Behavior Based Software Theft Detection

[1]Xinran Wang [1]Yoon-Chan Jhi [1,2]Sencun Zhu [2]Peng Liu
[1]Department of Computer Science and Engineering
[2]College of Information Sciences and Technology
Pennsylvania State University, University Park, PA 16802
{xinrwang, szhu, jhi}@cse.psu.edu; 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 characteristics of a program, can be used for software theft detection. We propose a system call dependence graph based software birthmark called *SCDG birthmark*, and examine how well it reflects unique behavioral characteristics of a program. To our knowledge, our detection system based on SCDG birthmark is the first one that is capable of detecting software *component* theft where only partial code is stolen. We demonstrate the strength of our birthmark against various evasion techniques, including those based on different compilers and different compiler optimization levels as well as two state-of-the-art obfuscation tools. Unlike the existing work that were evaluated through small or toy software, we also evaluate our birthmark on a set of large software. Our results show that SCDG birthmark is very practical and effective in detecting software theft that even adopts advanced evasion techniques.

Categories and Subject Descriptors

K.4.1 [**COMPUTERS AND SOCIETY**]: Public Policy Issues—*Intellectual property rights*

General Terms

Security

Keywords

Software Birthmark, Software Plagiarism, Software Theft, Dynamic Analysis

1. 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 230,000 registered open source projects as of Feb.2009), software theft has become a very serious concern to honest software companies and open source communities. As one example, in 2005 it was determined in a federal court trial that IBM should pay an independent software vendor Compuware $140 million to license its software and $260 million to purchase its services [1] because it was discovered that certain IBM products contained code from Compuware.

To protect software from theft, Collberg and Thoborson [10] proposed software watermark techniques. Software watermark is a unique identifier embedded in the protected software, which is hard to remove but easy to verify. However, most of commercial and open source software do not have software watermarks embedded. On the other hand, "a sufficiently determined attackers will eventually be able to defeat any watermark." [9]. As such, a new kind of software protection techniques called software birthmark were recently proposed [21, 26–28]. A software birthmark is a unique characteristic that a program possesses and can be used to identify the program. Software birthmarks can be classified as static birthmarks and dynamic birthmarks.

Though some initial research has been done on software birthmarks, existing schemes are still limited to meet the following five highly desired requirements: (R1) Resiliency to semantics-preserving obfuscation techniques [11]; (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; (R5) Independence to platforms such as operating systems and program languages. To see the limitations of the existing detection schemes with respect to these five requirements, let us break them down into four classes: (C1) static source code based birthmark [27]; (C2) static executable code based birthmark [22]; (C3) dynamic WPP based birthmark [21]; (C4) dynamic API based birthmark [26,28]. We may briefly summarize their limitations as follows. First, Classes C1, C2 and C3 cannot satisfy the requirement R1 because they are vulnerable to simple semantics-preserving obfuscation techniques such as outlining and ordering transformation. Second, C2, C3 and C4 detect only whole program theft and thus cannot satisfy R2. Third, C1 cannot meet R4 because it has to access source code. Fourth, existing C3 and C4 schemes cannot satisfy R5 because they rely on specific features of Windows OS or Java platform. Finally, none of the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'09, November 9–13, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-352-5/09/11 ...$5.00.

four class schemes has evaluated their schemes on large-scale programs.

In this paper, we propose behavior based birthmarks to meet these five key requirements. Behavior characteristics have been widely used to identify and separate malware from benign programs [8,16]. While two independently developed software for the same purpose share many common behaviors, one usually contains unique behaviors compared to the other due to different features or different implementations. For example, HTML layout engine Gecko engine [3] supports MathML, while another engine KHTML [4] does not; Gecko engine implements RDF (resource description framework) to manage resources, while KHTML engine 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. Small programs or components, which may not contain unique behaviors, are out of the scope of this paper.

A system call dependence graph (SCDG), a graph representation of the behaviors of a program, is a good candidate for behavior based birthmarks. In a SCDG, system calls are represented by vertices, and data and control dependences between system calls by edges. A SCDG shows the *interaction* between a program and its operating system and the interaction is an essential behavior characteristic of the program [8, 16]. Although a code stealer may apply compiler optimization techniques or sophisticated semantic-preserving transformation on a program to disguise original code, these techniques usually do not change the SCDGs. It is also difficult to avoid system calls, because a system call is 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. Although an exceptionally sedulous and creative plagiarist may correctly overhaul the SCDGs, the cost is probably higher than rewriting his own code, which conflicts with the intention of software theft. After all, software theft aims at code reuse with disguises, which requires much less effort than writing one's own code.

We develop system call dependence graph (SCDG) birthmarks for meeting these five key requirements. To extract SCDG birthmarks, automated dynamic analysis is performed on both plaintiff and suspect programs to record system call traces and dependence relation between system calls. Since system calls are low level implementation of interactions between a program and an OS, it is possible that two different system call traces represent the same behavior. Thus, we filter out noises, which cause the difference, from system call traces in the second step. Then, SCDGs are constructed and both plaintiff and suspect SCDG birthmarks are extracted from the SCDGs.

We exploit subgraph isomorphism algorithm to compare plaintiff and suspect SCDG birthmarks. Although subgraph isomorphism is NP-complete in general, it is tractable in this application. First, the size of SCDGs is limited by a predefined parameter (100 or 400 in our experiment). Second, SCDGs are not general graphs. Their characteristics such as various types of nodes, makes backtrack-based isomorphism algorithm efficient. Finally, the first matching suffices for software theft detection, whereas the traditional isomorphism testing finds all isomorphism pairs. Hence, the isomorphism testing on SCDGs is tractable and efficient in practice.

