Reviving Sequential Program Birthmarking for Multithreaded Software Plagiarism Detection

Zhenzhou Tian, Ting Liu, *Member, IEEE,* Qinghua Zheng, *Member, IEEE,* Eryue Zhuang, Ming Fan and Zijiang Yang, *Senior Member, IEEE*

Abstract—As multithreaded programs become increasingly popular, plagiarism of multithreaded programs starts to plague the software industry. Although there has been tremendous progress on software plagiarism detection technology, existing dynamic birthmark approaches are applicable only to sequential programs, due to the fact that thread scheduling nondeterminism severely perturbs birthmark generation and comparison. We propose a framework called TOB (Thread-oblivious dynamic Birthmark) that revives existing techniques so they can be applied to detect plagiarism of multithreaded programs. This is achieved by thread-oblivious algorithms that shield the influence of thread schedules on executions. We have implemented a set of tools collectively called TOB-PD (TOB based Plagiarism Detection tool) by applying TOB to three existing representative dynamic birthmarks, including SCSSB (System Call Short Sequence Birthmark), DYKIS (DYnamic Key Instruction Sequence birthmark) and JB (an API based birthmark for Java). Our experiments conducted on large number of binary programs show that our approach exhibits strong resilience against state-of-the-art semantics-preserving code obfuscation techniques. Comparisons against the three existing tools SCSSB, DYKIS and JB show that the new framework is effective for plagiarism detection of multithreaded programs. The tools, the benchmarks and the experimental results are all publicly available.

Index Terms—software plagiarism detection, multithreaded program, software birthmark, thread-oblivious birthmark

1 Introduction

Software plagiarism, an act of illegally copying others' code, severely affect both open source communities and honest software companies. The recent incidents include the lawsuit against Verizon by Free Software Foundation for distributing Busybox in its FIOS wireless routers [1], and the crisis of Skype's VOIP service for the violation of licensing terms of Joltid [2]. Unfortunately software plagiarism is easy to implement but difficult to detect. A study in 2012 [3] shows that about 5%-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 [4]–[6] are a few reasons that make software plagiarism detection a daunting task. Nevertheless, researchers welcomed this challenge and developed effective methods, of which software birthmarking [7], [8] is a popular and recently well studied technique. A birthmark is a set of characteristics extracted from a program that can be used to uniquely identify the program. Depending on whether its extraction relies on program runs, a software birthmark can be either considered static or dynamic. Static birthmarks, generated mainly by analyzing syntactic features, are usually ineffective against semantics-preserving obfuscations that can modify the syntactic structure of a program. Besides, static birthmarks are easily defeated by packing techniques [9], [10] which can make processed executables rather different in the static level. In contrast, dynamic birthmarks are extracted based on runtime behaviors and thus are believed to be more accurate reflections of program semantics and more robust against obfuscations [11]–[15]. Thus in this paper we mainly focus on dynamic software birthmark methods.

Despite the tremendous progress in software plagiarism detection technology, a new trend in software development greatly threatens its effectiveness. The trend towards multithreaded programs is creating a gap between the current software development practice and the software plagiarism detection technology, as the existing dynamic approaches remain optimized for sequential programs. Due to the perturbation caused by non-deterministic thread scheduling, existing birthmark generation and comparison are no longer applicable to modern software with multiple

- *Z. Tian is with the Ministry of Education Key Lab For Intelligent Networks and Network Security (MOEKLINNS), Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China, and with the School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an 710121, China. Email: tianzhenzhou@xupt.edu.cn.*
- *T. Liu and Q. Zheng are with the Ministry of Education Key Lab For Intelligent Networks and Network Security (MOEKLINNS), School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China. Email: {tingliu, qhzheng}@mail.xjtu.edu.cn.*
- *E. Zhuang and M. Fan are with the Ministry of Education Key Lab For Intelligent Networks and Network Security (MOEKLINNS), Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China. Email: {zhuangeryue, fanming.911025}@stu.xjtu.edu.cn.*
- *Z. Yang is with the Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008, USA, and with the Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China. Email: zijiang.yang@wmich.edu.*

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#define N 8
pthread_t mThread[N];
sem_t sem_a;
int tid;
void *run(void *data){
    sem_wait(&sem_a);
    tid = (int)data;
    printf("hello world from thread %d\n", tid);
    sem_post(&sem_a);
    if(((int)data)%2==0){
       usleep(0);}
    char str[25];
    sprintf(str,"%04X",(int)data);
    return NULL;}
int main(int argc, char *argv[]){
    int rc, i, j;
    sem_init(&sem_a, 0, 1);
    printf("input a number please: \n");
    scanf("%d", &i); j = i;
    for (i; i < N; i++){
      rc = pthread_create(&mThread[i], NULL, run, (void *)i);
      if (rc)
         printf("create thread failed. error code = %d\n", rc);}
    for (i = j; i < N; i++)
      pthread_join(mThread[i], NULL);
    printf("main thread finished\n");
    return 0;}
```

Fig. 1. A simple multithreaded program

TABLE 1
Similarity scores calculated between birthmarks of multiple runs of the sample program in Figure 1. The column headings indicate the number of threads.

(a) SCSSB

SCSSB	**1**	**2**	**4**	**6**	**8**
Cosine Distance	1.000	0.898	0.670	0.569	0.469
Jaccard Index	1.000	0.76	0.439	0.363	0.265
Dice Coefficient	1.000	0.864	0.609	0.532	0.403
Containment	1.000	0.864	0.619	0.57	0.412

(b) DYKIS

DYKIS	**1**	**2**	**4**	**6**	**8**
Cosine Distance	1.000	1.000	0.968	0.708	0.471
Jaccard Index	1.000	1.000	0.874	0.437	0.311
Dice Coefficient	1.000	1.000	0.926	0.6	0.459
Containment	1.000	1.000	0.984	0.6	0.482

threads.

The problems with existing dynamic birthmarks can be illustrated by a multithreaded program shown in Figure 1. The program is a test case in the WET project [16] with slight modifications. We apply two typical plagiarism detection algorithms on multiple runs of this program. One is called System Call Short Sequence Birthmark (SCSSB) [15] and the other is a more recent algorithm called DYKIS [17]. The details of both algorithms will be presented in Section 2. If the existing approaches fail to claim plagiarism on different runs of the same small program, even without any code modifications, the capability of such approaches is in doubt.

Dynamic birthmarks usually give quantitative measurement between 0 and 1 to indicate the similarity between two runs. A value of 1 indicates identicalness and 0 refers to complete difference. The measurement is given by applying metrics, such as Cosine distance, Jaccard index, Dice coefficient and Containment [13], [15], [18], [19], on the birthmarks obtained by a pair of executions. Tables 1(a) and 1(b) show the experimental results of SCSSB and DYKIS, where the column and row headings indicate the number of threads and the evaluation metrics, respectively. When there is only one thread, the program becomes sequential. Without non-deterministic thread scheduling, the executions are deterministic. As expected, the similarity scores are all 1.0, pointing out correctly that the runs are from identical programs. However, as the number of threads increase, the similarity scores quickly deteriorate. As explained in Section 2, in birthmark-based techniques a threshold ε is used to indicate plagiarism. A similarity score greater than $1 - \varepsilon$ indicates strong possibility of plagiarism, while a score less than ε indicates the opposite. Considering typical value of ε is between 0.15 and 0.35, SCSSB and DYKIS will not claim plagiarism when the number of threads is 6, and start to claim the runs are from different programs when the number of threads becomes 8.

The example illustrates that the existing dynamic birthmark based approaches are inadequate in detecting plagiarism of multithreaded programs because they neglect the effect of thread scheduling. Sequential program behavior is deterministically determined by system inputs, including I/O, DMA, interrupts, thus executions of highly similar programs under the same input are similar. This assumption no longer holds for multithreaded programs since thread schedules are a major source of non-determinism. For a program with n threads, each executing k steps, there can be as many as $(nk)!/(k!)^n > (n!)^k$ different thread interleavings, a doubly exponential growth in terms of n and k. This indicates that two executions under the same inputs can be very different, which invalidates the basic assumption of existing approaches.

In this paper, we present TOB (Thread-Oblivious dynamic Birthmark), a framework that can revive existing dynamic birthmarks such as SCSSB [15], DYKIS [17], JB [18] to handle multithreaded programs. Unlike many prior approaches [20], [21], TOB operates on binary executables rather than source code that is usually unavailable when birthmark techniques are used to obtain initial evidence of software plagiarism.

This paper extends our preliminary conference paper [22], and makes the following contributions:

- To the best of our knowledge, this is the first work

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2688383, IEEE Transactions on Software Engineering

that discusses the impact of thread scheduling on birthmark based software plagiarism detection, and proposes a solution to remedy the problem.

- We apply the var-gram [23] algorithm in birthmark generation. As far as we know, this is the first time this algorithm is used for such purpose. Our experiments confirm its effectiveness.
- We have implemented a set of tools collectively called `TOB-PD` (TOB based Plagiarism Detection tool) by integrating the principle of TOB with existing algorithms, including SCSSB [15], DYKIS [17] and JB [18]. The tools as well as the source codes are publicly available at website: `http://labs.xjtudlc.com/labs/wlaq/TAB-PD/site`.
- Our experiments on 418 versions of 35 different multithreaded programs show that the new tools are highly effective in detecting plagiarism and are resilient to most state-of-the-art semantics-preserving obfuscation techniques implemented in tools such as `SandMark` [4], `DashO` [24] and `UPX`. All benchmarks and the experimental data can also be downloaded from our website.

The remainder of the paper is organized as follows. Necessary background on software birthmarks are described in Section 2. In Section 3 we present the TOB framework that revives existing birthmarks. In Section 4 the approaches for comparing the TOB-revived birthmarks are discussed. The system design and implementation details are described in Section 5. Section 6 presents the empirical study, including the evaluation on resilience and credibility of the thread-oblivious birthmarks, the comparison with traditional SCSSB and the integration of TOB with DYKIS and JB. In Section 7 we discuss threats to the validity of our approaches. Section 8 reviews related work and finally we conclude the paper in Section 9.

2 Preliminaries

2.1 Birthmark based Plagiarism Detection

A software birthmark, whose classical definitions are as depicted in Definition 1 and Definition 2, is a set of characteristics extracted from a program statically or dynamically, that reflects intrinsic properties of the program and that can be used to identify the program uniquely.

Definition 1: **Software Birthmark [8].** 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 $p \Rightarrow f(p) = f(q)$.

Definition 2: **Dynamic Software Birthmark [25].** Let p be a program and I be an input to p. Let $f(p, I)$ be a set of characteristics extracted from p when executing p with 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 two conceptual descriptions, various implementable birthmarks have been developed by mining characteristics from different aspects of the program, of which the SCSSB [15] extracted from system calls, the DYKIS [13] extracted from executed instructions, the JB [18] extracted from executed Java APIs are several representative dynamic birthmarks.

The two conceptual definitions of software birthmarks require that if two programs are in copy relation, their birthmarks should be the exactly the same. But due to many practical issues in implementing specific birthmarks, even if q is a copy of p, their corresponding birthmarks are not identical. Thus in the literature of software birthmarking, the plagiarism of two programs is decided by a threshold ε and a function sim that computes the similarity score between their birthmarks. The range of a similarity score is between 0 and 1. Although a value of 0.25 was typically used as the threshold in previous studies, other values were used as well and we found that the choice was quite arbitrary. Thus in our work, we do not set ε to a particular value. Instead we analyze its impact on the performance under a wide range of values. Let p and $p_{\mathcal{B}}$ be the plaintiff program and its birthmark, and q and $q_{\mathcal{B}}$ be the defendant program and its birthmark. The plagiarism is decided with Equation 1, which gives a conceptual definition of sim that returns a three-value result: positive, negative or inconclusive.

$$sim\left(p_{\mathcal{B}}, q_{\mathcal{B}}\right) = \begin{cases} > 1-\varepsilon \quad positive:\ q\ is\ a\ copy\ of\ p \\ < \varepsilon \quad negative:\ q\ is\ not\ a\ copy\ of\ p \\ otherwise \quad inconclusive \end{cases} \tag{1}$$

A high quality birthmark manifests in that the ratio of incorrect classifications should be low enough for a certain ε. However, false negative is more intolerable than false positive, since birthmarking technique is not a proving techniques but rather a detecting technique of suspected copies [14], [15], [17], [26]. Generally in the literature, the following two properties of a birthmark should be satisfied to make it valid. We refer to the definitions [17] restated from the original descriptions of Myles [27] and Choi [19].

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 $sim\left(p_{\mathcal{B}}, q_{\mathcal{B}}\right) > 1 - \varepsilon$.

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 $sim\left(p_{\mathcal{B}}, q_{\mathcal{B}}\right) < \varepsilon$.

2.2 Three Representative Dynamic Birthmarks

2.2.1 SCSSB

SCSSB (System Call Short Sequence Birthmark) [15] is a dynamic birthmark extracted from executed system calls that are believed to be a fundamental runtime indicator of program behavior. Modification of system calls usually leads to incorrect programs, and therefore, a birthmark generated from sequence of system calls can be used to uniquely identify a program even after it has been modified. In SCSSB, the sets of k-length system call sequence is treated as the birthmarks. The birthmark scales well, and their experimental results show resilience against either evasion techniques enabled by different compilers or powerful obfuscations provided in Sandmark [4]. However, as non-deterministic thread scheduling perturbs the order of system calls, SCSSB becomes ineffective in handling multithreaded programs. This is confirmed by our empirical study in Section 6.

2.2.2 DYKIS

DYKIS (DYnamic Key Instruction Sequence birthmark) [17] is extracted from executed instructions of a binary executable. Rather than taking the whole instructions for birthmark generation, the authors propose the concept of key instructions. Specifically, they treat an instruction as key instruction if it is both value-updating and input-correlated, where value-updating refers to that the execution of an instruction will generate new values rather than transfer values, and input-correlated refers to that the execution of an instruction will introduce new taint labels (dynamic taint analysis is conducted to acquire this). DYKIS is then extracted from a key instruction sequence of length k. Operating directly on the assembly instructions, DYKIS is able to detect cross-platform plagiarism. Also, the experimental evaluation shows that it is resilient against various obfuscation techniques. However, suffering from the same issue as SCSSB, DYKIS cannot be applied for plagiarism detection of multithreaded programs.

