International Journal of Software Engineering
and Knowledge Engineering
Vol. 30, Nos. 11&12 (2020) 1667–1688

DOI: 10.1142/S0218194020400252

Plagiarism Detection of Multi-threaded Programs Using Frequent Behavioral Pattern Mining

Zhenzhou Tian*, Qing Wang and Cong Gao

School of Computer Science and Technology
Xi'an University of Posts and Telecommunications

Shaanxi Key Laboratory of Network
Data Analysis and Intelligent Processing
Xi'an, Shaanxi 710121, P. R. China
**tianzhenzhou@xupt.edu.cn*

Lingwei Chen and Dinghao Wu

College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802, USA

Received 10 July 2020
Revised 14 August 2020
Accepted 5 October 2020

Software dynamic birthmark techniques construct birthmarks using the captured execution traces from running the programs, which serve as one of the most promising methods for obfuscation-resilient software plagiarism detection. However, due to the perturbation caused by non-deterministic thread scheduling in multi-threaded programs, such dynamic approaches optimized for sequential programs may suffer from the randomness in multi-threaded program plagiarism detection. In this paper, we propose a new dynamic thread-aware birthmark FPBirth to facilitate multi-threaded program plagiarism detection. We first explore dynamic monitoring to capture multiple execution traces with respect to system calls for each multi-threaded program under a specified input, and then leverage the Apriori algorithm to mine frequent patterns to formulate our dynamic birthmark, which can not only depict the program's behavioral semantics, but also resist the changes and perturbations over execution traces caused by the thread scheduling in multi-threaded programs. Using FPBirth, we design a multi-threaded program plagiarism detection system. The experimental results based on a public software plagiarism sample set demonstrate that the developed system integrating our proposed birthmark FPBirth copes better with multi-threaded plagiarism detection than alternative approaches. Compared against the dynamic birthmark System Call Short Sequence Birthmark (SCSSB), FPBirth achieves 12.4%, 4.1% and 7.9% performance improvements with respect to union of resilience and credibility (URC), F-Measure and matthews correlation coefficient (MCC) metric, respectively.

Keywords: Software plagiarism; dynamic birthmark; multi-threaded program; frequent pattern.

1. Introduction

As modern social coding platforms, such as GitHub and CodeShare, have been emerging as one of the most vibrant and important information sources to software programming ecosystem, the incentive for the developers to copy or abuse the ready-to-use codes from others to expedite their own software developments increases as well. For example, as revealed in 2018, Redcore, a Chinese startup's "self-made" web browser, was found to plagiarize substantial code from Google Chrome. Due to the openness of Android, application (app) plagiarism has become even more prevalent through repackaging [1, 2] such that about 13% of apps hosted in third-party marketplaces are repackaged [3], which poses serious threats to the healthy development of software industry.

In order to detect the evolving software plagiarism, different birthmarking techniques [4–9] have been developed. In these methods, software birthmark, which is a set of features, is first extracted from a program to uniquely identify the programs, and then birthmark similarities are measured to determine the potential plagiarism between the programs. Compared to the static birthmark analysis on programs' lexical, grammatical or structural characteristics, dynamic birthmarking techniques [7, 10, 11] construct birthmarks using the captured execution traces from running the programs, which can depict the behaviors and semantics of the programs more accurately and thus enjoy better anti-obfuscation ability. However, due to the perturbation caused by non-deterministic thread scheduling in multi-threaded programs, existing dynamic approaches optimized for sequential programs may suffer from the randomness in plagiarism analysis for multi-threaded programs [12]. For instance, given an input, birthmarks extracted from multiple runs of the same multi-threaded program can be very different; in the extreme cases, such constructed birthmarks may even fail to detect plagiarism between a multi-threaded program and itself [13]. Two dynamic birthmarking methods (i.e. thread-related system call birthmark (TreSB) [13] and thread-oblivious birthmark (TOB) [12]) have been accordingly proposed, yet they still suffer from either weak universality or limitation of overall behavior understanding in multiple threads.

To address the challenge, we run a number of multi-threaded programs, and analyze their behaviors, from which we observe that the same input may generally enforce the same program function execution, while not all parts of the program get involved in thread interleaving, so that its multiple execution traces under the same input may be similar, but not identical. This calls for a sophisticated method to characterize the behavioral patterns from multiple execution traces. Inspired by the success of motif recognition in DNA sequence analysis where difference-tolerant motifs are extracted to identify common patterns of DNA sequence variations, in this paper, we would like to shift such a paradigm that generalizes motif formulation to abstract the behaviors of the multi-threaded programs through their execution traces. More specifically, we first explore dynamic monitoring to capture multiple execution traces for each multi-threaded program under the same input, and then use Apriori [14] to extract significant frequent patterns over execution traces, based on which, we construct a thread-aware birthmark, called FPBirth, to model the behavior of the multi-threaded program and reduce the impact of interleaving threads on multi-threaded program plagiarism detection. The contributions of this paper are summarized as follows:

