Detecting Software Theft via System Call Based Birthmarks

Xinran Wang, Yoon-Chan Jhi, Sencun Zhu
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802
Email: {xinrwang, jhi, szhu}@cse.psu.edu

Peng Liu
College of Information Sciences and Technology
Pennsylvania State University
University Park, PA 16802
Email: pliu@ist.psu.edu

***Abstract*—Along with the burst of open source projects, software theft (or plagiarism) has become a very serious threat to the healthiness of software industry. Software birthmark, which represents the unique characteristic of a program, can be used for software theft detection. We propose two system call based software birthmarks: SCSSB (System Call Short Sequence Birthmark) and IDSCSB (Input Dependant System Call Subsequence Birthmark), and examine how well they reflect unique behavioral characteristics of a program. To our knowledge, our detection system based on SCSSB and IDSCSB is the first one that is capable of software *component* theft detection where only partial code is stolen. We demonstrate the strength of our birthmarks against various evasion techniques, including those based on different compilers and different compiler optimization levels as well as those based on very powerful obfuscation techniques supported by SandMark. Unlike the existing work that were evaluated through small or toy software, we also evaluate our birthmarks on a set of large software (web browsers). Our results show that system call based birthmarks are very practical and effective in detecting software theft that even adopts advanced evasion techniques.**

***Keywords*-detection; plagiarism; birthmark;**

I. Introduction

Software theft is an act of reusing someone else's code, in whole or in part, into one's own program in a way violating the terms of original license. Along with the rapid developing software industry and the burst of open source projects (e.g., in SourceForge.net there were over 180,000 registered open source projects as of Aug.2008), software theft has become a very serious concern to honest software companies and open source communities. To protect software from theft, Collberg and Thoborson [1] proposed software watermark techniques. Software watermark is a unique identifier inserted into the protected software, which is hard to remove but easy to verify. However, theoretically, any watermark can be removed by sufficiently determined attackers [2]. As such, a new kind of software protection techniques called software birthmark were recently proposed [3]–[6]. A software birthmark is a unique characteristic that a program inherently possesses, which can be used to identify the program. Though some initial research has been done on software birthmarks, existing schemes are still limited in meeting the following highly desired requirements: (R1) Resiliency to semantics-preserving obfuscation techniques [7]; (R2) Capability to detect theft of components, which may be only a small part of the original program; (R3) Scalability to detect large-scale commercial or open source software theft; (R4) Applicability to binary executables, because the source code of a suspected software product often cannot be obtained until some strong evidences are collected. To see the limitations of the existing techniques with respect to above requirements, let us break them down into four classes: (C1) static source code based birthmark [3]; (C2) static executable code based birthmark [8]; (C3) dynamic whole program path(WPP) based birthmark [5]; (C4) dynamic API based birthmark [4], [6]. We briefly summarize their limitations as follows: C1, C2 and C3 techniques cannot satisfy requirement R1 because they are vulnerable to the techniques such as outlining and ordering transformation; C2, C3 and C4 detect only whole program theft thus cannot satisfy R2; C1 cannot meet R4 because it requires source code; none of the existing techniques has been evaluated on large-scale programs.

In this paper, we propose behavior based birthmarks for meeting these key requirements. Behavior characteristic has been widely used to identify malware from benign programs [9], [10]. While two software independently developed for the same purpose share many common behaviors, one usually contains unique behaviors compared to the other due to the difference in features and implementation details. For example, the Gecko HTML rendering engine [11] supports MathML, while KHTML [12] does not; Gecko implements RDF (resource description framework) to manage resources, while KHTML implements its own framework. The unique behaviors can be used as birthmarks for software theft detection. Note that we aim to protect large-scale software. The programs or components that are too small to bear unique behaviors are out of our scope.

A system call sequence is a good candidate for behavior based birthmarks because it shows the interaction between a program and the operating system, where the interaction

The work of Wang and Zhu was supported by CAREER NSF-0643906. The work of Jhi and Liu was supported in part by AFOSR MURI grant FA9550-07-1-0527, ARO MURI: Computer-aided Human Centric Cyber Situation Awareness, and NSF CNS-0905131. This work was also supported by AFRL award FA8750-08-C-0137.

is an essential behavioral characteristic of the program [9], [10]. Although a code stealer may apply compiler optimization techniques or sophisticated semantic-preserving transformation on a program to hide original code, these techniques usually do not change the sequence of system calls. It is also difficult to avoid system calls, because system calls are the only way for a user mode program to request kernel services in modern operating systems. For example, in operating systems such as Unix/Linux, there is no way to go through the file access control enforcement other than invoking open()/read()/write() system calls.

We develop two system call based dynamic birthmarks SCSSB (System Call Short Sequence Birthmark) and IDSCSB (Input Dependant System Call Subsequence Birthmark) to meet aforementioned key requirements. To extract SCSSB, short subsequences of system calls are collected from the whole system call sequence during the execution of a program with a given input. Observing that some system call short sequences in the set are commonly found in many other programs and hence they do not represent the unique behavior characteristic of the program, we establish a database of the common short sequences of system calls from various programs. SCSSB is then extracted by removing the commonly found short sequences from the system call short sequences of the program. To address noise injection attacks, we further propose IDSCSB, which involves slightly higher performance overhead than SCSSB. In IDSCSB, with two different inputs, we first extract two whole system call sequences that only include the system calls dependent to the individual input. The IDSCSB is generated from each system call sequence by excluding the system calls which appear in both system call sequences in common. In this way, we can remove noisy system calls that are intentionally injected. Our contributions are threefold:

- We proposed a novel type of birthmarks, which exploits short sequences or input dependent subsequences of system calls to represent unique behaviors of a program. Without requiring any source code from the suspect, the system call birthmark detection is a practical solution for reducing plaintiff's risks of false accusation before filing an intellectual property lawsuit.
- As one of the most fundamental runtime indicators of program behaviors, our system call birthmarks are resilient to various advanced obfuscation techniques. Our experiment results indicate that it not only is resilient to simple evasion techniques such as different compilers and different optimization levels, but also successfully discriminates code obfuscated by *SandMark* [13], a state-of-the-art obfuscator.
- To our best knowledge, SCSSB and IDSCSB are the first birthmarks that are proposed to detect software component theft. Moreover, unlike existing techniques that are evaluated with small or toy programs, we evaluate our birthmark on a set of large software (web browsers). Thus, our evaluation shows the proposed birthmarks are practical.

