Plagiarism Detection for Multithreaded Software Based on Thread-Aware Software Birthmarks

Zhenzhou Tian[1], Qinghua Zheng[1], Ting Liu[1]*, Ming Fan[1], Xiaodong Zhang[1], Zijiang Yang[2, 3]
[1] MOEKLINNS, Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China
[2] Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008, USA
[3] College of Computer and Technology, Xi'an University of Technology, 710048, China
{zztian,fanming.911025,oijiaoda}@stu.xjtu.edu.cn; {qhzheng,tingliu}@mail.xjtu.edu.cn; zijiang.yang@wmich.edu

ABSTRACT
The availability of inexpensive multicore hardware presents a turning point in software development. In order to benefit from the continued exponential throughput advances in new processors, the software applications must be multithreaded programs. 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 approaches remain optimized for sequential programs and cannot be applied to multithreaded programs without significant redesign. This paper fills the gap by presenting two dynamic birthmark based approaches. The first approach extracts key instructions while the second approach extracts system calls. Both approaches consider the effect of thread scheduling on computing software birthmarks. We have implemented a prototype based on the Pin instrumentation framework. Our empirical study shows that the proposed approaches can effectively detect plagiarism of multithread programs and exhibit strong resilience to various semantic-preserving code obfuscations.

Categories and Subject Descriptors
K.5.1 [**Legal Aspects of Computing**]: Hardware/Software Protection—*Copyrights, Licensing*; K.4.1 [**Computer and Society**]: Public Policy Issues—*Intellectual property rights*

General Terms
Experimentation, Security, Legal Aspects

Keywords
Software Birthmark, Plagiarism Detection, Multithreaded Program

1. INTRODUCTION
Software plagiarism is becoming a serious threat to the healthy development of the software industry. 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. Unfortunately software plagiarism is easy to implement but very difficult to detect. The unavailability of source code and the existence of powerful automated semantic-preserving code obfuscation tools [8] are a few reasons that make software plagiarism a daunting task. Nevertheless, researchers welcomed this challenge and developed effective methods. Software watermarking is one of the earliest and most widely adopted techniques. A watermark is a unique identifier embedded in a program before its distribution. Being hard to remove but easy to verify, watermarks can serve as a strong evidence for occurrences of software plagiarism. However, watermarks in a program may be eliminated by code obfuscations. It is also believed that a sufficiently determined attacker will eventually be able to defeat any watermark [7]. In order to address the problem, the concept of software birthmark was proposed. A birthmark is a set of characteristics extracted from a program that reflect the program's intrinsic properties and can be used to uniquely identify the program. As illustrated in [17], with proper algorithms birthmarks may identify software theft even after code obfuscations.

Despite the tremendous progress in software plagiarism detection technology, a new trend in software development greatly threatens its effectiveness. In recent years, from smartphones to servers, multicore processors are now ubiquitous. The availability of inexpensive multicore hardware presents a turning point in software development. In order for software applications to benefit from the continued exponential throughput advances in new processors, the applications must be multithreaded programs. 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 and cannot be applied to multithreaded without significant redesign.

Figure 1 shows a multithreaded program that is taken from a test case used in the WET [25] project with slight modifications. We apply two widely used software plagiarism detection approaches based on software birthmarks: Dynamic Key Instruction Sequence Birthmark (DKISB) [22] and System Call Short Sequence Birthmark (SCSSB) [24]. We execute the program multiple times under the same inputs. For each run we use DKISB or SCSSB to extract a software birthmark and then compare the similarity between the birthmarks across different runs. The similarity is computed using four different metrics, including Cosine distance, Jaccard index, Dice coefficient and Containment [22, 20, 6, 24], that are widely used in birthmark based plagiarism detection literature. According to its definition, a birthmark can uniquely

*Corresponding Author

ICPC'14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06...$15.00
http://dx.doi.org/10.1145/2597008.2597143

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include<stdlib.h>
#define N 8
pthread_t mThread[N];
void *run(void *data){
    int tid;
    tid =(int) data;
    printf("hello world from thread %d\n",tid);
     return NULL; }
int main(int argc, char *argv[]){
   int rc, i;
   int count;
   printf("input a number please: \n");
   scanf("%d",&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=0;i<N; i++)
       pthread_join(mThread[i], NULL);
   printf("main thread finished\n");
   return 0; }
```

Figure 1. A simple multithreaded program

identify the program from which the birthmark is extracted. Therefore, we expect highly similar birthmarks as we are executing the same program under the same inputs. That is, we expect current approaches to claim plagiarism in this experiment.

However, as shown in Table 1, the data contradict what we have expected. For DKISB, the similarity scores are between 0.55 and 0.85. As for SCSSB, no score is greater than 0.55. In most literature, a similarity score above 0.8 usually means definite plagiarism and a score below 0.2 usually means definite independent programs. Therefore, the widely used birthmark-based software plagiarism detection techniques fail to declare plagiarism on identical programs.

The example illustrates that the existing dynamic birthmark based approaches are inadequate in identifying plagiarism of multi-threaded programs because they neglect the effect of thread scheduling. Program behavior is deterministically determined by system inputs, including I/O, DMA, interrupts, in sequential programs. Thus the executions of highly similar programs under the same input should be very similar. This assumption no longer holds for multithreaded programs because 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 renders the existing approaches ineffective.

TABLE 1. Similarity scores calculated with four metrics for DKISBs and SCSSBs of multiple runs of the sample program

	DKISB	SCSSB
Cosine Distance	0.838	0.452
Jaccard Index	0.551	0.369
Dice Coefficient	0.678	0.51
Containment	0.735	0.477

In this paper, we present thread-aware algorithms that effectively detect plagiarism of multithreaded programs at the binary level. Unlike many existing approaches [14, 19, 11] that require source code, our approach uses binary because source code is usually unavailable when birthmark techniques are used to obtain the initial evidence of software plagiarism. We name our two approaches TW-DKISB (Thread Aware Dynamic Key Instruction Sequence Birthmark) and TW-SCSSB (Thread Aware System Call Short Sequence Birthmark) that amend the existing approaches of DKISB and SCSSB, respectively. We exploit two models to abstract the thread information during birthmark extraction. The similarity of birthmarks is computed using two matching algorithms on the four metrics, *i.e.* Cosine Distance, Jaccard Index, Dice Coefficient and Containment [22, 20, 6, 24].

We have implemented a prototype and conducted experiments on 134 versions of 24 multithreaded programs. The preliminary results show that our approach is effective for multithreaded software plagiarism detection. In addition, our approach exhibits strong resilience to both weak obfuscations obtained by various compiler optimizations, and strong obfuscations supported by obfuscators such as SandMark [8] and Allatori [3].