- A new and dynamic behavioral representation learning method for multi-threaded programs is proposed over their multiple execution traces through candidate set generation and frequent pattern mining. This allows a refined representation to preserve semantics of execution traces while tolerating differences among them as well.
- Based on extracted frequent patterns, a new thread-aware birthmark FPBirth is constructed, which is leveraged to design a multi-threaded program plagiarism detection system.
- Comprehensive experimental studies on a public software plagiarism sample set are conducted to demonstrate that FPBirth is a reliable thread-aware birthmark, and plagiarism detection system over it can achieve the state-of-the-art results, which also outperforms TreSB and TOB.

2. Related Work

In this section, we review the related work on birthmark-based software plagiarism detection. Since we target binaries, existing researches [23, 24] operating on the source code level will not be discussed, while there have already been several mature detection systems over source code [25, 26]. Basically, the birthmarking approaches without requiring the access to program source code fall into two categories: Static birthmark and dynamic birthmark.

2.1. *Static birthmark-based software plagiarism detection*

Tamada *et al.* [4] did the pioneer work of proposing the concept of static software birthmark and designing four kinds of birthmarks constant values in field variables (CVFV), sequence of method calls (SMC), inheritance structure (IS) and used classes (UC) for Java programs. Myles *et al.* [17] introduced k-gram-based static birthmarks, where sets of Java bytecode sequences of length k are taken as the birthmarks. Considering that API calls are usually an indispensable part of a program, Seokwoo *et al.* [27] proposed a static birthmark based on disassembled API calls from executables for detecting plagiarism of windows applications. These birthmarks are simply constructed based on the essential elements that are difficult to be tampered with, and thus significantly enforce syntactic and semantic information loss, rendering them susceptible to even simple code transformations [6]. Lim *et al.* [28, 29] proposed several static birthmarks with the basic idea of simulating the runtime behaviors of Java programs via conducting control flow analysis or analyzing stack flows. Similarly, Park [30] proposed to extract all possible sequences of object instructions from CFG of each method, and applied them for detecting common modules in Java packages. These methods show weakness against control flow obfuscation, and suffer from high time consumption since a mass of traces can be extracted if the CFG is complex. Luo *et al.* [16, 31] proposed an obfuscation-resilient method based on longest common subsequence of semantically equivalent basic blocks. Symbolic execution combined with theorem proving is utilized for birthmark extraction and comparison, which ensures high detection accuracy but also leads to the scalability issue. To alleviate the impact of code transformations, Esh [32] and Git [33] chose to lift the binary assembly code to intermediate languages such as LLVM-IR or BoogieIVL, on the basis of which canonicalization and normalization are performed to form the final representations for programs.

With the popularity of representation learning and deep learning, some works applying them to achieve binary code similarity detection have been emerging. Asm2Vec [34] generated vector representations for assembly sequences by designing and training a representation learning model that is improved upon the PV-DM [35] model, and compared the vectors with cosine similarity. Xu *et al.* [36] designed a deep neural network-based graph embedding model for processing program CFGs to vectors, followed with a siamese architecture to achieve similarity detection. INNEREYE [37] trained an LSTM-based neural network model to obtain embedding vectors for basic blocks and achieve basic-block similarity calculation, based on which program level similarity can be detected. Broadly speaking, the embedding vectors output by trained models can also be viewed as a kind of software birthmark. These methods demonstrate promising detection performance and resilience against code obfuscation attacks, while the limitation lies in construction of a large and high-quality database and a long-time training phase. There also exist plagiarism detection researches [1] targeting Android Apps. DroidMoss [3] took hash value of bytecode fragments as birthmark. ViewDroid [38] presented a functional view graph birthmark, which is vulnerable to dummy view insertion and encryption attacks.

2.2. *Dynamic birthmark-based software plagiarism detection*

Myles *et al.* [5] introduced the concept of dynamic software birthmark and suggested the whole program path (WPP) birthmark, which was demonstrated to be susceptible to loop transformations. Schuler [39] defined a dynamic birthmark for Java programs by observing how objects provided by the Standard APIs are used. Wang *et al.* [20] designed System Call Short Sequence Birthmark (SCSSB) and Input-Dependent System Call Subsequence Birthmark (IDSCSB) to address the problems with API-based techniques. To improve birthmark resilience against deep code obfuscations, core values identified with data flow analysis were organized as sequences and graphs to depict program behavior [9]. By taking the data and control dependency among system calls into birthmark construction, Wang *et al.* [40] proposed system call dependence graph birthmark (SCDGB). Patrick [41] proposed heap graph birthmark (HGB) for JavaScript via conducting heap memory analysis. As SCDGB and HGB utilize graph isomorphism for calculating birthmark similarity, these two methods suffer from the scalability issue. Dynamic API authority vectors (DAAV) [42] constructed a dynamic call graph but converted it to authority vectors with random walks to speed up similarity detection. However, the uncertainty caused by thread scheduling greatly affects the effectiveness of these traditional dynamic birthmarks, making them not suitable for multi-threaded programs.