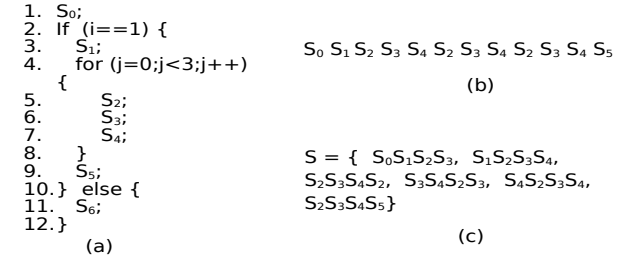


Figure 1. (a) An example program. S_0,...,S_6 denote system calls. (b) The system call trace generated by the execution of the example program with input $i = 1$. (c) 4-long system call sequence set of the system call trace.

II. Problem Formalization

A. Software Birthmarks

A software birthmark is an inherent characteristic of a program, which can uniquely identify the program. Before we formally define software birthmarks, we first define *copy*. Program q is a copy of program p, if q is exactly the same as p. q is still considered as a copy of p after a sophisticated software thief applies semantic preserving transformation such as obfuscation techniques and compiler optimization. Tamada et al. [3] and Myles et al. [5] define software birthmark and dynamic software birthmark as follows:

Definition 1: (Software Birthmark) Let p, q be programs or program components. Let $f(p)$ be a set of characteristics extracted from p. We say $f(p)$ is a birthmark of component p, only if both of the following conditions are satisfied:

1) $f(p)$ is obtained only from p itself.
2) program q is a copy of $p \Rightarrow f(p) = f(q)$.

Software birthmarks can be classified into static birthmarks and dynamic birthmarks. A static birthmark relies on syntactic structure of a program. Existing static birthmarks are vulnerable to simple semantic-preserving transformations [5]. On the other hand, dynamic birthmarks rely on the runtime behavior of a program, which is more difficult to be altered through the code obfuscation techniques. In this paper, we propose two dynamic birthmarks.

Definition 2: (Dynamic Software Birthmark) Let p, q be programs or program components. Let I be an input to p and q. Let $f(p, I)$ be a set of characteristics extracted from p by executing p with input I. $f(p, I)$ is a dynamic birthmark of p, only if both of the following conditions are satisfied:

1) $f(p, I)$ is obtained only from p executed with input I
2) q is a copy of $q \Rightarrow f(p, I) = f(q, I)$

B. System Call Birthmarks

Definition 3: (System Call Trace) Let p be a program or a program component. Let I be an input to p. A system call trace $T(p, I)$ is the trace of system calls called by program p during the execution of program p with input I.

Figure 1(b) shows an example of a system call trace. For simplicity, we show only the order of the system calls. The actual system call traces also contain the parameter values passed to the system calls and the return values.

Definition 4: (System Call Sequence Set) Let p be a program or a program component. Let I be an input to p. Let $T(p, I)$ be the trace of system calls called by p during the execution of p with input I. Then, k-long system call sequence set $S(p, I, k)$ is defined as follows:

$$S(p, I, k) = \{t \mid t \; is \; a \; substring \; of \; T(p, I) \; and \; |t| = k\}$$

Figure 1(c) shows an example of a 4-long system call sequence set.

C. System Call Short Sequence Birthmark

Short sequences of system calls have been widely used for intrusion detection systems to detect irregularities in the behavior of a program for many years [14]. Here, we use system call short sequences as a birthmark to detect similarity of two programs. We define system call short sequence birthmark as follows:

Definition 5: (SCSSB: System Call Short Sequence Birthmark) Let p be a program or a program component. Let I be an input to p. System call short sequence birthmark $SCSSB(p, I, k)$ is defined as a subset of a system call sequence set $S(p, I, k)$ that satisfies the following conditions:

1) q is a copy of $p \Rightarrow SCSSB(p, I, k) = SCSSB(q, I, k)$ for any I.
2) q is different from $p \Rightarrow SCSSB(p, I, k)$ should not appear in q's execution instances.

D. Measurement of Birthmark Similarity

The simplest measurement of similarity of two programs is *resemblance*. Assuming a birthmark is a set of values that represent unique characteristics of a program, we can define resemblance of two birthmarks A and B using set operations: $R(A, B) = \dfrac{|A \cap B|}{|A \cup B|}$. Here the $\cap$ and $\cup$ operations are set intersection and union operations, respectively, and the $||$ operation denotes set cardinality. However, a software plagiarizer can deliberately insert noise in a stolen program. As a consequence, the resemblance is decreased so that the stolen program can evade detections. In addition, the resemblance is not an accurate measurement for core component theft, because the core component may be only a small part of the whole program. To overcome the limitation of resemblance measurement, we define *containment* of two birthmarks.

Definition 6: (Containment) The containment of A in B is defined as:

$$C(A, B) = \frac{|A \cap B|}{|A|}$$

Here A is the birthmark of a plaintiff program or its component, and B is the birthmark of a suspect program.

