Exploiting Thread-Related System Calls for Plagiarism Detection of Multithreaded Programs

Zhenzhou Tian[a], Ting Liu[a,*], Qinghua Zheng[a], Ming Fan[a], Eryue Zhuang[a], Zijiang Yang[b,a]

[a]*MOEKLINNS, Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China*
[b]*Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008, USA*

Abstract

Dynamic birthmarking used to be an effective approach to detecting software plagiarism. Yet the new trend towards multithreaded programming renders existing algorithms almost useless, due to the fact that thread scheduling nondeterminism severely perturbs birthmark generation and comparison. In this paper, we redesign birthmark based software plagiarism detection algorithms to make such approach effective for multithreaded programs. Our birthmarks are abstractions of program behavioral characteristics based on thread-related system calls. Such birthmarks are less susceptible to thread scheduling as the system calls are the sources that impose thread scheduling rather than being affected. We have conducted an empirical study on a benchmark that consists of 234 versions of 35 different multithreaded programs. Our experiments show that the new birthmarks are superior to existing birthmarks and are resilient against most state-of-the-art obfuscation techniques.

Keywords: software plagiarism detection, software birthmark, multithreaded program, thread-aware birthmark

1. Introduction

Software plagiarism, ranging from open source code reusing to smartphone app repacking, severely affect both open source communities and software companies. It is widespread because software plagiarism is easy to implement but hard to detect. For example, a study in 2012 (Zhou et al., 2012) shows that about 5% to 13% of apps in the third-party app markets are copied and redistributed from the official Android market. The unavailability of source code and the existence of powerful automated semantics-preserving code obfuscation techniques and tools (Collberg et al., 2003; Wu et al., 2010; Jiang et al., 2007; Madou et al., 2006; Linn and Debray, 2003) are a few reasons that make software plagiarism detection a daunting task. Nevertheless, significant progress has been made to address this challenge. One of the most effective approaches is software birthmarking, where a set of characteristics, called birthmarks, are extracted from a program to uniquely identify the program. As illustrated in previous works (Myles and Collberg, 2004; Tian et al., 2013; Wang et al., 2009a; Zhang et al., 2014b; Luo et al., 2014; Ming et al., 2016), with proper algorithms birthmarks can identify software theft even after complex code obfuscations.

Despite the tremendous progress in software birthmarking, the trend towards multithreaded programming greatly threatens its effectiveness, as the existing approaches remain optimized for sequential programs. For example, birthmarks extracted from multiple runs of the same multithreaded programs can be very different due to the inherent non-determinism of thread scheduling. In this case software birthmarking fails to declare plagiarism even for simply duplicated multithreaded programs. In this paper, we introduce a thread-aware dynamic birthmark called TreSB (Thread-related System call Birthmark) that can effectively detect plagiarism of multithreaded programs. Being extracted by mining behavior characteristics from thread-related system calls, TreSB is less susceptible to thread scheduling as these system calls are sources that impose thread scheduling rather than being affected. In addition, unlike many approaches (Liu et al., 2006; Prechelt et al., 2002; Cosma and Joy, 2012), our approach operates on binary executables rather than source code. The latter is usually unavailable when birthmarking is used to obtain initial evidence of software plagiarism.

We have implemented a prototype based on the PIN (Luk et al., 2005) instrumentation framework, and conducted extensive experiments on an publicly available benchmark[1] consisting of 234 versions of 35 different multithreaded programs. Our empirical study shows that TreSB and its comparison algorithms are credible in differentiating independently developed programs, and resilient to most state-of-the-art semantics-preserving obfuscation techniques implemented in the best commercial and academic tools such as `SandMark` (Collberg et al., 2003) and `DashO` (Patki, 2008). In addition, a comparison of our method against two recently proposed thread-aware birthmarks show that TreSB outperforms both of them with respect to any of the three performance metrics `URC`, `F-Measure` and `MCC`.

*Corresponding author
Email addresses: `zztian@stu.xjtu.edu.cn` (Zhenzhou Tian), `tingliu@mail.xjtu.edu.cn` (Ting Liu), `qhzheng@mail.xjtu.edu.cn` (Qinghua Zheng), `fanming.911025@stu.xjtu.edu.cn` (Ming Fan), `zhuangeryue@stu.xjtu.edu.cn` (Eryue Zhuang), `zijiang.yang@wmich.edu` (Zijiang Yang)

[1]http://labs.xjtudlc.com/labs/wlaq/TAB-PD/site/download.html

The remainder of this paper is organized as following. Section 2 presents our thread-related system call birthmark after introducing the concept and definition of birthmarks. Section 3 describes our approach and prototype on exploiting our birthmarks to detect plagiarism of multithreaded programs. Section 4 presents the empirical study on an open benchmark, including the evaluation of its effectiveness and the performance comparison against existing methods. It also compares TreSB against another potential birthmark that also exploits thread-related system calls. Section 5 reviews related work, followed by conclusions and future work in Section 6.

2. Software Birthmarks

In this section we first give a brief review of the formal definitions of software birthmarks. We then introduce our thread-related system call birthmark TreSB with its implementation, and explain why it is suitable to serve as birthmark for multi-threaded programs.

2.1. Dynamic Software Birthmarks

A software birthmark, whose classical definition is given in Definition 1, is a set of characteristics extracted from a program that reflects intrinsic properties of the program and that can be used to identify the program uniquely. This definition leads to works (Tamada et al., 2004a; Myles and Collberg, 2005; Choi et al., 2009; Park et al., 2011) that extract birthmarks statically.

Definition 1. Software Birthmark (Tamada et al., 2004a). Let p be a program and f be a method for extracting a set of characteristics from p. We say $f(p)$ is a birthmark of p if and only if both of the following conditions are satisfied:

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

Generated mainly by analyzing syntactic features, static birthmarks tend to overlook operational behaviors of a program. As a result, they are usually ineffective against semantics-preserving obfuscations that can modify the syntactic structure of a program. Besides, static birthmarks are easily defeated by the packing techniques (Roundy and Miller, 2013; Guo et al., 2008) that add shells to the plagiarized program to evade detection. Executables processed with these techniques can become rather different in the static level, and static birthmark methods cannot be applied unless the shells can be firstly recognized and unpacked. Thus dynamic birthmarks, as defined in Definition 2, are introduced to remedy the problems. Comparing with static birthmarks, dynamic birthmarks are extracted based on runtime behaviors and thus are believed to be more accurate reflections of program semantics. It has been generally agreed that dynamic birthmarks are more robust against semantics-preserving code obfuscations (Tamada et al., 2004b; Wang et al., 2009a,b; Lim et al., 2009; Chan et al., 2013; Tian et al., 2015).

Definition 2. Dynamic Software Birthmark (Myles and Collberg, 2004). Let p be a program and I be an input to p. Let $f(p)$ be a set of characteristics extracted from p by executing p with input I. We say $f(p, I)$ is a dynamic birthmark of p if and 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 $p \Rightarrow f(p, I) = f(q, I)$.

Based on the above conceptual definitions, various implementable birthmark methods have been developed by mining behavior characteristics from different aspects. Representative dynamic birthmarks include SCSSB (Wang et al., 2009b) that is extracted from system calls, DYKIS (Tian et al., 2013, 2015) that is extracted from executed instructions, and Birthmarking (Schuler et al., 2007) that is extracted from executed Java APIs. As dynamic birthmark based plagiarism detection is essentially determined by the similarity of execution behaviors, the new trend towards multithreaded programming renders existing approaches ineffective. For a program with n threads, each executing k steps, there can be as many as $(nk)!/(k!)^n > (n!)^k$ different thread schedules or interleavings[2], a doubly exponential growth in terms of n and k. Since execution order plays a key role, birthmarks generated from multiple executions of the same program can be very different, thus erroneously indicating the same programs to be independently developed. In order to address this challenge, Tian et al. (Tian et al., 2014b) proposed the concept of thread-aware birthmark.

Definition 3. Thread-Aware Dynamic Software Birthmark (Tian et al., 2014b). Let p, q be two multithreaded programs. Let I be an input and s be a thread schedule to p and q. Let $f(p, I, s)$ be a set of characteristics extracted from p when executing p with I and schedule s. We say $f(p, I, s)$ is a dynamic birthmark of p if and only if both of the following conditions are satisfied:

- $f(p, I, s)$ is obtained only from p itself when executing p with input I and thread schedule s.
- Program q is a copy of $p \Rightarrow f(p, I, s) = f(q, I, s)$.

Similar to Definitions 1 and 2, Definition 3 provides a conceptual guideline without considering any implementation details. In practice it is almost impossible to predetermine a thread schedule and enforce the same thread scheduling across multiple runs, especially for the programs that have been obfuscated or even independently developed (Olszewski et al., 2009; Cui et al., 2011). Thus in our algorithm we mine execution characteristics that are not or little affected by non-deterministic thread schedules. That is, to make a birthmark thread-aware, we must ensure that $\forall_{s_1, s_2 \in S}, f(p, I, s_1) \approx f(p, I, s_2)$, where S denotes the set of all possible thread schedules of program p.

[2]In this paper, thread schedule or interleaving of a program refers to the order of threads whose instructions are executed in a valid execution.

