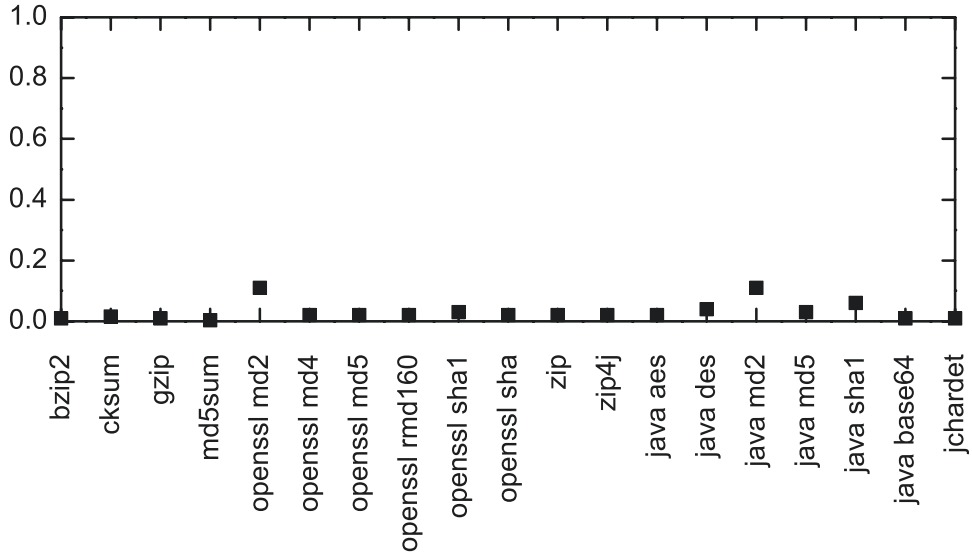

Fig. 9. VDG-based similarity scores (y-axis) of original JLex to other programs written in Java and C (x-axis)

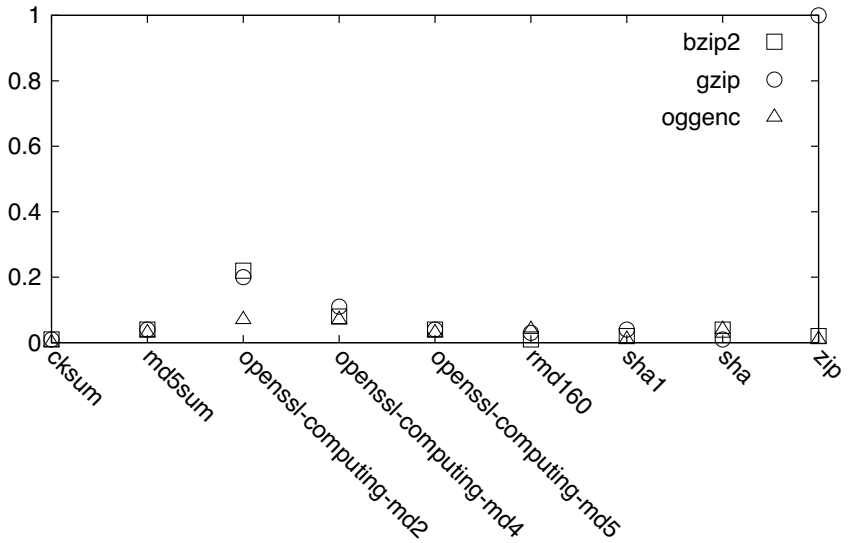

Fig. 10. VDG-based similarity scores (y-axis) of different programs (x-axis)

```
001:  x = 10 ;
002:  x = x + 1 ;
003:  y = ... (x - 1) ... ;
004:  x = x - 1 ;
005:  out(x) ;
006:  out(y) ;
```

(a) Simple Data Transformation

Original Code

```
001:  x = 0 ;
002:  y = 1 ; // y is tainted
003:  i = 0 ;
004:  while (i < 5) {
005:      i ++ ;
006:      x = x + i ;
007:      y = y * i ;
008:  }
009:  out(y) ;
```

Transformed Code

```
001:  x = 0 ;
002:  y = 1 ; // y is tainted
003:  i = 0 ;
004:  while (i < 5) {
005:      i ++ ;
006:      x = x + i ;
007:      y = (y + i - x) * i + x ;
008:  }
009:  out(y - x) ;
```

(b) More Complex Data Transformation

Fig. 11. Data Transformation Examples. Underlined codes are added by the transformation.

7.1.1 Data Transformation

Simple data transformations expose the core-values of the original program. Fig. 11(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 Fig. 11(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 [47]. 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.

7.1.2 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 have no motivation 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. To defend this, we can consider sliding over a stream of values so that we may keep only a small portion of a value sequence in memory during runtime.