The remainder of the paper is organized as follows. Section 2 introduces necessary concepts and describes our methods to extracting and comparing birthmarks. A prototype overview is also briefly described at the end of this section. Section 3 presents the empirical study, followed by the related works in Section 4. Finally we conclude the paper in Section 5.

2. THREAD AWARE BIRTHMARKS BASED PLAGIARISM DETECTION

2.1 Software Birthmarks

A software birthmark is a set of characteristics extracted from a program that reflects intrinsic properties of the program. Depending on whether its extraction relies on program runs, a software birthmark can be either considered static or dynamic. Generated mainly by analyzing syntactic features, static birthmarks tend to overlook operational behaviors of a program. As a result, they are ineffective against sematic-preserving obfuscations that can modify the syntactic structure of a program. 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. The approaches proposed in this paper are based on dynamic software birthmarks whose classical definition is given below.

Definition 1. (Dynamic Software Birthmark [21]) Let p, q be two programs or program components. Let I be an input to p and q. Let $f(p,I)$ be a set of characteristics extracted from p when executing p with input I. Then $f(p,I)$ is a dynamic birthmark of p only if both of the following conditions are satisfied:

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

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.

Definition 2. (Thread-Aware Dynamic Software Birthmark) Let p,q be two multithreaded programs or program components. 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 input I and thread schedule s. Then $f(p,I,s)$ is a dynamic birthmark of p 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 $p \Rightarrow f(p,I,s) = f(q,I,s)$.

2.2 Birthmarks for Individual Threads

Similar to Definition 1, Definition 2 provides an abstract guideline without considering implementation. In practice it is very difficult to predetermine a thread schedule and enforce the scheduling. Therefore instead of enforcing thread schedules in our algorithms, we try to shield their influence on executions. In order to do so, we annotate each event, either a system call or a key instruction, in an execution trace with thread identifier. We then project the trace on thread identifiers to obtain sub-traces, each of which belongs to a single thread. The birthmarks are extracted from the sub-traces that can remain same even under different thread schedules.

Formally, an execution trace $trace(p,I) = \langle e_1, e_2, \cdots, e_n \rangle$ is an ordered set, in which e_i ($1 \le i \le n$) is an instance of either a system call or a key instruction, along with the thread identifier that executes the instance. A key instruction is both value-updating (whose execution generates new values rather than migrate values, such as add and xor) and input-correlated (whose execution propagates taints from program inputs). Detailed description about key instructions and system calls, as well as the reasons that they are suitable for software birthmark generation are discussed in [22] and [24]. We use *e.in* and *e.tid* to denote the instance and thread identifier at an event e, respectively.

Definition 3. (Thread Slice) Given an execution trace $trace(p,I)$, we define its projection on thread t to be an ordered sub-set $Slice(p,I,t) = \langle e_i \mid e_i \in trace(p,I) \wedge e_i.tid = t \rangle$ of $trace(p,I)$. The projections of all the threads appearing in the trace form a partition of $trace(p,I)$, and each sub-set $Slice(p,I,t)$ is called a thread slice.

Definition 4. (Thread-Slice Birthmark) Let $Slice(p,I,t) = \langle e_1, e_2, \cdots, e_n \rangle$ be a thread slice of thread t when executing program p with input I. Let $Set(p,I,t,k) = \{ g_j \mid g_j = \langle e_j, e_{j+1}, \cdots, e_{j+k-1} \rangle, j \in \{1,2,\cdots,n-k+1\} \}$ be a set of k-grams generated by applying the k-gram algorithm [18] on the slice. We call the key-value pair set $Birth_p^I(k,t) = \{ \langle g_j', freq(g_j') \rangle \mid g_j' \in Set(p,I,t,k), and\ \forall j_1 \neq j_2, g_{j_1}' \neq g_{j_2}' \}$, where $freq(g_j')$ represents the frequency of g_j' occurred in $Set(p,I,t,k)$, as the thread-slice birthmark of $Slice(p,I,t)$.

2.3 Generation of Program Birthmarks

With the availability of the birthmarks for individual threads, we present two models, Slice Aggregation (SA) and Slice Set (SS), to generate software birthmarks for a multithreaded program. The SA model generates program birthmarks by aggregating all thread birthmarks into a single set of key-value pairs, where the keys are the unique k-grams obtained from all possible elements in each thread-slice birthmark, and the values are frequencies of correspondingly unique k-grams. If a key is owned by multiple thread-slice birthmarks, its frequencies are added to be the new value of the key. The SS model simply treats the key-value pair consisting of thread identifier and the slice birthmark as each element comprising the final program birthmark. Formally, the definition of the two model are described as follows:

Definition 5. (Slice Aggregation Model) The slice aggregation model is a map $f : SB \to PB$, where :

- $SB = \{ Birth_p^I(k,t) \mid 0 \le t \le m, t \in \mathbb{N} \}$ is the set of thread-slice birthmarks and m is the number of threads in the recorded trace.
- $PB = \cup_{sb} Birth_p^I(k,t)$ is the software birthmark of program p with input I, where t is the thread identifier of sb.
- For each element $\langle g_i, freq(g_i) \rangle \in PB$, the frequency of g_i is calculated as $freq(g_i) = \sum_{sb} freq(g_j)$ where $sb \in SB, g_j \in sb, and\ g_j = g_i$.

Definition 6. (Slice Set Model) The slice set model is a map $g : SB \to PB$, where:

- $SB = \{ Birth_p^I(k,t) \mid 0 \le t \le m, t \in \mathbb{N} \}$ is the set of thread-slice birthmarks and m is the number of threads in the recorded trace.
- $PB = \{ (t, Birth_p^I(k,t)) \mid sb \in SB \}$ is the software birthmark of program p with input I, where t is the thread identifier of each corresponding sb.

Based on the above discussions, the definition of TW-DKISB and TW-SCSSB can be formally described as follows.

Definition 7. (TW-DKISB and TW-SCSSB) Let $trace(p,I) = \langle e_1, e_2, \cdots, e_n \rangle$ be an execution trace and its corresponding thread-slice birthmark set be $SB = \{ Birth_p^I(k,t) \mid 0 \le t \le m \}$. Then program birthmark PB can be generated by applying either the SA model $f : SB \to PB$ or the SS model $g : SB \to PB$. We call the program birthmark PB as:

- TW-DKISB, if each element $e_i \in trace(p,I)$ is a key instruction.