This paper makes the following contributions:

- We proposed a new type of birthmarks, which exploit SCDGs to represent unique behaviors of a program. Without requiring any source code from the suspect, SCDG birthmark based detection is a practical solution for reducing plaintiff's risks of false accusation before filing a lawsuit related to intellectual property.
- As one of the most fundamental runtime indicators of program behaviors, the proposed system call birthmarks are resilient to various obfuscation techniques. Our experiment results indicate that they not only are resilient to evasion techniques based on different compilers or different compiler optimization levels, but also successfully discriminates code obfuscated by two state-of-the-art obfuscators.
- We design and implement Hawk, a dynamic analysis tool for generating system call traces and SCDGs. Hawk potentially has many other applications such as behavior based malware analysis. Detailed design and implementation of Hawk are present in this paper.
- To our knowledge, SCDG birthmarks are the first birthmarks which are proposed to detect software component theft. Moreover, unlike existing works that are evaluated with small or toy software, we evaluate our birthmark on a set of large software, for example web browsers. Our evaluation shows the SCDG birthmark is a practical and effective birthmark.

The rest of this paper is organized as follows. Section 2 introduces concepts and measurements about software birthmarks. In Section 3, we propose the design of SCDG birthmarks based software theft detection system. Section 4 presents evaluation results. We discuss limitation and future work in Section 5. Finally, we summarize related work in Section 6 and draw our conclusion in Section 7.

2. PROBLEM FORMALIZATION

This section first presents the definitions related to software birthmark and SCDG birthmark, and then introduces a metric to compare two SCDG birthmarks.

2.1 Software Birthmarks

A software birthmark is a unique characteristic that a program possesses and that can be used to identify the program. Before we formally define software birthmarks, we first define the meaning of *copy*. We say a program q is a copy of program p if q is exactly the same as p. Beyond that, q is still considered as a copy of p if it is the result of applying semantic preserving transformation (e.g., obfuscation techniques and compiler optimization) over p. The following definition of software birthmark and dynamic software birthmark is a restatement of the definition by Tamada et al. [27] and Myles et al. [21]:

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 p, only if both of the following conditions are satisfied:

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

Software birthmarks can be classified as 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 [21]. On the other hand, a dynamic birthmark relies on runtime behavior of a program, which is more difficult to be altered through code obfuscation techniques. The system call based birthmark we propose in this paper is a dynamic birthmark.

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 when executing p with input I. We say $f(p, I)$ is a dynamic birthmark of p, only if both of the following conditions are satisfied:

- $f(p, I)$ is obtained only from p itself when executing P with input I.
- program q is a copy of $q \Rightarrow f(p, I) = f(q, I)$

2.2 SCDG Birthmarks

Before we define SCDG birthmarks, we first define SCDG. A system call dependence graph (SCDG) is a graph representation of the behaviors of a program, where system calls are represented by vertices, and data and control dependences between system calls are represented by edges. To formally define SCDG, we first define dynamic dependence graph (DDG) [31] of a program run. Later, an SCDG can be constructed from a DDG.

Definition 3. (DDG: Dynamic Dependence Graph) The dynamic dependence graph of a program run is a 3-tuple graph $DDG = (N, E, \beta)$, where

- N is a set of nodes, and node $n \in N$ corresponds to an executed statement in the program run
- $E \subseteq N \times N$ is the set of dependence edges, and each edge $n_1 \rightarrow n_2 \in E$ corresponds to a dynamic data dependence or dynamic control dependence between statements n_1 and n_2. A data dependence exists between two executed statements if the set values used by one statement overlaps the set of value defined by the other. A control dependence is introduced if the execution of one statement depends on the values defined by the other statement, usually a predicate statement.
- $\beta : E \rightarrow T$ is a function assigning dynamic dependence types, either data or control, to edges.

Definition 4. (SCDG: System Call Dependence Graph) The system call dependence graph of a program run is a 4-tuple graph $SCDG = (N, E, \alpha, \beta)$, where

- N is a set of nodes, where node $n \in N$ corresponds to a system call execution in the program run
- $E \subseteq N \times N$ is the set of dependence edges, and each edge $n_1 \rightarrow n_2 \in E$ denotes that there exists a dependence path from n_1 to n_2 in the DDG of the program run.
- $\alpha : N \rightarrow S$ is a function assigning system call to nodes,
- $\beta : E \rightarrow T$ is a function assigning dependence types, either data or control, to edges.

Figure 1 presents an example to illustrate DDG and SCDG. Data and control dependences are plotted in solid and dashed lines, respectively. In Figure 2.2 (b), statement executions 2, 10 and 14 have data dependence on statement 1 because they use the file descriptor defined at 1. In Figure 2.2 (c), system call execution "7:lseek" has data dependence on "5:read" because of the dependence path $7 \rightarrow 6 \rightarrow 5$; system call execution "12:write" has data dependence on "10:read" due to the dependence path $12 \rightarrow 11 \rightarrow 10$; System call execution "4:open" has control dependence on "2:read" because statement 4 has control dependence on 3 and 3 has data dependence on 2 .

```
 1 :  fd1 = open(path1,"r",...);
 2 :  read(fd1, buf, ...);
 3 :  if (buf == "1") {
 4 :     fd2 = open(path2, "r", ...);
 5 :     n = read(fd2,buf,...);
 6 :     offset = n + 10;
 7 :    lseek(fd2,offset,...);
        ...
 8 :    close(fd2);
    }
 9 :  fd3 = open(path3,"w",...);
10:  read(fd1, buf, ...);
11:  strcpy(outbuf,buf);
12:  write(fd3,outbuf);
13:  close(fd3);
14:  close(fd1);
```

(a) statements

(b) DDG

(c) SCDG

Figure 1: An Example for DDG and SCDG

Next, we define subgraph isomorphism which will be used to compare similarity of SCDGs.

Definition 5. (Graph Isomorphism) A bijective function $f : N \rightarrow N'$ is a graph isomorphism from a graph $G = (N, E, \alpha, \beta)$ to a graph $G' = (N', E', \alpha', \beta')$ if

- $\forall n \in N, \alpha(n) = \alpha(f(n))$,
- $\forall e = (n_1, n_2) \in E, \exists e' = (f(n_1), f(n_2)) \in E'$ such that $\beta(e) = \beta(e')$,
- $\forall e' = (n'_1, n'_2) \in E', \exists e = (f^{-1}(n'_1), f^{-1}(n'_2)) \in E$ such that $\beta(e') = \beta(e)$

Definition 6. (Subgraph Isomorphism) A bijective function $f : N \rightarrow N'$ is a subgraph isomorphism from a graph $G = (N, E, \alpha, \beta)$ to a graph $G' = (N', E', \alpha', \beta')$ if there exists a subgraph $S \subset G'$ such that f is a graph isomorphism from G to S.