Considering the noticeable impact of thread scheduling on birthmarking techniques, Tian *et al.* [15] introduced the concept of thread-aware birthmark. Accordingly, two dynamic birthmarking methods TreSB [13] and TOB [12] were proposed to detect multi-threaded program plagiarism. TreSB made use of the thread-related system calls which are in charge of thread operations and management to depict the program behavior, which also makes it only applicable to multi-threaded programs. TOB chose to revive traditional dynamic birthmarks by designing a slicing-merging framework to alleviate the impact of thread interleaving non-determinism. Yet the assumption that event occurring in each thread is stable is not always true due to the interactions between threads. A common practice of these two methods is that they both extract birthmarks from a single execution trace corresponding to a certain input. Differently, our proposed FPBirth takes the frequent patterns mined from multiple execution traces collected from multiple runs of multi-threaded programs under the same given input as birthmark, which preserves the behavioral semantics and also improves the difference-tolerant ability.

3. Problem Statement

In this section, we first define the software plagiarism detection problem. A software birthmark, whose classical definition is given in Definition 1, is a set of characteristics extracted from a program that reflects intrinsic properties of the program and can hence be used to identify the program uniquely. Static birthmarks tend to overlook operational behaviors of a program. In this respect, dynamic birthmarks, as defined in Definition 2, are introduced to remedy this formulation. Dynamic birthmarks are extracted based on runtime behaviors and thus are believed to be more accurate reflections of program semantics. However, when multi-threaded programming gradually becomes the mainstream, the traditional birthmark techniques inevitably encounter a great obstacle in multi-threaded program plagiarism detection. The non-determinism brought by thread schedules in multi-threaded programs causes the execution behaviors to differ across different runs, while plagiarism is essentially determined based on similarity measuring over the execution behaviors. In order to address this challenge, the concept of thread-aware birthmark [15] is accordingly elaborated to abstract the behaviors of the multi-threaded programs through their execution traces, which is specified in Definition 3.

Definition 1 (Software Birthmark [4]). 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 implies that $f(p) = f(q)$.

Definition 2 (Dynamic Software Birthmark [5]). Let p be a program, I be an input to p and $f(p)$ be a set of characteristics extracted from p by executing p with input I. We say $f(p, I)$ is a dynamic birthmark of p if and only if both of the following conditions are satisfied:

— $f(p, I)$ is obtained only from p itself when executing p with input I.
— Program q is a copy of p implies that $f(p, I) = f(q, I)$.

Definition 3 (Thread-Aware Dynamic Software Birthmark [15]). Let p, q be two multi-threaded programs, I be an input, s be a thread schedule to p and q and $f(p, I, s)$ be a set of characteristics extracted from p when executing p with I and schedule s. We say $f(p, I, s)$ is a thread-aware 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 implies that $f(p, I, s) = f(q, I, s)$.

Obviously, these are abstract guidelines without considering any implementation feasibility. In practice, even if there is a plagiarism correlation between two programs, the constructed birthmarks may not be exactly the same. Therefore, for software birthmarking techniques, instead of enforcing exact birthmark matching, the plagiarism of two programs is decided by a similarity metric that computes the similarity score between their birthmarks and a threshold ε over this score with a range between 0 and 1. In our work, we measure the similarity between the original program p's birthmark and the suspect program q's birthmark $\mathrm{sim}(f(p, I, s), f(q, I, s))$ to determine the plagiarism. The higher the similarity, the more possible the suspect program q copies code from the original program p. We do not set ε to a fixed value, while analyze its impact on performance under a wide range of choices. As such, the plagiarism can be decided by Eq. (1), which gives a conceptual definition of sim that returns either positive, negative, or inconclusive.

$$\mathrm{sim}(p_f, q_f) = \begin{cases} \geq 1 - \varepsilon & q \text{ is a copy of } p, \\ < \varepsilon & q \text{ is not a copy of } p\,, \\ \text{otherwise} & \text{inconclusive.} \end{cases} \tag{1}$$

4. Proposed Method

In this section, we present the detailed method of how we construct thread-aware birthmarks for multi-threaded programs over their execution traces, which is illustrated in Fig. 1.

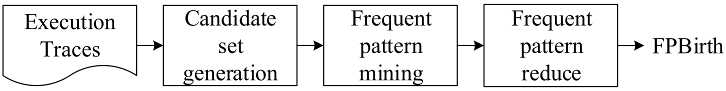


Fig. 1. The overview of FPBirth construction.