2.2.3 JB

Different from SCSSB and DYKIS that operate on binary executables, JB [18] is a dynamic birthmark for Java programs. It observes short sequences of Java Standard API calls received by individual objects. Similar to both SCSSB and DYKIS, an API call trace is chopped into a set of short sequences to generate birthmarks. Organizing traces on a per-object basis makes JB less susceptible to thread scheduling. However, we will demonstrate that it can be further improved with our TOB framework.

3 Thread-Oblivious Birthmarks

As illustrated by the example in Figure 1, thread scheduling makes the behavior of a multithreaded program non-deterministic even under a fixed input. The classical definition of dynamic software birthmark is no longer correct because $f(p, I) \neq f(q, I)$ even if q is a copy of p. In the following we give a definition suitable for multithreaded programs.

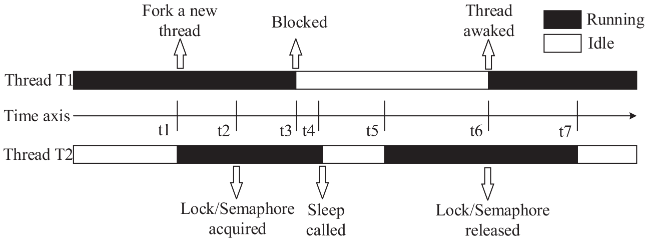


Fig. 2. A time window of executions between two threads of a multithreaded program

Definition 3: **Thread-Oblivious dynamic Birthmark.** 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 Definition 1 and Definition 2, Definition 3 provides an abstract 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 [28], [29]. Therefore instead of enforcing thread schedules in our algorithms, we try to shield the influence of thread schedules on executions. That is, to make a birthmark thread-oblivious, 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.

3.1 Intuition

Figure 2 depicts a typical execution snippet between two possible threads of a multithreaded program. Despite there can be many different thread interleavings, the order of the events happened in each single thread seems not affected by thread scheduling. That is, if e_1 happens before e_2 in an execution, the order of two events from the same thread will remain the same even under different thread interleavings.

Based on the observation, we project traces on individual threads, and then check whether two traces match each other after the projection. Consider the system call traces comprised of eight threads that correspond to the case as indicated by the last column in Table 1(a). As depicted in Figure 3, almost identical traces are observed (except subtle differences on the

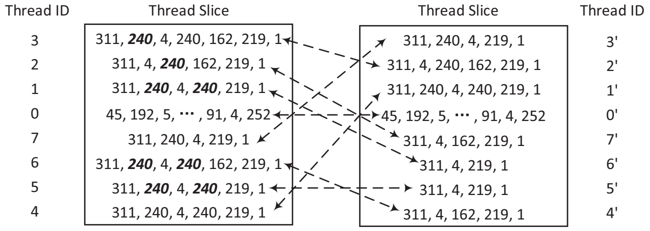


Fig. 3. Trace comparison on the thread level. System call No. is used to represent each system call instance.

system call *futex* numbered 240) after projecting the traces on individual threads. *Futex* is called only when it is likely that the program has to block for a longer time until the condition becomes true. Its occurrences show intrinsic randomness under different executions, as *futex* is essentially designed to reduce the number of system calls so as to improve performance. Thus in our implementation, as described in Section 5, we treat them as noises. After refining the traces, exact match among all the traces under the same inputs for the sample program can be observed.

To validate whether the hypothesis (that projected events within individual threads are relatively stable under different thread interleavings) also holds among real world multithreaded programs, we further investigated in detail the traces of ten multithreaded programs, including `blackscholes`, `bodytrack`, `fludanimate`, `canneal`, `dedup`, `ferret`, `freqmine`, `streamcluster`, `swaptions`, and `x264`, from the PARSEC benchmark suite [30]. These programs are typical complex multithreaded programs covering different domains, such as computer vision, video encoding, financial analytic, animation physics and image processing. They come with inputs of different sizes, and allow different number of threads to start with.

In our initial empirical study, we run each program with four threads [1] under the same inputs multiple times. After projecting the traces on individual threads, traces executed by `blackscholes`, `fludanimate`, `canneal` and `streamcluster` show perfect match. The projected traces executed by other programs show exact match under some inputs, while present subtle differences under some other inputs. After manual examination, we conclude that the subtle differences are due to the fact that a thread's behavior may be affected by other threads. In Section 5 we discuss this issue further and provide an optimization to alleviate the problem. To give a more intuitive view of the matching, we compute the average of similarity scores between the traces obtained under same inputs, with the TOB-revived SCSSBs (to be discussed in the following sections) for each program. The gray and white columns in Table 2 summarize the scores calculated without and with the trace refining. As indicated by the scores that are all near 1.0 in the white columns, our hypothesis applies to the real world multithreaded programs.

Considering that more threads usually lead to more complex thread interleaving, we vary the number of threads that each program starts with. The left figure in Figure 4 depicts how the average similarity scores change for each program as the number of threads increases. As it shows, the scores calculated with the TOB-revived birthmarks are all very high. There is also no significant variation across the x-axis. It indicates that the number of threads has negligible effect on the hypothesis.

In the third experiment, we validate whether the hypothesis holds under different workloads. The benchmark programs all come with input of different sizes, each representing a different workload. Specifically, we use five levels of inputs, arranged in ascending order in terms of the workloads, to drive the executions. The five levels include *Test*, *Simdev*, *Simsmall*, *Simmedium* and *Simlarge*. There is a large span between different levels of inputs. For example, the size of the *Test* input for `dedup` is just 6 byte, while the size of its *Simlarge* input is 184MB. Thus, the inputs provided by the benchmarks can properly test the interleaving situation of a multithreaded program under different levels of workloads. The right figure in Figure 4 summarizes the results. As it shows, the average similarity scores are all very high regardless of the workload, and do not exhibit significant differences. This indicates that the effect of different levels of workload on the hypothesis is trivial.

Finally, the thread interleaving can be affected by scheduling policies. There are basically three scheduling policies supported by the Linux kernel, including SCHED_OTHER, SCHED_FIFO (realtime first-in-first-out) and SCHED_RR (realtime round-robin). Since all the benchmark programs adopt the default scheduling policy (SCHED_OTHER), we make slight modifications to the source code in order to support the other two scheduling policies. Table 3 summarizes the average scores with respect to different scheduling policies. As the data shows, all the scores are close to 1.0 and score variations between different scheduling policies are trivial. The results indicate that the scheduling policy has little effect on the hypothesis.

Based on these observations, we propose a solution to shield the influence of thread scheduling in birthmark generation. Figure 5 depicts our approach of reviving existing birthmarks. Given a multithreaded execution trace, TOB projects the trace on individual threads. Each thread slice is insensitive to thread scheduling so we can create a birthmark using existing techniques. Then TOB uses two models to combine individual thread slice birthmarks into a thread-oblivious birthmark for the multithreaded execution. We use the rest of this section to explain our approach

[1] According to the document of the PARSEC benchmark suite, the actual number of threads can be higher.

TABLE 2
Average of the similarity scores between the traces obtained under same inputs. For simplicity, we use SA and SS to denote the TOB-revived birthmarks. The gray and white columns summarize the scores calculated without and with trace refining, respectively.

	Cosine Distance				Jaccard Index				Dice Coefficient				Containment			
	SA		SS		SA		SS		SA		SS		SA		SS	
blackscholes	0.990	1.000	0.989	1.000	0.935	1.000	0.936	1.000	0.965	1.000	0.965	1.000	0.970	1.000	0.970	1.000
bodytrack	0.995	1.000	0.992	1.000	0.873	1.000	0.873	1.000	0.929	1.000	0.921	1.000	0.937	1.000	0.944	1.000
canneal	0.972	1.000	0.979	1.000	0.944	1.000	0.948	1.000	0.960	1.000	0.964	1.000	0.963	1.000	0.968	1.000
dedup	0.941	0.991	0.819	0.989	0.568	0.991	0.502	0.989	0.714	0.994	0.619	0.989	0.738	0.997	0.672	0.989
ferret	0.896	1.000	0.860	1.000	0.579	0.995	0.727	1.000	0.703	0.998	0.783	1.000	0.720	1.000	0.827	1.000
fludanimate	0.943	1.000	0.950	1.000	0.906	1.000	0.924	1.000	0.924	1.000	0.935	1.000	0.927	1.000	0.938	1.000
freqmine	0.963	0.997	0.973	0.996	0.766	0.932	0.785	0.950	0.857	0.962	0.872	0.971	0.864	0.965	0.882	0.975
streamcluster	0.884	0.996	0.885	0.998	0.800	0.993	0.878	0.998	0.839	0.995	0.881	0.998	0.846	0.996	0.884	0.998
swaptions	0.989	0.992	0.983	0.988	0.922	0.978	0.918	0.988	0.956	0.986	0.943	0.988	0.966	0.993	0.953	0.988
x264	0.845	0.999	0.889	0.999	0.795	0.998	0.866	0.999	0.820	0.999	0.877	0.999	0.829	0.999	0.888	0.999

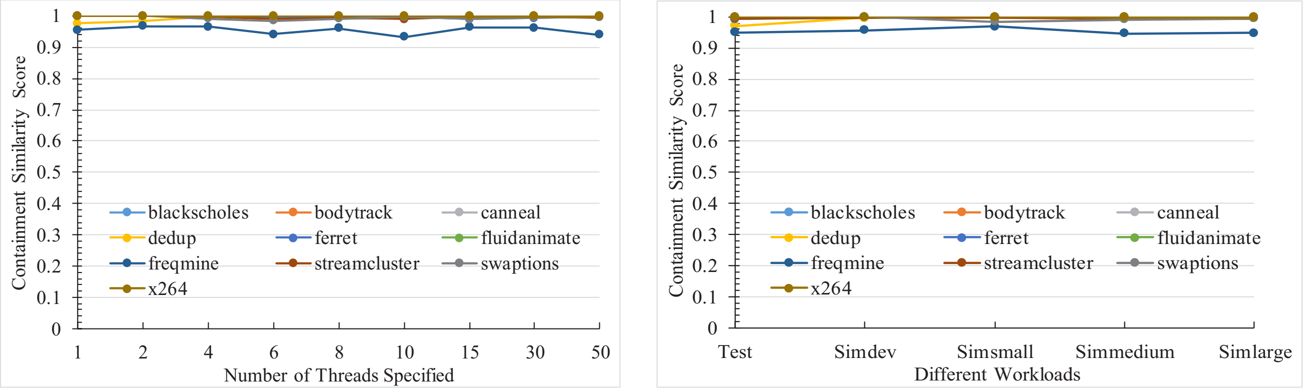


Fig. 4. Validation of the hypothesis under different number of threads and different workloads. Since similar results are observed, only Containment scores measured for SA birthmarks are given here to save space.

TABLE 3
Validation of the hypothesis under different scheduling policies. To save space, only results measured with SA birthmarks are given here.

	Cosine			Jaccard			Dice			Containment		
	OTHER	FIFO	RR	OTHER	FIFO	RR	OTHER	FIFO	RR	OTHER	FIFO	RR
blackscholes	1.000	1.000	0.999	1.000	1.000	0.999	1.000	1.000	0.999	1.000	1.000	0.999
bodytrack	1.000	1.000	1.000	0.994	0.989	0.990	0.997	0.994	0.995	0.999	0.997	0.999
canneal	1.000	0.999	0.999	0.999	0.998	0.998	0.999	0.999	0.998	0.999	0.999	0.999
dedup	0.993	0.980	0.992	0.973	0.960	0.976	0.983	0.971	0.985	0.992	0.979	0.992
ferret	0.999	0.999	0.999	0.986	0.989	0.985	0.993	0.994	0.992	0.997	0.998	0.997
fluidanimate	1.000	1.000	1.000	0.999	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
freqmine	0.996	0.995	0.996	0.912	0.902	0.894	0.951	0.945	0.940	0.954	0.948	0.944
streamcluster	0.996	0.996	0.998	0.990	0.990	0.994	0.993	0.993	0.996	0.996	0.996	0.998
swaptions	0.994	0.993	0.996	0.986	0.987	0.990	0.991	0.991	0.993	0.994	0.995	0.997
x264	1.000	1.000	0.999	0.998	0.998	0.998	0.999	0.999	0.999	1.000	0.999	0.999

in details. Note that although in this paper we focus on birthmarks in set format, birthmarks in other formats such as sequences or graphs can also utilize TOB to handle multithreaded programs. In the latter case appropriate algorithms for thread slice birthmark generations and comparisons need to be developed.

3.2 Birthmark for Individual Threads

In order to shield the influence of non-deterministic scheduling, TOB annotate each event in an execution trace with thread identifier. It then project the trace on thread identifiers to obtain sub-traces, each of which belongs to a single thread. As a result, the birthmarks extracted from the sub-traces can remain same even under different thread schedules.

Formally, let an execution trace of program p under input I be $trace(p, I) = \langle e_1, e_2, \cdots, e_n \rangle$. Each recorded event e_i is an instruction, a system call or an API, along with its thread identifier that is denoted as $e_i.tid$. We define its projection on

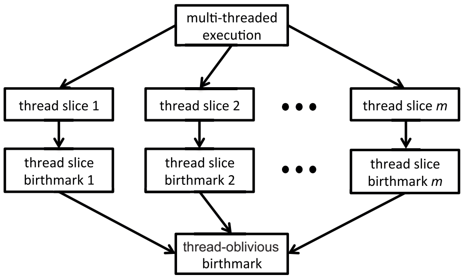


Fig. 5. TOB framework that revives traditional birthmark to thread-oblivious birthmark

thread t to be an ordered subsequence $slice\,(p, I, t) = \langle e_i | e_i \in trace\,(p, I) \wedge e_i.tid = t\rangle$. The projections of all the threads appearing in the trace form a partition of $trace(p, I)$, and we refer to each subsequence $slice(p, I, t)$ as a thread slice.