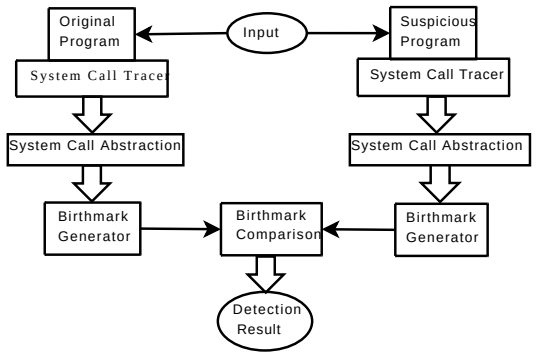


Figure 2. System Design Diagram

From above definition, although a plagiarist may be able to inject noise into B, as long as we can ensure $|A \cap B|$ will not change, the containment measurement will be robust to such attacks and suitable for core component theft detection.

III. System Design and Implementation

Figure 2 shows the generic system diagram of our software theft detection system. Given two programs in binary executable, one is the plaintiff program and the other is the suspect one, we first select inputs to feed both programs. Note that to avoid false positives and false negatives caused by program randomness, a number of different inputs should be tested. In the runtime we use *system call tracer* to record the whole system call sequence from each of the execution with the same input. Then, *system call abstraction* removes the system calls that are normally not related to program behaviors. Next, system call birthmarks are generated by *birthmark generator* and their similarity is measured. Finally, given a detection threshold, our system reports the detection result. Let us describe these components in detail.

A. System Call Tracer

The simplest way to record system call traces is to use a tool such as *strace* in Linux. Strace can record all system calls invoked by a process, but it does not provide the thread identifier when tracing a multi-threaded program. Thus, due to the scheduling of threads, strace cannot always generate accurate system call sequence birthmarks.

Therefore, we implemented SATracer based on Valgrind [15]. Valgrind is an open source dynamic emulator. It supports extensions called skins, which can dynamically instrument a program. SATracer records the system call traces of a program by running the program in its own emulation environment. It records the system call number as well as the process and thread numbers when a system call is invoked.

SATracer can also mark whether a system call is called by a specific component. This is useful for detecting software component theft because we need to know which system calls are invoked from which component of a plaintiff program. Specifically, there are two options. one method is

to check whether a system call is called by a subroutine of the component during the execution. We first prepare a list of all subroutines of the component in SATracer. The list is automatically generated by analyzing the source files of the component using parser Elsa [16]. Then, SATracer checks the execution stack of the running thread when a system call is called from one of the subroutines. If a subroutine in the execution stack exists in the prepared list, it must be that this system call was called by the component and hence it will be marked. Note that we assume that we have control to the compiling process of plaintiff programs so that symbol tables are kept in the executables. Alternatively, if we can compile the specific component to a dynamically linked library, a simpler method is used. Valgrind supplies a function which maps a subroutine to the library which the subroutine belongs to. Therefore, if a subroutine in the execution stack is called from the component library during the execution of a plaintiff program, it will be marked. In either way, we can generate a birthmark for a component of the plaintiff program.

B. System Call Abstraction

Since it is possible that two different system call sequences represent the same behavior, we do not extract birthmarks directly from the raw system call sequences. To address this, we abstract the system calls to a higher level in the following way. First, we ignore the system calls that apparently do not represent the behavior characteristic of a program. For example, a libc *malloc* function is normally implemented by system call *brk* and/or *mmap*. The *mmap* system call is used when extremely large segments are allocated. The *brk* system call changes the size of the heap to be larger or smaller as needed. It is normally called to grab a large chunk of memory and then split it as needed to get smaller chunks in C function *malloc*. As such, not every malloc in C need a system call and two programs with the same behavior may have very different memory management system call sequences. Fortunately, we can ignore all memory management system calls, because they do not represent the behavior characteristic of a program. Second, we consider aliases or multiple versions of a system call as the same in system call birthmarks. For example, *fstat(int fd, struct stat *sb)* system call is the same as *stat(const char *path, struct stat *sb)* except that *fstat* uses the file descriptor *fd* as its parameter instead of the file name *path* . We consider them the same. This not only reduces the sophistication of dealing with many different system calls, but also helps avoid the counterattack where an attacker replaces one system call with another. Finally, since failed system calls do not affect the behavior characteristic of a program, they are also ignored. For example, assume that a program opening a file fails at the first attempt and then succeed in the next time. Although system call *open* is called twice, the first failed call should be removed.

C. Birthmark Generator

Birthmarks are extracted from abstracted system calls provided by the previous step. As to the SCSSB birthmark, We extract it based on its definition. Condition 1 in Def. 5 tells us that given the same input the extracted system call sequence should be the same. Therefore, we should remove those loading-environment-dependent system calls. To achieve this, the program is run multiple times with the same input to find the common subsequences of the multiple system call traces. Condition 2 in Def. 5 tells us that the SCSSB of a program should be unique to the program; therefore, we should remove the (noisy) system calls common to the other programs. To do this, we establish a database of common system call short sequences by analyzing various sample programs in the wild. After that, we remove these noise from our system call short sequences and get the SCSSB birthmark.

D. Input Dependant System Call Subsequence Birthmarks

As the next section will show, SCSSB is robust to the existing obfuscation techniques. However, if an attacker can insert arbitrary system calls into the original program meanwhile preserving its original semantics (although we have not seen such automated tools yet), the original SCSSB could be polluted or even destroyed.