4.1. *Candidate set generation*

The thread interleaving in multi-threaded programs leads to changes in the program execution traces. To capture such unique behaviors so that the constructed birthmarks are difference-tolerant to the changes among execution traces, we take as input multiple execution traces from a multi-threaded program under the same input, and extract frequent behavioral patterns over execution traces to formulate birthmark. To improve the effectiveness of frequent pattern mining, pattern candidate set is first generated through pre-processor, gram-based slice and slice merging, which is displayed in Fig. 2.

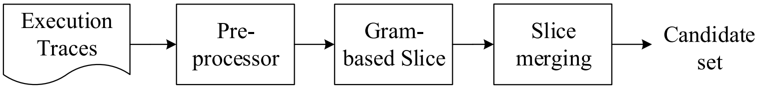


Fig. 2. Basic process of pattern candidate set generation.

4.1.1. *Pre-processor*

The captured execution traces consist of system calls related to program and thread operations, where each record in the system call sequence is specified as system call number, name and return value. However, the raw execution traces are not applicable for direct FPBirth extraction. First, those system calls that fail cannot correctly reflect the program's behaviors [16], which should be considered noises to be filtered out using their return values. For example, some system call serves to close the files; if there is a failure, this system call will be revoked multiple times until it succeeds. Second, those system calls that are invoked randomly may perturb the execution traces, which should be also removed. For example, *futex*, essentially designed to reduce the number of system calls for performance issue, is called only when the program is likely to be blocked for a longer time until the condition becomes true. Its occurrences show intrinsic randomness under different executions.

Another kind of system calls that are responsible for memory management, such as *mmap* and *brk*, also greatly depend on real-time memory chunk needs. To this end, we perform the pre-processing to prune these captured execution traces before fed to birthmark construction.

4.1.2. *Gram-based slice*

Due to its simplicity and scalability, k-gram model [17] in natural language processing is then used to slice up the pre-processed execution traces to form different subsequences of k continuous system calls. Given a pre-processed execution trace $s = (e_1, e_2, \ldots, e_n)$, a series of subsequences split by k-gram can be defined as

$$\mathrm{grams}(s, k) = \{g_i | g_i = (e_i, e_{i+1}, \ldots, e_{i+k})\}, \quad 1 \leq i \leq n - k + 1. \tag{2}$$

In this respect, execution traces can be transformed into a set of short sequences to facilitate fast pattern mining while not significantly compromising their important semantic information, which thus greatly ensures the integrity of trace contents.

4.1.3. *Slice merging*

To generate the candidate set for frequent pattern mining, we further merge all the short sequences sliced by k-gram over multiply execution traces of each multi-threaded program under the same input. In other words, one multi-threaded program with one input will specify one pattern candidate set. As such, given a multi-threaded program p and an input I, a pattern candidate set can be defined as

$$\mathrm{CanSet}_p^I = \bigcup_{i=1}^{m} \mathrm{grams}(s_i, k), \tag{3}$$

where s_i is program p's ith execution trace under input I and m is the number of execution traces.

4.2. ***Frequent pattern mining***

Frequent pattern mining is an important research topic in data mining [18], which searches for recurring relationships in a given dataset with frequency not less than minimum support threshold, and thus leads to discovery of associations among itemsets. Therefore, based on the generated candidate sets, we explore a frequent pattern mining method Apriori [14] to dig out the most representative behavioral patterns to birthmark each multi-thread program, which not only preserve semantics of execution traces, but also have strong ability to resist variations caused by thread interleaving.

The key of Apriori is the apriori knowledge that all non-empty subsets of a frequent itemset must also be frequent. Therefore, Apriori algorithm follows the iterative steps that frequent t-itemsets (i.e. itemsets that contain t items and have frequency not less than minimum support σ) are generated by joining frequent

$(t-1)$-itemsets with itself until no new frequent itemsets are identified. In this way, given a candidate set CanSet_p^I, the generated frequent pattern set over it can be defined as

$$\mathrm{FreSet}_p^I = \{f_i|\mathrm{count}(f_i) \geq \sigma, 1 \leq i \leq l)\}, \tag{4}$$

where f_i is ith frequent pattern in CanSet_p^I and l is the number of frequent patterns in FreSet_p^I.

To perform frequent pattern mining, the length of the input sequences k, which is decided by k-gram slices, must be appropriately considered: (1) excessive length will lead to an explosion in the number of iterations and itemset candidates, and the burden of program running, while (2) the length being too short may enforce short frequent itemset generation; since we utilize frequent itemsets as patterns to construct the birthmark, frequent itemsets being too short will not be able to depict any specific patterns and thus degrade their expressiveness and representativeness to execution traces and the corresponding birthmark's semantics and accuracy to the multi-threaded programs. That is to say, given the input sequences of length k, the length of frequent itemsets t may directly impact on the validity of the constructed birthmark. As such, the length of the input sequences k and the length range of the frequent itemsets t will be empirically evaluated in the experiments on the sample data to find the best trade-off between the effectiveness and efficiency for multi-threaded program plagiarism detection.