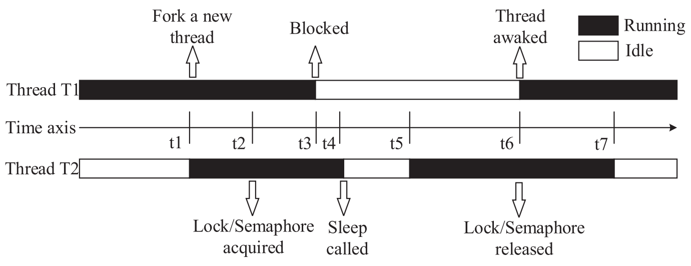

Figure 1: Execution snippet of a multithreaded program with two threads

In this paper, we propose a practical solution that address the challenge of plagiarism detection of multithreaded programs. The principle is similar to previous work (Wang et al., 2009a; Tian et al., 2014b; Wang et al., 2009b; Tian et al., 2016), where the authors argue that modifications of system calls usually leads to incorrect program behavior, and therefore, a birthmark generated from sequence of system calls can be used to identify stolen programs even after they have been modified. We argue that thread related system calls are intrinsic to a multithreaded program. They are the source to impose thread interleaving rather than being affected by the non-determinism. A random or deliberate modification to the thread synchronizations can result in very subtle errors and therefore they are the least possible code to be changed. Such hypothesis is confirmed by our empirical study.

2.2. Thread-Related System Call Birthmark

Figure 1 depicts a typical execution snippet between two possible threads of a multithreaded program. Thread $T1$ (possibly the main thread) forks a new thread $T2$ at time $t1$. Both threads execute concurrently until $T1$ is blocked and entered its idle period at time $t3$ due to unavailable shared resources (lock or semaphore) held by $T2$ since time $t2$. $T2$ continues its execution until time $t4$ when it invokes sleep and resume its execution at time $t5$. At time $t6$ $T2$ releases the shared resource, enabling $T1$ to resume its execution. Both threads execute concurrently again until time $t7$ when $T2$ terminates. The example shows that thread scheduling can be complex even for two threads. The existence of thread synchronization controls and context switches brings certain determinism to the execution. For example the usage of locks protects shared resources and prevents unexpected executions. Meanwhile, there are certain time segments, such as between $t1$ and $t3$ and between $t6$ and $t7$, multiple threads enjoys concurrency without any restrictions. Obviously depending on many factors such as system load, multiple executions, even under the same input, can produce different execution behaviors. This is the key reason that existing dynamic software birthmarking fails to uniquely identify multithreaded programs.

Despite the complex thread interleavings as illustrated in Figure 1, there must exist characteristics or rules that ensure the correct execution under the chaos. System calls that govern thread synchronization, priority setting, thread initiating and disposing, etc, are sources that enforce thread scheduling rather than being affected. They are also essential to the semantics and correct executions of a multithreaded program. We call them thread-related system calls and believe they form a favorable basis for generating thread-aware birthmarks. As summarized in Table 1, where Columns `No.` and `Name` give the system call ID and its name, we treat 65 system calls as thread-related. They accomplish tasks including thread and process management (such as creation, join and termination, capability setting and getting), thread synchronization, signal manipulating, as well as thread and process priority setting.

The fact that only a small portion, about 20%, of system calls are thread related may cast doubt on using them as the base for birthmarks. First of all, what if a multithreaded program does not even have thread related system calls except forking a bunch of threads? In theory it is possible but in reality, at least for the real programs we have studied, such programs do not exist. If there are no coordination and communication among threads, programmers may simply develop several sequential applications. On the other hand, we agree there may exist programs with minimal number of thread related system calls. Our empirical study shows that they have surprisingly strong capability to identify thefts and differentiate different program due to their subtle usage and rich varieties. For example, the *nanosleep* system call used in Figure 1 may never exist in another independently developed program, and removing it may likely leads to subtle errors. Even for the same types of thread operations such as forking a new thread, a program may use the *clone* while other independently developed programs may use *fork* or *vfork*. Due to their complicated parameter usages and subtle difference in their meanings, it is very difficult for automated tools or even programmers to change from one to another. Besides, the frequency and order of occurrences, which play important role in our birthmark generation, also contribute to the credibility and resilience of our approach.

Based on the above discussions, we propose to extract thread-aware dynamic birthmarks from thread-related system call sequences. A thread-related system call sequence $tos(p, I) = \langle e_1, e_2, \cdots, e_n \rangle$ consists of system calls recorded during the runtime of a multithreaded program p under input I, in which e_i is a thread-related system call instance. Usually, the thread-related system call sequences across multiple runs are difficult to compare directly. In order to address the problem, we adopt the k-gram algorithm (Myles and Collberg, 2005) to bound the sequences with a length k window, generating a set of fixed-length short subsequences called k-grams. Finally, as with the typical dynamic birthmarks such as SCSSB (Wang et al., 2009b), DYKIS (Tian et al., 2013, 2015) and Birthmarking (Schuler et al., 2007), our birthmark is a key-value pair set. The keys consist of all unique grams and values are the frequencies of the corresponding grams. Definition 4 gives the formal definition of TreSB, the birthmark based on thread-related system calls.

Definition 4. TreSB. Let $tos(p, I) = \langle e_1, e_2, \cdots, e_n \rangle$ be a thread-related system call sequence when executing program p with input I. A sub-sequence g_j of $tos(p, I)$ is a k-gram if $g_j = \left\langle e_j, e_{j+1}, \cdots, e_{j+k-1} \right\rangle$, where $1 \leq j \leq n - k + 1$. Let $gram(p, I, k) = \langle g_1, \ldots, g_{n-k+1} \rangle$ be the se-

Table 1: Thread-related system calls of Linux kernel version 3.2

No.	Name	No.	Name	No.	Name	No.	Name	No.	Name	No.	Name
1	exit	48	signal	81	setgroups	154	sched_setparam	175	rt_sigprocmask	243	set_thread_area
2	fork	53	lock	96	getpriority	155	sched_getparam	176	rt_sigpending	244	get_thread_area
7	waitpid	57	setpgid	97	setpriority	156	sched_setscheduler	177	rt_sigtimedwait	256	epoll_wait
11	execve	64	getppid	114	wait4	157	sched_getscheduler	178	rt_sigqueueinfo	258	set_tid_address
20	getpid	65	getpgrp	117	ipc	158	sched_yield	179	rt_sigsuspend	270	tgkill
26	ptrace	66	setsid	120	clone	159	sched_get_priority_max	184	capget	284	waitid
29	pause	67	sigaction	123	modify_ldt	160	sched_get_priority_min	185	capset	321	signalfd
34	nice	69	ssetmask	126	sigprocmask	161	sched_rr_get_interval	190	vfork	327	signalfd4
37	kill	72	sigsuspend	132	getpgid	162	nanosleep	238	tkill	331	pipe2
42	pipe	73	sigpending	136	personality	172	prctl	241	sched_setaffinity	346	setns
46	setgid	80	getgroups	147	getsid	174	rt_sigaction	242	sched_getaffinity		

quence of all the k-grams in $tos(p, I)$. The key-value pair set $p^I_{\mathcal{B}}(k) = \left\{(g_j, freq\left(g_j\right)) \,|g_j \in gram\,(p, I, k) \wedge \forall_{i \neq j}, g_i \neq g_j\right\}$, where $freq(g_j)$ represents the frequency of g_j that occurs in $gram(p, I, k)$, is a thread-related system call birthmark TreSB.

We also use $p_{\mathcal{B}}$ to represent a TreSB if removing the input symbol does not cause confusion. In addition, we define $kSet(p_{\mathcal{B}})$ to be the set of keys in the TreSB $p_{\mathcal{B}}$.

3. TreSB Based Software Plagiarism Detection

Obtaining birthmark is the first step towards plagiarism detection. The next step is to quantify the similarities and then decide whether plagiarism exists.

3.1. Similarity Calculation

In the literature of software birthmarking, the similarity between two programs is measured by the similarity of their birthmarks. Different methods of similarity measurements are used depending on different birthmark formats that in general are sequences, sets or graphs. For birthmarks in sequence format, their similarity can be computed with pattern matching methods, such as measuring the longest common subsequences (LCS) (Jhi et al., 2011; Zhang et al., 2012; Jhi et al., 2015). Birthmarks in set form are usually generated by abstracting sequences into shorter subsequence sets (Myles and Collberg, 2005; Schuler et al., 2007; Xie et al., 2010; Tian et al., 2014a, 2015), and then various metrics used in the field of information retrieval can be adopted for calculating the similarity between sets, including Dice coefficient (Choi et al., 2009), Jaccard index (Schuler et al., 2007), Cosine distance (Tian et al., 2013). Computing the similarity of graphs is relatively more complex. It is conducted by either graph monomorphism (Chan et al., 2013) or isomorphism algorithms (Wang et al., 2009a), or by translating a graph into a vector using algorithms such as random walk with restart (Chae et al., 2013).

Our birthmark is a set composed of key-value pairs, thus similarity calculation methods such as Cosine distance, Jaccard index, Dice coefficient and Containment can be used. Since these four metrics all have ever been used for computing birthmark similarity in previous studies, we implement all of them in our prototype for better comparison against existing works.