- TW-SCSSB, if each element $e_i \in trace(p,I)$ is a system call.

Example 1. Let's take the trace of system calls as an example to illustrate the process of generating TW-SCSSB. Suppose the following trace is recorded when executing program *p* with *I*.

$$trace(p,I)=\langle(t_1,open),(t_1,read),(t_1,write),(t_1,read),(t_2,read),(t_2,write),(t_1,close)\rangle$$

It can be observed that seven system calls were executed by two threads t_1 and t_2. According to the definition of thread slice, this trace can be split into the following two slices: $slice(t_1)=\langle(t_1,open),(t_1,read),(t_1,write),(t_1,read),(t_1,close)\rangle$ $slice(t_2)=\langle(t_2,read),(t_2,write)\rangle$.

The generated k-gram sets when $k=2$ are: $Set(p,I,t_1)=\{(open,read),(read,write),(write,read),(read,close)\}$ and $Set(p,I,t_2)=\{(read,write)\}$; the corresponding thread-slice birthmarks are: $Birth_p^I(2,t_1)=\{\langle(open,read),1\rangle,\langle(read,write),1\rangle,\langle(write,read),1\rangle,\langle(read,close),1\rangle\}$ and $Birth_p^I(2,t_2)=\{\langle(read,write),1\rangle\}$.

Finally, the TW-SCSSB of program *p* with input *I* generated with SA and SS model are respectively:

$$TW-SCSSB_{SA}(p,I,2)=\{\langle(open,read),1\rangle,\langle(read,write),2\rangle,\langle(write,read),1\rangle,\langle(read,close),1\rangle\}$$

$$TW-SCSSB_{SS}(p,I,2)=\{\langle t_1,Birth_p^I(2,t_1)\rangle,\langle t_2,Birth_p^I(2,t_2)\rangle\}.$$

2.4 Similarity Calculation

In the literature of birthmark based software plagiarism detection, the similarity between two programs is measured by the similarity of their birthmarks. In general, birthmarks mainly exist in three forms: sequences, sets and graphs. There are many methods for calculating similarity of sets that are widely adopted in the field of information retrieval, including Dice coefficient [6], Jaccard index [20], and Cosine distance [16]. Computing the similarity of graphs is relatively more complex. It is conducted by either graph monomorphism or isomorphism algorithms [5, 14] or translating a graph into a vector using algorithms such as random walk with restart [4]. In our work, we explore different methods to calculate the similarity of birthmarks generated with the SA and SS model.

2.4.1 Similarity Calculation Method for Birthmarks Generated with SA model

According to the definition of the SA model, the generated TW-DKISBs or TW-SCSSBs are in the form of key-value pair set, therefore similarity computation methods such as Cosine distance, Jaccard index, Dice coefficient and Containment can be used (It is worth nothing that all the four metrics have been used to compute birthmark similarities in previous studies. To prevent favoritism, we provide formal definitions of the four modified metrics, and implement all of them in our prototype to properly compare the performance with others' as shown in Section 3). However, since these metrics do not consider frequency of the elements, two different birthmarks may have the same result. In contrast, frequencies of k-grams in a birthmark are taken into consideration in our similarity calculation method. Specifically, each traditional metric is multiplied by a factor θ that reflects frequency similarity between two birthmarks. In the following we illustrate how to compute the factor θ:

For software birthmarks $A=\{\langle k_1,v_1\rangle,\langle k_2,v_2\rangle,\cdots,\langle k_n,v_n\rangle\}$ and $B=\{\langle k_1',v_1'\rangle,\langle k_2',v_2'\rangle,\cdots,\langle k_m',v_m'\rangle\}$, which may be either two TW-DKISBs or two TW-SCSSBs, let $S=keySet(A)\cup keySet(B)$. We construct a vector $\vec{A}=(a_1,a_2,\cdots,a_l)$, in which each element $a_i=\begin{cases}v_i, if\ S_i\in keySet(A)\\0, if\ S_i\notin keySet(A)\end{cases}$, where $1\le i\le l$ and v_i is the value of key S_i in A. Likewise $\vec{B}=(b_1,b_2,\cdots,b_l)$ can be constructed. Thus

$$\theta=\frac{min\left(\left|\vec{A}\right|,\left|\vec{B}\right|\right)}{max\left(\left|\vec{A}\right|,\left|\vec{B}\right|\right)},\ where\ \left|\vec{A}\right|=\sqrt{\sum_{a_i\in\vec{A}}a_i^2},\ \left|\vec{B}\right|=\sqrt{\sum_{b_i\in\vec{B}}b_i^2}.$$

The modified metrics are defined as following:

$$Ex-Cosine(A,B)=\frac{\vec{A}\bullet\vec{B}}{\left|\vec{A}\right|\left|\vec{B}\right|}\times\theta;\quad Ex-Jaccard(A,B)=\frac{|A\cap B|}{|A\cup B|}\times\theta;$$

$$Ex-Dice(A,B)=\frac{2|A\cap B|}{|A|+|B|}\times\theta;\quad Ex-Containment(A,B)=\frac{|A\cap B|}{|A|}\times\theta;$$

And the similarity of two SA generated birthmarks can be calculated with $Sim(A,B)=sim_c(A,B)$, where $c\in\{Ex-Cosine, Ex-Jaccard, Ex-Dice, Ex-Containment\}$.

2.4.2 Similarity Calculation Method for Birthmarks Generated with SS model

For software birthmarks generated by the SS model, we reduce the problem of calculating their similarity into finding a maximum weighted bipartite matching as illustrated in Figure 2. 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 $A=\{(t_1,Birth(t_1)),(t_2,Birth(t_2)),\cdots,(t_m,Birth(t_m))\}$ and $B=\{(t_1',Birth(t_1')),(t_2',Birth(t_2')),\cdots,(t_n',Birth(t_n'))\}$ be two TW-DKISBs or TW-SCSSBs, where birthmark *A* contains *m* thread-slice birthmarks and birthmark *B* contains *n* thread-

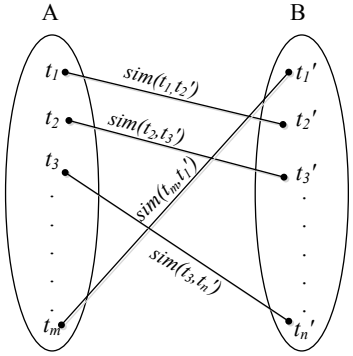

Figure 2. Schematic diagram of bipartite matching modeling for birthmarks generated by the SS model

slice birthmarks. A $m \times n$ similarity matrix is generated by comparing every thread-slice birthmark in A to each thread-slice birthmark in B using either of the four metrics listed in 2.4.1.