4.3. *Frequent pattern reduction*

Using frequent pattern mining over CanSet_p^I, we may generate the frequent pattern set FreSet_p^I with a large number of frequent patterns, where according to the implementation of Apriori algorithm, the resulting patterns with shorter length are obviously more than the ones with longer length. On the one hand, shorter patterns are weaker than longer ones in representing program-specific semantic behaviors for less context; on the other hand, shorter patterns themselves may be embedded in longer patterns, which has a major drawback to cause the redundancy, and thus mislead the effect of the constructed birthmark over frequent patterns. Therefore, the removal of such short frequent patterns is indispensable.

More specifically, we here propose a pattern removing method before constructing the birthmark, named insignificant pattern removing, where all the frequent patterns that are included in others as continuous subsequences are insignificant and should be removed. For example, given the pattern "ABCDE", the following pattern "ABC" becomes insignificant because it is a complete substring and gives no extra information, while the pattern "ADE" will be retained due to its variation on "ABCDE".

Finally, the refined frequent pattern set is used to construct the thread-aware dynamic software birthmark for the program. Note that, for dynamic birthmarks, the number of pattern occurrences is related to the execution behavior of the program to some extent; that is, birthmark similarity should be measured over pattern frequency instead of pattern existence. To facilitate such a similarity calculation, we further transform the frequent pattern set into key-value pair set where the keys represent the frequent patterns and the values refer to their corresponding frequencies. This key-value pair set acts as the program's dynamic birthmark under a specified input, named FPBirth. Accordingly, given a frequent pattern set FreSet_p^I, FPBirth can be defined as

$$\text{FPBirth}_p^I = \{\langle f_i, \text{sup}(f_i)\rangle | f_i \in \text{FreSet}_p^I\}, \tag{5}$$

where f_i is ith frequent pattern in FreSet_p^I and $\text{sup}(f_i)$ is the frequency of pattern f_i (i.e. support count).

Algorithm 1 gives the pseudo-code on FPBirth generation. Lines 2 to 3 generate candidate set. The function *grams* returns a set of short sequences and the *merge* function merges all short sequences to return the pattern candidate set. Line 4 uses Apriori as the frequent pattern mining algorithm. Apriori algorithm follows the iterative steps that frequent t-itemsets are generated by joining frequent

Algorithm 1. Extracting FPBirth

```
Input:
    $s$: a pre-processed execution trace consisted of system calls
    $k$: the window size used to generate $k$-grams
    $\sigma$: minimum support
    $t$: the length range of frequent patterns
    $m$: the number of execution traces
Output:
    $FPBirth_p$: the birthmark FPBirth, which is a key-value pair set
 1: $FPBirth_p \leftarrow \langle\rangle$
 2: $short\_sequences = grams(s, k)$    ▷ Slicing $s$ with a window of length $k$
 3: $CanSet_p = merge(short\_sequences, m)$    ▷ To generate the candidate set
 4: $FreSet_p = apriori(CanSet_p, \sigma, t)$    ▷ To generate the frequent pattern set
 5: for $pattern_a \in FreSet_p$ do
 6:     for $pattern_b \in FreSet_p$ do
 7:         if $pattern_a.complete\_substring(pattern_b)$ then
 8:             remove $pattern_a$    ▷ Insignificant pattern removing
 9:         else
10:             $FPBirth_p \leftarrow FPBirth_p \oplus \langle pattern_a, frequency\rangle$
11:         end if
12:     end for
13: end for
14: return $FPBirth_p$
```

$(t-1)$-itemsets with itself until no new frequent itemsets are identified. Lines 5–13 reduce the frequent patterns. The refined frequent pattern set and its corresponding frequency constitute key-value pairs that are used to construct thread-aware dynamic birthmarks for the program. The algorithm finally returns FPBirth_p.

The most time-consuming phase of Algorithm 1 is frequent pattern mining. The number of iterations in this phase depends on the length of the input sequences k decided by k-gram slices and the parameters t and σ. As described in Sec. 4.2, given the input sequences of length k, the length of frequent itemsets t may impact on the number of iterations and the validity of the constructed birthmark, and thus they will be empirically evaluated in the experiments on the sample data to find the best trade-off between the birthmark effectiveness and construction efficiency. The validity and authenticity of the mining results of the Apriori algorithm will be affected by the parameter σ. A large σ may result in filtering out some important but infrequent itemsets, and in extreme cases, no frequent itemsets can be mined. On the contrary, a small σ may result in insignificant frequent itemsets, and furthermore, the time overhead may increase while performance deteriorates significantly. This paper considers that $\sigma = 4$ is a reasonable choice for the experimental dataset, which will be detailed in Sec. 6.1; that is, itemsets that have frequency no less than 4 are frequent itemsets.