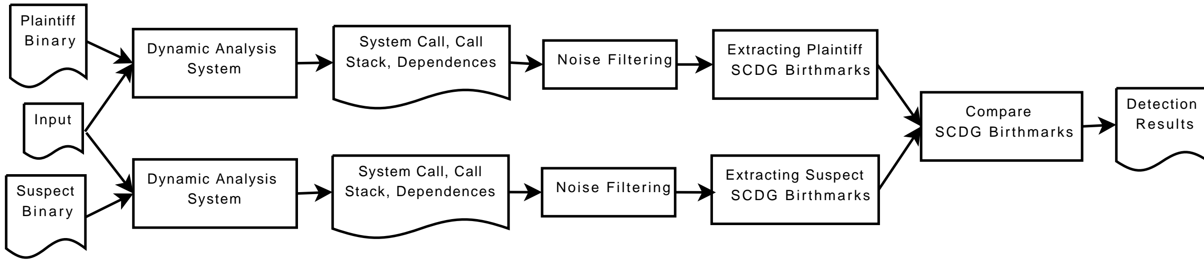

Figure 2: System Overview

Definition 7. (γ-Isomorphism) A graph G is γ-isomorphic to G' if there exists a subgraph $S \subseteq G$ such that S is subgraph isomorphic to G', and $|S| \geq \gamma|G|$, $\gamma \in (0, 1]$.

Definition 8. (SCDGB: System Call Dependence Graph Birthmark) Let p be a program or program component. Let I be an input to p, and $SCDG_p$ be system call dependence graph of the program run with input I. SCDG birthmark $SCDGB_p$ is the subgraph of the graph $SCDG_p$ that satisfies the following conditions:

- program or component q is a copy of p and $SCDG_q$ be system call dependence graph of the program run of q with input $I \Rightarrow SCDGB_p$ is subgraph isomorphic to $SCDG_q$.
- program or component q is different from p and $SCDG_q$ be system call dependence graph of the program run of q with input $I \Rightarrow SCDGB_p$ is not subgraph isomorphic to $SCDG_q$.

According to our definition of SCDG birthmark, program q is regarded as plagiarized from program p if the SCDG birthmark of p is subgraph isomorphic to SCDG of q. Although as shown in our experiment SCDG birthmark is robust to state-of-the-art obfuscation techniques, for robustness to unobserved and unexpected attacks, we relax subgraph isomorphism to γ-isomorphism in our detection. q is regarded as plagiarized from that of p, if the SCDG birthmark of p is γ-isomorphic to SCDG of q. We set $\gamma = 0.9$ in experiments because we believe that overhauling 10% of a SCDG birthmark is almost equivalent to changing the behavior of a program.

3. SYSTEM DESIGN

3.1 System Overview

Figure 2 shows the overview of our system. It consists of four stages: dynamic analysis, noise filtering, SCDG birthmark extraction, and birthmark comparison. Let us summarize each of the steps before dealing with details in later subsections.

Dynamic Analysis. In the first step, automated dynamic analysis is performed on both plaintiff and suspect programs to record their system call traces. For both programs, we feed in the same input. Besides system calls, the call stack for each system call and the dependence relation among system calls are calculated and recorded.

Noise Filtering. System calls are low level implementation of interactions between a program and an OS. It is possible that two different system call traces represent the same behavior, e.g., because of the existence of many system calls that are dependent on runtime environment. Therefore, we filter out noises from system call traces before extracting birthmarks.

SCDG Birthmarks Extraction. We aim to detect component theft. Therefore, in this step, we first identify the system calls invoked by the component of interest in a plaintiff program and then extract SCDGs for the component. Then, we divide SCDGs of the component into subgraphs, and refine the subgraphs by removing common nodes that are also found in SCDGs of several unrelated programs. Finally, the remaining subgraphs are considered as birthmarks of the plaintiff component.

Although it is possible to choose the SCDG of the whole suspect program for comparison, the graph's size would be too large to efficiently test subgraph isomorphism. As such, we also divide the SCDG of a suspect program into subgraphs.

Birthmark Comparison. Once both plaintiff and suspect birthmarks are extracted, we examine the birthmarks for a r-isomorphism using relaxed VF subgraph isomorphism algorithm [14]. To increase the efficiency, three forms of pruning are performed to reduce the search space.

3.2 Dynamic analysis

In this subsection, we first briefly introduce our dynamic analysis system. Then, we describe the design details of the dynamic instrumentation. Finally, deferred reference counting is introduced and discussed to improve performance.

Our dynamic analysis system consists of Valgrind [23] and Hawk, as shown in Figure 3. Valgrind is a generic framework to instrument machine code at runtime, and Hawk is a plugin tool we designed and developed for Valgrind. Valgrind and Hawk work together to generate system call traces and their dependences. Specifically, Valgrind takes a binary client program, which is a plaintiff or a suspect program in our case, and an input to the client program for dynamic analysis. Then, it decompiles the client's machine code, one small code block at a time, in a just-in-time, execution-driven fashion. It disassembles the code block into an architecture-neutral intermediate representation (IR) block. In Hawk, every memory byte and register of the client program is *shadowed* by a dependence set, which is a set of system calls it depended on. Hawk instruments the IR block given by Valgrind with analysis code. The analysis code is used to update the shadow values of the client program's memory locations and registers. Then, the instrumented IR block is converted back into machine code by Valgrind and executed. The resulting translation is stored in a code cache and thus it can be reused without calling the instrumenter again. Valgrind also provides system call hooks for Hawk to instrument system calls of client programs. When a system call of a client program is invoked, Hawk create a new node for the system call as well as the dependence edges between the new node and the other ones.