$$SimMatrix(A,B)=\begin{pmatrix} sim_c(t_1,t_1') & sim_c(t_1,t_2') & \cdots & sim_c(t_1,t_n') \\ sim_c(t_2,t_1') & sim_c(t_2,t_2') & \cdots & sim_c(t_2,t_n') \\ \vdots & \vdots & \ddots & \vdots \\ sim_c(t_m,t_1') & sim_c(t_m,t_2') & \cdots & sim_c(t_m,t_n') \end{pmatrix},$$

where $c \in \{Ex-Cosine, Ex-Jaccard, Ex-Dice, Ex-Containment\}$. Then 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 $l=min(m,n)$, $u_i \in keyset(A)$, $v_i \in keyset(B)$, $u_i \neq u_j$ *if* $i \neq j$, $v_i \neq v_j$ *if* $i \neq j$, and $\sum_i^l sim_c(u_i,v_j)$ has the maximum value among all the matchings. Finally, the similarity of two SS model generated software birthmarks are calculated with the following formula:

$$Sim(A,B)=\frac{\sum_{(t_i,t_j')\in MaxMatch(A,B)} sim_c(t_i,t_j')\times(count(t_i)+count(t_j'))}{\sum_{i=1}^{m} count(t_i)+\sum_{j=1}^{n} count(t_j')}$$

where $count(t_i)=|keySet(Birth(t_i))|$, $count(t_j')=|keySet(Birth(t_j'))|$.

Example 2. We take TW-SCSSBs generated by the SS model as an example to illustrate the process of similarity comparison. Suppose the following trace is recorded when executing program p' with I, where p' is a copy of program p shown in Example 1. $trace(p',I)=\langle(t_1',open),(t_2',read),(t_1',read),(t_2',write),(t_1',write),(t_1',read),(t_1',close)\rangle$.

For this trace, the TW-SCSSB generated using the SS model is: $TW-SCSSB_{SS}(p',I,2)=\{\langle t_1', Birth_{p'}^I(2,t_1')\rangle,\langle t_2', Birth_{p'}^I(2,t_2')\rangle\}$ where $Birth_{p'}^I(2,t_1')=\{\langle(open,read),1\rangle,\langle(read,write),1\rangle,\langle(write,read),1\rangle,\langle(read,close),1\rangle\}$ and $Birth_{p'}^I(2,t_2')=\{\langle(read,write),1\rangle\}$.

Assume *Ex-Jaccard* is used as the metric to compute the similarity between each slice birthmark of the two TW-SCSSBs, that is $sim_{Ex-Jaccard}(t_1,t_2')=\frac{|Birth_p^I(2,t_1)\cap Birth_{p'}^I(2,t_2')|}{|Birth_p^I(2,t_1)\cup Birth_{p'}^I(2,t_2')|}\times\theta=\frac{1}{8}$. It leads to a similarity matrix $\begin{pmatrix}1 & 0.125\\ 0.125 & 1\end{pmatrix}$, base on which a maximum matching can be found that $MaxMatch=\{(t_1,t_1'),(t_2,t_2')\}$. Finally, similarity of the two birthmarks generated by the SS model, $A=TW-SCSSB_{ss}(p,I,2)$ and $B=TW-SCSSB_{ss}(p',I,2)$, is computed as $Sim(A,B)=\frac{1\times 8}{10}+\frac{1\times 2}{10}=1$.

2.5 Plagiarism Detection

The purpose of extracting birthmarks and calculating their similarity is to eventually determine whether there exists plagiarism. Considering that there are other random factors such as DMA and interrupts besides thread scheduling, multiple similarity scores are computed by executions under multiple inputs. The average of the scores is taken as an evidence of plagiarism.

Formally, Let P_A and P_B be two programs under test. Let the birthmarks extracted from the programs using either the SS model or the SA model under the inputs $I_1, I_2, \cdots, I_n$ be $A_1, A_2, \cdots, A_n$ and $B_1, B_2, \cdots, B_n$ respectively. The similarity between P_A and P_B can be calculated by $Sim(P_A,P_B)=\frac{\sum_{j=1}^{n} Sim(A_j,B_j)}{n}$, whose value is between 0 and 1. We then decide whether there exists plagiarism according to the similarity scores and a threshold ε as follows:

$$Sim(P_A,P_B)=\begin{cases}\geq 1-\varepsilon & P_A,P_B \text{ are classified as copies}\\ \leq \varepsilon & P_A,P_B \text{ are classified as independent}\\ otherwise & inconclusive\end{cases}$$

2.6 Tool Overview

Figure 3 depicts the overview of our prototype in which we have implemented all the techniques discussed in this section. Given the plaintiff (original program) and the defendant (suspicious program) in binary code, and a set of inputs, our prototype executes both programs with the same input one by one. Meantime, the dynamic analysis module monitors the executions and identifies the key instructions and the system calls in real time. The monitoring produces two sequences. After sequences of both plaintiff and defendant programs are available, they are fed into the pre-processor to filter out noise and generate formatted elements that constitute valid execution traces. Then the birthmark generator performs thread projection, extracts thread-slice birthmarks and generates TW-DKISBs and TW-SCSSBs with either the SA or the SS model. Next the similarity scores are computed between two TW-DKISBs or two TW-SCSSBs by the similarity calculator. Finally, the plagiarism decider judges whether the defendant is innocent or guilty according to the average of similarity scores computed under different inputs and the given threshold ε.

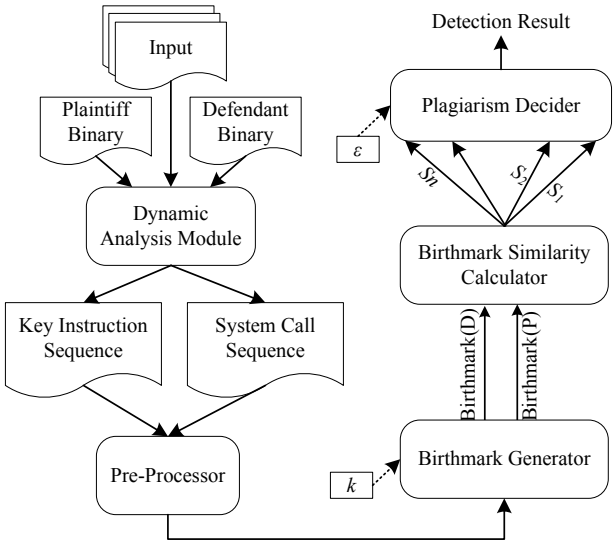

Figure 3. Overview of thread-aware birthmarks based plagiarism detection system

3. EXPERIMENTS AND EVALUATION

A high quality birthmark manifests in that the ratio of false classifications (both inconclusive and incorrectly classified cases are treated as false classifications) should be low enough for a specific ε. To be specific, the similarity scores calculated between a program and its derivative versions generated by applying semantic-preserving code transformations should be high enough so as to recognize copies, while scores between independently developed programs should be low enough to distinguish them. Generally in the literature, the following two properties of a birthmark should be satisfied to make it valid. We restate them by referring to the descriptions of Myles [18] and Seokwoo [6].