TOB further abstract each subsequence into a set of short sequences. Two methods are used to partition a sequence. One is the typical k-gram algorithm that is widely used in birthmark generation literature [13], [15], [18], [27]. which partitions a sequence with a length k windows, generating a set of fixed-length short sequences called k-grams. The other is the var-gram algorithm [23] that divides a subsequence into variable-length short sequences called var-grams. var-gram algorithm is widely used in behavior pattern mining, and is believed to be more natural description of program behavior [31], [32]. In this paper, it is applied for the first time to extract birthmarks.

Similar to the typical dynamic birthmarks such as SCSSB, DYKIS, and JB, the set of key-value pairs is taken as the birthmark for each thread slice. The keys and values consist of all unique grams and their corresponding frequencies, respectively. Definition 4 gives the definition of thread slice birthmark.

Definition 4: **Thread Slice Birthmark.** Given a thread slice $slice(p, I, t)$, we treat the key-value pair set $birth\,(p, I, t) = \{(g, freq\,(g, I, t))\}$ as the thread slice birthmark of $slice(p, I, t)$, where g is a unique gram and $freq(g, I, t)$ is its frequency in $gram(p, I, t)$.

3.3 Thread-Oblivious Birthmark Generation

With the availability of birthmark for each individual thread, TOB provides two models, Slice Aggregation (SA) and Slice Set (SS), to generate thread-oblivious birthmarks for multithreaded programs. Defined in Definition 5, SA generates birthmarks by aggregating all thread slice birthmarks into a single set of key-value pairs, where the keys are the unique k-grams or var-grams in any of the thread slice birthmark, and the values are frequencies of corresponding k-grams or var-grams. If a key is owned by multiple thread slice birthmarks, its frequency is the sum of the individual frequencies. As described in Definition 6, SS treats the thread identifiers as the keys and their corresponding thread slice birthmarks as the values. The two models produce thread-oblivious birthmarks $birth^{SA}(p, I)$ and $birth^{SS}(p, I)$ from thread slice birthmarks. We also use $birth(p, I)$ to represent a birthmark if we do not care whether it is obtained by SA or SS.

Definition 5: **Slice Aggregation Model.** The slice aggregation model is a map $f\,:\,\{birth\,(p, I, t)\,|0 \leq t < m\} \xrightarrow{f} birth^{SA}(p, I)$, where m is the number of threads and $birth^{SA}(p, I) = \{(g, freq(g, I))|g \in \bigcup\limits_{0 \leq t < m} gram(p, I, t), freg(g, I) = \sum\limits_{0 \leq t < m} (freq(g, I, t))\}$.

Definition 6: **Slice Set Model.** The slice set model is a map $f\,:\,\{birth\,(p, I, t)\,|0 \leq t < m\} \xrightarrow{f} birth^{SS}(p, I)$, where m is the number of threads and $birth^{SS}(p, I) = \{(t, birth(p, I, t))\,|0 \leq t < m\}$

4 Plagiarism Detection with TOB-Revived Birthmarks

The approaches provided by TOB framework lead to four types of thread-oblivious birthmarks, i.e., SA birthmarks and SS birthmarks extracted with k-gram or var-gram algorithms, when reviving a traditional dynamic birthmark. In this section, we describe how to detect plagiarism on top of these birthmarks.

4.1 Similarity Calculation

Different similarity calculation methods should be adopted depending on the specific format of birthmarks. State-of-the-art birthmarks mainly exist in three forms: sequences, sets and graphs. For birthmarks in sequence format, their similarity can be computed with pattern matching methods, such as calculating the longest common subsequences [33] [34]. Birthmarks in set form are usually generated by dividing sequences into short subsequence to make comparisons efficient. SCSSB, DYKIS, as well as many other birthmark methods [8], [13], [18], [27], [35], [36] utilize such principle. Then various methods widely used in the field of information retrieval are adopted for calculating the similarity between sets, including Dice coefficient [19], Jaccard index [18], and Cosine distance [13]. Computing the similarity of graphs is relatively more complex. It is conducted by either graph monomorphism [11] or isomorphism algorithms [20], or by translating a graph into a vector using algorithms such as random walk with restart [37], or possibly by mapping graphs to values, with techniques such as the Centroid [38], [39] and structure measurement [40]. Unfortunately these existing methods cannot be directly applied for comparing thread-oblivious birthmarks.

4.1.1 *Similarity Calculation for SA Birthmarks*

According to its definition, $birth^{SA}(p, I)$ is in the format of key-value pair set, therefore similarity computation methods such as Cosine distance, Jaccard index, Dice coefficient and Containment can be used[2]. However, two different thread-oblivious birthmarks may be considered identical because these metrics do not consider frequency of the elements. In order to address this issue we add a factor θ to each of the traditional metric to take into consideration the frequencies of the keys in a birthmark.

Given a SA birthmark $A = \{(k_1, v_1), (k_2, v_2), \cdots, (k_n, v_n)\}$, let $kSet(A)$ be the set of keys in A. That is, $kSet(A) = \{k_1, k_2, \cdots, k_n\}$. Given a second SA birthmark $B = \{(k'_1, v'_1), (k'_2, v'_2), \cdots, (k'_m, v'_m)\}$, let $U = kSet(A) \cup kSet(B)$. We convert set U to vector $\overrightarrow{U} = \left\langle k''_1, k''_2, \cdots, k''_{|U|} \right\rangle$ by assigning an arbitrary order to the elements in U. Let vector $\overrightarrow{A} = \left\langle a_1, a_2, \cdots, a_{|u|} \right\rangle$, where

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

Likewise we define $\overrightarrow{B} = \left\langle b_1, b_2, \cdots, b_{|U|} \right\rangle$. And we calculate θ as:

$$\theta = \frac{min\left(\left|\overrightarrow{A}\right|, \left|\overrightarrow{B}\right|\right)}{max\left(\left|\overrightarrow{A}\right|, \left|\overrightarrow{B}\right|\right)}$$

where $\left|\overrightarrow{A}\right| = \sqrt{\sum\limits_{a_i \in \overrightarrow{A}} a_i^2}$, and $\left|\overrightarrow{B}\right| = \sqrt{\sum\limits_{b_i \in \overrightarrow{B}} b_i^2}$.

Thus the modified metrics are defined as following:

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

The similarity of two SA birthmarks, $sim_c(A, B)$, can be calculated with *Ex-Cosine*(A, B), *Ex-Jaccard*(A, B), *Ex-Dice*(A, B) or *Ex-Containment*(A, B), by specifying c to either of the four metrics.

4.1.2 Similarity Calculation for SS Birthmarks

The SS birthmarks are also in the format of key-value pair set. But unlike SA birthmarks where the keys are grams and values are corresponding frequencies, each key-value pair in SS birthmarks consists of a thread identifier and its thread slice birthmark. Although the keys, i.e. thread identifiers, have clear physical meaning, such definition actually makes it difficult to compare SS birthmarks. This is because a thread identifier is just a label that is assigned by another forking thread during dynamic execution. This means thread identifies can vary across different runs of same or difference programs.

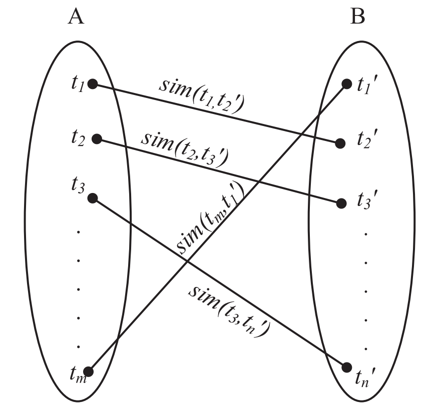


Fig. 6. Schematic diagram of bipartite matching modelling for birthmarks generated by the SS model

Since the main purpose of software birthmarking is to detecting rather than proving plagiarism, false negative is more serious than false positive. Considering that further investigation can be conducted once plagiarism is detected, we calculate the maximal similarity between two SS birthmarks. To achieve this, we reduce the problem of calculating similarity between SS birthmarks into finding a maximum weighted bipartite matching as illustrated in Figure 6. In particular, each node marked by a thread identifier corresponds to a thread slice birthmark of the thread, and a weighted edge denotes the similarity between two thread slice birthmarks.

Formally, let $A = \{(t_1, birth\,(p, I, t_1)), \cdots, (t_m, birth\,(p, I, t_m))\}$ and $B = \{(t'_1, birth\,(p', I, t'_1)), \cdots, (t'_n, birth\,(p', I, t'_n))\}$ be two SS birthmarks. A $m \times n$ similarity matrix is generated by comparing between every pair of thread slice birthmarks in A and B using either of the four metrics defined in Section 4.1.1. Note that the definition can be applied to thread slice birthmarks directly. For example, when Ex-Jaccard is chosen, $sim_c(t_1, t'_1)$ =Ex-Jaccard$(birth(p, I, t_1), birth(p', I, t'_1))$.

$$simMatr\,(A,B) = \begin{pmatrix} sim_c\,(t_1,t'_1) & \ldots & sim_c\,(t_1,t'_n) \\ sim_c\,(t_2,t'_1) & \ldots & sim_c\,(t_2,t'_n) \\ \vdots & \ddots & \vdots \\ sim_c\,(t_m,t'_1) & \ldots & sim_c\,(t_m,t'_n) \end{pmatrix}$$

A valid match can be found by applying any maximum weighted bipartite matching algorithms, formally denoted as $MaxMatch\,(A, B) = \{(u_1, v_1), (u_2, v_2), \cdots, (u_l, v_l)\}$ where $l = min\,(m, n)$, $u_i \in kset\,(A)$, $v_i \in kset\,(B)$, $u_i \neq u_j\ if\ i \neq j$, $v_i \neq v_j\ if\ i \neq j$, and $\sum\limits_{i}^{l} sim_c\,(u_i, v_i)$ has the maximum value among all possible matchings. Finally, the similarity of two SS birthmarks are calculated with the following formula: $sim\,(A, B) =$

[2] All of the four metrics were used to compute birthmark similarities. We implemented all of them after slight modification in our prototype.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2688383, IEEE Transactions on Software Engineering

$$\frac{\sum_{\left(t_i,t'_j\right)\in MaxMatch(A,B)} sim_c\left(t_i,t'_j\right)\times\left(cnt(t_i)+cnt\left(t'_j\right)\right)}{\sum\limits_{i=1}^{m} cnt(t_i)+\sum\limits_{j=1}^{n} cnt\left(t'_j\right)},$$

where $cnt\left(t_i\right) = \left|kSet\left(birth\left(p,I,t_i\right)\right)\right|$, and $cnt\left(t'_j\right) = \left|kSet\left(birth\left(p',I,t'_j\right)\right)\right|$.

4.2 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, are also possible, even though executions faithfully reveal program behavior under a particular input. For example, two independently developed programs adopting standard error-handling subroutines may exhibit identical behavior under error-inducing inputs [17]. In order to alleviate this problem, the calculation of the similarity score between two programs is based on various birthmarks obtained under multiple inputs.

Let p and q be the plaintiff and defendant. Given a set of inputs $\{I_1, I_2, \cdots, I_n\}$ to drive the execution of the programs, we obtain n pair of birthmarks (for each birthmark type) $\{(A_1, B_1), (A_2, B_2), \cdots, (A_n, B_n)\}$. The similarity score between programs p and q is calculated by $sim\left(p,q\right) = \sum\limits_{i=1}^{n} sim\left(A_i, B_i\right) \Big/ n$. That is, the existence of plagiarism between p and q is decided by the average of similarity scores of their birthmarks and the threshold ε.

5 Design and Implementation

We have applied the TOB framework to revive three typical birthmarks, including SCSSB [14], DYKIS [17] and JB [18]. This leads a set of tools that are collectively called `TOB-PD`. All the tools follow the same workflow as depicted in Figure 7, except for some differences in the implementation of each module. For example, the tracer for DYKIS is implemented on top of `PIN` [41] to monitor executed instructions of binary executables, while the tracer for JB is implemented with `ASM` [42] to monitor API calls of Java programs. To avoid redundancy, in this paper we mainly discuss the implementation details of applying TOB to revive SCSSB. We name the enhanced thread-oblivious birthmarks SCSSB^{SA} and SCSSB^{SS} accordingly.

5.1 Tracer

The tracer for SCSSB is called `sysTracer`, which is implemented as a `PIN` [41] plugin. By instrumenting each call point to recognize the system calls and collect required information, `sysTracer` produces a system call sequence for each program run. The format of the sequence is illustrated in Figure 8, where each line corresponds to a system call record. A system call record contains a thread identifier, a system call number with which we uniquely identify each system call, the name of the system call, parameters when it is called, and its return value. These attributes are separated with "#".

5.2 Pre-Processor

The raw sequences extracted by `sysTracer` are not appropriate for birthmark generation yet. The preprocessor needs to parse the system call records and eliminate unnecessary information. First, failed calls are eliminated from the system call sequence because they do not affect the behavior characteristics of a program [14], [15]. This is accomplished by checking the return value attribute of each system call record. In addition, as discussed in Section 3.1, the nature of system call `futex` determines that its occurrences vary across multiple executions. Besides, the memory management system calls such as `mmap`, `brk` etc. also show intrinsic randomness across different runs. We treat all the records corresponding to these system calls as noises, and remove them from the sequence.

We know that for multithreaded programs multiple executions under the same input may vary significantly, which is the reason that we are focusing on more stable thread slices for our birthmark generation. However, even thread slices may vary due to the facts that a thread's behavior may be affected by other threads and there exist other random factors like OS-state related operations. In particular, a system call that has been executed in one thread in the first run may appear in another thread in the second run.

Since such randomness are unavoidable, we execute a program multiple times under the same input to obtain a series of traces. We then select two most similar traces for plaintiff and defendant within the `TraceSelector`, implemented as a sub-module of the pre-processor. In our implementation, by default `TraceSelector` select traces based on their Ex-Containment similarity, but other similarity metrics can be used as well. Effectiveness of this optimization will be evaluated in Section 6.3.4.