5. FPBirth-based Software Plagiarism Detection

In the previous section, we mine the frequent patterns over the execution traces to construct thread-aware dynamic birthmark FPBirth. Here, we further leverage such birthmarks for multi-threaded program plagiarism detection.

5.1. *Similarity calculation and plagiarism detection*

Using FPBirth, we can effectively and dynamically birthmark a multi-threaded program under a specified input. However, an FPBirth birthmark merely abstracts part of the semantics and behaviors of the program under a single input, based on which, the plagiarism detection decision is clearly biased and not reliable. For instance, two different programs may adopt the same standard exception handling mechanism, while any inputs that invoke the exception handling will enforce the same behavioral patterns for both programs. To address this issue, we formulate different inputs and perform multiple executions for each multi-threaded program under each of these inputs to cover as many functional blocks as possible, so that we can construct a series of FPBirth birthmarks to thoroughly represent the semantics and behaviors of the program. Given an original program p, a suspect program q and a set of inputs $\{I_1, I_2, \ldots, I_d\}$, we accordingly generate a set of FPBirth birthmark pairs for p and q, which can be denoted as $\{(\text{FPBirth}_p^{I_1}, \text{FPBirth}_q^{I_1}), \ldots, (\text{FPBirth}_p^{I_d}, \text{FPBirth}_q^{I_d})\}$. Instead of evaluating the similarity between a single pair of birthmarks, we calculate the similarities for all pairs of birthmarks and take their mean value as the measure of software similarity between p and q, which can be denoted as follows:

$$\mathrm{sim}(p_f, q_f) = \sum_{i=1}^{d} \mathrm{sim}(\mathrm{FPBirth}_p^{I_i}, \mathrm{FPBirth}_q^{I_i})/d. \tag{6}$$

Based on $\mathrm{sim}(p_f, q_f)$ and Eq. (1), we can obtain the final plagiarism detection results, where the threshold ε is adjustable for different sample dataset. Note that we aim to outline a general paradigm to explore the similarity between p and q, where the measure models can be instantiated in different ways. In this paper, we employ cosine similarity for measurement, since it is commonly used in high-dimensional positive spaces with the outcome being neatly bounded in $[0, 1]$.

5.2. *System implementation*

Figure 3 depicts the overview of our FPBirth-based birthmarking system. The plaintiff represents the original program owned by its developer while the defendant represents the suspicious program that may have plagiarized the plaintiff. The system mainly consists of four modules: the dynamic monitor for capturing program execution traces, the extractor for extracting FPBirth birthmarks, the similarity calculator for computing birthmark similarity, and the decision maker that determines whether the defendant is guilty.

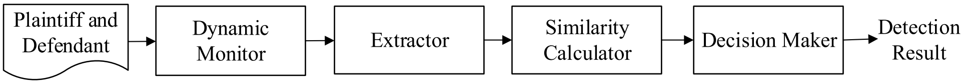


Fig. 3. Overview of FPBirth-based software plagiarism detection prototype.

Dynamic monitor module, which is implemented as a PIN plugin [19], monitors program executions. Specifically, using the PIN dynamic instrumentation framework and API functions *PIN_AddSyscallEntryFunction* and *PIN_AddSyscallExitFunction*, monitoring and analysis code is instrumented before and after system call invoking positions, respectively, to capture relevant system call information during program execution. The form of each system call is specified as < thread identifier, system call number, system call name, specific parameter, return value >. The extractor module handles the execution traces captured by the dynamic monitor module, and performs the construction of FPBirth birthmark through candidate set generation, frequent pattern mining and frequent pattern reduction. In the similarity calculator, similarity scores are computed between the birthmarks of plaintiff and defendant under different inputs, and the decision maker decides the plagiarism using the threshold over the mean of similarity scores.

6. Experimental Results and Analysis

In this section, we conduct experimental studies using a public software plagiarism sample set to fully evaluate the performance of our developed FPBirth-based system in multi-threaded program plagiarism detection.

6.1. *Experimental setup*

We evaluate the effectiveness of our proposed detection system over FPBirth on a public software plagiarism sample set [13], including 234 multi-threaded programs of different versions, derived from a series of obfuscations (e.g. SandMax, Zelix, UPX) over 35 benchmark multi-threaded programs, which are shown in Table 1. The parameter settings to implement our model for evaluation are specified as: $k = 6$ for k-gram slice, which is also the length of the input sequences for frequent pattern mining, minimum support $\sigma = 4$, the length of frequent patterns ranging in $t \in [3, 6]$; for each input, $m = 4$ for the number of execution traces captured. As for the baselines, we compare our approach with two multi-threaded program plagiarism detection methods TreSB [13] and TOB [12] and system call-based dynamic birthmark technique SCSSB [20].