To address the system call injection attack, next we propose an input dependent system call subsequence birthmark (IDSCSB), which introduces slightly higher performance overhead than SCSSB. We observe that many system calls in a system call sequence are independent to the input and do not reflect the semantic characteristic of a program for a given input. A system call is said to be *dependant* to input if any of the following conditions is true when the input changes: (1) the system call disappears in the system call sequence; (2) parameters to the system call changes; (3) return value of the system call changes. We consider input independent system calls as noise, because an attacker may deliberately inject them. Therefore, we only extract input dependent system calls as a birthmark.

Definition 7: (IDSCSB: Input Dependant System Call Subsequence Birthmark) Let p be a program or a program component. Let I be an input to p and J another input. Let $T(p, I)$ and $T(p, J$ be system call traces generated by executing program p with input I and J, respectively. Input dependent system call subsequence birthmark is defined as:

$$IDSCSB(p, I) = \{s | s \in T(p, I) \; and \; s \notin T(p, J)\}$$

We also use the *containment* measurement to compare two IDSCSB birthmarks, but revise its definition for IDSCSB birthmarks by replacing the $\cap$ operation with the computation of the longest common subsequence (LCS) and replacing $||$ operation with the length of a system call trace. Note that LCS does not require every subsequence to be a continuous segment of the mother sequence. For example,

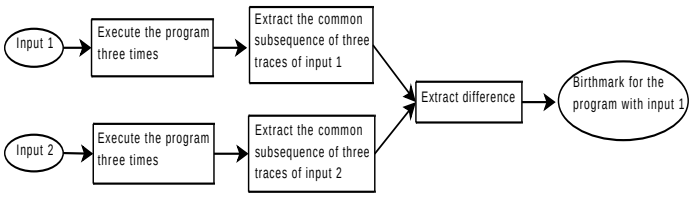


Figure 3. The process of extracting IDSCSB Birthmarks

both {1, 6, 120} and {2, 24} are valid subsequences of the value sequence {1, 2, 6, 24, 120}. In this sense, it is resilient to noise injection attacks.

Extracting IDSCSB Birthmarks Figure 3 shows the process of extracting IDSCSB Birthmarks. First, we prepare two different inputs, and generate the system call sequence for one input. To remove environmental noise, we extract the common subsequence of the multiple system call traces obtained by running a program multiple times with the same input. In our experiment, we run the target Second, we generate the system call sequence for the other input. The same method as in the first step is used to remove noise. Finally, the results from above two steps are compared and the system calls that are independent to both of the inputs are identified. The IDSCSB birthmark is generated by extracting the system calls which appear only in the result of the first (or the second, depending on which input we would use to detect plagiarism) input.

There are two additional implementation details. First, some parameters and return values of system calls such as "file id" and "process id" are ignored because the parameters vary when the execution environment changes. Second, to control the length of the system call log, large parameters over 32 bytes are hashed by the MD5 algorithm and only their hash values are recorded.

IV. Evaluation

In Section 1 we mentioned four key requirements on software theft detection. It is easy to see R4 (Applicability to binary executables) is already met by our design. In this section, we evaluate the performance of SCSSB and IDSCSB birthmarks with respect to four primary criteria: (M1) capability to detect whole program theft (M2) capability to detect component theft for large-scale programs, (M3) credibility to independently developed program, and (M4) resiliency to obfuscation. These four criteria contain more than R1, R2 and R3 because of M3.

In this section, we will first demonstrate the strength of our birthmarks against evasion techniques that apply different compilers or different compiler optimization levels. Then, we will evaluate criteria M1, M2, M3 and M4 for both SCSSB and IDSCSB against some advanced obfuscation techniques and 15 real-world large applications.

For ease of presentation, before showing the results we first introduce SandMark [13], a tool developed at the University of Arizona for analyzing and processing Java byte code. We use SandMark because it is the only free and powerful software with a comprehensive list of fully functioning code obfuscation algorithms. Note that our system call birthmark extractor built upon Valgrind reads only x86 Linux executables. To feed the extractor with Java applications, we convert Java class files to x86 executable using GCJ 4.1.2, the GNU ahead-of-time Compiler for the Java language.

For code obfuscation analysis, SandMark implements 39 byte code obfuscators. Dividing an array to multiple arrays, splitting an array element, promoting all primitive data types to classes, wrap and move a part of a class into a new class, merging two variables in a longer variable, and encrypting string variables are some of the features that SandMark provides for data obfuscation. For control obfuscation, SandMark can insert opaque predicates to every conditional branch, reorder instructions, inline/merge/interleave methods, randomly insert opaque branches within a basic block. Besides, SandMark can alter method-signatures by adding or reordering parameters, change class inheritance structures, and thwart static decompilation.

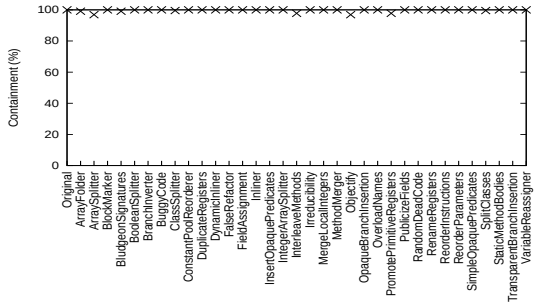
A. Impact of Compiler Optimization Levels

Changing compiler optimization levels is a type of semantic preserving transformation techniques which may be used by a software plagiarist to avoid detection. Here, we evaluated the impact of compiler optimization levels on system call based birthmarks. A set of programs were used: bzip2 (a popular lossless compression tool), gzip (a popular lossless compression tool) and oggenc (an encoding tool for Ogg Vorbis, a free lossy audio compression format). To make them easy to compile with several different compilers, single compilation-unit source code (bzip2.c, gzip.c and oggenc.c) were used. [1] We used five optimization switches (-O0,-O1,-O2,-O3 and -Os) of GCC to generate executables of different optimization levels (e.g., bzip2-O0, bzip2-O3, etc.) for each program. The generated executables were executed with the same input and a system call sequence was recorded for each executable. We compared the system call sequences, and found that applying optimization options did not change the system call sequences of bzip2 and gzip while the system call sequences for oggenc with optimization options -O3 and -Os had only one less "write" system calls compared to the executables with optimization options -O0, -O1 and -O2. This result shows that system call based birthmarks are robust to compiler optimization.