5.3 Birthmark Generator

Following the procedure depicted in Figure 5, the birthmark generator performs thread slicing on a preprocessed trace, generates thread slice birthmarks and then produce the thread-oblivious birthmarks using the SA and SS models. Both k-gram and var-gram algorithms are implemented. There are several algorithms for mining var-grams and we choose Teiresias [23], a well-known algorithm initially developed for discovering rigid patterns in unaligned biological sequences. It outperforms other var-gram algorithms [43], [44] in that it is capable of discovering all patterns without enumerating the solution space.

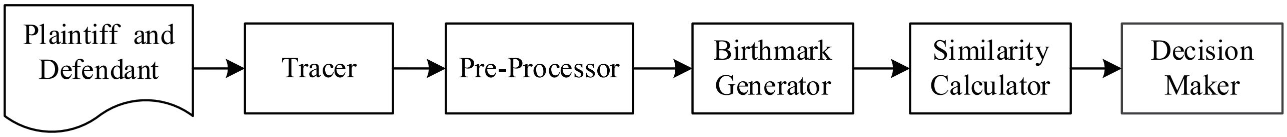
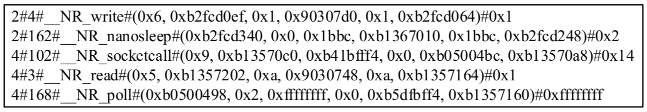


Fig. 7. Overview of the TOB based plagiarism detection tool TOB-PD



```
2#4#__NR_write#(0x6, 0xb2fcd0ef, 0x1, 0x90307d0, 0x1, 0xb2fcd064)#0x1
2#162#__NR_nanosleep#(0xb2fcd340, 0x0, 0x1bbc, 0xb1367010, 0x1bbc, 0xb2fcd248)#0x2
4#102#__NR_socketcall#(0x9, 0xb13570c0, 0xb41bfff4, 0x0, 0xb05004bc, 0xb13570a8)#0x14
4#3#__NR_read#(0x5, 0xb1357202, 0xa, 0x9030748, 0xa, 0xb1357164)#0x1
4#168#__NR_poll#(0xb0500498, 0x2, 0xffffffff, 0x0, 0xb5dfbff4, 0xb1357160)#0xffffffff
```

Fig. 8. Instances of elements comprising the system call sequence

5.4 Similarity Calculator & Decision Maker

These two modules implement the algorithms discussed in Sections 4.1 and 4.2, respectively. In the similarity calculator, scores are computed with each of the four similarity metrics for SA birthmarks. For SS birthmarks, we use each of the four metrics to calculate the weight of the edges in the bipartite graph. This produces four $SimMatr$ correspondingly. For each $SimMatr$, the Kuhn-Munkres algorithm [45] is used to find a maximum weighted bipartite matching. The decision maker decides plagiarism by the average value of multiple similarity scores against a predefined threshold ε that varies between 0 and 0.5.

6 EXPERIMENTS AND EVALUATION

We have conducted extensive experiments for evaluating the effectiveness of our TOB method. Table 4 lists the names and some other basic information of our benchmarks. The columns under *#Ver* give the number of versions of each program, where Column *Total* gives the total number of versions including the original program and its obfuscated versions, while the other four columns S_1, S_2, S_3 and S_4 give the number of obfuscated versions generated with different obfuscation strategies. The meaning of S_1, S_2, S_3 and S_4 are explained below. Column *Size* gives the number of kilobytes in the largest version, with its version number listed in Column *Version*. Following is a summary of our testing environment.

- The benchmarks consist of 35 mature multi-threaded software implemented in C/C++ or Java.
- We process the programs with the following obfuscation strategies for evaluating the resilience of our methods.
  - S_1: We use two different compliers `gcc` and `llvm` with various optimization levels to compile source code.
  - S_2: We apply publicly available code obfuscators, including `Sandmark`, `Zelix KlassMaster`, `Allatori`, `DashO`, `Jshrink`, `ProGuard` and `RetroGuard`, that provide strong obfuscations.
  - S_3: We utilize packing tool `UPX` to obfuscate binaries.
  - S_4: We generate programs that adopt different scheduling policies.
- We evaluate the credibility of our methods with independently developed programs in the same and different categories.
- All the programs are executed with multiple inputs that represent different workloads. And a program is executed four times for each input. All the execution traces are collected on a desktop (with Intel(R) i5-3270@3.2GHz CPU and 2.0GB Memory) that runs the Linux (Ubuntu 12.04) system.

With these settings, we mainly evaluate the resilience and credibility of the thread-oblivious birthmarks on the specific TOB implementation SCSSB^{SA} and SCSSB^{SS}. The overall quality of the TOB-revived birthmarks are further compared with the original SCSSB [15] and between each other[3], with respect to three performance metrics. Finally, we illustrate the effectiveness and easiness of applying the TOB framework to two other typical dynamic birthmarks.

It should be noted that, for birthmarks generated with the k-gram algorithm, different values of k lead to different birthmarks even for the same execution trace. In previous studies [15], [17], [18], [22] where k-gram is also used to generate birthmarks, setting the value of k to 4 or 5 is believed to be a proper compromise between accuracy and efficiency. To be fair, when reviving these traditional birthmarks and comparing with them, we adopt the same k values.

6.1 Validation of Resilience Property

6.1.1 Resilience to Different Compilers and Optimization Levels

Different compilers and optimization levels usually lead to different binaries from the same source code. We firstly validate whether the revived birthmarks can handle such relatively weak semantics-preserving code transformations. Specifically, we choose the ten

[3] Note that not all programs can start execution with a specified number of threads. Even if the number of threads can be specified, there is no guarantee that certain number can be maintained when program runs. Thus, for the sake of fairness, when comparing the traditional and TOB-revived birthmarks, we do not manually specify but let a program itself to determine the number of threads to handle different workloads.

TABLE 4
Benchmark programs

Name	Size(kb)	Version	#Ver					Name	Size(kb)	Version	#Ver				
			Total	S_1	S_2	S_3	S_4				Total	S_1	S_2	S_3	S_4
pigz	294	2.3	23	19	-	1	2	luakit	153.4	d83cc7e	1	-	-	-	-
lbzip	113.3	2.1	1	-	-	-	-	midori	347.6	0.4.3	1	-	-	-	-
lrzip	219.2	0.608	1	-	-	-	-	seaMonkey	760.9	2.21	1	-	-	-	-
pbzip2	67.4	1.1.6	1	-	-	-	-	Daisy	201.9	SIR	37	-	35	1	-
plzip	51	0.7	1	-	-	-	-	Elevator	92.1	SIR	44	-	42	1	-
rar	511.8	5.0	1	-	-	-	-	Groovy	59.5	SIR	44	-	42	1	-
cmus	271.6	2.4.3	1	-	-	-	-	Pool	205.7	SIR	30	-	28	1	-
mocp	384	2.5.0	1	-	-	-	-	blackscholes	23	Parsec3.0	23	19	-	1	2
mp3blaster	265.8	3.2.5	1	-	-	-	-	bodytrack	3,368	Parsec3.0	13	9	-	1	2
mplayer	4,300	r34540	1	-	-	-	-	fludanimate	126.6	Parsec3.0	23	19	-	1	2
sox	55.2	14.3.2	1	-	-	-	-	canneal	414.7	Parsec3.0	23	19	-	1	2
arora	1,331	0.11	1	-	-	-	-	dedup	388	Parsec3.0	23	19	-	1	2
chromium	80,588	28.0.1500.71	1	-	-	-	-	ferret	2,150	Parsec3.0	23	19	-	1	2
dillo	610.9	3.0.2	1	-	-	-	-	freqmine	227.6	Parsec3.0	23	19	-	1	2
Dooble	364.4	0.07	1	-	-	-	-	streamcluster	103	Parsec3.0	23	19	-	1	2
epiphany	810.9	3.4.1	1	-	-	-	-	swaptions	94	Parsec3.0	23	19	-	1	2
firefox	59,904	24.0	1	-	-	-	-	x264	896.3	Parsec3.0	23	19	-	1	2
konqueror	920.1	4.8.5	1	-	-	-	-								

TABLE 5
Statistical differences of the `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

PARSEC benchmark[4] programs and a compression program `pigz` as the experimental subjects. Two compilers `llvm3.2` and `gcc4.6.3` are used to compile the source code of each program with different optimization levels (`-O0`, `-O1`, `-O2`, `-O3` and `-Os`) and the debug option (`-g`) switched on or off. Such setup leads to 20 different executables for each experimental subject[5]. Table 5 gives the statistical differences on the size, number of functions, number of instructions, number of blocks and number of function calls between the 20 binaries of `pigz`. The data indicate that even weak code transformations can make significant differences to the produced binaries.

Pair-wise similarity is calculated between the binaries of each program with our methods. Figure 9 illustrates the distribution of the similarity scores. There are four subfigures, each corresponding to one the four metrics utilized for similarity computation. In each subfigure, the horizontal axis represents the thread-oblivious birthmarks, the vertical axis represents the percentage of birthmark pairs belonging to each similarity range as specified in the legend. For simplicity, we mark SA and SS birthmarks generated with k-gram algorithm as SA_K and SS_K in the figures. Correspondingly birthmarks generated with var-gram algorithm are marked as SA_V and SS_V.

It can be observed that the majority scores are in the 90%-above region for either kind of birthmark regardless of the similarity metrics. These observations indicate that the thread-oblivious birthmarks exhibit strong resilience against obfuscations caused by different compilers and optimization levels.

6.1.2 Resilience to Obfuscation Tools

In this section, we evaluate the resilience of our methods against advanced obfuscation techniques. In particular, we use the Java bytecode obfuscation tool `SandMark` [4] to generate a group of obfuscated versions, which are then converted to x86 executables by `GCJ` [46]. Since `Sandmark` can only obfuscate Java programs, thus the 4 multithreaded Java programs from the SIR Benchmark Suite[6], including `Daisy`, `Elevator`, `Groovy` and `Pool` are used as the experimental subjects. We design similar experiments as those conducted in [14], [17] to measure the resiliency of the thread-oblivious birthmarks against single obfuscations, where only one obfuscation technique is applied at a time, and multiple obfuscations, where multiple obfuscation techniques are applied at the same time.

a) Single obfuscation

We apply the 39 obfuscation techniques implemented in `Sandmark` on each program one at a time and generate a series of obfuscated versions. In order to ensure correctness of these transformations, all obfuscated versions are tested with a set of inputs and versions with wrong outputs are eliminated. We finally obtained 116 successfully obfuscated versions. There are 40 failed transformations. The failures are because certain obfuscation techniques cannot be applied, `GCJ` fails to compile obfuscated versions, or obfuscated versions no longer give correct outputs.

[4] http://parsec.cs.princeton.edu/

[5] Ten binaries are generated for `bodytrack`, since we fail to compile its source code with `llvm`.

[6] http://sir.unl.edu/content/sir.php

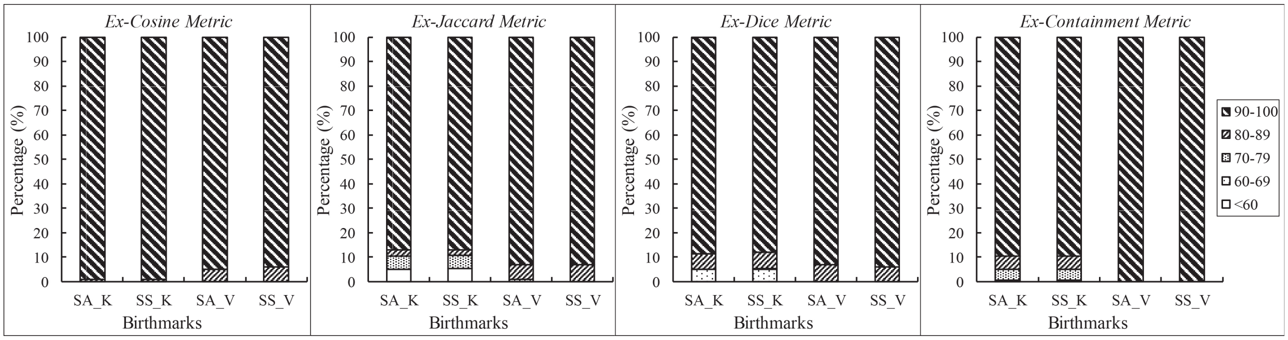


Fig. 9. Similarity distribution graph for birthmarks of the copies generated with different compilers and optimization levels

b) Multiple obfuscations

As indicated by the 40 failures in the prior section, transformations with a single obfuscation are not always successful. Not surprisingly, applying multiple obfuscations simultaneously can significantly increase the failure rate. To facilitate our experiments, we adopt the method used in [14], [17] where the obfuscators in SandMark are classified into two categories: data obfuscators and control obfuscators. Only the obfuscators in the same category are applied simultaneously to the same experimental subject.

Besides the SandMark obfuscators, six other commercial and open source obfuscation tools, including Zelix KlassMaster[7], Allatori[8], DashO[9], JShrink[10], ProGuard[11] and RetroGuard[12] that support layout, data and control flow obfuscations are also selected. Under the requirement of semantic equivalence between the original and the transformed programs, we turn on as many obfuscators as possible for each tool. Together with the two categories of SandMark, such setup leads to eight deeply obfuscated versions for each experimental subject.

The similarity scores are calculated between the birthmarks of each original program and all its obfuscated versions. According to the experimental results, most scores locate in the 70%-above region. In the cases where Ex-Cosine metric is used, most scores are in 90%-100% interval. These observations give a strong evidence that the thread-oblivious birthmarks are resilient to the semantics-preserving code obfuscations.

6.1.3 Resilience to Packing Tools

The packing tools [9], [10], which implement various binary obfuscation techniques as well as compression and encryption techniques, are widely used to hide the maliciousness of malware or to protect software from illegal modification and cracking. Such techniques may be used to evade plagiarism detection. This strategy can defeat most static birthmarks as it significantly modifies the syntax of a program.

The only publicly available packing tool we know that handles ELF, the executable file format under Linux, is UPX[13]. We process the previously used binaries with UPX. Similarity scores are calculated between birthmarks of the original programs and their corresponding UPX-packed versions. The results show that the majority scores are above 70%. It indicates that the thread-oblivious birthmarks are resilient against packing techniques.