Table 1. Benchmark multi-threaded programs.

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

6.2. *FPBirth evaluation*

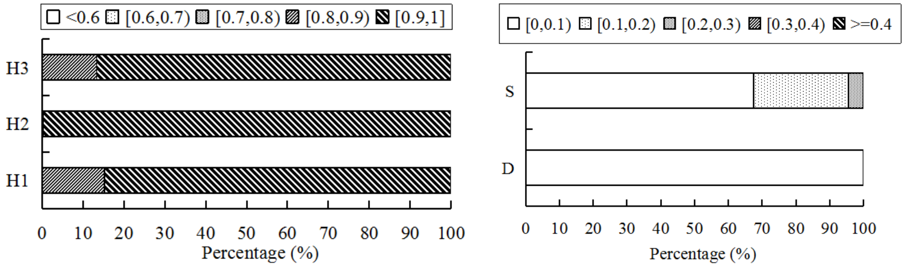
With these settings, we mainly evaluate the resilience and credibility of the thread-aware birthmark FPBirth [6], which can be described as follows [5]:

- **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 $\text{sim}(p_f, q_f) \geq 1 - \varepsilon$.
- **Credibility.** Let p and q be independently developed programs. A birthmark is credible if it can differentiate the two programs, that is $\text{sim}(p_f, q_f) < \varepsilon$.

In other words, resilience reflects the ability of birthmark to be resistant to all kinds of semantic-retention code obfuscations, while credibility characterizes the ability of birthmark to distinguish independently developed software.

6.2.1. *Resilience evaluation*

In this experiment, the benchmark program is taken as the original program while the obfuscated program is taken as the suspect program so that a series of original-suspect comparison pairs are formulated to evaluate the resilience of FPBirth. The experimental results regarding similarity distribution under three different obfuscations (H1, H2 and H3) are illustrated in Fig. 4(a), where H1 uses different compilers and optimizations (e.g. llvm, gcc, o0-oS) for weak obfuscation, H2 applies professional obfuscation tools (e.g. SandMark, Zelix, ProGuard) for strong obfuscation and H3 uses UPX for packing. We can observe that most of the comparison pairs enforce a similarity higher than 0.9; this indicates that FPBirth birthmark enjoys an excellent resistance to the obfuscation strategies involved in this public dataset.



(a) Resilience evaluation

(b) Credibility evaluation

Fig. 4. FPBirth evaluation on resilience and credibility.

6.2.2. *Credibility evaluation*

In this experiment, the programs independently developed are selected from the dataset to evaluate the credibility of FPBirth. Specifically, the selected instances include 6 multi-threaded compression/decompression software, 7 web browsers and 5 audio player software. We use FPBirth to birthmark software and then calculate the similarity between them. Figure 4(b) shows the distribution of similarity over similar software and different software, where *S* stands for software included in the same category and *D* represents software distributed in different categories. We can see that the similarity between software belonging to different categories is very low, with the mean similarity below 0.1. This indicates that FPBirth birthmark can effectively distinguish different kinds of software. Due to their remarkable consistency in functions, the similarity between software in the same category is slightly higher, but most of them still fall into a very low range. There are few comparison pairs with a similarity between 0.2 and 0.3 as their designs adopt the same algorithm or both rely on some functional modules. For example, the average similarity between browser Dooble and Epiphany is 0.28, since both ones use WebKit layout engines. Overall, FPBirth performs well in differentiating independently developed software.

6.3. *Comparisons with traditional birthmark techniques*

We also compare FPBirth with TreSB [13] and TOB [12], two traditional thread-aware birthmark techniques and SCSSB [20], a dynamic birthmark technique based on system calls by detection effect and time cost.

6.3.1. *Comparative analysis on detection effect*

In this section, to quantitatively validate the effectiveness of different birthmark methods, we use union of resilience and credibility (URC) [21], `F-Measure`, matthews correlation coefficient (MCC) [22] and area under the curve (AUC) as the performance measures.