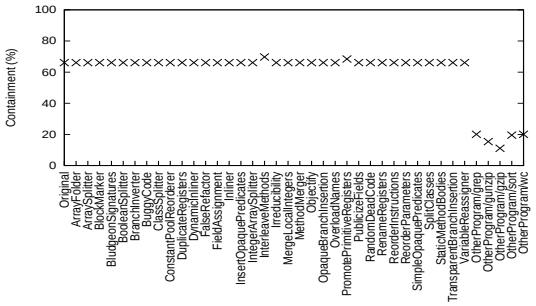
B. Impact of Different Compilers

A software plagiarist may also use a different compiler to avoid detection. To evaluate the impact of applying different compilers, we compared system call sequences with three

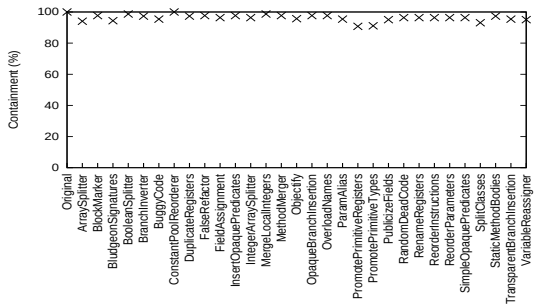
[1] http://people.csail.mit.edu/smcc/projects/single-file-programs/



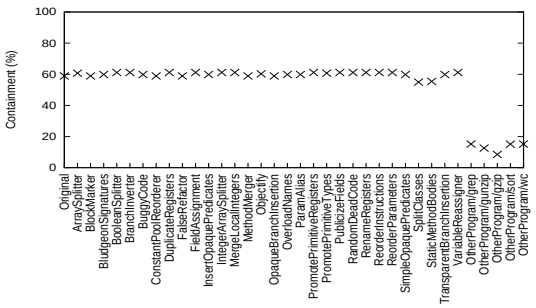
(a) Comparison between original JLex and its obfuscated ones



(b) Comparison between JFlex and different programs (JLex and its obfuscated ones)



(c) Comparison between original JFlex and its obfuscated ones



(d) Comparison between JLex and different programs (JFlex and its obfuscated ones)

Figure 4. The resiliency and credibility of SCSSB system call birthmark.

compilers: GCC, TCC and Watcom. We used the three compilers to generate executables for each of the three programs (e.g., bzip2-gcc, bzip2-tcc) we used before. The generated executables were executed with the same input and a system call sequence is recorded for the each executable. We compared the results of GCC to the results of TCC and Watcom. While the system call sequences of TCC and GCC (both with default optimization levels) are exactly the same, the system call sequences of GCC and Watcom look different. By checking the compilers, we found that the differences were caused by different standard C libraries used by the compilers. Both GCC and TCC use *glibc* while Watcom uses its own implementation. Although the system call sequences between GCC and Watcom looked different, we found such differences can be removed by our *system call abstraction* component. As such, our proposed birthmarks can survive under different compilers.

C. SCSSB Experiment I: M1, M3, M4

To show the resiliency of birthmark SCSSB to obfuscation techniques in whole program theft, we use JLex and JFlex. JLex and JFlex, both written in Java, are two individual programs that were written for the same purpose. They understand the same input syntax, and generate very similar lexical analyzers. The authors of each program claim that the two projects do not share any code in common. We verified this claim by manually comparing both programs using code comparison features of SandMark.

Each of JLex and JFlex is compared to the obfuscated versions of itself using SCSSB with 5-long system call sequences. As a dynamic analysis, SCSSB may not reliably justify (non-)theft based on a single high containment score. Hence, in this experiment, we use 20 different inputs and compute the average containment scores. The result is shown in Figure 4, where the x-axis shows totally 34 obfuscation techniques[2], and the y-axis shows the containment scores. From Figure4(a) and Figure4(c), we can observe that the containment scores between a original program and its obfuscated versions are more than 90%.

In order to verify the credibility of system call birthmarks on independently developed but similar software, JLex is compared to original/obfuscated JFlex while JFLex is also compared to original/obfuscated JLex. In addition, we compare JLex and JFlex to five totally different programs (grep, gunzip, gzip, sort, and wc). From Figure4(b) and Figure4(d), we can observe that the containment scores between JLex and JFlex obfuscated versions or between JFLex and JLex obfuscated versions are less than 67%, between JLex/JFlex and other different programs are no more than 20%. Hence, with appropriate detection thresholds, the detection system based on SCSSB can accurately report the detection results.