6.1.4 Resilience to Different Scheduling Policies

A possible approach that plagiarized program adopts to evade detection is to exploit different scheduling policies different from the original program. As shown in Section 3.1, the scheduling policy a multithreaded program adopts has insignificant effect on the executions if viewed at the thread level. Thus, even if the defendant program adopts a different policy, we can simply compare the plaintiff with the defendant by trying all possible policies. This is doable as the source code of the plaintiff is always available, by modifying which we can set different policies. But we believe our methods still work even if we do not adjust the scheduling policy of the plaintiff to the same as the defendant.

To validate it, we generate three versions that adopt SCHED_OTHER, SCHED_FIFO and SCHED_RR, respectively, for the previously used C/C++ programs. We use the version that adopts the default policy as the plaintiff. We process the other two versions that adopt FIFO and RR policy with UPX, and take them as the the defendants. The experimental results that all scores are above 80% indicate that the TOB-revived birthmarks are not affected by scheduling policies.

6.2 Validation of Credibility Property

Credibility of a birthmark is evaluated by its capability of distinguishing independently developed

[7] http://www.zelix.com/klassmaster
[8] http://www.allatori.com
[9] https://www.preemptive.com/products/dasho
[10] //www.e-t.com/jshrink.html
[11] http://proguard.sourceforge.net
[12] http://java-source.net/open-source/obfuscators/retroguard
[13] http://upx.sourceforge.net/

programs. Three types of widely used multithreaded Linux applications are selected as our experimental subjects, including 6 compression software (`lbzip2`, `lrzip`, `pbzip2`, `pigz`, `plzip` and `rar`), 10 web browsers (`arora`, `chromium`, `dillo`, `Dooble`, `luakit`, `midori`, `epiphany`, `firefox`, `konqueror` and `seaMonkey`), and 5 audio players (`cmus`, `mocp`, `mp3blaster`, `mplayer` and `sox`).

Firstly, we validate whether the TOB-revived birthmarks can distinguish programs in different categories. Similarity scores between the 6 compression programs and the 5 audio players are computed. According to the experimental results, the majority of the scores are below 10%. These data indicate that thread-oblivious birthmarks have strong credibility in distinguishing independently developed programs.

Distinguishing programs in the same category is more challenging because they may overlap greatly in their functionality. Figure 10 depicts the similarity score distribution for the ten web browsers. It can be observed that about 90% of the scores are below 30%. Also as illustrated by Columns Avg in Table 6, the average scores are all around 0.1. Similar results are observed between the compression software and between the audio players.

There are several similarity scores above 40%. This is because some of the browsers share the same layout engine. Our manual inspection discovers that five of the browsers (`arora`, `Dooble`, `epiphany`, `luakit` and `midori`) are all Webkit-based. In order to observe the effect of overlapped functionality on thread-oblivious birthmarks, we give the average similarity scores between these Webkit-based browsers in Column Avg+, and the average similarity scores between Webkit-based and non-Webkit-based browsers in Column Avg-. It can be observed that the values in Column Avg+ are 3 to 5 times greater than those values in Column Avg-. Since the goal 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.

6.3 Comparison with Traditional Birthmarks

This section compares the overall performance of the TOB-revived SCSSBs against the original SCSSB. We utilize the three evaluation metrics adopted in [17], including `URC`, `F-Measure` and `MCC`. `URC` measures resilience and credibility, while the other two are more comprehensive metrics introduced for amending the problem of `URC` that focuses only on the rate of correct classifications. All the comparison pairs of programs from Section 6.1 to Section 6.2 are taken as the experimental subjects.

6.3.1 Performance Evaluation with Respect to URC

Resilience and credibility reflect from different aspects the qualities of a birthmark. `URC` (Union of Resilience and Credibility) [47], defined below, is a metric proposed for evaluating the overall performance of birthmarks that considers both aspects.

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

In the definition R represents the ratio of correctly classified pairs where there exists plagiarism and C represents the ratio of correctly classified pairs where there is no plagiarism. 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 $\forall (p, q) \in EP$, q is a copy of p, and JP be the set of pairs such that $\forall (p, q) \in JP$, a plagiarism detection method believes that q copies p. Similarly, let EI 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 formally 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 depends 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 exist and non-exist at the same time. Figure 11 shows the results. In each subfigure, the data for SCSSB as well as its TOB-revived SA and SS birthmarks are depicted by the lines marked with square, triangle and circle symbols, respectively.

It can be observed from the figures that the SA and SS birthmarks do not exhibit significant difference regardless of similarity metrics. Meanwhile, both have greater `URC` values than SCSSB's across the x-axis. It can also be observed that SCSSB's curves are closer to the curves of its TOB-revived versions when Ex-Cosine is adopted, indicating similarity calculation using such metric is less sensitive to thread scheduling. Moreover, the curves of var-gram generated birthmarks are above the corresponding curves of k-gram generated birthmarks, especially for SCSSB. This indicates the superiority of applying var-gram algorithm to birthmark generation.

6.3.2 Performance Evaluation with F-Measure and MCC

As explained in work [17], `URC` mainly measures the rate of correct classifications, while inconclusiveness is considered as incorrect classification. Thus, `URC` gives better results with higher value of ε in Figure 11. As the value of ε increases, the chance of inconclusiveness becomes smaller, leading to less incorrect classifications.

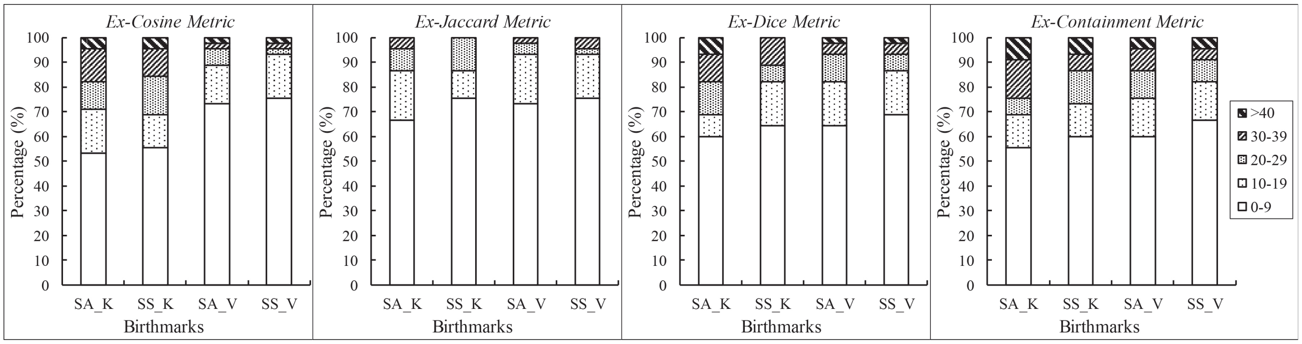


Fig. 10. Similarity distribution graph for birthmarks of the web browsers

TABLE 6
Credibility evaluation of the thread-oblivious birthmarks using software in the same category

	K-GRAM						VAR-GRAM					
	SA			SS			SA			SS		
	Avg	Avg+	Avg-	Avg	Avg+	Avg-	Avg	Avg+	Avg-	Avg	Avg+	Avg-
Ex-Cosine	0.137	0.334	0.078	0.133	0.314	0.079	0.075	0.156	0.041	0.078	0.167	0.041
Ex-Jaccard	0.090	0.213	0.046	0.072	0.163	0.034	0.068	0.128	0.042	0.068	0.132	0.040
Ex-Dice	0.134	0.322	0.078	0.103	0.238	0.056	0.100	0.189	0.069	0.092	0.173	0.060
Ex-Containment	0.166	0.364	0.111	0.135	0.281	0.087	0.128	0.208	0.097	0.106	0.186	0.068

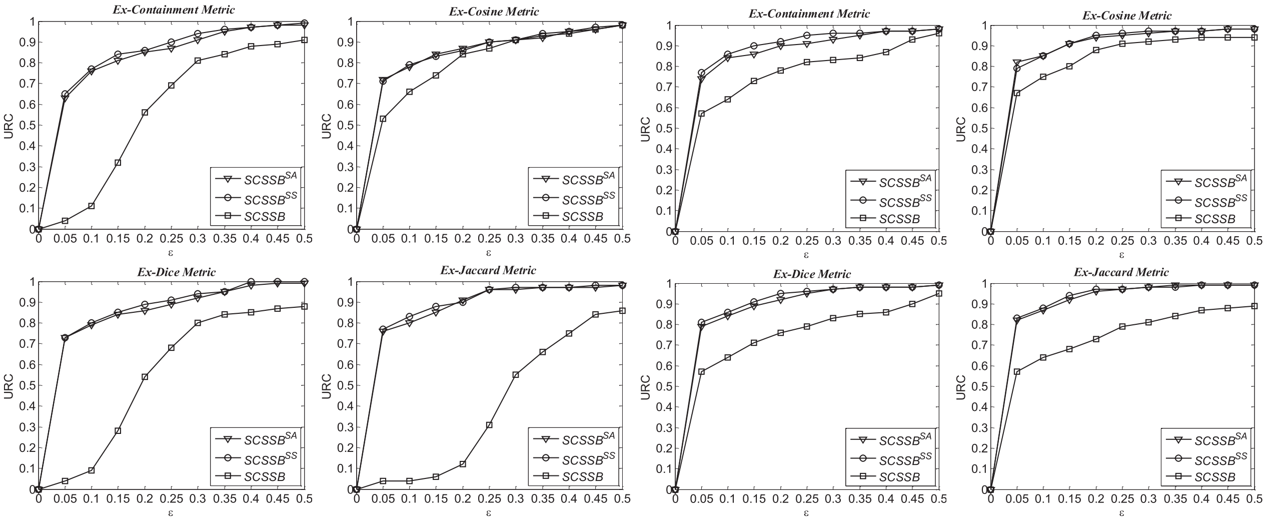


Fig. 11. Performance evaluation with respect to URC. The left four figures depict the curves for birthmarks generated with k-gram, the right four figures depict the curves for birthmarks generated with var-gram

To address the problem, similarly as in work [17], the birthmark methods are further compared against two other metrics, F-Measure and MCC (Matthews Correlation Coefficient) [48]. However, these two metrics cannot be directly applied as they mainly measure binary classifications. Thus, the definition of sim is revised as following by removing the inconclusiveness:

$$sim\left(p_{\mathcal{B}}, q_{\mathcal{B}}\right) = \begin{cases} \geq 1-\varepsilon & q \ is \ a \ copy \ of \ p \\ < 1-\varepsilon & q \ is \ not \ a \ copy \ of \ p \end{cases} \tag{4}$$

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

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

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.

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

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

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

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

Figure 12 depicts the experimental results with respect to F-Measure and MCC, respectively. Overall, similar results are observed as in the evaluation against URC. The TOB-revived birthmarks almost always outperform SCSSB across the whole x-axis.

More specifically, consider the curves summarizing the results with respect to F-Measure. The TOB-revived birthmarks outperform SCSSB mainly in the right region with relatively small thresholds. With relatively large thresholds the performance is very similar. Taking the upper-left figure that depicts results for k-gram generated birthmarks and calculated with Ex-Containment as an example, SCSSB performs almost as well as its TOB-revived ones until the value of ε becomes smaller than 0.55. But its F-Measure value decreases sharply when adopting a smaller threshold. To see the reason for the sharp decrease, we check the specific Precision, Recall and F-Measure values under different thresholds. According to the data, there is almost no difference between the Precision values of SCSSB and its TOB-revived ones under all thresholds. Also, the Recall value of SCSSB is almost identical with its TOB-revived ones, until it decreases sharply from threshold 0.5.

The reason for the sharp decrease on Recall value is the following. Due to the combined impacts from obfuscations and thread interleavings, the SCSSB of plagiarized pairs are greatly affected, which leads to low similarity scores. As the Recall values of SCSSB indicate, only about 58% scores are above 0.75, and only about 6% scores are above 0.9. But for the TOB-revived ones, there are about 90% scores that are above 0.9. Such results indicate that the thread-oblivious birthmarks are resilient to obfuscations and thread interleavings.

Figure 13 shows that SCSSB can differentiate plagiarized pairs and independent pairs even though the range of similarity scores is small. This explains the almost identical Precision values between SCSSB and its TOB-revived ones. In birthmark based plagiarism detection literature, plagiarism is determined by the similarity score and a threshold. Unfortunately, without abundant real-world plagiarism samples, deciding a threshold value is an arbitrary decision. In the ideal but unrealistic case, we hope the similarity scores for plagiarized pairs are all 1.0, and for independent pairs are all 0. Thus, we believe the greater the difference between the two type of scores, the better a birthmark method is. As indicated by the boxplots in Figure 13, the TOB-revived birthmarks exhibit much better distinct similarity scores. In particular, the TOB-revived birthmarks achieve 100% detection accuracy at $\varepsilon = 0.34$, where neither false positives nor false negatives are observed.

6.3.3 Comparing the Birthmarks with AUC Analysis

As discussed above, the birthmark methods exhibit different performance under different thresholds. To give an intuitive comparison, we compute the AUC (Area Under the Curve) values for each method with respect to the URC, F-Measure and MCC metrics. The AUC value gives a proper overall performance summary for each birthmark method. A larger value of AUC indicates better birthmark quality. The results are summarized in the white areas of Table 7. It can be observed that the AUC values of the TOB-revived birthmarks are all larger than those of SCSSB's.

We quantify the performance gains $PerGain$ by taking the original SCSSB as baseline. The quantification indicates the improvement of each thread-oblivious birthmark against the original birthmark, with respect to the same similarity metric and the same performance evaluation metric.