Given two TreSBs $p_{\mathcal{B}} = \{(k_1, v_1), \cdots, (k_n, v_n)\}$ and $q_{\mathcal{B}} = \left\{\left(k'_1, v'_1\right), \cdots, (k'_m, v'_m)\right\}$, let $U = kSet(p_{\mathcal{B}}) \cup kSet(q_{\mathcal{B}})$. We convert set U to vector $\vec{U} = \left\langle k''_1, \cdots, k''_{|U|}\right\rangle$ by assigning an arbitrary order to the elements in U. Let vector $\overrightarrow{p_{\mathcal{B}}} = \langle a_1, a_2, \cdots, a_{|U|}\rangle$, where

$$a_i = \begin{cases} v_i, & if\ k''_i \in kSet\,(p_{\mathcal{B}}) \\ 0, & if\ k''_i \notin kSet\,(p_{\mathcal{B}}) \end{cases}$$

Likewise we define $\overrightarrow{q_{\mathcal{B}}} = (b_1, b_2, \cdots, b_{|U|})$. The four metrics that quantify the similarities between $p_{\mathcal{B}}$ and $q_{\mathcal{B}}$ are defined as:

$$\begin{aligned} Ex-Cosine\,(p_{\mathcal{B}}, q_{\mathcal{B}}) &= \frac{\vec{p_{\mathcal{B}}} \bullet \vec{q_{\mathcal{B}}}}{\left|\vec{p_{\mathcal{B}}}\right|\left|\vec{q_{\mathcal{B}}}\right|} \times \theta, \\ Ex-Jaccard\,(p_{\mathcal{B}}, q_{\mathcal{B}}) &= \frac{|p_{\mathcal{B}} \cap q_{\mathcal{B}}|}{|p_{\mathcal{B}} \cup q_{\mathcal{B}}|} \times \theta, \\ Ex-Dice\,(p_{\mathcal{B}}, q_{\mathcal{B}}) &= \frac{2|p_{\mathcal{B}} \cap q_{\mathcal{B}}|}{|p_{\mathcal{B}}| + |q_{\mathcal{B}}|} \times \theta, \\ Ex-Containment\,(p_{\mathcal{B}}, q_{\mathcal{B}}) &= \frac{|p_{\mathcal{B}} \cap q_{\mathcal{B}}|}{|p_{\mathcal{B}}|} \times \theta \end{aligned}$$

where

$$\theta = \frac{min\left(\left|\vec{p_{\mathcal{B}}}\right|, \left|\vec{q_{\mathcal{B}}}\right|\right)}{max\left(\left|\vec{p_{\mathcal{B}}}\right|, \left|\vec{q_{\mathcal{B}}}\right|\right)}$$

and

$$\left|\overrightarrow{p_{\mathcal{B}}}\right| = \sqrt{\sum\nolimits_{a_i \in \vec{p_{\mathcal{B}}}} a_i^2}, \qquad \left|\overrightarrow{q_{\mathcal{B}}}\right| = \sqrt{\sum\nolimits_{b_i \in \vec{q_{\mathcal{B}}}} b_i^2}$$

The similarity of two TreSBs is represented by $Sim\,(p_{\mathcal{B}}, q_{\mathcal{B}}) = sim_c\,(p_{\mathcal{B}}, q_{\mathcal{B}})$, where $c \in \{Ex-Cosine, Ex-Jaccard, Ex-Dice, Ex-Containment\}$.

3.2. Plagiarism Detection

The purpose of extracting birthmarks and calculating their similarity is to eventually determine whether there exists plagiarism. False negatives are possible due to sophisticated code obfuscation techniques that camouflage stolen software. One of our goals is to make our approach resilient to these techniques and tools. False positives, on the other hand, are also possible, even though executions faithfully represent program behavior under a particular input vector. For example, two independently developed programs adopting standard error-handling subroutines may exhibit identical behavior under error-inducing inputs. In order to alleviate this problem, multiple similarity scores are computed for birthmarks obtained under multiple inputs, and the average of the scores is used to decide plagiarism.

Let p and q be the plaintiff and defendant programs, respectively. Given a set of inputs $\{I_1, I_2, \cdots, I_n\}$ to drive the execution of the programs, we obtain n pair of TreSBs $\{(p_{\mathcal{B}_1}, q_{\mathcal{B}_1}), (p_{\mathcal{B}_2}, q_{\mathcal{B}_2}), \cdots, (p_{\mathcal{B}_n}, q_{\mathcal{B}_n})\}$. The similarity score between program p and q is calculated by $Sim(p,q) = \sum_{i=1}^{n} sim(p_{\mathcal{B}_i}, q_{\mathcal{B}_i}) \Big/ n$, whose value is between 0 and 1. The existence of plagiarism between p and q is then decided according to the average similarity score and a predefined threshold ε as follows:

$$sim(p,q) = \begin{cases} > 1-\varepsilon & positive: \ q \ is \ a \ copy \ of \ p \\ < \varepsilon & negative: \ q \ is \ not \ a \ copy \ of \ p \\ otherwise & inconclusive \end{cases} \quad (1)$$

3.3. Implementation

Figure 2 gives an overview of our TreSB based birthmarking tool, where plaintiff and defendant represent the original program and the program suspected of plagiarism, respectively. The tool consists of several modules. The first module, `TreTracer`, is implemented as a `PIN` (Luk et al., 2005) plugin to monitors program executions. It recognizes and records the thread-related system calls by instrumenting call sites dynamically. Table 1 lists the 65 thread-related system calls of Linux kernel v3.2, on which platform we evaluate the effectiveness of the TreSB method. The output of this module is a thread-related system call sequence under a particular input vector. Each record in the sequence consists of a system call number and its corresponding system call name, and the return value during the execution.

The raw sequences extracted by the `TreTracer` are not appropriate to be directly used for birthmark generation. Since failed calls do not affect the behavior characteristics of a program (Wang et al., 2009a; Tian et al., 2014b; Wang et al., 2009b), we treat them as noise and delete them from the sequences. This is accomplished by checking the return value of each record. Also, in order to further reduce the impact of thread scheduling as well as other random factors such as os-state related operations, each program is executed multiple times with the same input. We then select two most similar sequences for further analysis. These tasks are accomplished by an optimization module.

The birthmark generator obtains TreSBs from the thread-related system call sequences passed from the optimizer. In the similarity calculator, scores are computed between the birthmarks of plaintiff and defendant programs with respect to each of the four similarity metrics as described in section 3.1. Finally a decision is made using Equation 1, where a default value of $\varepsilon = 0.3$ is adopted. However, users can adjust its value depending on how strong the plagiarism evidence is desired. A ε value between 0.15 and 0.35 has been used in prior work (Choi et al., 2009; Schuler et al., 2007; Tian et al., 2014b; Chae et al., 2015).

4. Experiments and Evaluation

A high quality birthmark must exhibit low ratio of incorrect classifications for a certain ε. This can be quantified by the resilience and credibility properties (Myles and Collberg, 2005; Choi et al., 2009). In order to demonstrate the merit of our birthmarking technique, we center our empirical study on the two properties.

Property 1: **Resilience.** Let p be a program and q be a copy of p generated by applying semantics-preserving code transformations τ. A birthmark is resilient to τ if $sim(p_{\mathcal{B}}, q_{\mathcal{B}}) > 1 - \varepsilon$.

Property 2: **Credibility.** Let p and q be independently developed programs that may accomplish the same task. A birthmark is credible if it can differentiate the two programs, that is $sim(p_{\mathcal{B}}, q_{\mathcal{B}}) < \varepsilon$.

4.1. Experimental Setup

We have conducted extensive experiments for evaluating the effectiveness of our method on an open benchmark established by Tian et al. (Tian et al., 2014b). Table 2 lists basic information about the benchmark. Column *#Ver* gives the number of versions of each program including the original program and its obfuscated versions. Column *Size* lists the number of kilobytes of the largest version, with its version number listed in Column *Version*. In the following we summarize our testing environment.

- The benchmark consists of 234 versions of 35 mature multithreaded software implemented in C or Java, including:
  - six compression/decompression software: `pigz`, `lbzip`, `lrzip`, `pbzip2`, `plzip` and `rar`.
  - five audio players: `cmus`, `mocp`, `mp3blaster`, `mplayer` and `sox`.
  - ten web browsers: `arora`, `chromium`, `dillo`, `dooble`, `epiphany`, `firefox`, `konqueror`, `luakit`, `midori` and `seaMonkey`.
  - four Java programs from the JavaG benchmark: `Crypt`, `Series`, `SparseMat` and `SOR`.
  - ten programs from the PARSEC 3.0 benchmark: `blackschole`, `bodytrack`, `fludanimate`, `canneal`, `dedup`, `ferret`, `freqmine`, `streamcluster`, `swaption`, `x264`.
- We evaluate the resilience of TreSB against relatively weak obfuscations provided by two different compilers `gcc` and `llvm` with various optimization levels.
- We evaluate the resilience of TreSB against strong obfuscations implemented in special obfuscators, including `Sandmark`, `Zelix KlassMaster`, `Allatori`, `DashO`, `Jshrink`, `ProGuard` and `RetroGuard`.
- We evaluate the resilience of TreSB against packing tool `UPX` which can obfuscate binaries.
- We evaluate the credibility of TreSB with independently developed programs.

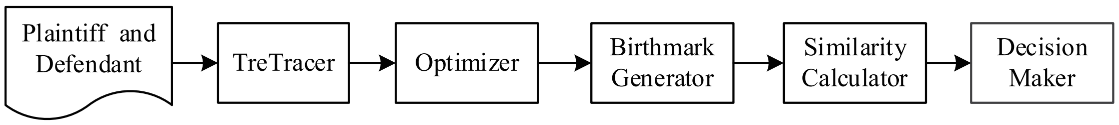

Figure 2: Overview of the TreSB based software plagiarism detection tool

Table 2: Benchmark programs.

Name	Size(kb)	Version	#Ver	Name	Size(kb)	Version	#Ver	Name	Size(kb)	Version	#Ver
pigz	294	2.3	21	chromium	80,588	28.0.1500.71	1	SOR	593.3	JavaG1.0	44
lbzip	113.3	2.1	1	dillo	610.9	3.0.2	1	blackschole	12.5	Parsec3.0	2
lrzip	219.2	0.608	1	Dooble	364.4	0.07	1	bodytrack	647.5	Parsec3.0	2
pbzip2	67.4	1.1.6	1	epiphany	810.9	3.4.1	1	fludanimate	46.4	Parsec3.0	2
plzip	51	0.7	1	firefox	59,904	24.0	1	canneal	414.7	Parsec3.0	2
rar	511.8	5.0	1	konqueror	920.1	4.8.5	1	dedup	127.2	Parsec3.0	2
cmus	271.6	2.4.3	1	luakit	153.4	d83cc7e	1	ferret	2,150	Parsec3.0	2
mocp	384	2.5.0	1	midori	347.6	0.4.3	1	freqmine	227.6	Parsec3.0	2
mp3blaster	265.8	3.2.5	1	seaMonkey	760.9	2.21	1	streamcluster	102.7	Parsec3.0	2
mplayer	4,300	r34540	1	Crypt	518.1	JavaG1.0	43	swaption	94	Parsec3.0	2
sox	55.2	14.3.2	1	Series	593.3	JavaG1.0	43	x264	896.3	Parsec3.0	2
arora	1,331	0.11	1	SparseMat	593.3	JavaG1.0	43				

- We compare the overall performance of the TreSB method with two latest thread-aware birthmarks SCSSB$_{SA}$ and SCSSB$_{SS}$ (Tian et al., 2014b), as well as the traditional birthmark SCSSB (Wang et al., 2009b), with respect to three widely used metrics including `URC`, `F-Measure`, and `MCC`.
- We propose another birthmark called TreCxtB that also exploits thread-related system calls, and compare it against TreSB.