7.1.3 Loop Rewriting

Another possible counterattack is rewriting a loop in a reverse order. However, automatic loop reversing is quite difficult because it might 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, only reversing the loop counter variable 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.

7.1.4 Other Obfuscations

While we have experimented with several popular *automated* obfuscation tools including KlassMaster [15], Thicket [14], and SandMark [12], we discuss here a few others in the literature.

DonQuixote[4] obfuscates Java programs based on dynamic name resolution techniques [48]. It implements two types of obfuscations. The first type is called API Blinder, which hides reference and invocation to classes, methods and fields. This does not affect VaPD at all since VaPD is a dynamic method and only cares about runtime values. The second type is string encryption, which hides strings in Java programs. Note again that VaPD is based on runtime values, not static strings, and thus this type obfuscation has limited effect on VaPD.

There are a number of other strong (non-automated) data obfuscations. Chow, Johnson, and Gu [49] proposed several encoding methods, including polynomial coding and residue number coding. Polynomial coding is briefly discussed in Section 7.1.1. Residue number coding makes use of Chinese Remainder Theorem and is often used for high-precision arithmetic [50]. In such case, VaPD will not be able to catch the original runtime values as they are encoded with a modular base. However, this encoding is usually not applied widely for normal arithmetic due to its increased cost. Moreover, arbitrary division using residue coding is impossible. This is manifested that this method has not been implemented in those available obfuscation tools. Chow *et al.* [51] proposed a white-box DES implementation that can hide a key in the program. VaPD will not be able to catch the hidden secrets in this case; however, this method is prohibitively expensive for the obfuscation purpose.

Lastly, obfuscation techniques that aim at thwarting static program analysis do not affect VaPD, simply because VaPD relies on the runtime value sequences, not static program text based features. Linn and Debray [52] proposed obfuscations techniques to improve resistance to static disassembly. Kanzaki *et al.* [53] proposed to use self-modification mechanism for program protection. Both try to thwart certain static program analysis, but do not affect runtime program execution behavior, and thus does not have any impact on VaPD.

4. http://se-naist.jp/DonQuixote/

7.2 Core-Part Plagiarism

Core-part plagiarism detection is a harder problem. In such cases, only some part of a program or software product 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 $\mathcal{V}(P)$ be a value based characteristic such as a value sequence extracted from P, a program or a module. Then the value based method can be applied to a subproblem of the core-part plagiarism detection where $I_{\mathrm{PM}} = I_{\mathrm{SM}}$. This often happens in the situations where two software products serve the same purpose. In this case, we can directly search in $\mathcal{V}(\mathrm{S})$ of the suspect program, for $\mathcal{V}(\mathrm{PM})$ of the plaintiff module to check whether the module has been plagiarized and where it is located in the suspect program. For example, in the case of web browser layout engine plagiarism, given an input URL I, we can first obtain $\mathcal{V}(\mathrm{PM})$ from the plaintiff layout engine module; then, using the same input I we can obtain $\mathcal{V}(\mathrm{S})$ from the suspect program. If the plaintiff program and the suspect program use the same layout engine, then $\mathcal{V}(\mathrm{PM})$ and part of $\mathcal{V}(\mathrm{S})$ (*i.e.*, $\mathcal{V}(\mathrm{SM})$) bear significantly similar patterns. Therefore, we can search for $\mathcal{V}(\mathrm{PM})$ in $\mathcal{V}(\mathrm{S})$.

7.3 Semantics Analysis Based Malware Detection

Some malware detection methods use similar techniques as the clone or software plagiarism detection. While the clone detection techniques mainly focus on syntactic similarities of the program source code to find duplicated code fragments, malware detection and plagiarism detection must utilize some indicators of the program semantics to deal with the automated obfuscation techniques which the plagiarizers and malware writers may easily use to avoid detection.

For example, Alzarooni [54] proposed a semantic signature to detect malware variants. A semantic signature is expressed as sequences of program states (i.e., registers and memory locations) representing the evaluation of the execution environment. Our proposed technique (VaPD) is similar in that its semantic indicator is the runtime output values written to the registers and memory locations as a result of the execution of machine instructions. However, to filter out irrelevant values, VaPD uses a dynamic taint analysis technique with the input of the program as the taint seed while Alzarooni's work uses a dynamic backward slicing technique.

7.4 Limitations

Our technique bears the following limitations. First, besides the ability of extracting value sequences from the entire scope of the plaintiff program, 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 7.2. However, we have not yet comprehensively evaluated this issue with real world test subjects. In such case, a more efficient and scalable program emulator or logger other than QEMU may be needed.