$$PerGain = \frac{AUC_{tob} - AUC_{org}}{AUC_{org}} \times 100\%$$

, where AUC_{tob} and AUC_{org} represent the AUC value of the thread-oblivious birthmark and the original birthmark respectively. For example, the $PerGain$ value with respect to Ex-Containment similarity and URC metric for SCSSB^{SA} generated with k-gram is:

$$\frac{0.822 - 0.56}{0.56} \times 100\% = 47\%$$

The average and maximal performance gains are summarized in the last row of Table 7. As the data indicate, TOB-revived birthmarks improve the original birthmark. The maximum performance gains happen for those SS birthmarks generated with k-gram and calculated with the Ex-Jaccard similarity, where 129%, 46% and 94% improvements are obtained with respect to URC, F-Measure and MCC metrics, respectively. Additionally, it can be observed that the AUC values based on var-gram are larger than those based on k-gram, indicating the superiority of applying var-gram algorithm to birthmark generation.

6.3.4 Evaluation of the TraceSelector Optimization

As mentioned in Section 5.2, we perform an optimization that chooses two most similar sequences from plaintiff and defendant programs to reduce the randomness of thread interleaving. In this section, we evaluate the impact of such optimization.

To simulate the worst case caused by thread interleavings, two least similar traces are selected from the executions of the plaintiff and defendant programs. The gray areas in Table 7 summarize the AUC values without optimization. By comparing with the values

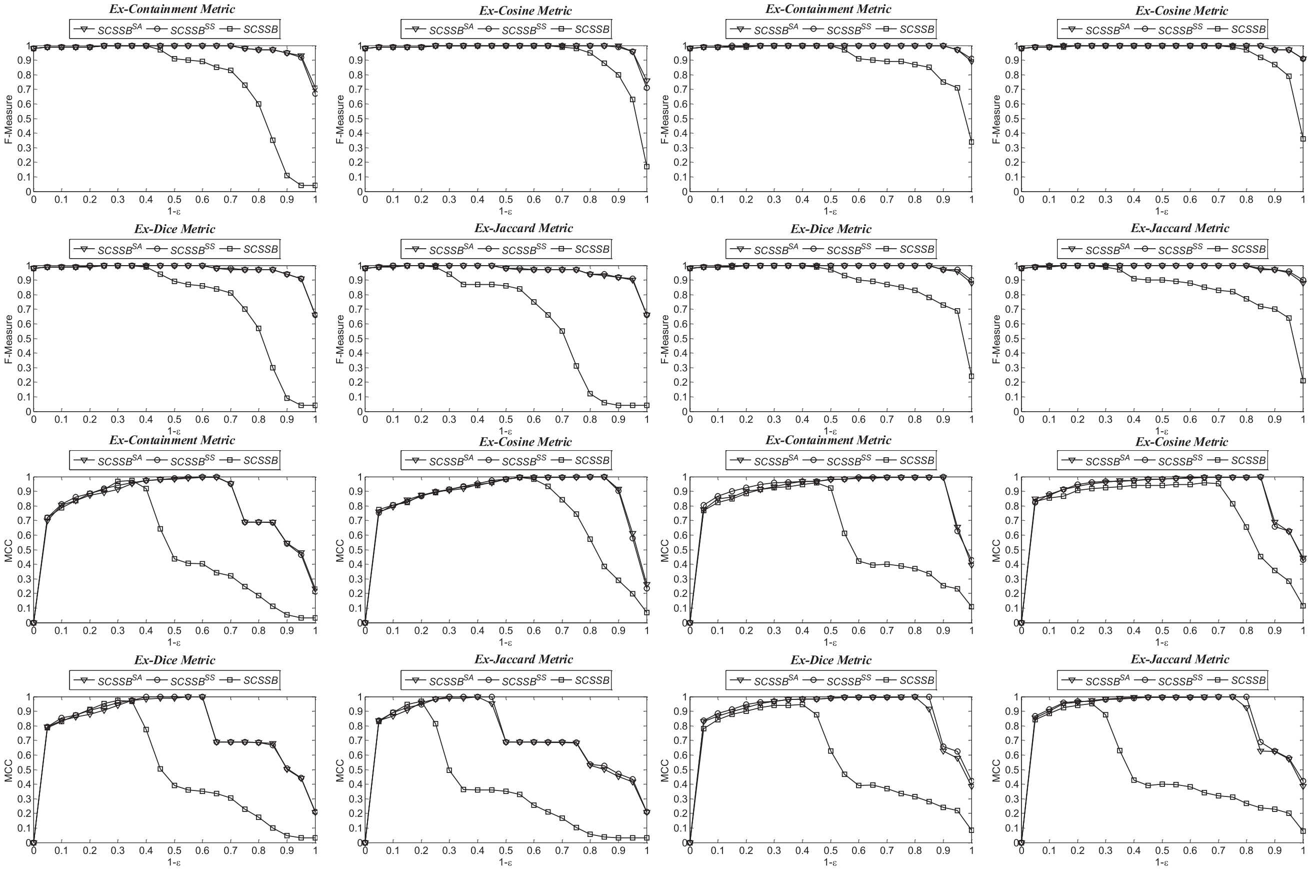
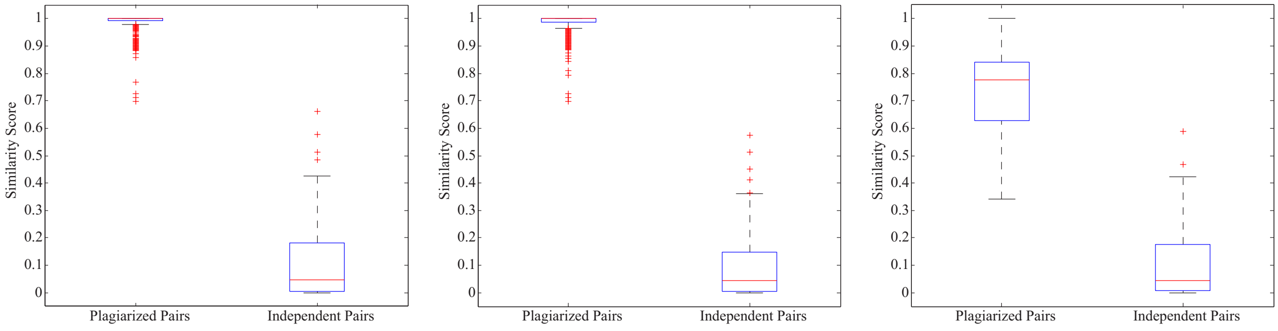


Fig. 12. F−Measure and MCC curves for the birthmark methods. The upper-left four figures depict the F-Measure curves for birthmarks generated with k-gram under each similarity metric, the upper-right four figures depict the F−Measure curves for birthmarks generated with var-gram, the bottom-left four figures and the bottom-right four figures similarly depict the MCC curves for birthmarks generated with k-gram and var-gram respectively.



(a) SCSSB^{SA}

(b) SCSSB^{SS}

(c) SCSSB

Fig. 13. Boxplots that summarize the statistical distribution of similarity scores corresponding to the very upper-left figure in Figure 12

TABLE 7
AUC analysis results

(a) With respect to URC evaluation metric

	K-GRAM						VAR-GRAM					
	SCSSB^{SA}		SCSSB^{SS}		SCSSB		SCSSB^{SA}		SCSSB^{SS}		SCSSB	
Ex-Containment	0.878	0.822	0.874	0.837	0.45	0.56	0.875	0.856	0.884	0.875	0.58	0.749
Ex-Cosine	0.883	0.834	0.873	0.835	0.787	0.799	0.895	0.884	0.897	0.884	0.624	0.821
Ex-Dice	0.885	0.845	0.88	0.857	0.426	0.543	0.889	0.879	0.895	0.889	0.538	0.739
Ex-Jaccard	0.878	0.864	0.88	0.872	0.236	0.38	0.903	0.899	0.909	0.903	0.462	0.726
PerGain(%)	-	59\127	-	61\129	-	-	-	16\24	-	17\24	-	-

(b) With respect to F-Measure evaluation metric

	K-GRAM						VAR-GRAM					
	SCSSB^{SA}		SCSSB^{SS}		SCSSB		SCSSB^{SA}		SCSSB^{SS}		SCSSB	
Ex-Containment	0.978	0.98	0.976	0.979	0.72	0.782	0.993	0.994	0.994	0.995	0.778	0.918
Ex-Cosine	0.989	0.989	0.986	0.988	0.92	0.938	0.993	0.993	0.992	0.993	0.818	0.959
Ex-Dice	0.972	0.974	0.971	0.976	0.703	0.769	0.991	0.992	0.992	0.993	0.753	0.9
Ex-Jaccard	0.96	0.966	0.964	0.968	0.579	0.663	0.988	0.99	0.99	0.992	0.698	0.867
PerGain(%)	-	26\46	-	26\46	-	-	-	9\14	-	9\14	-	-

(c) With respect to MCC evaluation metric

	K-GRAM						VAR-GRAM					
	SCSSB^{SA}		SCSSB^{SS}		SCSSB		SCSSB^{SA}		SCSSB^{SS}		SCSSB	
Ex-Containment	0.823	0.802	0.821	0.808	0.483	0.51	0.894	0.894	0.903	0.904	0.389	0.619
Ex-Cosine	0.889	0.876	0.88	0.874	0.734	0.744	0.892	0.894	0.893	0.893	0.468	0.774
Ex-Dice	0.795	0.782	0.793	0.788	0.461	0.496	0.879	0.883	0.894	0.896	0.371	0.586
Ex-Jaccard	0.728	0.73	0.737	0.739	0.338	0.381	0.868	0.877	0.884	0.885	0.318	0.5
PerGain(%)	-	56\92	-	57\94	-	-	-	47\75	-	48\77	-	-

between the gray columns without optimization and white columns with optimization, we can see that the performance of the original SCSSB always gets improved after the optimization. We use the following equation to quantify the improvement achieved by the optimization:

$$OptiGain = \frac{AUC_{opt} - AUC_{noOpt}}{AUC_{noOpt}} \times 100\%$$

As shown by the $OptGain$ values in Table 8, the performance gains achieved by the optimization are significant. Conclusion can be drawn that such optimization helps to a large extent alleviate the problem of SCSSB in applying to multithreaded programs. Yet as indicated by the data in Table 7, it is still not adequate to handle the disturbance of thread interleavings. This also demonstrates the significant impact thread interleaving could enforce on traditional SCSSB. On the other hand, the improvement for the TOB-revived ones are much less significant, indicating much less impact of thread interleaving on thread-oblivious birthmarks. Besides, as shown by the negative values, the optimization sometimes can make the overall performance of TOB-revived birthmarks worse.

The optimization, which pre-select more similar execution traces of the plaintiff and defendant, improve the similarity scores for both plagiarized pairs and independent pairs. It means that the optimization enhances the resilience but weakens the credibility, as larger similarity scores bring not only less false negatives but also more false positives. Consider the two bold font values -6.4 and 24.4 in Table 8, which are the $OptGain$ values for SCSSB^{SA} and SCSSB (generated with k-gram and measured with Ex-Containment similarity) with respect to URC. Table 9 gives their corresponding resilience (reflected by R in equation 3), credibility (reflected by C in equation 3) and URC values. The gray columns summarize the values without the optimization, and the white columns summarize the values with the optimization.

As the data show, resilience of both SCSSB and SCSSB^{SA} is enhanced after the optimization, while credibility of both birthmarks is weakened. However, the degree for the resilience promotion of SCSSB are rather obvious compared with the degree of its credibility reduction, leading to a significant increase in its URC values. On the other hand, the degree for credibility reduction of SCSSB^{SA} are more obvious than the degree of its resilience promotion, resulting in minor decrease in its URC values. Similarly, the negative $OptGain$ values in terms of MCC can also be explained, as the number of false positives the

TABLE 8
$OptGain$ values for the TraceSelector optimization

		URC			F-Measure			MCC		
		SCSSB^{SA}	SCSSB^{SS}	SCSSB	SCSSB^{SA}	SCSSB^{SS}	SCSSB	SCSSB^{SA}	SCSSB^{SS}	SCSSB
K-GRAM	Ex-Containment	**-6.4**	-4.2	**24.4**	0.2	0.3	8.6	-2.6	-1.6	5.6
	Ex-Cosine	-5.5	-4.4	1.5	0.0	0.2	2.0	-1.5	-0.7	1.4
	Ex-Dice	-4.5	-2.6	27.5	0.2	0.5	9.4	-1.6	-0.6	7.6
	Ex-Jaccard	-1.6	-0.9	61.0	0.6	0.4	14.5	0.3	0.3	12.7
V-GRAM	Ex-Containment	-2.2	-1.0	29.1	0.1	0.1	18.0	0.0	0.1	59.1
	Ex-Cosine	-1.2	-1.4	31.6	0.0	0.1	17.2	0.2	0.0	65.4
	Ex-Dice	-1.1	-0.7	37.4	0.1	0.1	19.5	0.5	0.2	58.0
	Ex-Jaccard	-0.4	-0.7	57.1	0.2	0.2	24.2	1.0	0.1	57.2

optimization brings are larger than the number of false negatives it reduces. Birthmarking is a detecting technique of suspected plagiarisms rather than a proving technique [14], [15], [17], false negative is more critical than false positive. Thus, we believe the optimization is proper and necessary.

6.4 Performance of the Analysis

This section presents the analysis performance of SCSSB and its TOB-revived ones. Both the birthmark methods involve basically two phases: the dynamic analysis phase that traces program executions (Phase I) and the static detection phase that extracts birthmarks and calculates similarities (Phase II).

For Phase I, our measurement indicates that on average a program is observed to become about 3 times slower once `PIN` and our tracer plugin are attached. The tracing overhead is observed smaller for larger inputs. It is because the overhead imposed by `PIN`'s runtime environment alone (that is runs a program using `PIN` without any instrumentation or analysis) become trivial compared to the total runtime.

For Phase II, no significant difference on calculation overhead is observed between the methods. Specifically, for k-gram generated birthmarks, Phase II takes on average 54ms, 58ms and 67ms for processing a trace pair for SCSSB^{SA}, SCSSB^{SS} and SCSSB, respectively. For var-gram generated birthmarks, the corresponding average time are 2.43s, 2.44s, and 2.43s, where the variable-pattern mining costs the most time.