It should be noted that, with all other factors the same, different values of k leads to different TreSBs. Fortunately, as it has been confirmed in previous papers (Tian et al., 2013, 2015; Wang et al., 2009a; Schuler et al., 2007) where k-grams are also used to generate birthmarks, setting the value of k to 4 or 5 is a proper compromise between accuracy and efficiency. In our evaluation we set $k = 5$ as adopted in (Tian et al., 2014b) and (Wang et al., 2009b), since they are the works that we mainly compare with. As discussed in Section 3.2, programs are executed multiple times under different inputs. However, we always give the same inputs to the plaintiff and defendant programs. In our experiments, whenever available, we utilize the inputs that are distributed with the benchmarks. For example, eighteen testing audio, image and text files that are distributed with `pigz` are used to drive the executions of `pigz` and its obfuscated versions in our experiments. For other programs such as the ten web browsers, we feed them with multiple websites such as `https://en.wikipedia.org/wiki/Plagiarism`.

4.2. Validation of Resilience Property

4.2.1. Resilience to Different Compilers and Optimization Levels

Stolen software is often compiled with different compilers or compiler optimization levels to evade detection. In this experiment, we choose the multithreaded compression software `pigz-2.3` as the experimental subject. Compiled with two compilers `llvm3.2` and `gcc4.6.3`, along with multiple optimization levels (`-O0`, `-O1`, `-O2` `-O3` and `-Os`) and the debug option (`-g`) switched on or off, we obtain 20 different executables. The statistical characteristics of the 20 binaries, obtained by using the disassembler IDA Pro, are summarized in Table 3. The table gives the statistical differences on the size, the number of functions, the number of instructions, the number of basic blocks and the number of function calls. The data indicate that even weak code transformations can make significant differences to the produced binaries.

Since the 20 binaries are obtained from the same source code, our approach is resilient to these weak code transformations if the similarity scores are high, indicating the existence of plagiarism. Figure 3 illustrates the distribution graph of the similarity scores calculated between the birthmarks of the 20 `pigz` binaries, where the vertical axis represents the metrics adopted for similarity computation, and the horizontal axis represents the percentage of birthmark pairs belonging to each range as specified in the legend. It can be observed that the similarity scores are all above 0.8, except for Ex-Jaccard, and the majority are above 0.9. Even for Ex-Jaccard, only a tiny fraction of the similarity scores are between 0.7-0.8 and all others are above 0.8. For a threshold value of 0.3, our technique will claim the existence of plagiarism for all the experiments. The experimental results indicate that TreSB exhibits strong resilience against the obfuscations caused by different compilers and optimization levels.

4.2.2. Resilience to Advanced Obfuscation Tools

In this group of experiments, we evaluate the resilience of TreSB against advanced obfuscation techniques available in sophisticated tools. Specifically, we conduct experiments on the 165 obfuscated versions of the 4 programs from the JavaG Benchmark, including `Crypt`, `Series`, `SparseMat` and `SOR`.

Table 3: Statistical differences between `pigz` versions generated with different compilers and optimization levels

	Size(Kb)	**#Functions**	**#Instructions**	**#Blocks**	**#Calls**
Max.	295	415	22178	3734	2376
Min.	84	342	13860	2672	1031
Avg.	151.75	380.25	16269	3068.9	1206.8
Stdev.	60.53	23.4	2679	286.58	280.9

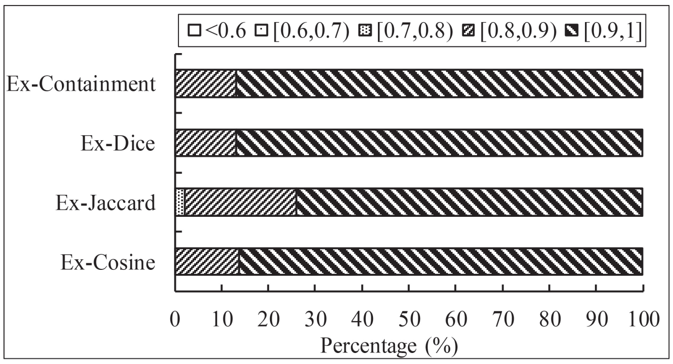

Figure 3: Similarity scores between `pigz` versions generated with different compilers and optimization levels.

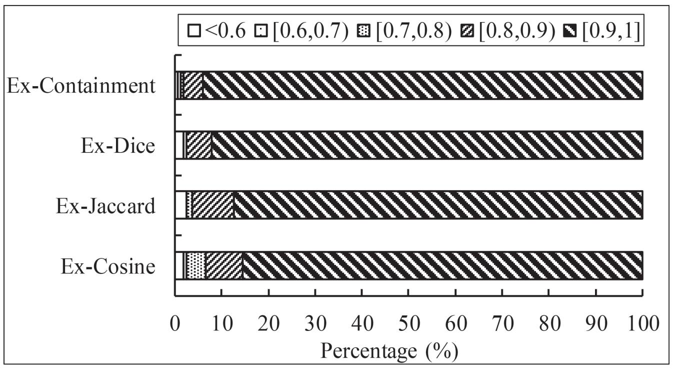

Figure 4: Similarity scores between a program and its obfuscated versions

These 165[3] versions are obtained by either applying the 39 obfuscation techniques implemented in the obfuscation tool `SandMark` (Collberg et al., 2003) to each program one at a time, or by applying multiple obfuscation techniques implemented in six commercial and open source obfuscation tools, including `Zelix KlassMaster`[4], `Allatori`[5], `DashO`[6], `JShrink`[7], `ProGuard`[8] and `RetroGuard`[9], simultaneously to a single program, on the premise of ensuring semantics equivalence between each original and the transformed programs. Semantic equivalence is confirmed via empirical study rather than theoretical proof. We consider a transformed program is equivalent to the original one if they produce the same outputs in our experiments.

The similarity scores are calculated between the original program and one of its obfuscated versions. Figure 4 gives the similarity score distribution with the vertical and horizontal axes indicating the same meaning as in Figure 3. It can be observed that the majority scores locate in the 0.7-above region. Yet as indicated by the white bars in the figure, there exist some similarity scores that are relatively low. We checked the experimental data, and found that these low scores all happened between each program and its `Allatori`-obfuscated version. It seems that TreSB is susceptible to the obfuscation produced by `Allatori`. By manually investigating the recorded execution traces, we find that an extra thread is added in the `Allatori`-obfuscated version, compared with its original version during runtime. The new thread decrypts strings that are encrypted during the obfuscation process of `Allatori`, which introduces extra thread-related system calls. For example, *NR_waitpid* and *NR_pipe*, which do not appear in the execution of the original version, appear in the trace of `Allatroi`-obfuscated `Crypt`. We anticipate an approach to defeat our algorithm is to introduce new threads that emit thread-related system calls to disguise plagiarism. To address this issue, we can add a preprocessing module. After firstly projecting a trace on individual threads, the module performs maximal matching among all subtraces of the plaintiff and defendant to detect unmatched subtraces. The unmatched traces are very likely to be the noise introduced by the obfuscation and thus can be removed. Figure 5 gives the distribution graph after performing such trace filtering. It can be observed that all scores are above 0.6. It indicates that TreSB is resilient to such anticipated obfuscation techniques.

[3]Note that with our experimental setup, there should be 180 obfuscated versions if all the obfuscations are successfully applied. Yet some of the obfuscations fail to transform the programs or fail to transform them into semantically equivalent executables.

[4]http://www.zelix.com/klassmaster

[5]http://www.allatori.com

[6]https://www.preemptive.com/products/dasho

[7]https://www.e-t.com/jshrink.html

[8]http://proguard.sourceforge.net

[9]http://java-source.net/open-source/obfuscators/retroguard

4.2.3. Resilience to Packing Tools

The packing tools or packers (Roundy and Miller, 2013; Guo et al., 2008; Kim et al., 2010), which implement various binary obfuscation techniques as well as compression and encryption techniques, are widely used to hide malicious malware or to protect proprietary software from illegal modification and cracking. Such techniques may be used to evade plagiarism detection. These tools can defeat static birthmarks as they significantly modify the original programs.