We also notice that a code plagiarist may attempt to hide by heavily transforming a stolen program through a series of obfuscators. Therefore, evaluating resiliency of SCSSB against multiple obfuscation techniques applied to

[2]We could not test all 39 obfuscators because some of them failed in transforming JLex and JFlex

Table I
MULTIPLE OBFUSCATION TECHNIQUES APPLIED TO JLEX AND JFLEX

(a) Control obfuscation

Obfuscators	JLex	JFlex
Transparent Branch Insertion	√	√
Simple Opaque Predicates	√	√
Inliner	√	
Insert Opaque Predicates	√	√
Dynamic Inliner	√	
Interleave Methods	√	√
Method Merger	√	√
Opaque Branch Insertion		√
Reorder Instructions	√	√

(b) Data obfuscation

Obfuscators	JLex	JFlex
Array Splitter		
Array Folder	√	
Integer Array Splitter	√	√
String Encoder		
Promote Primitive Registers	√	
Variable Reassigner	√	√
Promote Primitive Types		
Duplicate Registers	√	√
Boolean Splitter	√	√
Merge Local Integers	√	√

a 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 raises practical issues of maintaining the correctness of a target program and its efficiency. For example, we attempted to apply all the 39 obfuscation techniques of SandMark to each of JLex and JFlex, but, after trying several obfuscation orders, only some of them could be successfully applied. To address this problem, we selected obfuscation techniques from two groups following the classification of Collberg et al. [7]: data obfuscation and control obfuscation. We created four test programs by transforming JLex and JFlex through the two groups of obfuscators. As shown in Table I, we could apply eight control obfuscators and seven data obfuscators to JLex and seven control obfuscators and five data obfuscators to JFlex.

We compared the four multi-obfuscated JLex and JFlex to their original programs. The containment scores of JLex to control obfuscated JLex and data obfuscated JLex are 87.9% and 85.2%, respectively. The containment scores of control and data obfuscated JFlex compared to original JFlex are both 96%. This experiment shows that SCSSB is also effective in detecting heavily obfuscated programs.

D. SCSSB Experiment II : M2

In this experiment, we demonstrate SCSSB's ability to detect stolen components, using the layout engines in web browsers. A layout engine is a software component that renders web contents (such as HTML, XML, image files, etc.) combined with formatting information (such as CSS, XSL, etc.) onto the display units or printers. It is not only the core components of a web browser, but also used by many applications that need to render and/or edit web documents.

Gecko [11] is an layout engine used in all Mozilla software and its derivatives. We compute the containment of Gecko in a number of browsers using both SCSSB birthmarks with and without noise. These web browsers include Epiphany, Firefox, Flock, Songbird, Kazehakase, Amaya, Konqueror and Dillo. The first five web browsers are Gecko-based, and the other three are not. Table II(a) shows their relation with the Gecko engine.

Table II

(a) The first set of programs

Program	Type	Gecko Engine
Firefox 3.0.4	Web Browser	Yes
Flock 1.0.8	Web Browser	Yes
Epiphany 2.22.2	Web Browser	Yes
Kazehakase 0.5.2	Web Browser	Yes
Songbird 0.2.5	Media player Web Browser	Yes
Konqueror 3.5.10	Web Browser	No
Amaya 10	Web Browser	No
Dillo 0.8.6	Web Browser	No

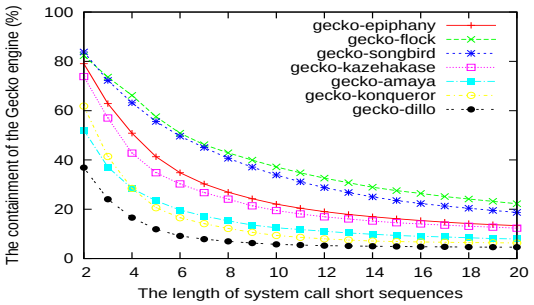
(b) The second set of programs

Program	Type	Gecko Engine
Opera 9.52	Web Browser	No
Evolution 2.22.3	Email Client	No
Gimp 2.4.5	Graph Editor	No
Kile 2.0.0	Latex Editor	No
Open Office 2.4.1	Office	No
Totem 2.22.1	Movie Player	No
Pdfedit 0.3.2	PDF Editor	No

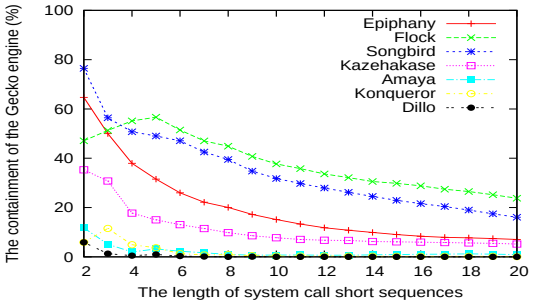
To feed the web browsers the same input, we launch a web browser, open the web site "http://en.wikipedia.org/wiki/Rome", and quit whenever we record the system call sequence. For Firefox, we record the system call trace of the target component (i.e., Gecko). For the other browsers, we recorded their system call sequences through out entire program. Fig5(a) shows the SCSSB containment scores of Gecko, with the x-axis representing the lengths of system call sequences. Although we can observe that the containment of Gecko in Gecko-based browsers is larger than in non-Gecko browsers, the difference is not significant enough for us to draw any conclusion. This indicates that two different programs may overlap significantly in their system call sequence sets. As a result, SCSSB is not a good birthmark without removing noise here. Therefore, we must eliminate noise in the system call sequence sets to obtain a useful SCSSB.

To see the effect of noise removal, we use a set of different programs (shown in Table II(b)) to prune, from the system call sequence sets of the browsers we have tested, the noisy system call sequences that are commonly found in other programs. In this set of programs, only *Opera* is a web browser and we generate its system call sequence sets as before. For the other programs, we do the following: launch the program, open a file and then quit. Their system call traces are recorded during the operations. Figure 5(b) shows the containment of Gecko in these browsers, using SCSSB with noise removal. It shows significant differences

between Gecko-based browsers and non-Gecko browsers. We can also see that five is a good choice for the length of short system call sequences in distinguishing Gecko-based browsers from non-Gecko browsers.



(a) The containment score before noise removal



(b) The containment score after noise removal

Figure 5. The containment of Gecko in the browsers.