6.5 Applying TOB to Other Birthmarks

In this section, we further demonstrate the application of TOB on two other representative dynamic birthmarks. One is DYKIS [17] that is based on executed key instructions, and the other one is JB [18] that is based on executed APIs of a Java program. For simplicity, we use the k-gram algorithm only. We use DYK_TR, DYK_SA, and DYK_SS to represent the original DYKIS and its TOB-revived versions using SA and SS models, respectively. Similarly we have JB_TR, JB_SA, and JB_SS for JB.

6.5.1 Reviving DYKIS with TOB

For DYKIS [17], we use the 20 `pigz` binaries generated with different compilers and optimization levels as the experimental subjects. Figure 14 illustrates the distribution graph of the similarity scores calculated between the birthmarks of the 20 `pigz` binaries. As it shows, all the scores of thread-oblivious birthmarks are above 70%, while for DYKIS there are quite a number of scores below 70%. It indicates the effectiveness of applying the TOB framework on DYKIS.

6.5.2 Reviving JB with TOB

For JB [18], the 4 Java programs as well as their 149 single and deep obfuscated versions are used as the experimental subjects. Similarity scores are calculated between the original Java programs and their obfuscated versions. No significant differences are observed between the original JB and its TOB-revived ones. This is because JB is extracted from API call sequences at object level. Similar to the TOB framework that slices traces by thread, JB essentially slices traces by Java objects that greatly mitigates the effect of thread scheduling. However, JB is only applicable to Java programs, while our TOB framework can be applied to transforming existing dynamic birthmarks, including JB, into thread-oblivious ones. In addition, the larger average and minimum scores as summarized in Table 10 show that the TOB-revived birthmarks indeed improve the original JB, although not significantly.

7 Threats To Validity

Dynamic birthmarks are extracted from execution traces, therefore execution monitoring is necessary. It is an undeniable fact that the monitoring itself may affect the thread interleavings during the execution of a multithreaded program. Many other factors such as workload, scheduling policies, and runtime environments affect the interleavings as well. However, we do not believe these issues cause an unfair comparison against traditional birthmarks.

For a multithreaded program, it is possible that the effect of scheduling causes it to execute different

TABLE 9
Impact analysis of the optimization to birthmark credibility and resilience

Threshold	SCSSB^{SA} generated with k-gram						SCSSB generated with k-gram					
	Resilience		Credibility		URC		Resilience		Credibility		URC	
0	0	0	0	0	0	0	0	0	0	0	0	0
0.05	0.84	0.86	0.63	0.5	0.72	0.63	0.02	0.02	0.66	0.53	0.04	0.04
0.1	0.88	0.9	0.8	0.66	0.84	0.76	0.02	0.06	0.82	0.63	0.04	0.11
0.15	0.94	0.95	0.89	0.71	0.91	0.81	0.03	0.21	0.89	0.71	0.05	0.32
0.2	0.95	0.95	0.91	0.77	0.93	0.85	0.19	0.43	0.96	0.79	0.31	0.56
0.25	0.95	0.95	0.93	0.8	0.94	0.87	0.37	0.58	0.97	0.84	0.53	0.69
0.3	0.99	1	0.96	0.84	0.98	0.91	0.45	0.71	0.97	0.94	0.62	0.81
0.35	1	1	0.97	0.91	0.98	0.95	0.61	0.74	0.98	0.96	0.75	0.84
0.4	1	1	0.98	0.95	0.99	0.97	0.73	0.81	0.99	0.96	0.84	0.88
0.45	1	1	0.99	0.96	0.99	0.98	0.8	0.81	0.99	0.96	0.84	0.88
0.5	1	1	1	0.97	1	0.98	0.81	0.84	0.99	0.99	0.89	0.91

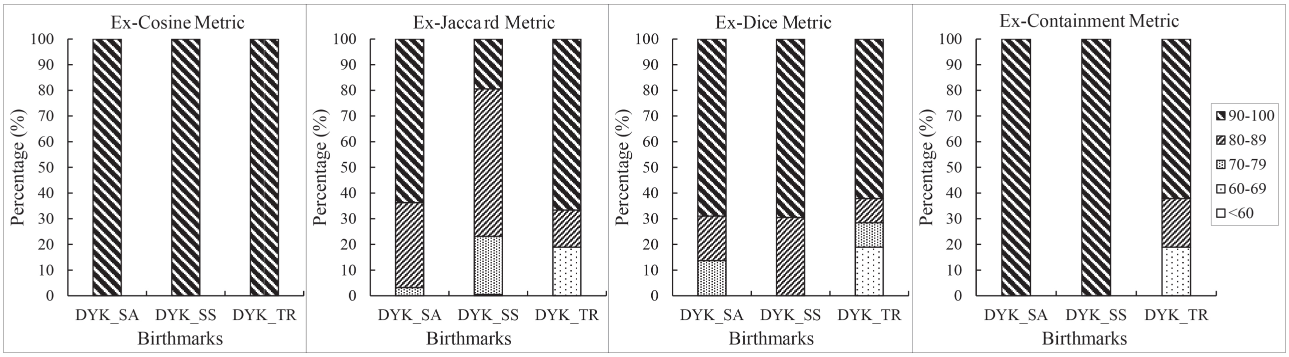


Fig. 14. Similarity distribution graph for DYKIS and its TOB-revived thread-oblivious ones

TABLE 10
Effectiveness of applying the TOB framework to JB

	JB_SA			JB_SS			JB_TR		
	Avg.	**Max.**	**Min.**	**Avg.**	**Max.**	**Min.**	**Avg.**	**Max.**	**Min.**
Ex-Cosine	0.983	1.000	0.554	0.983	1.000	0.602	0.983	1.000	0.555
Ex-Jaccard	0.970	1.000	0.530	0.979	1.000	0.618	0.963	1.000	0.470
Ex-Dice	0.976	1.000	0.503	0.978	1.000	0.527	0.964	1.000	0.453
Ex-Containment	0.986	1.000	0.545	0.985	1.000	0.596	0.978	1.000	0.508

paths across multiple runs even under the same input. In such cases, the execution traces of multiple runs apparently can become different even after projection on individual threads, which may lead to the failure of our methods. But as indicated by our experiments conducted on the various types of real multithreaded programs, the thread-oblivious birthmarks always illustrate good performance. Thus we believe such cases rarely happen in practical programs. Besides, as discussed in Section 5 and Section 6.3.4, we adopt an optimization that select two most similar traces from plaintiff and defendant for further birthmark generation. Thus even if the mentioned cases happen, the optimization helps alleviate the problem.

The thread-oblivious birthmarks improve upon traditional birthmark with TOB framework. Thus they suffer the same limitation of dynamic birthmarks in exhaustively covering all behaviors of a program. In the experiments, despite large number of executions, the inputs still constitute only a small proportion of the whole input space. This is the fundamentally challenge for all dynamic birthmarks. One way to alleviate the concerns is to combine with testing techniques. We take it as one of our future work.

Effectiveness of the TOB framework is mainly evaluated on whole program plagiarism detection, where a complete program is copied and then disguised through various automatic semantics-preserving transformations. One problem plagiarism detection researches face is the lack of real-world plagiarism cases [49], [50]. In recent years, whole program plagiarism on mobile markets starts to rise, and many of the stolen apps have been processed with obfuscation techniques to evade plagiarism detection. According to a recent study [51], about 5%-13% of apps in the third-party app markets are copied and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2688383, IEEE Transactions on Software Engineering

redistributed from the official Android market. This potentially provides rich real-world plagiarism cases. Yet, identifying potential pairs of apps that plagiarism may exist is extremely labor-intensive. Also, tracing apps needs nontrivial efforts because our tracers support the monitoring of binary executables and Java bytecodes. Thus, we take it as one of our future work.

Besides whole program plagiarism, there exists many cases that only part or a library of a program is copied. The main problem of using dynamic birthmarks to detect partial plagiarism is that they are mainly based on the similarity of program executions. Therefore, if there is only a small portion of code being copied, these approaches give low similarity scores. Improved upon existing dynamic birthmarks, the thread-oblivious birthmarks suffer the same problem. A straightforward solution is to instrument only the suspicious part. But this requires manual efforts and domain knowledge.

8 Related Work

Broadly speaking, the research areas related to our work include software watermarking [25], [52] which protects software copyrights and detects piracy, plagiarism detection, as well as code clone detection [53], [54] and malware identification [55], [56] that detect clones or maliciousness by characterizing software with features. In this section we focus on the discussion of birthmark based software plagiarism detection techniques. Works targeting source code will not be discussed here, and there have already been many mature detection systems and tools [20], [21], [57].

8.1 Static birthmark based software plagiarism detection

Myles and Collberg [27] proposed k-gram based static birthmarks, where sets of Java bytecode sequences of length k are taken as the birthmarks. The similarity between two birthmarks was calculated through set operations that ignore the frequency of elements in the set. Although being more robust than birthmarks proposed Tamada [8], the birthmarks were still vulnerable to code transformation attacks. Weighted k-gram based static birthmarks [47] improved upon Myles and Collberg's [27] by taking the frequency of each k-length operation code sequence into consideration. However, the improvement in detection ability seems minor while introducing extra cost in computing change rate of k-gram frequencies. A static birthmark based on disassembled API calls from executables is put forward by Seokwoo et al. [19] to detect plagiarism of windows applications. The requirement for de-obfuscating binaries before applying their method is too restrictive and thus reduces its availability. Park [12] proposed a static birthmark by extracting all possible sequences of object instructions from a CFG of each method, and applied it for detecting common modules in Java packages. Yet this method suffered high time consumption since a mass of traces can be extracted if the CFG is complex. Hemel et al. [58] suggested three methods to find potential cloned binaries within a program repository by simply treating binaries as normal files. Specifically, similarity between two binaries were evaluated by calculating the ratio of shared string literals, by calculating the compression ratio, and by computing binary deltas. Since no syntactic or semantic attributes of binary executables are considered, efficiency is assured but low detection accuracy is expected. Lim used control flow information that reflected runtime behaviors to supplement static approaches [59]. Recently he proposed to analyze stack flows obtained by simulating operand stack movements to detect copies [60]. But they are only available to Java programs. An obfuscation-resilient method based on longest common subsequence of semantically equivalent basic blocks was proposed by Luo et. al. [61]. 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. In addition, symbolic execution combined with theorem proving is not scalable.

There are also some work focusing on detecting plagiarism for smartphone applications. DroidMOSS [3] detects plagiarism by applying fuzzing hashing on instruction sequences. Yet simple obfuscations, such as noise injection can evade the detection of DroidMOSS, since no semantic information is used. DNADroid [62] achieves plagiarism detection by constructing and comparing program dependence graphs between methods. Since considering data dependencies, this method are more robust. ViewDroid [51] proposes the feature view graph birthmark by capturing users' navigation behaviors. But it's vulnerable to dummy view insertion and encryption attacks.

8.2 Dynamic birthmark based software plagiarism detection

Myles and Collberg [7] suggested the whole program path (WPP) birthmark generated by compressing a whole dynamic control flow trace into a directed acyclic graph form to uniquely identify program. Even with compression the method does not scale, and it's susceptible to various loop transformations. Schuler [18] treated Java standard API call sequences at object level as dynamic birthmarks for Java programs. Such approach exhibited better performance than WPP birthmark, but they also pointed out that their method was affected by thread scheduling. Similar principle was applied in Tamada's work [63], where API calls of windows executables executed during runtime were used to derive two kind birthmarks: Sequence of API Function Calls (EXESEQ) and Frequency of API Function Calls (EXEFREQ). Apparently

API based birthmarks are all language dependent. To address the problem Wang et al. [15] proposed System Call Short Sequence birthmark (SCSSB), that treat the sets of k-length system call sequences as birthmarks. However, as we illustrated it has limited applicability to multithreaded programs.

Liu et al. [34], [49] suggested to characterize software with core values and applied it to software and algorithm plagiarism detection. Tian et al. [17], [64] proposed the DYKIS birthmark based on dynamic key instruction sequences. By introducing dynamic taint analysis into birthmark generation, these birthmark methods were resilient to various semantics-preserving code transformations. LoPD [65], a program logic based approach was designed for software plagiarism detection by leveraging symbolic execution and weakest precondition reasoning to find semantic dissimilarities. Despite these methods are resilient, they all suffer the scalability problem, since they all operates on the instruction granularity, and either taint analysis or symbolic execution with constraint solving is computational non-trivial.

By integrating data flow and control flow dependency analysis, Wang et al. [14] proposed a system call dependency graph based birthmark, and graph isomorphism is utilized for calculating similarity between birthmarks. Patrick et al. [11] proposed a heap graph birthmark for JavaScript utilizing heap memory analysis, and graph monomorphism algorithm was applied for similarity computation. But to be effective, these graph based birthmarks require that the programs under protection to have prominent referencing structures. Also, since graph isomorphism and monomorphism algorithms are NP-complete in general, several thousand nodes will make the methods impractical to use.

9 Conclusion

As multithreaded software become increasingly more popular, current dynamic software plagiarism detection technology geared toward sequential programs are no longer sufficient. This paper fills the gap by proposing a thread-oblivious software plagiarism detection framework (TOB) that revives existing dynamic software birthmarks. We have developed a set of tools collectively called `TOB-PD` by applying the TOB framework to three typical dynamic birthmarks, including SCSSB, DYKIS and JB. The extensive experiments conducted on 418 versions of 35 different programs show that the proposed approaches are not only accurate in detecting plagiarism of multithreaded programs but also robust against most state-of-the-art semantics-preserving obfuscation techniques. In addition, a suite of benchmarks of multithreaded programs are complied. We believe there will be more research on plagiarism detection for multithreaded programs. The existence of such benchmarks will be beneficial for researchers to conduct experiments and present their findings. The benchmarks, the TOB-PD tools as well as the experimental data are all available.

Our work addresses the challenges of applying dynamic birthmark based approaches for whole program plagiarism detection of multithreaded software. As far as we know, this is the first work that discusses the impact of thread scheduling on birthmark based plagiarism detection, and the first work that propose thread-oblivious birthmarks for solving the problem systemically. In recent years, whole program plagiarism of mobile apps has becomes a serious problem. About 5% to 13% of apps in third-party app markets are copied and redistributed from the official Android market. We plan to conduct case studies and optimize our approaches for this domain.