In this section, resilience of TreSB is evaluated on the binaries in the benchmark that are processed with packing techniques, including the previously used `pigz`, `Crypt`, `Series`,

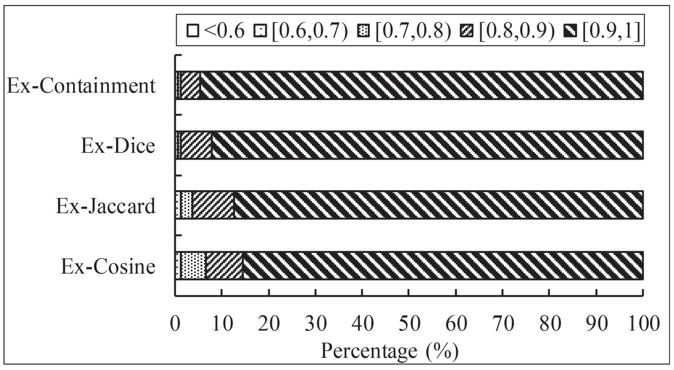

Figure 5: Similarity scores with trace filtering performed

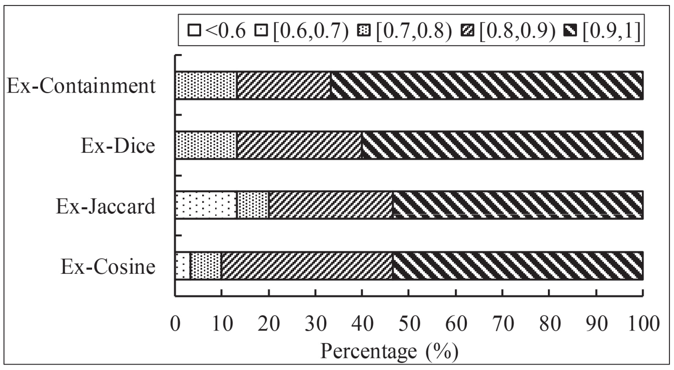

Figure 6: Similarity scores between a program and its UPX-packed versions

`SparseMat`, `SOR` as well as 10 other multithreaded programs `blackschole`, `bodytrack`, `fludanimate`, `canneal`, `dedup`, `ferret`, `freqmine`, `streamcluster`, `swaption` and `x264`. These programs all have versions packed by `UPX`[10], the only publicly available packing tool for the ELF-format (executable file format under Linux).

Figure 6 depicts the distribution of the similarity scores calculated between birthmarks of the original programs and their corresponding `UPX`-packed versions. It can be observed the majority of the scores are above 0.7 and all scores are above 0.6. For a default threshold value of 0.3, there are very few cases, when adopting Ex-Jaccard and Ex-Cosine metrics, TreSB is uncertain about the existence of plagiarism.

4.3. Validation of Credibility Property

Credibility of TreSB is evaluated by its capability of distinguishing independently developed programs. Three widely used types of software are selected as our experimental subjects, including six multithreaded compression software (`lbzip2`, `lrzip`, `pbzip2`, `pigz`, `plzip` and `rar`), ten web browsers (`arora`, `chromium`, `dillo`, `Dooble`, `epiphany`, `firefox`, `konqueror`, `luakit`, `midori` and `seaMonkey`), and five audio players (`cmus`, `mocp`, `mp3blaster`, `mplayer` and `sox`).

[10]http://upx.sourceforge.net/

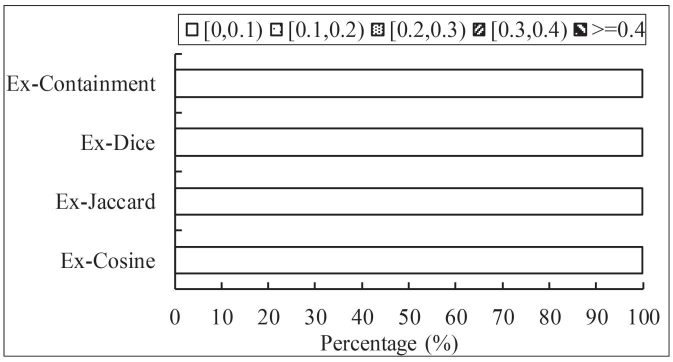

Figure 7: Similarity scores for software in different categories

4.3.1. Distinguishing Programs in Different Categories

In this group of experiments, similarity scores between the compression programs and audio players are computed. Since the comparison is between software that accomplish totally different tasks, we expect very low similarities. As shown in Figure 7, the results are as expected. All the similarity scores are below 0.1, indicating good credibility of TreSB in distinguishing programs without much in common.

4.3.2. Distinguishing Programs in Same Categories

Distinguishing programs in the same category is more challenging because they overlap greatly in their functionality. Figure 8 depicts the distribution of the similarity scores calculated between the ten web browsers. It can be observed that the majority of the scores are below 0.1, indicating good capability of TreSB in distinguishing similar but independently developed programs. However, there exist a few cases where the similarity scores are above 0.3, which shows that TreSB is unsure about whether plagiarism exists.

In order to better interpreting the data, Table 4 gives the max and average similarity scores in the first two columns after the names of the matrices. Consistent with the distribution graph, while the average scores are well below 0.1, the maximum similarity scores are all above 0.25. After careful examination of the programs we have found that some of the browsers share the same layout engine. Specifically, five of the browsers (`arora`, `Dooble`, `epiphany`, `luakit` and `midori`) are `Webkit`-based while the others utilize different layout engines. Column `Avg+` lists the average similarity scores between those five `Webkit`-based browsers, and Column `Avg-` gives the average scores between the `Webkit`-based and non-`Webkit`-based browsers. As expected, the values in Column `Avg+` are 6 to 10 times bigger than the values in Column `Avg-`.

Since the goal of TreSB is to detect whole program plagiarism, we believe the experimental results show strong credibility for real-world applications where certain libraries are shared. If there exist trivial programs that simply calls the same third-party functions, it is hard to give a conclusive judgment even with manual examination.

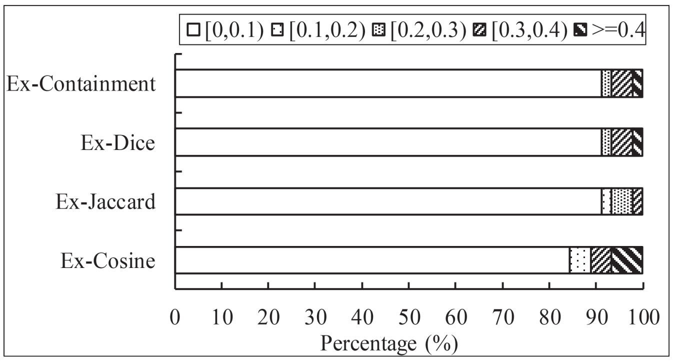

Figure 8: Similarity scores for software in the same category

Table 4: Credibility evaluation using ten web browsers

	Max	Avg	Avg+	Avg-
Ex-Cosine	0.584	0.068	0.178	0.023
Ex-Jaccard	0.373	0.030	0.092	0.008
Ex-Dice	0.509	0.049	0.142	0.015
Ex-Containment	0.512	0.055	0.144	0.024

4.4. *Comparing with Other Birthmarks*

This section compares the performance of TreSB against two birthmarks SCSSB$_{SA}$ and SCSSB$_{SS}$ (Tian et al., 2014b) that adapt SCSSB (Wang et al., 2009b) for multithreaded programs, as well as the original SCSSB. All the programs from Section 4.2 to Section 4.3 are taken as the experimental subjects.

4.4.1. *Performance Evaluation with Respect to URC*

As discussed earlier, resilience and credibility reflect from different aspects the qualities of a birthmark. `URC` (Union of Resilience and Credibility) (Xie et al., 2010), defined below, is a metric that considers both aspects.

$$\texttt{URC} = 2 \times \frac{R \times C}{R + C} \tag{2}$$

In the equation, R represents the ratio of correctly classified pairs where plagiarism exists and C represents the ratio of correctly classified pairs where plagiarism does not exist. The value of `URC` ranges from 0 to 1, with higher value indicating a better birthmark. Let EP be the set of pairs of programs such that $\forall\,(p, q) \in EP$, plagiarism indeed exists between q and p, and JP be the set of pairs such that $\forall\,(p, q) \in JP$, a plagiarism detection method believes that plagiarism exists between q and p. Similarly, let EI to be the set of pairs such that $\forall\,(p, q) \in EI$, q and p are independent, and JI be the set of pairs that are deemed independent by a plagiarism detection method. R and C are defined as:

$$R = \frac{|EP \cap JP|}{|EP|} \quad \text{and} \quad C = \frac{|EI \cap JI|}{|EI|}. \tag{3}$$

As indicated by Equation 1, the detection result relies on the value of threshold ε. Therefore in the experiments we vary the value of ε from 0 to 0.5. Note that ε cannot be greater than 0.5, otherwise plagiarism can be claimed to be existing and non-existing at the same time. Figure 9 depicts the experimental results, where each subfigure corresponds to the specific metric utilized for similarity computation. As indicated by the pink lines, TreSB always performs better than the other three birthmark methods.

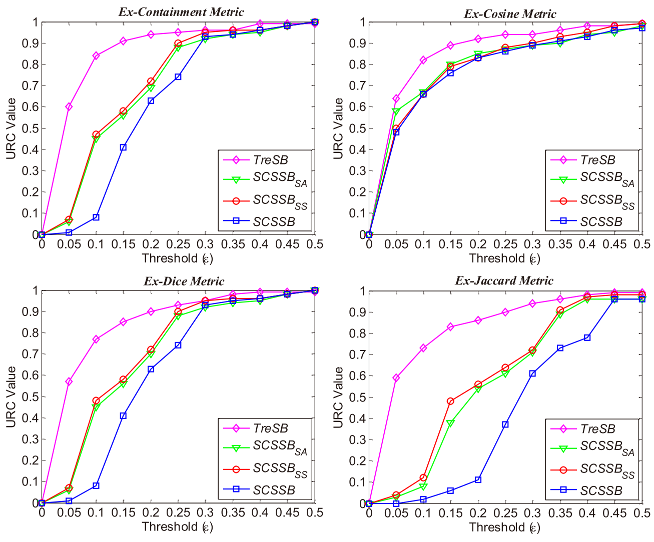

Figure 9: Performance evaluation with respect to the `URC` metric.