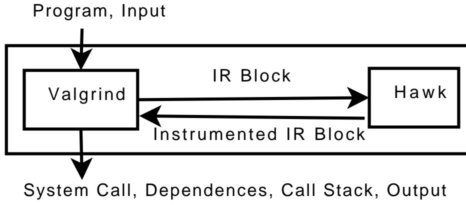

Figure 3: Dynamic Analysis System

Next, we presents the details of IR instrumentation. The Valgrind IR majorly consists of five types: load memory IR, store memory IR, get register IR, put register IR, and expression IR. The first four types of IR are used to read or write values from memory and registers to the temporary variables of an IR block. In Hawk, the instrumentation to the first four types of IR is just transferring of shadow values between a temporary variable and a register/memory location. All actual operations are performed by expression IR. An expression IR is abstracted as $t_d = op(t_1, t_2, ..., t_n)$ (n<=4 in practice), where $t_1, t_2, ..., t_n$ denote the set of temporary variables used by the IR and t_d denotes the temporary variable defined. The instrumentation of the expression IR is defined by $sh(t_d) = \cup(sh(t_1), sh(t_2), ..., sh(t_n)$. Figure 4 shows an example of an IR block and its instrumentation.

Besides IR instrumentation, Hawk also instruments the system calls of a client program to create new system call nodes and dependence edges. When a system call occurs, a handler function of Hawk is called. The system call information (number, index, parameter and result) and its call stack is recorded within the function. In addition, a new system call node is created, and dependence edges between the new system call node and previous nodes are established by the shadow values of the system call's input parameters. For example, the *read* system call in X86 Linux uses register *ebx* as the input parameter, which stores a file descriptor id. Dependence edges are created between this *read* system call node and the nodes in the shadow values of ebx. Finally, the new system call node is assigned to the shadow variables of the system call's output parameters. For example, the *eax* register is the return variable of the *open* system call, storing the descriptor id of an opened file . The newly created *open* system call node is assigned to the shadow value of the *eax* register.

For large programs, Hawk may generate a great number of intermediate dependence sets. Thus, a garbage collector for dependence sets is needed. There are several ways to implement a garbage collector, such as reference counting, mark-sweep and copy collection. Here we use reference counting instead of mark-sweep or copy collection because the number of dependence sets during execution are huge and tracing would be prohibitively slow. Also, there are no cycles to cause problems. However, a disadvantage of reference counting is that frequent update of reference count may hurt performance. This is a severe problem in our case, because every instrumentation of an IR may need to update reference count. To solve this problem, we exploit deferred reference counting [6]. Deferred reference counting was originally used to reduce the cost of maintaining reference counts by avoiding adjustments when the reference is stored in the stack. In our case, we avoid updating references on temporal variables due to the short lifetime of the temporal

```
0x4000B02: addl %edx,4(%eax)
—— IMark(0x4000B02, 3) ——
 *1:   t9 = GET:I32(0)                  # get %eax
  2:   sh(t9) = sh(%eax)
 *3:   t8 = Add32(t9,0x4:I32)           # add address
  4:   sh(t8) = sh(t9)
 *5:   t2 = LDle:I32(t8)                #load
  6:   sh(t2) = sh(memory(t8))
 *7:   t1 = GET:I32(8)                  # get %edx
  8:   sh(t1) = sh(%edx)
 *9:   t0 = Add32(t2,t1)                # addl
 10:   sh(t0) = sh(t2) ∪ sh(t1)
*11:   STle(t8) = t0                    # store
 12:   sh(memory(t8)) = sh(t0)

0x4000B05: movl 0x2E0(%ebx),%eax
—— IMark(0x4000B05, 6) ——
*13:   PUT(60) = 0x4000B05:I32          # put %eip
*14:   t11 = GET:I32(12)                # get %ebx
 15:   sh(t11) = sh(%ebx)
*16:   t10 = Add32(t11,0x2E0:I32)       # add
 17:   sh(t10) = sh(t11)
*18:   t12 = LDle:I32(t10)              # load
 19:   sh(t12) = sh(memory(t10))
*20:   PUT(0) = t12                     # put %eax
 21:   sh(%eax) = sh(t12)
```

Figure 4: IR instrumentation Example. Statements with mark * are original IRs. Instrumentation IRs are pseudo code for brevity.

variables. During the execution, dependence sets cannot be reclaimed as soon as their reference counts become zero. Because there might still be references to them from temporal variables, such sets are added to a zero count table (ZCT) instead. The dependence sets in the ZCT are scanned at the end of code blocks, and any sets with zero reference count are reclaimed.

Currently, Hawk does not trace control dependence for efficiency concerns. Our experiments in section 4 show that data dependence alone is powerful enough for software theft detection.

3.3 Noise Filtering

As a low level abstraction of the interaction between a program and the OS, system call sequences contain noise. Due to the noise system calls, the same behavior could be represented by two different system call sequences. We filter out the noises from system call traces in the following ways. First, some types of system calls are ignored because they apparently do not represent the behavior characteristic of a program. For example, the system call *gettimeofday* returns the elapsed time since Epoch in seconds and microseconds. Many programs periodically call *gettimeofday* with no significant impact on their behaviors; therefore, we remove *gettimeofday* if no other system calls depend on them. Another example is related to memory management system calls. 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, while the *brk* system call changes the size of the heap to be larger or smaller as needed. Normally, C function *malloc* first grabs a large chunk of memory and then splits it as needed to get smaller chunks. As such, not every malloc in C involves a system call and two identical programs may have very different memory management system call sequences. Therefore, we ignore all memory management system calls. Second, some types of system calls are considered as the same in system call birthmarks, because some system calls provide multiple versions that take slightly different parameters for convenience.

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*. By considering such system calls identical, we can not only reduce the sophistication of dealing with many different system calls, but also address 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, when a program tries to open a file, it may fail at the first time but succeed at the second time. Although system call *open* is called twice, here the first failed call should be removed.