E. IDSCSB Experiment I : M1, M3, M4

In the third experiment, we use JLex and JFlex again to show the obfuscation resiliency of IDSCSB in whole program plagiarism. Each of JLex and JFlex is compared to the obfuscated versions of itself, using IDSCSB. The containment scores between original and obfuscated JLex are 100%. We also observed 100% containment with JFlex.

In order to verify the credibility of system call birthmarks, we did the similar experiments using SCSSB. The containment scores between JLex and obfuscated JFlex or between JFLex and obfuscated JLex are less than 46%, between JLex/JFlex and other programs are no more than 7%. Hence, with appropriate thresholds, the detection system based on IDSCSB can accurately report plagiarism.

We also compared the four multi-obfuscated JLex and JFlex (data obfuscated JLex, control obfuscated JLex, data obfuscated JFlex, and control obfuscated JFlex) to their original programs for IDSCSB. We observed containment scores of the multi-obfuscated JLex/JFlex compared to corresponding original versions were all 100%. This experiment shows that IDSCSB is very effective in detecting heavily obfuscated plagiarisms, outperforming SCSSB.

F. IDSCSB Experiment II : M2

Next we evaluate IDSCSB on web browsers and their layout engines. There are three steps to generate input dependent system call sequence birthmarks. First, we generate a system call sequence for an input. As the input to each browser, we follow a simple scenario: launch the web browser, visit the web site "http://en.wikipedia.org/wiki/Germany", and quit. To remove noise, we run the program three times with the same input to find the common subsequence of the three system call traces. Second, we generate a system call sequence for another input "http://www.us.gov". The same method as in the first step is used to remove noise. Finally, the result from above two steps are compared. The input dependent system call sequence birthmark is generated by extracting system calls which appear only in the result from the first step.

We also generated IDSCSB for inputs "http://www.cnn.com" and "http://www.msnbc.com". The result shown in Figure 6 indicates significant differences between Gecko-based browsers and non-Gecko browsers.

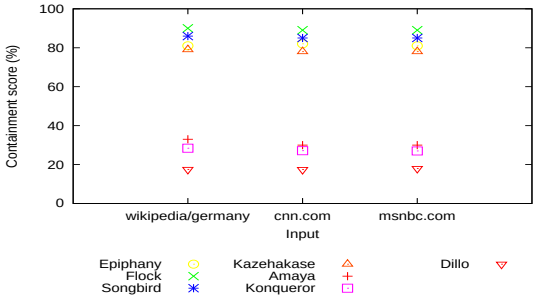


Figure 6. The containment of Gecko in the browsers.

V. Discussion

A. Counterattacks

One of the possible counterattacks to our system call based birthmarks is the *system call injection attack*. An attacker may insert arbitrary system calls in the plagiarism program to reduce the containment score of SCSSB without compromising its original semantics. However, this attack would not bypass the detection of IDSCSB. The containment score of IDSCSB detection will not decrease because (1) these injected system calls will likely be filtered out in the first place, (2) we use longest common subsequence (LCS) to identify similarity between two system call sequences, thus IDSCSB is robust to noise injection attack by its nature.

Another type of possible attacks is the *system call reordering attack*. An attacker may change the order of system calls in the execution path to fool SCSSB and IDSCSB. However, it has quite limited applicability to reorder system calls without affecting the semantics of an original program, due to many reasons including data and control dependencies between the system calls. Moreover, reordering of system calls often affects semantics of the machine instructions surrounding the system calls, which makes the attack much

harder to be accomplished. We will study the feasibility of such attacks in the future.

B. Limitations

Both SCSSB and IDSCSB bear the following fundamental limitation. First, they do not apply if the program of interest does not involve any system calls or has very few system calls, for example, when there are only arithmetic operations in the program. Second, they are not applicable to the programs which do not have unique system call behaviors. For example, the only behavior of a sorting program is to read an unsorted file and print the sorted data. This behavior, which is common to other sorting programs or even irrelevant programs, is not unique. As such, our tool should be used with caution, especially for tiny common programs with few system calls. Third, as a detection system, it bears the same limitation of intrusion detection systems; that is, there exists a fundamental tradeoff between false positives and false negatives. The detection result of our tool depends on the threshold a user defines. To have higher confidence, one should use a large threshold, thus it is likely to increase false negative. In contrast, reducing the detection threshold will increase false positive. Unfortunately, without many real-world plagiarism samples, we are unable to show some concrete results on such false rates although we have showed system call birthmarks exist for all the programs we studied. As such, rather than applying our tool to "prove" software plagiarisms, in practice one may use it to collect some initial evidences before taking further investigations, which often involve nontechnical actions. More discussion can be found in our follow-up work [17].

VI. Related Work

We roughly group the literature into two categories: software birthmark and clone detection.

Software Birthmark: There are four classes of software birthmark.

Static source code based birthmark: Tamada [3] *et al.* proposed four types of static birthmark: Constant Values in Field Variables Birthmark (CVFV), Sequence of Method Calls Birthmark (SMC), Inheritance Structure Birthmark (IS) and Used Classes Birthmark (UC). All of the four types are vulnerable to obfuscation techniques mentioned in [8]. In addition, they need to access source code and only work for object-oriented programming language.

Static executable code based birthmark: Myles and Collberg [8] proposed a opcode-level k-gram based static birthmark. Opcode sequences of length k are extracted from a program and k-gram techniques which were used to detect similarity of documents are exploited to the opcode sequences. Although the k-gram static birthmark is more robust than Tamadas birthmark, it is still strongly vulnerable to some well-known obfuscations such as statement reordering, junk instruction insertion and other semantic-preserved transformation techniques such as compiler optimization.