4.4.2. *Evaluation with F-Measure and MCC*

Equation 1 indicates that the birthmark based plagiarism detection give three-value results. If the similarity score of two birthmarks is between ε and $1 - \varepsilon$, there is no definite answer whether plagiarism exists. The inconclusiveness reflects the nature of birthmark based techniques, which are mostly used for collecting evidence rather than proving or disproving the existence of plagiarism. Such three-value outcome also explains the reason that `URC` gives better results with higher value of ε in Figure 9. This is because `URC` mainly measures the rate of correct classifications, where inconclusiveness is considered as incorrect classification. As the value of ε increases, the chance of inconclusiveness becomes smaller, leading to less incorrect classifications.

To address the problem, we further compare the birthmark methods against two other metrics, `F-Measure` and `MCC` (Matthews Correlation Coefficient) (Matthews, 1975), that are widely used in the areas of information retrieval and machine learning. However, these two metrics cannot be directly applied as they mainly measure binary classifications. Thus in the following, we revise the definition of *sim* by removing the inconclusiveness:

$$sim\,(p_{\mathcal{B}}, q_{\mathcal{B}}) = \begin{cases} \geq \varepsilon & q \textit{ is a copy of } p \\ < \varepsilon & q \textit{ is not a copy of } p \end{cases} \tag{4}$$

`F-Measure` is based on the weighted harmonic mean of *Precision* and *Recall*:

$$\texttt{F-Measure} = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{5}$$

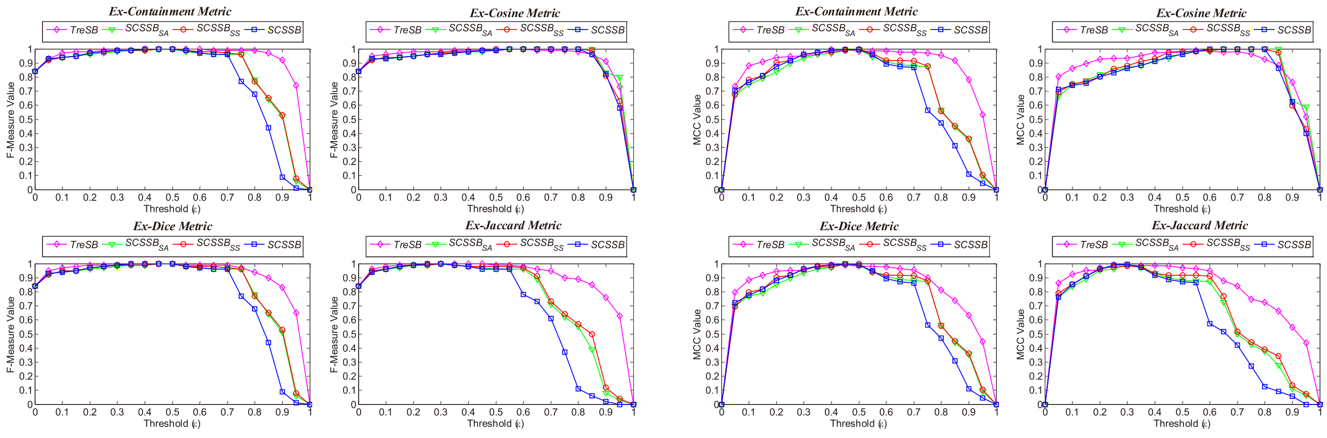

Figure 10: Performance evaluation with respect to `F-Measure` and `MCC`

where *Precision* and *Recall* are defined as following:

$$Precision = \frac{|EP \cap JP|}{|JP|} \quad \text{and} \quad Recall = \frac{|EP \cap JP|}{|EP|}$$

`MCC`, defined below, is regarded as one of the best metrics that evaluate true and false positives and negatives by a single value.

$$\texttt{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{6}$$

where TP, TN, FP and FN are the number of true positives, true negatives, false positives and false negatives, respectively. They can be computed with the following formulas:

$$TP = |EP \cap JP|\,; \qquad FN = |EP \cap JI|$$

$$FP = |EI \cap JP|\,; \qquad TN = |EI \cap JI|$$

The values of `F-Measure` are between 0 and 1, and `MCC` between -1 and 1, with closing to 1 indicating better quality. The experimental results are depicted in Figure 10 . The left four sub-figures give results of `F-Measure` and the right four sub-figures show the results of `MCC`. Note that the values between -1 and 0 never appear in our experiments so the scale in the figure for `MCC` is between 0 and 1. It can be observed that TreSB almost always performs better than all the other birthmarks across the whole x-axis.

4.4.3. Comparing Birthmarks with AUC Analysis

For more specific and intuitional comparison, we compute `AUC` (Area Under the Curve) for each birthmarking method with respect to the `URC`, `F-Measure` and `MCC` metrics. Note that larger value of `AUC` indicate better birthmark quality. The experimental results are summarized in the Table 5, where SA and SS denote birthmarks SCSSB$_{SA}$ and SCSSB$_{SS}$, respectively. As it shows, the `AUC` values of TreSB are all larger than that of the other birthmarks'.

We quantify the performance gains by taking the original SCSSB as baseline. That is, we compute the improvement of each thread-aware birthmark against SCSSB with respect to the same similarity metric and the same performance evaluation metric using the following equation:

$$PerGain = \frac{AUC_{tab} - AUC_{scssb}}{AUC_{scssb}} \times 100\%$$

where AUC_{tab} and AUC_{scssb} represent the `AUC` value of a thread-aware birthmark and SCSSB, respectively. Note that both AUC_{tab} and AUC_{scssb} in the equation are relative. Their values vary with respect to different similarity metrics and performance evaluation metrics. For example, the AUC_{scssb} value with respect to Ex-Cosine similarity and URC metric is 0.776, while its value with respect to Ex-Dice similarity and URC metric is 0.619. Therefore, the corresponding $PerGain$ values for TreSB are:

$$\frac{0.857-0.776}{0.776} \times 100\% = 10\%, and \ \frac{0.843-0.619}{0.619} \times 100\% = 36\%$$

respectively. The last row in Table 5 gives the average and maximal performance gains of each birthmark against SCSSB. It can be observed TreSB is significantly better that other birthmarks.

4.5. Effectiveness of the Sequence Selection

As mentioned in Section 3.3, we conduct an optimization by pre-selecting two most similar sequences from plaintiff and defendant programs to reduce the randomness of thread interleaving. To evaluate the necessity of this optimization, in the first four rows of Table 6 we give the `AUC` values of comparisons between birthmarks generated from two most dissimilar sequences. It can be observed the data are not as good as those in Table 5. In order to quantify the performance degradation we use the following equation and give the results in the last row `PerDegr`.

$$PerDegr = \frac{\sum\limits_{simMetrics} AUC_{opt} - AUC_{noOpt}}{\sum\limits_{simMetrics} AUC_{opt}} \times 100\%$$

We can see that the performance of all birthmark methods are improved after the optimization, indicating the necessity and

Table 5: Comparison Using AUC Values

	URC				F-Measure				MCC			
	SCSSB	SA	SS	TreSB	SCSSB	SA	SS	TreSB	SCSSB	SA	SS	TreSB
Ex-Containment	0.618	0.693	0.709	0.863	0.802	0.847	0.853	0.942	0.704	0.737	0.753	0.868
Ex-Cosine	0.776	0.816	0.793	0.857	0.919	0.933	0.924	0.938	0.814	0.837	0.827	0.861
Ex-Dice	0.619	0.694	0.71	0.843	0.802	0.847	0.853	0.927	0.705	0.74	0.754	0.84
Ex-Jaccard	0.412	0.564	0.591	0.827	0.691	0.768	0.784	0.912	0.602	0.665	0.687	0.818
PerGain (%)	-	17\37	19\43	47\101	-	6\11	7\13	17\32	-	6\10	7\14	21\36

Table 6: Effectiveness of Sequence Selection

	URC				F-Measure				MCC			
	SCSSB	SA	SS	TreSB	SCSSB	SA	SS	TreSB	SCSSB	SA	SS	TreSB
Ex-Containment	0.344	0.614	0.635	0.753	0.659	0.799	0.809	0.874	0.577	0.711	0.723	0.761
Ex-Cosine	0.74	0.794	0.792	0.767	0.876	0.923	0.908	0.881	0.769	0.83	0.817	0.781
Ex-Dice	0.343	0.614	0.631	0.778	0.659	0.798	0.809	0.888	0.578	0.712	0.721	0.79
Ex-Jaccard	0.098	0.4	0.465	0.734	0.503	0.687	0.722	0.859	0.436	0.604	0.636	0.75
PerDegr (%)	37.1	12.5	10.0	10.6	16.1	5.5	4.9	5.8	16.5	4.1	4.1	9.0

benefit of applying the optimization. It can also be observed that the performance degradation of SCSSBs are more significant than that of thread-aware birthmarks, which correctly reflect the significant impact of thread interleaving on traditional SCSSB and the fact that thread-aware birthmarks are needed for multithreaded programs.

4.6. Alternative Approach

So far, we have discussed and evaluated TreSB extracted from a sequence composed of thread-related system calls. Besides TreSB, there are other ways to extract birthmarks based on thread-related system calls. In this section, we discuss an alternative birthmark called TreCxtB (short for thread-related system call with context birthmark) that considers the context of individual thread-related system calls.

4.6.1. Thread-related System Call with Context Birthmark

Figure 11 depicts the context of a thread-related system call in a trace, which includes the system calls immediately before and after it. Formally, given an execution trace $trace(p, I) = \langle s_1, s_2, \cdots, s_n \rangle$ consisted of system calls recorded during the runtime of program p under input I, a subsequence $tos(p, I) = \langle e_1, e_2, \cdots, e_m \rangle$ consisted of just thread-related system calls can be obtained. For each thread-related system call e_i in $tos(p, I)$, we define its k-prefix as the nearest k system calls executed preceding e_i in $trace(p, I)$, and its k-suffix as the nearest k system calls executed succeeding e_i in $trace(p, I)$. The context of e_i is then defined as the sequence that concatenates the k-prefix, e_i, and the k-suffix. Similarly to TreSB, TreCxtB is defined as the key-value pair set consists of all unique context and their corresponding frequencies. Algorithm 1 gives the pseudo-code on TreCxtB generation.