3.4 Extraction of SCDG Birthmarks

⊙ *Extraction of Plaintiff Birthmarks.* There are four steps to extract SCDG birthmarks for a plaintiff component. First, by analyzing the system call trace whose noise has been removed, we determine whether a system call is called by a plaintiff 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 implementation options. One method is to use a special list, L, containing information on the functions belonging to the plaintiff component. List L can be automatically generated by processing the source code of the plaintiff component with a tool such as Elsa [2]. Then, whenever a system call is called, Hawk can notice whether the system call is called by the plaintiff component by searching in the call stack (containing callers of the system call) for any occurrence of the functions listed in L. Note that we can preserve the symbol table of the plaintiff component because we have control over the compilation of plaintiff source codes. Alternatively, a simpler method which does not need to maintain list L is available. If we can compile the plaintiff component into a dynamic linked library (DLL), Hawk can simply use a utility function provided in Valgrind to retrieve the DLL component that contains any of the callers of the invoked system call.

Second, an SCDG and a dynamic call tree are built from the system call traces corresponding to the the plaintiff component. Building an SCDG is trivial given nodes (system calls) and edges (dependences). Besides SCDG, we also build a dynamic call tree, which will be used to partition an SCDG in later steps. A dynamic call tree here is a tree with subroutine calls as internal nodes and system calls as leaf nodes. A node's parent is its caller and its children are its callees. The path from a leaf system call node to the root node is the call stack of the system call. The process of generating a dynamic call tree is also trivial: we just need to merge all the call stacks.

Third, we divide an SCDG into subgraphs. Because an extracted SCDG may be too large to efficiently compute subgraph isomorphism and/or too specific for the plaintiff program, they are not directly used as birthmarks. As a component theft is mostly likely to happen over a subroutine, we partition the graph based on dynamic call tree. That is, we extract a subtree from the dynamic call tree, and the leaf nodes in the subtree and their dependence relation consist of an SCDG birthmark candidate. The partition process is as the following. We first set a parameter m, which is used to guarantee that the subgraph is not too large or too specific for the partition. Then, the dynamic call tree of the selected component is traversed by a depth first search algorithm. When a tree node is visited, the number of leaf nodes in this subtree is calculated. If the number is less than m, the subtree is selected and search within the subtree is skipped. The process is finished when all noes in the dynamic call tree is traversed.

Finally, we remove the SCDG subgraphs which represent common behaviors. This step is necessary because we need to find the unique behavior of plaintiff components as birthmarks. For this purpose, a set of training programs are used. The set of programs should include programs which have a similar component with the plaintiff program but are indeed completely unrelated programs. The SCDG subgraphs are compared with the SCDGs of the training set. All SCDG subgraphs which are subgraph isomorphism to the SCDGs of the training set are removed, and finally, the remaining subgraphs become birthmarks.

⊙ *Extraction of Suspect Birthmarks.* As mentioned earlier, we assume that there are no source code and symbolic debugging information of a suspect program. Hence, it is difficult to identify the suspicious components in a suspect program, not to mention extracting SCDGs from them. Thus, we have to extract SCDG birthmarks based on the SCDG of the whole suspect program. Specifically, we partition the whole SCDG according to dynamic call tree, as in the case for plaintiff birthmark extraction, except that we choose m to be several times larger.

3.5 Birthmark Comparison

Once both the plaintiff and suspect SCDG birthmarks are extracted, they are compared using (relaxed) VF subgraph isomorphism algorithm [14]. $n * m$ pairs subgraph isomorphism testing are needed in principle, where n and m are the numbers of plaintiff and suspect birthmarks, respectively. Fortunately, most pairs can be pruned through three forms of loseless pruning.

Pruning Search Space First, SCDG birthmarks smaller than an interesting size K or the types of system calls smaller than T are excluded from both plaintiff and suspect birthmarks. For software theft detection, we only need to locate birthmark pairs of non-trivial ones, which, if found, can provide strong evidence for proving theft. Second, based on the definition of $\gamma - isomorphism$, a SCDG birthmark pair (g, g') can be excluded if $|g'| < \gamma|g|$, where g and g' are SCDG birthmarks of plaintiff and suspect programs, respectively. Finally, a SCDG birthmark pair can be pruned based on the characteristics of SCDGs. In this paper, we use vertex histogram as the characteristics of SCDGs and the similarity between two vertex histograms can be used for pruning. Specifically, a plaintiff SCDG birthmark g is represented by vertex histogram $h(g) = (n_1, n_2, ..., n_k)$, where n_i is the frequency of the ith kind of vertices, and a suspect SCDG birthmark g' is represented by $h(g') = (m_1, m_2, ..., m_k)$. We define the difference between $h(g)$ and $h(g')$ as $d(h(g), h(g')) = \sum_{i=0}^{k} e_i$, where $e_i = n_i - m_i$ if $n_i > m_i$ and $e_i = 0$ if $n_i \leq m_i$. Based on the definition of $\gamma - isomorphism$, the pair (g, g') can be excluded if $d < (1 - \gamma)|g|$.

Computational Feasibility. Because our SCDG birthmark involves subgraph isomorphism testing, we discuss the computation feasibility of the testing. As mentioned in [19], although subgraph isomorphism is NP-complete in general, research in the past three decades has shown that some algorithms are reasonably fast on average and become com-

putationally intractable in a few cases [12,13]. For instance, algorithms based on backtracking and look-ahead, e.g., Ullmann's algorithm [30] and VF [14], are suitable with graphs of thousands of nodes.

In addition to the general tractability issue, the characteristics of graphs and the needs for a specific application also reduce the computational cost [19]. In our application, the computational cost are reduced for the following reasons. First, the size of SCDGs is limited by a predefined parameter (100 or 400 in our experiment). Second, SCDGs are not ordinary graphs. Their characteristics such as various types of nodes, makes backtrack-based isomorphism algorithm efficient. Last, the first matching suffices for software theft detection, whereas the traditional testing finds all isomorphism pairs. Hence, the isomorphism testing on SCDGs is tractable and efficient in practice.