Property 1. (Resilience) Let p be a program and p' be a derivative version generated by applying semantic-preserving code transformation τ to p. We say a birthmark B_p is resilient to τ if $Sim(p, p') \geq 1 - \varepsilon$.

Property 2. (Credibility) Let p and q be independently developed programs which may accomplish the same task. We say a birthmark B_p is credible if $Sim(p, q) \leq \varepsilon$.

In the following sections, we firstly evaluate the TW-DKISBs and the TW-SCSSBs generated by the SA and SS models against the above two properties to check if they are valid software birthmarks. The quality of our birthmarks are further compared with other birthmarks using the AUC metric. It should be noted that either TW-DKISBs or TW-SCSSBs are generated with *k-gram* algorithm, which means the birthmarks are different when choosing different values of k. However, as it has been confirmed in most previous papers [22, 24, 20] where *k-gram* is also used to generate birthmarks, setting the value of k to be 4 or 5 is a proper compromise between accuracy and efficiency. Hence, all experiments conducted in the following adopt a k value of 5.

3.1 Validation of Resilience Property

3.1.1 Resiliency to Different Compilers and Optimization Levels

A software plagiarist may try to evade detection by choosing a different compiler or changing compiler optimization levels. This can be considered as a relatively weak semantic preserving code transformation technique. We choose a multithreaded compression software pigz-2.3 as the experimental object to evaluate the resilience property of our thread-aware birthmarks against different compilers and optimization levels.

In the experiment, two different compilers LLVM3.2 and GCC4.6.3 are used to compile the source code of pigz with five optimization levels (-O0, -O1, -O2, -O3 and -Os) and the debug option (-g) switched on or off, thus generating totally 20 different executables. The statistical characteristics of the binaries are collected using the disassembler IDA Pro. Table 2 shows some statistical differences (maximum value, minimum value, average value, and the standard deviation) among the executables. It can be observed that these binaries vary significantly with respect to the characteristics listed in the table headings (size of the binary, number of functions, number of instructions, number of blocks and number of calls).

TABLE 2. Statistical differences for all versions of pigz generated with different compilers and optimization levels

	Size(kb)	NoF.	NoI.	NoB.	NoC.
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

Each pair of the generated binaries of pigz are executed with the same input. TW-DKISBs and TW-SCSSBs are generated and their similarity is computed. As discussed in Section 2.5, results based on a single input may not be credible, therefore we conducted experiments with 18 different inputs (all the experiments conducted below are conducted with multiple inputs by default). To measure the resilience capability, the default threshold value $\varepsilon = 0.25$ is used.

Table 3(a) summarizes the results of the experiment, where the heading "Avg." indicates the average similarity score of all comparison pairs, "Min." indicates the minimum value computed among all pairs, and "Acc." indicates the accuracy of our prototype. The accuracy is calculated by the ratio of pairs classified as copies. It can be observed that TW-DKISBs generated by either SA or SS model are very accurate regardless of what types of similarity metrics are adopted. In particular, the accuracy is nearly 1.0 when Ex-Cosine is selected to compute similarity. TW-SCSSBs exhibit similar accurate results as TW-DKISBs except for the Ex-Jaccard metric where an accuracy of merely 0.47 is reached. This is due to the fact that most of the similarity scores for this case are around 0.7 with a threshold value $\varepsilon = 0.25$.

3.1.2 Resiliency to Special Obfuscation Tools

In this section, we evaluate the resilience of the thread-aware birthmarks against advanced obfuscation techniques available in sophisticated tools. Unfortunately the only binary obfuscator we found is a commercial obfuscator called CloakWare Security Suite. Publically available obfuscators or to say packers such as Upx and WinUpack only implement code compression, encryption and packing obfuscations. Executables processed with these techniques will become rather different, bringing great challenge to static birthmark based approaches, since an unpack process is needed first to restore the original binary executables before analysis. However, since executables processed with them must be decompressed or decrypted during runtime in order to be executed, making dynamic birthmarks have innate immunity to them. Therefore in our empirical study we choose the Java byte code obfuscation tool SandMark [8] that implements a series of advanced semantic-preserving transformation techniques. We use it with 15 application obfuscations, 7 class obfuscations and 17 method obfuscations to generate a group of obfuscated versions. Several commercial and open source obfuscators including Allatori, DashO, Jshrink, ProGuard and RetroGround (in the following we call these tools Allatori-Series) are also selected to generate semantic equivalent derivative versions since the latest SandMark supports Java 1.4 only. In addition, because our system works on binary executables, obfuscated versions are converted to x86 executables with GCJ, the GNU ahead-of-time compiler for java.

TABLE 3. Experimental results for resilience evaluation of TW-DKISBs and TW-SCSSBs

(a) Resilience evaluation against obfuscations caused by different compilers and optimization levels

	TW-DKISB						TW-SCSSB					
	SA Model			SS Model			SA Model			SS Model		
	Avg.	Min.	Acc.	Avg.	Min.	Acc.	Avg.	Min.	Acc.	Avg.	Min.	Acc.
Ex-Cosine	1	1	1	1	1	1	0.921	0.875	1	0.912	0.862	1
Ex-Jaccard	0.917	0.656	0.832	0.945	0.747	0.937	0.752	0.623	0.474	0.759	0.626	0.474
Ex-Dice	0.95	0.785	1	0.967	0.849	1	0.851	0.765	1	0.85	0.761	1
Ex-Containment	0.95	0.698	0.916	0.972	0.747	0.968	0.851	0.758	0.997	0.85	0.758	0.997

(b) Resilience evaluation against obfuscations provided by SandMark

	TW-DKISB						TW-SCSSB					
	SA Model			SS Model			SA Model			SS Model		
	Avg.	Min.	Acc.	Avg.	Min.	Acc.	Avg.	Min.	Acc.	Avg.	Min.	Acc.