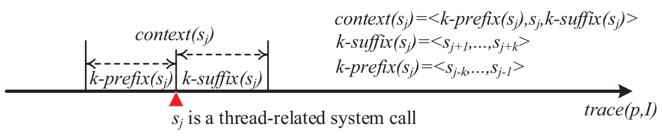
context(s_j)=<k-prefix(s_j),s_j,k-suffix(s_j)>
k-suffix(s_j)=<s_{j+1},...,s_{j+k}>
k-prefix(s_j)=<s_{j-k},...,s_{j-1}>
context(s_j)
k-prefix(s_j)
k-suffix(s_j)
s_j is a thread-related system call
trace(p,I)

Figure 11: Context of a thread-related system call

Algorithm 1 Extracting TreCxtB

```
Input:
    trace: an execution trace consisted of system calls
    k: the scope for determining prefix and suffix
Output:
    $cxt_{\mathcal{B}}$: the birthmark TreCxtB, which is a key-value pair set
 1: $cxt_{\mathcal{B}} \leftarrow \langle\rangle$
 2: for each system call $s$ in trace do
 3:     if $s$ is a thread-related system call then
 4:         $k-pre = prefix(s, k)$
 5:         $k-suf = suffix(s, k)$    ▷ Obtain the nearest $k$ system calls preceding and succeeding $s$
 6:         $cxt = concat(k-pre, s, k-suf)$    ▷ Generate the context for $s$
 7:         if $cxt_{\mathcal{B}}.keyset.contains(cxt)$ then
 8:             $++cxt_{\mathcal{B}}.getkey(cxt)$    ▷ Update the value of key $cxt$ in $cxt_{\mathcal{B}}$
 9:         else
10:             $cxt_{\mathcal{B}} \leftarrow cxt_{\mathcal{B}} \oplus \langle cxt, 1\rangle$    ▷ Add the new key-value pair to $cxt_{\mathcal{B}}$
11:         end if
12:     end if
13: end for
```

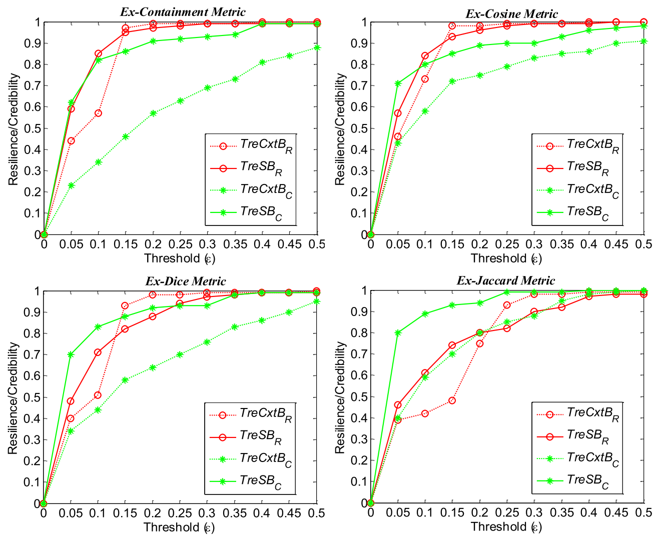

Figure 12: Comparing the resilience and credibility of TreCxtB and TreSB. We use $TreSB_R$ and $TreSB_C$ to denote the R and C of TreSB, and use $TreCxtB_R$ and $TreCxtB_C$ to denote the R and C of TreCxtB.

4.6.2. *Performance Evaluation of TreCxtB*

Same as TreSB, we evaluate the performance of TreCxtB with respect to the three performance metrics `URC`, `F-Measure` and `MCC`, respectively. Note that TreCxtB depends on a factor k that determines the range of its contexts. Table 7 summarizes the `AUC` values as well as the Average/Maximum *PerGain* values of TreCxtB with different k values. As it shows, TreCxtB exhibits the best performance (the largest average and maximum *PerGain* values with respect to all metrics among the tested k values) when $k = 1$. With increasing k values, both the average and maximum *PerGain* values with respect to either performance metric decrease. There is no need to test more k values, as larger k makes two TreCxtB more dissimilar, which decreases resilience while increases credibility of TreCxtB. In other words, larger k leads to more false negatives but less false positives. With birthmarking being a detecting technique of suspected copies, false negative is more critical than false positive(Tian et al., 2015). Besides, larger k incurs more computational cost. Thus $k = 1$ is the best choice for TreCxtB.

As indicated by `AUC` and *PerGain* values in Table 7, when $k = 1$ TreCxtB is significantly better than SCSSB and outperforms SCSSB$_{SA}$ and SCSSB$_{SS}$. But it is no better than TreSB. We also consider R and C as defined in equation 3. Figure 12 illustrates the resilience (reflected by R) and credibility (reflected by C) of TreCxtB and TreSB. From the figures, we can see that the R curve of TreCxtB is below that of TreSB, while the C curve of TreCxtB is occasionally above that of TreSB. It indicates that TreSB is no better than TreCxtB in terms of the resilience against semantics-preserving obfuscations. The main shortcoming of TreCxtB lies in its credibility of distinguishing programs when there is no plagiarism.

5. Related Work

In this section we discuss related work on birthmark based software plagiarism detection. Since we target binaries, previous researches assuming availability of source code are not discussed here.

Static birthmark based plagiarism detection: Myles et al. (Myles and Collberg, 2005) proposed k-gram based static birthmarks, where sets of Java bytecode sequences of length k are taken as the birthmarks. Although being more robust than birthmarks proposed by Tamada (Tamada et al., 2004a), the birthmarks were still vulnerable to code obfuscation attacks. A static birthmark based on disassembled API calls from executables is put forward by Seokwoo et al. (Choi et al., 2009), yet the requirement for de-obfuscating binaries before applying their method is too restrictive and thus reduces its availability. An obfuscation-resilient method based on longest common subsequence of semantically equivalent basic blocks was proposed (Luo et al., 2014). They utilized symbolic execution to extract from basic blocks symbolic formulas, whose pair-wise equivalence are compared via a theorem prover. Being static analysis method, accuracy can not be assured since it has difficulty in handling indirect branches. There are also some works focusing on detecting plagiarism for smartphone applications. DroidMOSS (Zhou et al., 2012) detects plagiarism by applying fuzzing hashing on instruction sequences. Yet simple obfuscations such as noise injection can invalidate the method. ViewDroid (Zhang et al., 2014a) proposes the feature view graph birthmark by capturing users' navigation behaviors. But it's vulnerable to dummy view insertion and encryption attacks.

Dynamic birthmark based plagiarism detection: Myles et al. (Myles and Collberg, 2004) suggested the whole program path (WPP) birthmark generated by compressing a whole dynamic control flow trace into a directed acyclic graph to uniquely identify a program. Even with compression the method does not scale, and it's susceptible to various loop transformations. Schuler (Schuler et al., 2007) treated Java API call sequences at object level as dynamic birthmarks for Java programs. Such approach gave better performance than WPP birthmark, but they also pointed out that their method was affected by thread scheduling. Wang et al. (Wang et al., 2009b) proposed System Call Short Sequence birthmark (SCSSB), which treated the sets of k-length system call sequences as birthmarks. As illustrated in this paper, although dynamic birthmarks exhibit certain resilience to syntax modifications, they are not suitable for plagiarism detection of multithreaded programs.

Thread aware birthmarks: There has been very few work that target birthmark based plagiarism detection of multithreaded programs. To the best of our knowledge, SCSSB$_{SA}$ and SCSSB$_{SS}$ proposed by Tian et al. (Tian et al., 2014b) are the only two birthmarks that consider the impact of thread scheduling. Their principle was to extract birthmarks based on the events in individual threads. Yet the assumption that events happened in each thread is stable is not always true, as events happened in each thread are variable due to the interactions between threads. In addition, such isolated approach cannot catch the overall program behavior, especially thread in-

Table 7: `AUC` values of TreCxtB

	k=1			k=2			k=3		
	URC	F-Measure	MCC	URC	F-Measure	MCC	URC	F-Measure	MCC
Ex-Containment	0.674	0.903	0.749	0.718	0.879	0.753	0.697	0.843	0.735
Ex-Cosine	0.782	0.924	0.81	0.775	0.897	0.799	0.724	0.858	0.772
Ex-Dice	0.726	0.903	0.773	0.756	0.879	0.774	0.706	0.845	0.752
Ex-Jaccard	0.748	0.883	0.767	0.667	0.826	0.724	0.551	0.766	0.675
PerGain (%)	27\82	13\28	11\27	25\62	9\20	9\20	13\34	4\11	5\12

teractions that are a crucial component of multithreaded programs. Therefore, although the method is able to alleviate the impact of non-deterministic thread scheduling to some extent, false negatives are not uncommon, especially when the thread interplay is complex. In addition, their method of calculating maximum similarity score through bipartite graph matching may lead to false positives, since scores calculated between independent programs tends to be higher. As verified by our experiments, our approach based on thread related system calls is superior in terms of both accuracy and efficiency.

6. Conclusion and Future Work

As multithreaded programming becomes increasingly more popular, existing dynamic software plagiarism detection techniques geared towards sequential programs are no longer sufficient. This work fills the gap by proposing a new thread-aware birthmarking technique. The primary contributions of this paper include the following:

- We proposed a new kind of thread-aware birthmark called TreSB, which works efficiently for plagiarism detection of multithreaded programs. There has been very few work dealing with the impact of thread scheduling on plagiarism detection.
- We implemented a tool and evaluated its effectiveness. The experiments on 234 versions of 35 programs show that our approach is not only accurate in detecting plagiarism of multithreaded programs, but also resilient to most state-of-the-art semantics-preserving obfuscation techniques.
- We compared TreSB against two latest and currently only existing thread-aware birthmarks SCSSB$_{SA}$ and SCSSB$_{SS}$. The comparison results with respect to three metrics including `URC`, `F-Measure`, and `MCC` indicate our approach is superior.
- We suggested an alternative birthmark generating approach called TreCxtB that also exploits thread-related system calls. The experiments show that TreCxtB outperforms SCSSB$_{SA}$ and SCSSB$_{SS}$.