Finally, our search space pruning phase can effectively identify and discard the spurious SCDG pairs which are obviously not isomorphic to each other, avoiding detailed isomorphism testing. As a result, only a small portion of SCDG pairs are really tested in the case of a real software theft. Thus, our detection is computationally efficient in our specific problem settings.

4. EVALUATION

In Section 1 we mentioned five key requirements on software theft detection. It is easy to see R4 and R5 are already met by our design. In this section, we evaluate the performance of SCDG birthmarks with respect to three primary criteria: (M1) capability to detect component theft for large-scale programs, (M2) credibility to independently developed program, and (M3) resiliency to obfuscation. These three criteria contain more than R1, R2 and R3 because of M2.

In the following, we first discuss the implementation of our system and environmental setup. Then, we evaluate criteria M1 and M2 for SCDG birthmarks with over 30 real-world large application programs. Finally, we evaluate criteria M3 against evasion techniques that apply different compilers, different compiler optimization levels, or obfuscation techniques.

4.1 Implementation and Environmental Setup

We implemented a prototype of SCDG birthmark based software theft detection system. The entire system consists of about 5,000 lines of C/C++ code and 1000 lines of Tcl script code. Our implementation of the γ-isomorphism testing algorithm was adopted from VFlib[1]. We used tree.hh[2], an STL-like C++ tree class, for dynamic call tree representation and operation. The version of Valgrind we used is 3.3.1. The entire detection system runs under Ubuntu 8.04. For detection purpose, we fed the same input and perform the same operation (spell checking) if applicable; otherwise, an appropriate input and a simple operation was provided. In our experiment, we set $\gamma = 0.9$, minimal size of SCDG birthmarks $K = 15$, and maximal size of SCDG birthmarks m 100 for plaintiff programs and 400 for training programs and testing sets (i.e., suspect programs), respectively.

4.2 Effectiveness of SCDG

We chose two subject program components for experiments: Gecko layout engine for web browser and GNU Aspell spell checker. Before we give details on the subject components' SCDG birthmarks and show the effectiveness, we first introduce the training program set and the testing program set.

Training Program Set. The following programs were part of the training program collection: Dillo, Yudit, Meld, Gimp, Totem, Pdfedit and Dia. Dillo was chosen for its similar web content rendering behavior with Gecko engine. Yudit was chosen for its similar spell checking behavior with Aspell. Others were chosen for general common behaviors. Each training program was executed under our dynamic analysis system and performed a simple operation and then quit. We fed one of our authors' home page url as input to Dillo and quit it after the home page was displayed. The home page html was also fed to Yudit and performed spell checking and quit. For other programs, appropriate input and a simple operation were provided (e.g. giving Gimp a gif file and then adjust color balance) and then quit. Table 1 shows the statistics for the SCDGs of the training program set. Note that for the training program set, we have already known that none of them contains our subject components.

Table 1: Training set statistics

Program	Version	Type	SCDG Node #	SCDG Edge #
Dillo	0.8.6	Web Browser	2612	1510
Yudit	2.4.1	Text Editor	4576	1023
Meld	1.1.5.1	Diff Viewer	12314	7084
Gimp	2.4.5	Graph Editor	59372	5972
Totem	2.22.1	Media Player	21865	6762
Pdfedit	0.3.2	PDF Editor	8937	4867
Dia	0.96.1	Diagram Drawing	27145	29185

Table 2: Testing set statistics

Program	Version	Type	SCDG Node #	SCDG Edge #
Flock	2.0.3	Web Browser	21337	9343
Epiphany	2.22.2	Web Browser	16864	9011
Konqueror	3.5.10	Web Browser	11850	5589
Amaya	10	Web Browser	42701	23958
Opera	9.52	Web Browser	58485	21361
Songbird	1.1.2	Web Browser	37103	25547
Galeon	2.0.7	Web Browser	19825	7450
AbiWord	2.4.6	Word Processor	12975	5642
KWord	1.6.3	Word Processor	15408	6630
LyX	1.5.3	Latex Editor	21977	18656
Texmaker	1.6	Latex Editor	6897	3223
Kile	2.0.0	Latex Editor	50937	24615
Gedit	2.22.3	Text Editor	25113	5867
Bluefish	1.0.7	Text Editor	10952	3502
GNU Emacs	22.2.1	Text Editor	14807	4734
Vim	7.1.138	Text Editor	2582	1979
Pidgin	2.5.2	Messenger	10816	8014
Kopete	0.12.7	Messenger	16319	7144
Kmess	1.5	Messenger	10830	6247
GnoCHM	0.9.9	CHM Viewer	21191	8354
Evince	2.22.2	Doc. Viewer	16179	7095
GV	3.6.3	Doc. Viewer	6508	3267
Quod Libet	1.0	Media Player	15839	10725
Evolution	2.22.3	Email Client	13798	6787

Testing Program Set. We evaluated Gecko SCDG birthmarks and Aspell SCDG birthmarks against 24 large application programs shown in Table 2. Each test program was executed under our dynamic analysis system and perform a simple operation and then quit. Again, we fed the home page url as input to all browsers, and performed spell checking if applicable, and then quit after the home page was displayed.