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 more 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 which are often not available, 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. The similarity threshold and the credibility and resilience requirements are discussed by Myles and Collberg [25], Lim *et al.* [55], and Mahmood *et al.* [56]. While we do not have experimental results on real-world software plagiarism cases, our experimental results clearly show the different scores between similar and irrelevant software. For all the independent software we evaluated, the similarity scores are all below 0.3, with most of them below 0.1, while for the similar or obfuscated software, the scores are close to 1.0.

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 [42]. Anti-taint attacks and some countermeasures (*e.g.*, tainting the program counter when a tainted conditional is tested [57]) are well summarized by Cavallaro *et al.* [58]. Addressing those issues is orthogonal and out of the scope of this research.

Finally, our prototype implementation is based on QEMU [43], a processor emulator, on the x86 platform. We have experimented with executables compiled from C and Java. In general, our approach is applicable to other platforms and programming languages and systems as runtime core values are difficult to avoid, but we have not tested on those platforms and systems such as .NET, JVM, and other hardware platforms.

7.5 Future and Ongoing Work

As our future work, we are interested in examining 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 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. As an extension to VaPD, we are conducting a study on using *symbolic values* and *constraints*, which reveal the relationship between concrete runtime values, to compare program similarity. Symbolic values and constraints, combined with LCS, can naturally lead to more precise modeling of program semantics. In addition, it would be interesting to study the impact of emulation-based obfuscators such as Themida and Code Virtualizer [59] on VaPD's performance. Such obfuscators encode original program into a specially designed bytecode instruction set and run the bytecode program in an emulator. Our our value-based detection method may be able to handle such obfuscators.

8 Conclusion

Obfuscation resilient code characterization is important for many code analysis applications, including code theft detection. Prior techniques have limited applicability in detecting code theft because they either require source code analysis or cannot handle automated obfuscation tools. Motivated by an observation that some outcome values computed by machine instructions survive various semantics-preserving code transformations, we have proposed techniques to extract and refine runtime values from the outcome of machine instructions. Leveraging such invariant values, our technique is resilient to various control and data obfuscation techniques. To our best knowledge, our work is the first one to explore the existence of *runtime core-values* and use them to characterize code fragments. The proposed approach directly examines executable files and does not need to access the source code of suspicious programs. We have analyzed a number of real world programs, including five XML parsers, bzip2, gzip, oggenc, openssl, zip, and JLex, along with a comprehensive set of obfuscation techniques in Sandmark, KlassMaster, Thicket, and Loco/Diablo. Our results show that the value-based method is effective in identifying software plagiarism.

Acknowledgments

The authors would like to thank Jonas Maebe of University of Ghent for his help in compiling and using *Loco* and *Diablo*; Semantic Designs, Inc. for donating *C/C++ obfuscators*. This work was supported by NSF CCF-1320605, NSF CNS-1223710, NSF CNS-0905131, NSF CNS-0916469, AFOSR FA9550-07-1-0527 (MURI), ARO W911NF-09-1-0525 (MURI), ARO W911NF-13-1-0421 (MURI), and AFRL FA8750-08-C-0137. Xiaoqi Jia was supported by National Natural Science Foundation

of China (NSFC) under Grant No. 61100228, the National High-tech R&D Program of China under Grant No. 2012AA013101, and the Strategic Priority Research Program of the Chinese Academy of Sciences under Grant No. XDA06030601 and XDA06010701.

REFERENCES

[1] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of 2nd Working Conference on Reverse Engineering (WCRE '95)*, 1995, pp. 86–95.

[2] 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.

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

[4] J. Krinke, "Identifying similar code with program dependence graphs." in *Proceedings of Eighth Working Conference on Reverse Engineering (WCRE '01)*, 2001, pp. 301–309.

[5] T. Kamiya, S. Kusumoto, and K. Inoue., "CCFinder: a multilinguistic token-based code clone detection system for large scale source code." *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[6] 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, pp. 321–330.

[7] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, ser. ESEC-FSE '07, 2007, pp. 55–64.

[8] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, 2007, pp. 96–105.

[9] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," The Univeristy of Auckland, Tech. Rep. 148, Jul. 1997.

[10] C. S. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, 1998, pp. 184–196.

[11] C. Wang, "A security architecture for survivability mechanisms," Ph.D. dissertation, University of Virginia, Charlottesville, VA, USA, 2001, adviser-John Knight.

[12] C. Collberg, G. Myles, and A. Huntwork, "Sandmark–a tool for software protection research," *IEEE Security and Privacy*, vol. 1, no. 4, pp. 40–49, 2003.

[13] M. Madou, L. Van Put, and K. De Bosschere, "Loco: An interactive code (de)obfuscation tool," in *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM '06)*, 2006, pp. 140–144.

[14] Semantic Designs, Inc., "Thicket™," http://www.semanticdesigns.com.

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

[16] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD '06)*, 2006, pp. 872–881.