In this paper, TreSB is mainly evaluated on the detection of whole program plagiarism, where a complete program is copied and disguised through code obfuscation techniques. In recent years, whole program plagiarism of mobile apps has become a serious problem. About 5 to 13 percent of apps in the third-party app markets are copied and redistributed from the official Android market. We plan to conduct case studies for this domain. On the other hand, while whole program plagiarism detection is very useful in practice, there are also many cases that only part of a program is copied, such as the web browser experiment where the Webkit layout engine is utilized in multiple browsers. We will explore whether our approach can be adapted to detect partial plagiarisms.

To our best knowledge there do no exist obfuscation techniques that particularly target multithreaded programs. However, it is inevitable that such obfuscations will surface once plagiarism detection techniques as TreSB are being used. In the future we plan to investigate obfuscation techniques that can potentially defeat thread-aware birthmarks. Based on our investigation we will continually optimize TreSB and design other thread-aware birthmarks to defend against these obfuscations.

Acknowledgement

This work was supported by the National Natural Science Foundation of China (91218301, 91418205, 61472318, 61428206), Key Project of the National Research Program of China (2013BAK09B01), Ministry of Education Innovation Research Team (IRT13035), the Fundamental Research Funds for the Central Universities, and the National Science Foundation (NSF) under grant CCF-1500365. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

Chae, D.-K., Ha, J., Kim, S.-W., Kang, B., Im, E. G., 2013. Software plagiarism detection: a graph-based approach. In: Proceedings of the 22nd ACM International Conference on Information & Knowledge Management (CIKM '13). ACM, pp. 1577–1580.

Chae, D.-K., Kim, S.-W., Cho, S.-J., Kim, Y., 2015. Effective and efficient detection of software theft via dynamic api authority vectors. Journal of Systems and Software 110, 1–9.

Chan, P. P., Hui, L. C., Yiu, S.-M., 2013. Heap graph based software theft detection. Information Forensics and Security, IEEE Transactions on 8 (1), 101–110.

Choi, S., Park, H., Lim, H.-i., Han, T., 2009. A static api birthmark for windows binary executables. Journal of Systems and Software 82 (5), 862–873.

Collberg, C., Myles, G., Huntwork, A., 2003. Sandmark-a tool for software protection research. IEEE Security & Privacy 1 (4), 40–49.

Cosma, G., Joy, M., 2012. An approach to source-code plagiarism detection and investigation using latent semantic analysis. Computers, IEEE Transactions on 61 (3), 379–394.

Cui, H., Wu, J., Gallagher, J., Guo, H., Yang, J., 2011. Efficient deterministic multithreading through schedule relaxation. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11). ACM, pp. 337–351.

Guo, F., Ferrie, P., Chiueh, T.-C., 2008. A study of the packer problem and its solutions. In: the 11th International Symposium on Recent Advances in Intrusion Detection (RAID '08). Springer, pp. 98–115.

Jhi, Y.-C., Jia, X., Wang, X., Zhu, S., Liu, P., Wu, D., 2015. Program characterization using runtime values and its application to software plagiarism detection. Software Engineering, IEEE Transactions on 41 (9), 925–943.

Jhi, Y.-C., Wang, X., Jia, X., Zhu, S., Liu, P., Wu, D., 2011. Value-based program characterization and its application to software plagiarism detection. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). pp. 756–765.

Jiang, L., Misherghi, G., Su, Z., Glondu, S., 2007. Deckard: Scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th International Conference on Software Engineering (ICSE '07). IEEE Computer Society, pp. 96–105.

Kim, M.-J., Lee, J.-Y., Chang, H.-Y., Cho, S., Wilsey, P. A., 2010. Design and Performance Evaluation of Binary Code Packing for Protecting Embedded Software against Reverse Engineering. In: the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC '10). pp. 80–86.

Lim, H.-i., Park, H., Choi, S., Han, T., 2009. A method for detecting the theft of java programs through analysis of the control flow information. Information and Software Technology 51 (9), 1338–1350.

Linn, C., Debray, S., 2003. Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03). ACM, pp. 290–299.

Liu, C., Chen, C., Han, J., Yu, P. S., 2006. GPLAG: Detection of software plagiarism by program dependence graph analysis. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '06). ACM, New York, NY, USA, pp. 872–881.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., Hazelwood, K., 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05). ACM, New York, NY, USA, pp. 190–200.

Luo, L., Ming, J., Wu, D., Liu, P., Zhu, S., 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14). ACM, pp. 389–400.

Madou, M., Van Put, L., De Bosschere, K., 2006. Loco: An interactive code (de) obfuscation tool. In: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '06). ACM, pp. 140–144.

Matthews, B. W., 1975. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. Biochimica et Biophysica Acta (BBA)-Protein Structure 405 (2), 442–451.

Ming, J., Zhang, F., Wu, D., Liu, P., Zhu, S., 2016. Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection (accepted). Reliability, IEEE Transactions on, 1–1.

Myles, G., Collberg, C., 2004. Detecting software theft via whole program path birthmarks. In: the 7th Information Security International Conference (ISC '04). Springer, pp. 404–415.

Myles, G., Collberg, C., 2005. K-gram based software birthmarks. In: Proceedings of the 2005 ACM Symposium on Applied Computing (SAC '05). ACM, New York, NY, USA, pp. 314–318.

Olszewski, M., Ansel, J., Amarasinghe, S., 2009. Kendo: efficient deterministic multithreading in software. ACM Sigplan Notices 44 (3), 97–108.

Park, H., Lim, H.-i., Choi, S., Han, T., 2011. Detecting common modules in java packages based on static object trace birthmark. The Computer Journal 54 (1), 108–124.

Patki, T., 2008. Dasho java obfuscator, http://www.cs.arizona.edu/ collberg/teaching/620/2008/assignments/tools/dasho/index.html.

Prechelt, L., Malpohl, G., Philippsen, M., 2002. Finding plagiarisms among a set of programs with JPlag. Journal of Universal Computer Science 8 (11), 1016–1038.

Roundy, K. A., Miller, B. P., 2013. Binary-code obfuscations in prevalent packer tools. ACM Computing Surveys 46 (1), 4.

Schuler, D., Dallmeier, V., Lindig, C., 2007. A dynamic birthmark for java. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07). ACM, pp. 274–283.

Tamada, H., Nakamura, M., Monden, A., Matsumoto, K.-i., 2004a. Design and evaluation of birthmarks for detecting theft of java programs. In: IASTED Conference on Software Engineering (IASTED '04). pp. 569–574.

Tamada, H., Okamoto, K., Nakamura, M., Monden, A., Matsumoto, K.-i., 2004b. Dynamic software birthmarks to detect the theft of windows applications. In: International Symposium on Future Software Technology (ISFST '04). Vol. 20.

Tian, Z., Liu, T., Zheng, Q., Tong, F., Fan, M., Yang, Z., 2016. A new thread-aware birthmark for plagiarism detection of multithreaded programs (accecpted). In: Proceedings of the 38th International Conference on Software Engineering (ICSE '16 Companion). pp. 1–1.

Tian, Z., Zheng, Q., Fan, M., Zhuang, E., Wang, H., Liu, T., 2014a. DBPD: A dynamic birthmark-based software plagiarism detection tool. In: the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE '14). pp. 740–741.

Tian, Z., Zheng, Q., Liu, T., Fan, M., 2013. DKISB: Dynamic key instruction sequence birthmark for software plagiarism detection. In: 2013 IEEE International Conference on High Performance Computing and Communications (HPCC '13). IEEE, pp. 619–627.

Tian, Z., Zheng, Q., Liu, T., Fan, M., Zhang, X., Yang, Z., 2014b. Plagiarism detection for multithreaded software based on thread-aware software birthmarks. In: Proceedings of the 22nd International Conference on Program Comprehension (ICPC '14). ACM, pp. 304–313.

Tian, Z., Zheng, Q., Liu, T., Fan, M., Zhuang, E., Yang, Z., 2015. Software plagiarism detection with birthmarks based on dynamic key instruction sequences. Software Engineering, IEEE Transactions on 41 (12), 1217–1235.

Wang, X., Jhi, Y.-C., Zhu, S., Liu, P., 2009a. Behavior based software theft detection. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09). ACM, pp. 280–290.

Wang, X., Jhi, Y.-C., Zhu, S., Liu, P., 2009b. Detecting software theft via system call based birthmarks. In: Annual Computer Security Applications Conference (ACSAC '09). IEEE, pp. 149–158.

Wu, Z., Gianvecchio, S., Xie, M., Wang, H., 2010. Mimimorphism: A new approach to binary code obfuscation. In: Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10). ACM, pp. 536–546.

Xie, X., Liu, F., Lu, B., Chen, L., 2010. A software birthmark based on weighted k-gram. In: IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS '10). Vol. 1. IEEE, pp. 400–405.

Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P., 2014a. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In: Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '14). Citeseer, pp. 25–36.

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

Zhang, F., Wu, D., Liu, P., Zhu, S., 2014b. Program logic based software plagiarism detection. In: IEEE 25th International Symposium on Software Reliability Engineering (ISSRE '14). IEEE, pp. 66–77.

Zhou, W., Zhou, Y., Jiang, X., Ning, P., 2012. Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the 2rd ACM conference on Data and Application Security and Privacy (CODASPY '12). ACM, pp. 317–326.