Ex-Cosine	0.999	0.974	1	0.997	0.921	1	0.994	0.979	1	0.967	0.942	1
Ex-Jaccard	0.965	0.928	1	0.969	0.920	1	0.864	0.757	1	0.908	0.844	1
Ex-Dice	0.982	0.959	1	0.984	0.948	1	0.924	0.855	1	0.939	0.890	1
Ex-Containment	0.984	0.977	1	0.986	0.962	1	0.925	0.849	1	0.939	0.882	1

(c) Resilience evaluation against obfuscations provided by Alloatori-Series

	TW-DKISB						TW-SCSSB					
	SA Model			SS Model			SA Model			SS Model		
	Avg.	Min.	Acc.	Avg.	Min.	Acc.	Avg.	Min.	Acc.	Avg.	Min.	Acc.
Ex-Cosine	0.997	0.963	1	0.993	0.893	1	0.993	0.972	1	0.955	0.870	1
Ex-Jaccard	0.929	0.818	1	0.930	0.774	1	0.786	0.568	0.818	0.842	0.653	0.966
Ex-Dice	0.962	0.889	1	0.959	0.826	1	0.875	0.723	0.967	0.897	0.755	1
Ex-Containment	0.967	0.920	1	0.966	0.846	1	0.877	0.653	0.933	0.898	0.749	0.999

ConGzip and OffBzip2, two java programs that accomplish multi-threaded gzip and bzip2 compression respectively using the java zip library and At4J [2] library, are developed and selected as the experimental objects. Each obfuscation technique implemented in the SandMark and Allatori-Series is applied to the two programs to generate a series of obfuscated versions. To ensure the semantic equivalence after these transformations, all obfuscated versions are tested with a set of inputs to check for correctness, and failed ones are removed. Finally, 58 different versions are generated for ConGzip, including 29 SandMark obfuscated versions and 29 Allatori-Series obfuscated ones. Only 35 different versions are generated for OffBzip2, since SandMark failed to apply any of its transformations due to its poor support for new java features used in OffBzip2.

Next the thread-aware birthmarks are extracted and their similarity scores are computed between each of the original program and its obfuscated versions. Table 3(b) and 3(c) show the experimental results of ConGzip with respect to SandMark and Allatori-Series respectively. We can see that both TW-DKISBs and TW-SCSSBs are accurate (with a lowest value of 0.818), and the average scores vary from 0.8 to 1.0. Similar results are observed in the experiments of OffBzip2. It indicates that our thread-aware birthmarks can correctly identify almost all the derivatives generated with various semantic-preserving code transformations. These results are strong evidence that our birthmarks are resilient and robust.

3.2 Validation of Credibility Property

In the following experiments, credibility of TW-DKISBs and TW-SCSSBs is evaluated by checking the capability of distinguishing independently developed programs. Three types of software that are widely used in Linux are selected as the experimental objects, including 6 multithreaded compression software (lbzip2, lrzip, pbzip2, pigz, plzip and rar), 10 web browsers (arora, chromium, dillo, Dooble, epiphany, firefox, konqueror, luakit, midori and seaMonkey), and 5 audio players (cmus, mocp, mp3blaster, mplayer and sox).

Although software of the same categories usually overlap greatly in their functionalities, they can be rather different at the source code level if implemented independently due to different algorithms adopted, different design patterns applied, different coding styles, etc.

In the experiments conducted, only TW-SCSSBs are evaluated for the browsers, since the recorded key instructions are too large to handle. But for the audio players and compression programs, both TW-DKISBs and TW-SCSSBs are evaluated.

TABLE 4. Experimental results for credibility evaluation of TW-DKSIBs and TW-SCSSBs

(a) Credibility evaluation of TW-SCSSBs using 10 web browsers

	SA Model					SS Model				
	Avg.	Max.	Acc.	Avg+.	Avg-.	Avg.	Max.	Acc.	Avg+.	Avg-.
Ex-Cosine	0.127	0.57	0.8	0.323	0.081	0.125	0.57	0.8	0.311	0.08
Ex-Jaccard	0.07	0.36	0.933	0.188	0.037	0.057	0.365	0.956	0.147	0.03
Ex-Dice	0.112	0.496	0.822	0.278	0.07	0.089	0.485	0.889	0.223	0.051
Ex-Containment	0.118	0.588	0.822	0.281	0.078	0.096	0.588	0.878	0.226	0.06

(b) Credibility evaluation using software of different categories

	TW-DKISB						TW-SCSSB					
	SA Model			SS Model			SA Model			SS Model		
	Avg.	Max.	Acc.	Avg.	Max.	Acc.	Avg.	Max.	Acc.	Avg.	Max.	Acc.
Ex-Cosine	0.083	0.631	0.873	0.071	0.61	0.891	0.108	0.611	0.818	0.1	0.616	0.836
Ex-Jaccard	0.025	0.181	1	0.018	0.165	1	0.043	0.269	0.964	0.042	0.305	0.982
Ex-Dice	0.042	0.283	0.945	0.03	0.26	0.982	0.072	0.418	0.891	0.066	0.435	0.909
Ex-Containment	0.065	0.336	0.927	0.051	0.338	0.982	0.077	0.481	0.909	0.072	0.485	0.936

The experimental results are very accurate for both TW-DKISBs and TW-SCSSBs. Due to space limitation we list results for the web browsers only.

As summarized in Table 4(a) for the web browsers, the accuracy values (defined by the ratio of pairs classified as independent) for TW-SCSSBs generated by either the SA or SS model are all very high regardless of what types of similarity metrics are adopted. The corresponding average scores are all around 0.1. Also, the maximum values computed under each scheme are counted. Besides, we note that five of the browsers (arora, Dooble, epiphany, luakit and midori) are Webkit-based while the others utilize different layout engines. Therefore we also count the average similarity scores just between these five Webkit-based browsers indicated by "Avg+", and the average similarity scores between the Webkit-based and non-Webkit-based browsers indicated by "Avg-". As expected, the values in the "Avg+" columns are four to five times bigger than values in the "Avg-" columns. It indicates that our birthmarks have the potential to recognize shared modules used in different programs, implying the possibility of applying them to library or partial plagiarism detection.

In this section, similarity between the 6 compression programs and the 5 audio players are computed with respect to each scheme, to evaluate the credibility of TW-DKISBs and TW-SCSSBs against independently implemented software in different categories. Similarly, the average scores, the maximum scores and the accuracies are computed and summarized in Table 4(b). The results are good enough to prove the credibility of our birthmarks.