[17] H. Tamada, M. Nakamura, A. Monden, and K. ichi Matsumoto, "Design and evaluation of birthmarks for detecting theft of Java programs," in *IASTED Conference on Software Engineering (IASTED SE '04)*, February 2004, pp. 569–574, innsbruck, Austria.

[18] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991.

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

[20] 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.

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

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

[23] J.-H. Ji, G. Woo, and H.-G. Cho, "A source code linearization technique for detecting plagiarized programs," in *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE '07)*, 2007, pp. 73–77.

[24] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proceedings of the 2005 ACM symposium on Applied computing (SAC '05)*, 2005, pp. 314–318.

[25] G. Myles and C. S. Collberg, "Detecting software theft via whole program path birthmarks," in *Proceedings of 7th International Conference on Information Security (ISC '04)*, 2004.

[26] 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, pp. 274–283.

[27] 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.

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

[29] F. Bellard, "Tiny C compiler," http://bellard.org/tcc/.

[30] Open Watcom Contributors, "Open Watcom," http://www.openwatcom.org.

[31] A. Sebjornsen, 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, pp. 117–128.

[32] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, November 2014.

[33] 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)*, 2009, pp. 280–290.

[34] F. Zhang, Y. Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA '12)*. ACM, 2012, pp. 111–121.

[35] F. Zhang, D. Wu, P. Liu, and S. Zhu, "Program logic based software plagiarism detection," in *Proceedings of the 25th annual International Symposium on Software Reliability Engineering (ISSRE 2014)*, November 2014.

[36] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party Android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy (CODASPY '12)*, 2012.

[37] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2012.

[38] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang, "Plagiarizing smartphone applications: Attack strategies and defense techniques," in *Engineering Secure Software and Systems, Lecture Notes in Computer Science*, 2012, pp. 106–120.

[39] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on Android markets," *Computer Security–ESORICS 2012*, pp. 37–54, 2012.

[40] H. Huang, S. Zhu, P. Liu, and D. Wu, "A framework for evaluating mobile app repackaging detection algorithms," in *Proceedings of the 6th International Conference on Trust and Trustworthy Computing (TRUST '13)*, 2013.

[41] F. Zhang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2014)*, July 2014.

[42] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the Network and Distributed System Security Symposium (NDSS '05)*, 2005.

[43] F. Bellard, "Qemu, a fast and portable dynamic translator," in *ATEC '05: Proc. of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41.

[44] I. D. Baxter, C. Pidgeon, and M. Mehlich, "DMS: Program transformations for practical scalable software evolution," in *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, 2004, pp. 625–634.

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

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

[47] S. Drape, A. Majumdar, and C. Thomborson, "Slicing aided design of obfuscating transforms," in *6th IEEE/ACIS International Conference on Computer and Information Science (ICIS '07)*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 1019–1024.

[48] H. Tamada, M. Nakamura, A. Monden, and K. ichi Matsumoto, "Introducing dynamic name resolution mechanism for obfuscating system-defined names in programs," in *Proc. IASTED International Conference on Software Engineering (IASTED SE 2008, 598-074)*, February 2008, pp. 125–130.

[49] S. T. Chow, Y. Gu, and H. J. Johnson, "Tamper resistant software encoding," Jan. 11 2005, uS Patent 6,842,862.

[50] D. Knuth, *The Art of Computer Programming, Volume Two, Seminumerical Algorithms*. Addison-Wesley, 1998.

[51] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot, "A white-box DES implementation for DRM applications," in *Digital Rights Management*. Springer, 2003, pp. 1–15.

[52] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security (CCS 2003)*. ACM, 2003, pp. 290–299.

[53] Y. Kanzaki, A. Monden, M. Nakamura, and K.-i. Matsumoto, "Exploiting self-modification mechanism for program protection," in *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC 2003)*. IEEE, 2003, pp. 170–179.

[54] K. M. A. Alzarooni, "Malware variant detection," Ph.D. dissertation, Unversity College London, 2012.

[55] H. il Lim, H. Park, S. Choi, and T. Han, "A method for detecting the theft of Java programs through analysis of the control flow information," *Information and Software Technology*, vol. 51, no. 9, pp. 1338–1350, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584909000469

[56] Y. Mahmood, Z. Pervez, S. Sarwar, and H. Ahmed, "Similarity level method based static software birthmarks," in *High Capacity Optical Networks and Enabling Technologies, 2008. HONET 2008. International Symposium on*, Nov 2008, pp. 205–210.

[57] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, "Dynamic spyware analysis," in *USENIX Annual Technical Conference*. USENIX, 2007, pp. 233–246.

[58] L. Cavallaro, P. Saxena, and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," in *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '08)*, 2008, pp. 143–163.

[59] Oreans Technologies, "Code Virtualizer," http://www.oreans.com/codevirtualizer.php.