Dynamic WPP based birthmark: Myles and Collberg [5] proposed a whole program path (WPP) based dynamic birthmark. WPP is originally used to represent the dynamic control flow of a program. WPP birthmark is robust to some control flow obfuscations such as opaque prediction, but is still vulnerable to many semantic-preserving transformation such as loop unwinding. Moreover, WPP birthmark may not work for large-scale programs due to overwhelming volume of WPP traces.

Dynamic API based birthmark: Tamada et al. [6], [18] also introduced two types of dynamic birthmark for Windows applications: Sequence of API Function Calls Birthmark (EXESEQ) and Frequency of API Function Calls Birthmark (EXEFREQ). In EXESEQ, the sequence of Windows API calls are recorded during the execution of a program. These sequences are directly compared to find similarity. In EXEFREQ, the frequency of each Windows API calls are recorded during the execution of a program. The frequency distribution is used as the birthmark. Schuler et al. [4] proposed a dynamic birthmark for Java. The call sequences to Java standard API are recorded and the short sequences at object level are used as a birthmark. Their experiments showed that API birthmarks are more robust to obfuscation than WPP birthmark in their evaluation. Unlike the Java or Windows API based birthmarks that are platform dependent, system call birthmarks can be used on any platform. In addition, system call birthmarks are more robust to counter-attacks than API-based ones. To evade API-based birthmarks, attackers may hide API calls by embedding their own implementation of some API routines. However, there are no easy ways to replace "system calls" without recompiling the kernel because because system call is the only way to gain privilege in modern operating systems. More importantly, existing API-based birthmarks have not been evaluated to protect core components theft.

Clone Detection: A close research field to software birthmark is clone detection. Clone detection is a technique to find the duplicate code ("clones") in a large-scale program. Existing techniques for clone detection can be classified into four categories: String-based [19], AST-based [20], [21], Token-based [22]–[24] and PDG-based [25], [26]. *String-based:* Each line of source code is considered as a string and the whole program is considered as a sequence of strings. A code fragment is labelled as clone if the corresponding sequence of strings is the same as another code fragment from original program. *AST-based:* The abstract syntax trees (AST) are extracted from programs by analyzing their syntax. Then the ASTs are directly compared. If there are common subtrees, clone may exist. *Token-based:* A program is first parsed to a sequence of tokens. The sequences of tokens are compared to find clone. *PDG-based:* A program

dependency graph is a graph which represents the control flow and data flow relations between the statements in a program procedure. To find clone, two PDGs are extracted from two programs (by some static analysis tools) and compared to find relaxed subgraph isomorphism.

Besides to be used to decrease code size and facilitate maintenance, clone detection can be also be used to detect software plagiarism. However, existing clone detection techniques are not robust to code obfuscation. String-based schemes are fragile even by simply renaming identifiers in programs. AST-based schemes are resilient to identifier renaming, but weak against statement reordering and control replacement. Token-based schemes are resilient to identifier renaming, but 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 existing schemes. However, PDG-based is still vulnerable to many semantics-preserving transformations such as inline and outline functions and opaque predicates. Moreover, all clone detection techniques need to access source code.

VII. Conclusion

In this paper, we propose two system call based software birthmarks: SCSSB and IDSCSB. We evaluate them using a set of real world programs. Our experiment results show that all the plagiarisms obfuscated by the *SandMark* tool are successfully discriminated. Unlike existing schemes that are evaluated with small or toy software, we evaluate our birthmarks (SCSSB and IDSCSB) with a set of large-scale software (web browsers). The results show that SCSSB and IDSCSB are effective and practical in detection of core component theft of large-scale programs.

Acknowledgment

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.

References

[1] C. Collberg and C. Thomborson, "Software watermarking: Models and dynamic embeddings," in *Principles of Programming Languages 1999*, Jan. 1999.

[2] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, and M. Stepp, "Dynamic path-based software watermarking," in *Proceedings of the Conference on Programming Language Design and Implementation*, 2004.

[3] H. Tamada, M. Nakamura, A. Monden, and K. ichi Matsumoto, "Design and evaluation of birthmarks for detecting theft of java programs," in *Proc. IASTED International Conference on Software Engineering*, 2004.

[4] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for java," in *ASE '07: Proc. of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007.

[5] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," in *ISC*, 2004, pp. 404–415.

[6] H. Tamada, K. Okamoto, M. Nakamura, and A. Monden, "Dynamic software birthmarks to detect the theft of windows applications," in *International Symposium on Future Software Technology 2004 (ISFST 2004)*, 2004.

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

[8] G. Myles and C. S. Collberg, "K-gram based software birthmarks," in *SAC*, 2005.

[9] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer, "Behavior-based spyware detection," in *Proceedings of the 15th conference on USENIX Security Symposium*, 2006.

[10] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proc. of ESEC/FSE*, 2008.

[11] "Gecko," http://en.wikipedia.org/wiki/Gecko_layout_engine.

[12] "KHTML," http://en.wikipedia.org/wiki/KHTML.

[13] 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.

[14] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, 1996.

[15] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*.

[16] "Elsa: An Elkhound-based C++ parser," http://scottmcpeak.com/elkhound/.

[17] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *Proc. of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, 2009.

[18] 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, Technical Report, 2007.

[19] B. S. Baker, "On finding duplication and near duplication in large software systems." in *Proc. of 2nd Working Conf. on Reverse Engineering*, 1995.

[20] 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.

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

[22] T. Kamiya, S. Kusumoto, and K. Inoue., "CCFinder: a multilinguistic token-based code clone detection system for large scale source code." *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, 2002.

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

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

[25] 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*, 2006.

[26] J. Krinke, "Identifying similar code with program dependence graphs." in *Proc. of 8th Working Conf. on Reverse Engineering*, 2001.