3.3 Comparison with Traditional Birthmarks

3.3.1 Performance Evaluation Metric

To compare the performance of our birthmarks with others, we utilize the AUC metric adopted in [9], namely the area under the F-Measure curve. However, they model the problem of plagiarism detection as a binary decision problem (guilty and innocent), while the detection result given by our system may be guilty, innocent and inconclusive according to the formula in Section 2.5. So based on the implication of precision and recall, the precision and recall for our plagiarism detection method are customized to the following definitions:

$$Precision = \frac{|EP \cap JP| + |EI \cap JI|}{|JP| + |JI|}\ ;\ Recall = \frac{|EP \cap JP| + |EI \cap JI|}{|EP| + |EI|}$$

where EP represents the set of comparison pairs that have plagiarism, and JP represents the set of comparison pairs that are judged plagiarism by our tool. Similarly, EI represents the set of comparison pairs that are independent, and JI represents the set of comparison pairs that are judged as independent by our tool. The weighted harmonic mean of precision and recall, namely the F-Measure metric, is defined as:

$$F - Measure = \frac{2 \times Precision \times Recall}{Precision + Recall}.$$

As mentioned in Section 2.5, the detection result of our approach relies on the adopted value of threshold ε. By increasing the value of ε from 0 to 0.5, we can correspondingly draw an F-Measure curve. And the area under the curve (AUC) is used as the metric to evaluate the performance of our birthmarks against others over the entire space of threshold ε.

3.3.2 Comparison Result

In this group of experiments, performance of TW-DKISBs and TW-SCSSBs are compared with traditional DKISBs and SCSSBs. All the comparison pairs that appear in Section 3.1 and 3.2 are taken as the experimental objects, which means all comparisons conducted in Section 3.1 to validate the resilience property with various semantic-preserving transformation generated copies constitute the set EP, and all comparisons conducted in Section 3.2 to validate the credibility property with various independently developed programs constitute the set EI. Similarly, JP is consisted of comparison pairs detected as copies among all pairs, and JI consists of pairs detected as independent.

TABLE 5. AUC analysis results

	$TW-DKISB_{SA}$	$TW-DKISB_{SS}$	$DKISB$	$TW-SCSSB_{SA}$	$TW-SCSSB_{SS}$	$SCSSB$
Ex – Cosine	0.93	0.934	0.93	0.935	0.926	0.933
Ex – Jaccard	0.887	0.908	0.874	0.682	0.744	0.537
Ex – Dice	0.907	0.923	0.9	0.804	0.828	0.721
Ex – Containment	0.909	0.923	0.9	0.807	0.829	0.721

Due to space limitations, we only give the F-Measure curves for system call sequence based birthmarks, as depicted in Figure 4. There are four subfigures, each of which corresponds to one of the four metrics selected to compute similarity. In each subfigure, the lines in green and red represent the F-Measure curves of TW-SCSSBs generated with the SA model and the SS model respectively. The blue lines represent the F-Measure curve plotted for SCSSB. It can be observed that the two TW-SCSSBs do not exhibit significant difference in their F-Measure values, but both are greater than that of SCSSBs' across the whole x-axis in each subfigure except for the Ex-Cosine metric subfigure in which the two TW-SCSSBs perform as good as traditional SCSSB.

For a more specific and scientific comparison, we compute the AUC for each birthmark with respect to each similarity metric, and summarize the results in Table 5. As it shows, the AUCs of thread-aware birthmarks are all greater than that of other birthmarks' no matter what similarity metric is adopted. This indicates that TW-DKISBs and TW-SCSSBs perform better than DKISB and SCSSB. Also, it can be observed that the performance improvement between TW-DKISBs and DKISB is minimal, while the improvement between TW-SCSSBs and SCSSB is significant (with a maximum performance gain of 38.5% for TW-SCSSB generated by the SS model using the Ex-Jaccard Metric). This indicates that system call sequence based birthmarks are more easily affected by thread scheduling. In addition, the Ex-Cosine metric is less sensitive to thread scheduling compared with the other three metrics.

4. RELATED WORK

Broadly speaking, the research areas related to our work include software watermarking [7], plagiarism detection, code clone detection, and malware identification. In this section we focus on the discussion of birthmark based software plagiarism detection techniques. We group relevant works into two categories: static and dynamic. Works targeting source code are neglected since there have already been many mature detection systems and tools [14, 19, 11].

Static binary code based birthmarks: Myles and Collberg [18] proposed a k-gram based static birthmark for Java, where sets of Java bytecode sequences of length k are taken as the birthmarks and similarity between two birthmarks are calculated through set operations. The frequencies of elements in the sets are ignored. They evaluated their birthmark using several small java programs. Although the birthmark shows good robustness, it is vulnerable to code transformation attacks. Lim proposed to use control flow information that reflects runtime behaviors to supplement static approaches [13]. More recently Lim proposed to analyze stack flows obtained by simulating the operand stack movements to detect copies [12].

Dynamic software birthmarks: Myles and Collberg [17] suggested to use whole program path generated by compressing a whole dynamic control flow trace into WPP form to uniquely identify program. Schuler [20] treated Java standard API call sequences at object level as dynamic birthmarks for java programs. Such approach exhibited better performance than WPP birthmarks. Tamada [21] introduced two API based birthmarks for windows executables extracted at runtime: Sequence of API Function Calls (EXESEQ) and Frequency of API Function Calls (EXEFREQ). However these methods are all language dependent. To address the problem Wang et al. [24] proposed two dynamic birthmarks based on system calls: System Call Short Sequence birthmark (SCSSB) and Input Dependent System Call Subsequence birthmark (IDSCSB). By integrating data flow and control flow dependency analysis, [23] proposed a system call dependency graph based birthmark (SCDG). Recently [10, 26] suggested to characterize software with core values and applied it to software and algorithm plagiarism detection. A heap graph birthmark based on heap memory analysis is proposed by Patrick et al. [5], and graph monomorphism algorithm is used to compute the similarity. In their experiments, the birthmark was applied to the detection of module plagiarism in JavaScript, and the results were highly accurate.