[1]http://amalfi.dis.unima.it/graph
[2]http://www.aei.mpg.de/∼peekas/tree

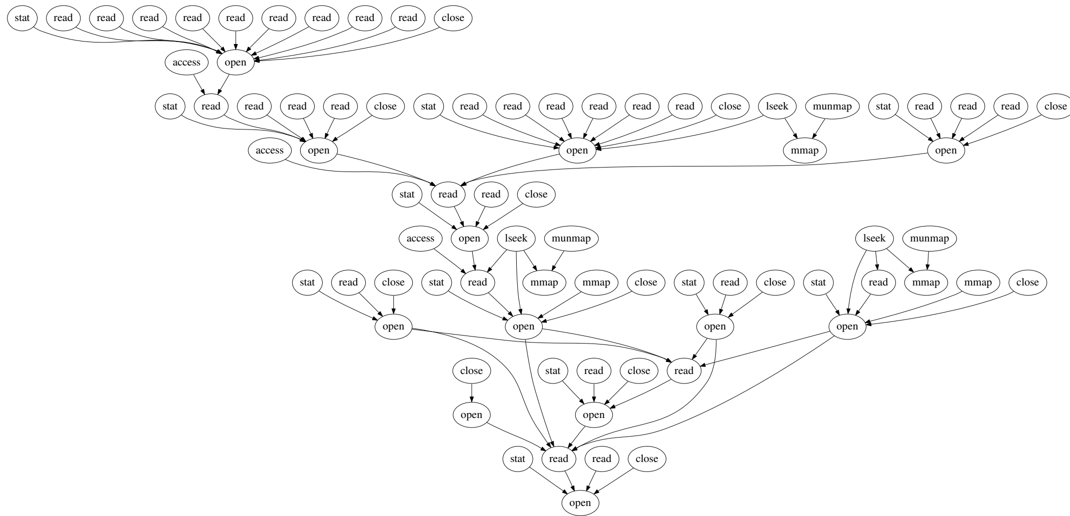

Figure 5: An Example Birthmark extracted From Aspell

We fed the home page html to all word processors, text editors, instant messengers and email clients and performed spell checking if applicable and quit. For other programs, appropriate input and a simple operation were provided and then quit. Table 2 shows statistics for the SCDGs of the test program set. Note that for most of programs in the testing sets, we do not have the preknowledge whether they contain Gecko and/or Aspell or not; that is, our test is a blind test.

Experiment of GNU Aspell. In this experiment, we test whether a program in the testing set contained a small software component – GNU Aspell spell checker. GNU Aspell is the standard spell checker for the GNU software system. It can either be used as a component (library) or as an independent spell checker. As a software component, it has been widely used in word processors, document editors, text editors and instant messengers.

We extracted birthmarks of Aspell from a standalone program Aspell 0.60.5. The extracted SCDG graph contains 481 nodes and 659 edges. One SCDG birthmark, shown in Figure 5, was generated after compared with SCDGs of the training programs set (i.e., after removing the common SCDGs). The Aspell SCDG birthmark was compared with SCDGs of the programs in the testing set. The result is that five programs, including Opera, Kword, Lyx, Bluefish, Pidgin, contain the Aspell Birthmark. It shows that each of the five programs contain Aspell component, while others not. This result was confirmed by manually checking the programs in the testing set.

Experiment of Gecko Engine. In this experiment, we study SCDG birthmarks using web browsers and their layout engines. 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 other applications that require the rendering (and editing) of web contents. Gecko [3], which is the second most popular layout engine on the Web, is an layout engine used in most Mozilla software and its derivatives.

We extracted Gecko SCDG birthmarks from Firefox 3.0.4. The extracted SCDG graph contains 726 nodes and 1048 edges. Two SCDG birthmarks were extracted after comparing with the training program set. Figure 6 shows an example SCDG birthmark of Gecko. The two Gekco SCDG birthmarks were compared with SCDGs of testing programs set. The result is that four programs, including Flock, Epiphany, SongBird and Geleon, contain one of the two Gecko Birthmarks. It shows that each of the four programs contain Gecko components, while others not. This result was confirmed by manually checking the programs of the testing set.

4.3 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 (the second-most popular lossless compression tool for Linux), gzip (a lossless compression tool) and oggenc (a command-line encoding tool for Ogg Vorbis, a non-proprietary lossy audio compression scheme). To avoid incompatible compiler features, single compilation-unit source code (bzip2.c, gzip.c and oggenc.c) were used [3]. 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, a system call trace was recorded, and finally SCDGs were generated for each executable. We compared the system call sequences and found that applying optimization options does not change the system call traces and SCDGs of bzip2 and gzip, while the system call traces for oggenc with optimization options (-O3 and -Os) contain only one "write" system call less than that with optimization options (-O0, -O1 and -O2). This result shows that system call based SCDG birthmarks are robust to compiler optimization.

4.4 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

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

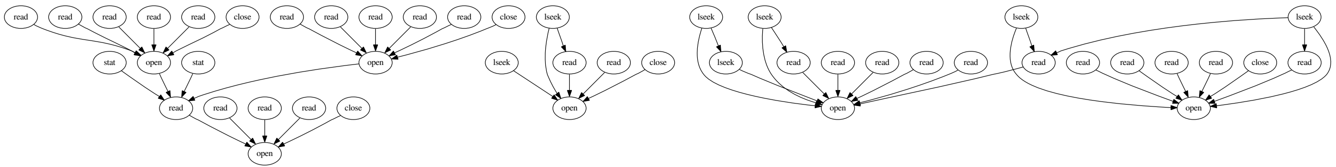

Figure 6: An Example Birthmark Extracted from Gecko

three 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 trace and SCDG is recorded for the each executable. We used GCC result to compare with TCC and Watcom results. The results show that the system call traces and SCDGs are exactly the same between TCC and GCC (both with default optimization levels). The system call traces between 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, not because of the compilers themselves. Both GCC and TCC use *glibc*, while Watcom uses its own implementation. Three types of differences are found. First, different but equivalent system calls are used between the two libc implementation such as *stat* and *stat64*. Second, failed system calls appear many times in one result, but not in the other. Last, some differences caused by memory management system calls. Fortunately, as mentioned in Section 3.3, such differences can be removed by our *noise filtering* step. As such, our proposed birthmarks can survive under the three different compilers in this experiment.

4.5 Impact of Obfuscation Techniques

Obfuscation is another type of semantic preserving transformation techniques. There are two types of obfuscation tools: source code based and binary based. A source code obfuscator accepts a program source file and generates another functionally equivalent source file which is much harder to understand or to reverse-engineer. A binary obfuscator exploits binary rewriting technique for obfuscation. We evaluated the impacts of obfuscation techniques over two state-of-the-art obfuscation tools. For source code obfuscation, we used the commercial product *Semantic Designs Inc's* C obfuscator that implements abstract syntax tree (AST) based code transformation. Its features include (but not limited to) identifier scrambling, format scrambling, loop rewriting, and if-then-else rewriting. For binary code obfucation, we used control flow flattening implemented in Loco based on Diablo link-time optimizer [20]. 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;}}'. We used the two obfuscators to obfuscate and then compile each of the three programs we used before. The generated executables were executed with the same input and a system call trace and SCDG were recorded for the each executable. The system call traces and SCDGs between the original programs and obfuscated programs are exactly the same. This results show that SCDG birthmark is robust to these two state-of-the-art obfuscation tools.