Acknowledgement

The research was supported in part by National Key Research and Development Program of China (2016YFB1000903), National Science Foundation of China under grants (91418205, 61472318, 61532015, 61532004, 61672419, 61632015, 61602369, 71501156, 61373116), Fok Ying-Tong Education Foundation (151067), Ministry of Education Innovation Research Team (IRT13035), Science and Technology Project in Shaanxi Province of China (2016KTZDGY04-01, 2016GY-092), and the Fundamental Research Funds for the Central Universities. 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. T. Liu is the corresponding author.

References

[1] [Online]. Available: http://sourceauditor.com/blog/tag/lawsuits-on-open-source/.

[2] [Online]. Available: http://www.martinsuter.net/blog/2009/08/skype-joltid-licensing-dispute-epic-ma-screwup.html.

[3] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proc. ACM Conf. Data and Application Security and Privacy (CODASPY '12)*, 2012, pp. 317–326.

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

[5] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang, "Mimimorphism: A new approach to binary code obfuscation," in *Proc. ACM Conf. Computer and Communications Security (CCS '10)*. ACM, 2010, pp. 536–546.

[6] C. Linn and S. K. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. ACM Conf. Computer and Communications Security (CCS '03)*, 2003, pp. 290–299.

[7] G. Myles and C. S. Collberg, "Detecting software theft via whole program path birthmarks," in *Proc. Int. Conf. Information Security (ISC '04)*, 2004, pp. 404–415.

[8] H. Tamada, M. Nakamura, and A. Monden, "Design and evaluation of birthmarks for detecting theft of Java programs," in *IASTED Conf. on Software Engineering (IASTEDSE '04)*, 2004, pp. 569–574.

[9] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys*, vol. 46, no. 4, 2013.

[10] M.-J. Kim, J.-Y. Lee, H.-Y. Chang, S. Cho, and P. A. Wilsey, "Design and Performance Evaluation of Binary Code Packing for Protecting Embedded Software against Reverse Engineering," in *IEEE Int. Symp. Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC '10)*, 2010, pp. 80–86.

[11] P. P. F. Chan, L. C. K. Hui, and S.-M. Yiu, "Heap graph based software theft detection," *IEEE Trans. Information Forensics and Security*, vol. 8, no. 1, pp. 101–110, 2013.

[12] H. Park, H. il Lim, S. Choi, and T. Han, "Detecting common modules in Java packages based on static object trace birthmark," *Computer Journal*, vol. 54, no. 1, pp. 108–124, 2011.

[13] Z. Tian, Q. Zheng, T. Liu, and M. Fan, "DKISB: Dynamic key instruction sequence birthmark for software plagiarism detection," in *IEEE Int. Conf. High Performance Computing and Communications. (HPCC '13)*, 2013, pp. 619–627.

[14] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *Proc. ACM Conf. Computer and Communications Security (CCS '09)*. ACM, 2009, pp. 280–290.

[15] X. Wang, Y. Jhi, S. Zhu, and P. Liu, "Detecting software theft via system call based birthmarks," in *Annual Computer Security Applications Conference (ACSAC '09)*, 2009, pp. 149–158.

[16] X. Zhang and R. Gupta, "Whole execution traces," in *Proc. Annual IEEE/ACM Int. Symp. Microarchitecture (MICRO '04)*. IEEE Computer Society, 2004, pp. 105–116.

[17] Z. Tian, Q. Zheng, T. Liu, M. Fan, E. Zhuang, and Z. Yang, "Software plagiarism detection with birthmarks based on dynamic key instruction sequences," *IEEE Trans. Software Engineering*, vol. 41, no. 12, pp. 1217–1235, 2015.

[18] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for Java," in *Proc. IEEE/ACM Int. Conf. Automated Software Engineering (ASE '07)*, 2007, pp. 274–283.

[19] S. Choi, H. Park, H. il Lim, and T. Han, "A static api birthmark for windows binary executables," *Journal of Systems and Software*, vol. 82, no. 5, pp. 862–873, 2009.

[20] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," in *Proc. ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD '06)*, 2006, pp. 872–881.

[21] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with jplag," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016–1038, 2002.

[22] Z. Tian, Q. Zheng, T. Liu, M. Fan, X. Zhang, and Z. Yang, "Plagiarism detection for multithreaded software based on thread-aware software birthmarks," in *Proc. Int. Conf. Program Comprehension (ICPC '14)*, 2014, pp. 304–313.

[23] I. Rigoutsos and A. Floratos, "Combinatorial pattern discovery in biological sequences: The teiresias algorithm." *Bioinformatics*, vol. 14, no. 1, pp. 55–67, 1998.

[24] [Online]. Available: DashO, https://www.preemptive.com/products/dasho.

[25] C. S. Collberg, E. Carter, S. K. Debray, A. Huntwork, J. D. Kececioglu, C. Linn, and M. Stepp, "Dynamic path-based software watermarking," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '04)*, 2004, pp. 107–118.

[26] Z. Tian, T. Liu, Q. Zheng, F. Tong, D. Wu, S. Zhu, and K. Chen, "Software plagiarism detection: A survey," *Journal of Cyber Security*, vol. 3, pp. 52–76, 2016.

[27] G. Myles and C. Collberg, "K-gram based software birthmarks," in *Proc. ACM Symp. Applied Computing (SAC '05)*, 2005, pp. 314–318.

[28] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," *ACM SIGPLAN Notices*, vol. 44, no. 3, pp. 97–108, 2009.

[29] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang, "Efficient deterministic multithreading through schedule relaxation," in *Proc. ACM Symp. Operating Systems Principles (SOSP '11)*. ACM, 2011, pp. 337–351.

[30] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[31] A. Wespi, M. Dacier, and H. Debar, "Intrusion detection using variable-length audit trail patterns," in *Int. Symp. Recent Advances in Intrusion Detection (RAID '00)*. Springer, 2000, pp. 110–129.

[32] C. Li, B. Wang, and X. Yang, "Vgram: Improving performance of approximate queries on string collections using variable-length grams," in *Proc. Int. Conf. Very Large Data Bases (VLDB '07)*. VLDB Endowment, 2007, pp. 303–314.

[33] Y.-C. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, "Value-based program characterization and its application to software plagiarism detection," in *Proc. Int. Conf. Softw. Eng. (ICSE '11)*, 2011, pp. 756–765.

[34] F. Zhang, Y. chan Jhi, D. Wu, P. Liu, and S. Zhu, "A first step towards algorithm plagiarism detection," in *Proc. Int. Symp. Software Testing and Analysis (ISSTA '12)*, 2012, pp. 111–121.

[35] Z. Tian, T. Liu, Q. Zheng, F. Tong, M. Fan, and Z. Yang, "A new thread-aware birthmark for plagiarism detection of multithreaded programs," in *Int. Conf. Software Engineering Companion (ICSE '16)*, 2016, pp. 734–736.

[36] Z. Tian, T. Liu, Q. Zheng, M. Fan, E. Zhuang, and Z. Yang, "Exploiting thread-related system calls for plagiarism detection of multithreaded programs," *Journal of Systems and Software*, vol. 119, pp. 136–148, 2016.

[37] D.-K. Chae, J. Ha, S.-W. Kim, B. Kang, and E. G. Im, "Software plagiarism detection: a graph-based approach," in *Pro. ACM Int. Conf. Information and Knowledge Management (CIKM '13)*. ACM, 2013, pp. 1577–1580.

[38] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *Proc. Int. Conf. Sof. Eng.(ICSE'14)*. New York, NY, USA: ACM, 2014, pp. 175–186.

[39] K. Chen, P. Wang, L. Y., W. X., N. Zhang, H. H., Z. W., and L. P., "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale," in *USENIX Security Symposium (USENIX Security'15)*, Washington, D.C., USA, 2015, pp. 659–674.

[40] Y. Qu, X. Guan, Q. Zheng, T. Liu, L. Wang, Y. Hou, and Z. Yang, "Exploring community structure of software call graph and its applications in class cohesion measurement," *Journal of Systems and Software*, vol. 108, pp. 193–210, 2015.

[41] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '05)*, 2005, pp. 190–200.

[42] [Online]. Available: ASM, http://asm.ow2.org/.

[43] I. Jonassen, J. F. Collins, and D. G. Higgins, "Finding flexible patterns in unaligned protein sequences." *Protein Science*, vol. 4, no. 8, pp. 1587–1595, 1995.

[44] M.-F. Sagot and A. Viari, "A double combinatorial approach to discovering patterns in biological sequences," in *Combinatorial Pattern Matching*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, vol. 1075, pp. 186–208.

[45] H. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics*, vol. 52, no. 1, pp. 7–21, 2005.

[46] [Online]. Available: GCJ, https://gcc.gnu.org/java/.

[47] X. Xie, F. Liu, B. Lu, and L. Chen, "A software birthmark based on weighted k-gram," in *IEEE Int. Conf. Intelligent Computing and Intelligent Systems (ICIS '10)*, 2010, pp. 400–405.

[48] B. W. Mathews, "Comparison of the predicted and observed secondary structure of T4 phage lysozyme," *Biochimica Et Biophysica Acta (bba) - Protein Structure*, vol. 405, pp. 442–451, 1975.

[49] Y.-C. Jhi, X. Jia, X. Wang, S. Zhu, P. Liu, and D. Wu, "Program characterization using runtime values and its application to software plagiarism detection," *IEEE Trans. Software Engineering*, vol. 41, no. 9, pp. 925–943, 2015.

[50] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[51] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. ACM Conf. Security and Privacy in Wireless and Mobile Networks (WiSec '14)*, 2014, pp. 25–36.

[52] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *IEEE Trans. Software Engineering*, vol. 28, no. 8, pp. 735–746, 2002.

[53] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large

scale source code," *IEEE Trans. Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[54] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: memory comparison-based clone detector," in *Proc. Int. Conf. Softw. Eng. (ICSE '11)*, 2011, pp. 301–310.

[55] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, "A semantics-based approach to malware detection," *ACM Trans. Programming Languages and Systems*, vol. 30, no. 5, pp. 25:1–25:54, 2008.

[56] S. Chaki, C. Cohen, and A. Gurfinkel, "Supervised learning for provenance-similarity of binaries," in *Proc. ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD '11)*, 2011, pp. 15–23.

[57] G. Cosma and M. Joy, "An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis," *IEEE Trans. Computers*, vol. 61, pp. 379–394, 2012.

[58] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proc. Working Conf. Mining Software Repositories (MSR '11)*, 2011, pp. 63–72.

[59] H. il Lim, H. Park, S. Choi, and T. Han, "A method for detecting the theft of Java programs through analysis of the control flow information," *Information and Software Technology*, vol. 51, no. 9, pp. 1338–1350, 2009.

[60] H. il Lim and T. Han, "Analyzing stack flows to compare Java programs," *IEICE Trans. Information and Systems*, vol. 95-D, no. 2, pp. 565–576, 2012.

[61] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proc. ACM SIGSOFT Symp. Found. Softw. Eng. (FSE '14)*, 2014, pp. 389–400.

[62] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *Proc. Eur. Symp. Research in Computer Security (ESORICS '12)*. Springer, 2012, pp. 37–54.

[63] H. Tamada, K. Okamoto, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Dynamic software birthmarks to detect the theft of windows applications," in *Int. Symp. Future Software Technology (ISFST '04)*, 2004, pp. 1–6.

[64] Z. Tian, Q. Zheng, M. Fan, E. Zhuang, H. Wang, and T. Liu, "DBPD: A dynamic birthmark-based software plagiarism detection tool," in *Int. Conf. Software Engineering and Knowledge Engineering (SEKE '14)*, 2014, pp. 740–741.

[65] F. Zhang, D. Wu, P. Liu, and S. Zhu, "Program logic based software plagiarism detection," in *IEEE Int. Symp. Software Reliability Engineering (ISSRE '14)*, 2014, pp. 66–77.

Zhenzhou Tian received the B.S. degree and Ph.D. degree in computer science and technology from Xi'an Jiaotong University, China, in 2010 and 2016, respectively. He is currently a lecturer in the School of Computer Science and Technology at Xi'an University of Posts and Telecommunications. His research interests include trustworthy software, software plagiarism detection, and software behavior analysis.

Ting Liu received his B.S. degree in information engineering and Ph.D. degree in system engineering from School of Electronic and Information, Xi'an Jiaotong University, Xi'an, China, in 2003 and 2010, respectively. Currently, he is an associate professor of the Systems Engineering Institute, Xi'an Jiaotong University. His research interests include smart grid, network security and trustworthy software.

Qinghua Zheng received the B.S. degree in computer software in 1990, the M.S. degree in computer organization and architecture in 1993, and the Ph.D. degree in system engineering in 1997 from Xi'an Jiaotong University, China. He was a postdoctoral researcher at Harvard University in 2002. He is currently a professor in Xi'an Jiaotong University, and the dean of the Department of Computer Science. His research areas include computer network security, intelligent e-learning theory and algorithm, multimedia e-learning, and trustworthy software.

Eryue Zhuang received the B.S. degree in software and microelectronics from Northwestern Polytechnical University, China, in 2014. She is currently working toward the M.S. degree in the Department of Computer Science and Technology at Xi'an Jiaotong University, China. Her research interests include trustworthy software and user behavior analysis.

Ming Fan received the B.S. degree in computer science and technology from Xi'an Jiaotong University, China, in 2013. He is currently working toward the Ph.D. degree in the Department of Computer Science and Technology at Xi'an Jiaotong University, China. His research interests include trustworthy software and malware detection of Android Apps.

Zijiang Yang is a professor in computer science at Western Michigan University. He holds a Ph.D. from the University of Pennsylvania, an M.S. from Rice University and a B.S. from the University of Science and Technology of China. Before joining WMU he was an associate research staff member at NEC Labs America. He was also a visiting professor at the University of Michigan from 2009 to 2013. His research interests are in the area of software engineering with the primary focus on the testing, debugging and verification of software systems. He is a senior member of IEEE.