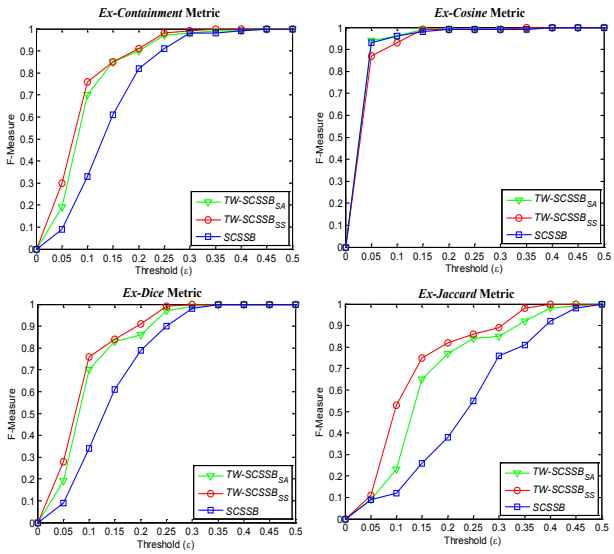

Figure 4. F-Measure curves for *TW-SCSSB$_{SA}$, TW-SCSSB$_{SS}$, and SCSSB*

5. CONCLUSION

As multithreaded software become increasingly more popular, current dynamic software plagiarism detection technology geared toward sequential programs are no longer sufficient. This paper is a first step to fill the gap by proposing thread aware software plagiarism detection techniques. The primary contributions of this paper are as following:

- Two thread-aware dynamic birthmarks TW-DKISB and TW-SCSSB are proposed for plagiarism detection of

multithreaded programs. Our algorithms are able to extract the birthmarks at binary level without the need for source code or java bytecode. In particular, TW-DKISB can be extracted independent of operating systems.

- We have developed a prototype and evaluated the effectiveness of our algorithms. The extensive experiments show that our proposed approaches are not only accurate in detecting plagiarism of multithreaded programs but also robust against semantic-preserving obfuscations.
- A suite of benchmarks is 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 are available at:

  http://labs.xjtudlc.com/labs/benchmark.html

To the best of our knowledge, our work is the first that addresses the challenges of applying dynamic birthmark based approaches for plagiarism detection of multithreaded programs. We envision a future when most applications are multithreaded programs and we plan to continue the research to improve accuracy and efficiency of the proposed algorithms.

6. ACKNOWLEDGMENTS

The research was supported in part by National Science Foundation of China under Grant (91118005, 91218301, 61221063, 61203174), National High Technology Research and Development Program 863 of China under Grant (2012AA011003), Cheung Kong Scholar's Program, Key Projects in the National Science and Technology Pillar Program of China (2012BAH16F02), and the Fundamental Research Funds for the Central Universities.

7. REFERENCES

[1] http://sourceauditor.com/blog/tag/lawsuits-on-open-source/.
[2] At4J library. http://www.at4j.org/download.php.
[3] Allatori obfuscator. http://www.allatori.com/.
[4] Chae D K, Ha J, Kim S W, et al. Software plagiarism detection: a graph-based approach[C].In: CIKM 2013.ACM 2013, 1577-1580.
[5] Chan P, Lucas C K. Heap Graph Based Software Theft Detection[J]. IEEE Transactions on Information Forensics and Security, 2013.
[6] Choi S, Park H, et al. A static API birthmark for Windows binary executables[J]. Journal of Systems and Software. 2009, 82(5): 862-873.
[7] Collberg C, Carter E, Debray S, et al. Dynamic path-based software watermarking[C]. In: PLDI '04.New York, NY, USA: ACM, 2004.
[8] Collberg C, Myles G R, Huntwork A. Sandmark-a tool for software protection research[J]. Security & Privacy, IEEE. 2003, 1(4): 40-49.
[9] Fukuda K, Tamada H. A Dynamic Birthmark from Analyzing Operand Stack Runtime Behavior to Detect Copied Software[C]. In: SNPD '13. IEEE, 2013: 505-510.
[10] Jhi Y, Wang X, Jia X, et al. Value-based program characterization and its application to software plagiarism detection[C]. In: ICSE '11.New York, NY, USA: ACM, 2011. 756-765.
[11] Ji J, Woo G, Cho H. A source code linearization technique for detecting plagiarized programs[J]. SIGCSE Bull.2007.
[12] Lim H I, Taisook H A N. Analyzing Stack Flows to Compare Java Programs[J]. IEICE TRANSACTIONS on Information and Systems, 2012, 95(2): 565-576.
[13] Lim H, Park H, Choi S, et al. A method for detecting the theft of Java programs through analysis of the control flow information[J]. Information and Software Technology, 2009, 51(9): 1338-1350.
[14] Liu C, Chen C, et al. GPLAG: detection of software plagiarism by program dependence graph analysis[C]. In: KDD, 2006. 872-881.
[15] Luk C, Cohn R, Muth R, et al. Pin: building customized program analysis tools with dynamic instrumentation[C]. In: PLDI '05.New York, NY, USA: 2005.
[16] Mcmillan C, Grechanik M, Poshyvanyk D. Detecting similar software applications[C]. In: ICSE 2012.Piscataway, NJ, USA: IEEE Press, 2012. 364-374.
[17] G. Myles and C. Collberg. Detecting software theft via whole program path birthmarks, in Proc. Inf. Security 7th Int. Conf. (ISC 2004),Palo Alto, CA, Sep. 27–29, 2004, pp. 404–415.
[18] Myles G, Collberg C. K-gram based software birthmarks[C]. In: SAC '05. New York, NY, USA: ACM, 2005. 314-318.
[19] Rechelt L, Malpohl G, Philippsen M. Finding plagiarisms among a set of programs with JPlag[J]. Journal of universal computer science, 2002, 8(11): 1016-1038.
[20] Schuler D, Dallmeier V, Lindig C. A dynamic birthmark for java[C]. In: ASE '07.New York, NY, USA: ACM, 2007. 27.
[21] Tamada H, Okamoto K, et al. Dynamic software birthmarks to detect the theft of windows applications[C]. In International Symposium on Future Software Technology. Xian, China, 2004.
[22] Tian Z, Zheng Q, Liu T, et al. DKISB: Dynamic Key Instruction Sequence Birthmark for Software Plagiarism Detection[C]. In: HPCC'13.Zhang Jia Jie, Hu Nan: IEEE, 2013.
[23] Wang X, Jhi Y, Zhu S, et al. Behavior based software theft detection[C]. In: CCS '09.New York, NY, USA: ACM, 2009. 280-290.
[24] Wang X, Jhi Y, Zhu S, et al. Detecting Software Theft via System Call Based Birthmarks[C].In:ACSAC'09.Washington, DC, USA: IEEE Computer Society, 2009. 149-158.
[25] Zhang X, Gupta R. Whole execution traces[C]. Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2004: 105-116.
[26] Zhang F, Jhi Y, Wu D, et al. A first step towards algorithm plagiarism detection[C]. In: ISSTA 2012.New York, NY, USA: ACM, 2012. 111-12.