5. DISCUSSION

5.1 Robustness

Besides those tested obfuscations in Section 4.5, SCDG birthmark is robust to most types of obfuscation techniques presented in [11] including split or merge variables, promote scalars to objects, convert static data to procedure, change encoding, change variable life, split or merge arrays, reorder arrays, unroll loop, reorder expression, extend loop condition, reorder statements and so on. Although they can significantly alter the source code and binary code appearance, they do not change system calls and dependence relations between system calls. As a consequence, SCDG birthmark is robust to the all these obfuscations.

5.2 Counterattacks

One of the possible counterattacks to SCDG birthmark is the *system call injection attack*. An attacker may insert a great number of system calls in the plagiarism program without compromising its original behaviors. This counterattack may avoid our detection when a small m is used for generating suspect birthmarks. However, this counterattack may not work in practice because (1) these injected system calls will most likely be filtered out in the noise filtering phase, and (2) system calls are very costly due to context switching between the kernel mode and the user mode. A large number of injected system calls will result in great slowdown of the plagiarism program, which thwart a plagiarist from using this counterattack. Moreover, a bigger m can be always used to defeat such counter attacks. It is a tradeoff between performance and robustness.

Another possible counterattack is the system call reorder attack. Although this attack changes the order of system calls in traces, it leaves SCDGs unaltered. Two or more system calls can be reordered only when they are not bounded by dependences. Otherwise, reordering could break dependences, and thus cause behavior change or program errors. As such, this counterattack can not evade our detection.

There may exist other counterattacks which overhaul SCDG birthmarks. For these unknown and unexpected attacks, SCDG birthmark is still robust to some extent, because it is tolerant to a certain percentage overhauling by using γ-isomorphism.

5.3 Limitations and Future Work

SCDG birthmark bears the following fundamental limitation. First, it will 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, it is 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 parameters (m and γ) a user defines. To have higher confidence, one should use large parameters, thus it is likely to increase false negative. In contrast, reducing these parameters may 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 SCDG 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.

We will study the impact of input on SCDG birthmarks as our future work. As a dynamic birthmark, SCDG birthmark requires the original program and the suspicious program to be fed with the same input. This requirement sometimes is difficult to meet. For example, a software plagiarist may illegally use a real time computer vision library as a part of their motion understanding software, whereas the original program uses the library for different purposes, say face recognition.

We will also examine how the birthmark of a component changes over software versions. A component of software might be upgraded frequently; some updates may change the implementation of the component significantly, which may include totally redesigned programming interfaces. These changes can invalidate the birthmarks computed before the updates. Will these changes help a plagiarist evade the detection by stealing an old version of the component? To answer this question, we will investigate the similarity of the birthmarks of different versions of the Gecko layout engine in the future.

6. 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 [27], static executable code based birthmark [22], dynamic WPP based birthmark [21], and dynamic API based birthmark [26, 28].

Static source code based birthmark: Tamada [27] *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 [22]. In addition, they need to access source code and only work for object-oriented programming language.

Static executable code based birthmark: Myles and Collberg [22] 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 [21] 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. [28, 29] 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. [26] 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 counterattacks 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 as four categories: String-based [5], AST-based [7, 17], Token-based [15, 24, 25] and PDG-based [18, 19].

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 control flow flattening and opaque predicates. Moreover, all clone detection techniques need to access source code.

7. CONCLUSION

In this paper, we proposed the SCDG software birthmark. We evaluated it with a set of real world programs. Our experiment results showed that all the plagiarisms obfuscated by the two state-of-the-art tools were successfully discriminated. Unlike existing schemes that are evaluated with small or toy software, we evaluate our birthmarks with a set of

large-scale software. The results showed that SCDG Birthmark is effective and practical in detection of components theft of large-scale programs.

8. 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. 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.

9. REFERENCES

[1] [online]Available from World Wide Web: `http://news.zdnet.com/2100-3513_22-5629876.html`.

[2] Elsa: An elkhound-based c++ parser, `http://www.cs.berkeley.edu/~smcpeak/elkhound`.

[3] The gecko engine, `http://en.wikipedia.org/wiki/Gecko_layout_engine`.

[4] Khtml engine, http://en.wikipedia.org/wiki/KHTML.

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

[6] H. G. Baker. Minimizing reference count updating with deferred and anchored pointers for functional data structures. *ACM SIGPLAN Notices*, 29:29–9, 1994.

[7] 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.

[8] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of ESEC/FSE*, 2008.

[9] 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.

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

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

[12] C. Hoffman. *Group-theoretic Algorithms and Graph Isomorphism.* Springer Verlag, 1982.

[13] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *ACM Symp. on Theory of Computing*, 1974.

[14] J. E. Hopcroft and J. K. Wong. Performance evaluation of the vf graph matching algorithm. In *Processing of 10th Int. Conf. on Image Analysis and Processing*, 1999.

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

[16] 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.

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

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

[19] 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.

[20] M. Madou, L. Van Put, and K. De Bosschere. Loco: An interactive code (de)obfuscation tool. In *Proceedings of ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM '06)*, pages 140–144, 2006.

[21] G. Myles and C. Collberg. Detecting software theft via whole program path birthmarks. In *ISC*, pages 404–415, 2004.

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

[23] 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*.

[24] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *Universal Computer Science*, 2000. Available from World Wide Web: `citeseer.ist.psu.edu/article/prechelt01finding.html`.

[25] 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.

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

[27] 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.

[28] 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.

[29] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K. ichi Matsumoto. Design and evaluation of dynamic software birthmarks based on api calls. Technical report, Nara Institute of Science and Technology, 2007.

[30] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[31] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *Processing of 14th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2006.