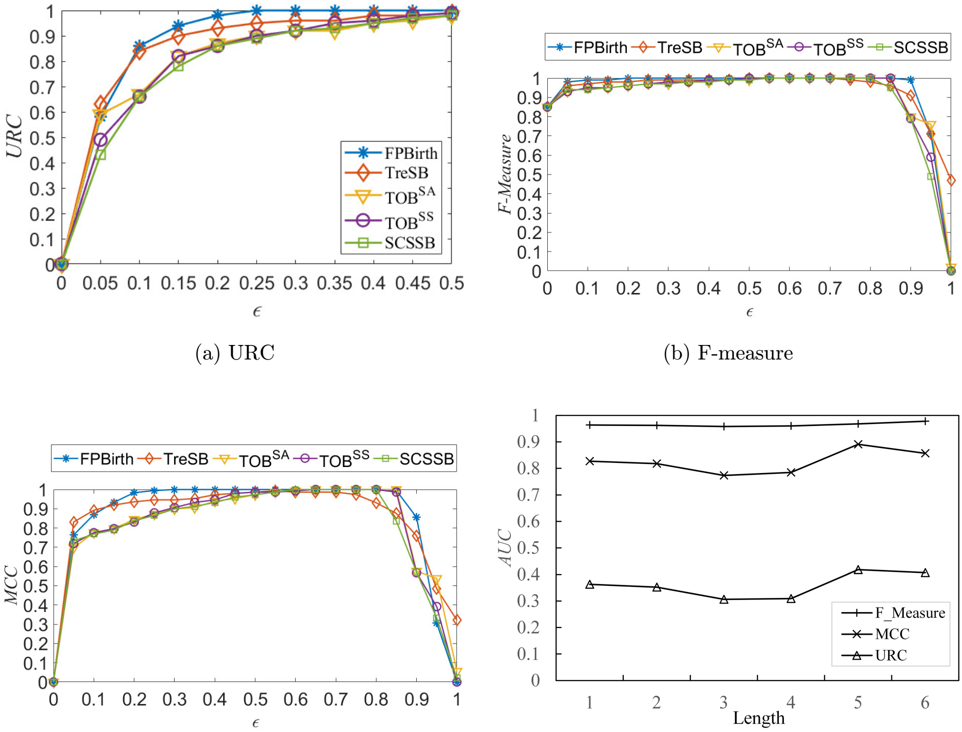
(i) URC. URC is an indicator designed for comprehensively measuring the birthmarks in terms of resilience and credibility:

$$\mathrm{URC} = 2 \times \frac{R \times C}{R + C}, \tag{7}$$

where R represents the ratio of plagiarism pairs correctly classified to all comparison pairs with plagiarism, and C represents the ratio of independently developed pairs correctly classified to all comparison pairs with independence (i.e. without plagiarism). The value of URC is between 0 and 1, and the higher the URC, the better the performance of the birthmark. According to the criteria given in Eq. (1), the plagiarism detection result is decided by the threshold ε. We set the effective value range of the threshold as 0–0.5, that is, $1 - \varepsilon \geq \varepsilon$. Figure 5(a) shows the comparative results between FPBirth and other birthmark techniques under different thresholds. As the blue line shows, FPBirth performs better than the other three birthmarking methods.

(ii) F-Measure and MCC. F-Measure and MCC are commonly used in the field of information retrieval and data mining. In this regard, the "uncertain" part of the criteria given in Eq. (1) is removed here, and plagiarism detection is described as a binary classification problem:

$$\mathrm{sim}(p_f, q_f) = \begin{cases} \geq \varepsilon & q \text{ is a copy of } p, \\ < \varepsilon & q \text{ is not a copy of } p. \end{cases} \tag{8}$$



(a) URC

(b) F-measure

(c) MCC

(d) Pattern length

Fig. 5. Comparative analysis on detection performance and pattern length.

For F-Measure measurement, the harmonic average of precision and recall is used here, which is described as

$$\text{F-Measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{9}$$

MCC is an evaluation metric considering true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN), and can be used to make a reasonable assessment of test effectiveness in the case of unbalanced positive and negative samples, which is denoted as

$$\text{MCC} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FP})(\text{TN} + \text{FN})}}. \tag{10}$$

Figures 5(b) and 5(c), respectively, show the comparison results between FPBirth and other birthmark techniques under different thresholds, where FPBirth outperforms TreSB, TOB and SCSSB in most measurements.

(iii) AUC. With the help of AUC, we can further perform the quantitative analysis of the technical performance of each birthmark with respect to URC, F-Measure and

Table 2. Comparison of birthmark techniques over AUC.

	SCSSB	TOB^{SA}	TOB^{SS}	TreSB	FPBirth
URC	0.394	0.404	0.402	0.431	0.443
F-Measure	0.916	0.933	0.925	0.952	0.954
MCC	0.820	0.839	0.834	0.875	0.885

MCC. Table 2 summarizes the specific AUC values of different measure metrics for each birthmark technique. It can be observed that all three AUC values of FPBirth are higher than those of traditional birthmark methods, which indicates that FPBirth can cope better with multi-threaded program plagiarism detection.

To get a more intuitive sense of the performance differences among the birthmark methods, we quantify their performance gains using PerGain [12], which takes the SCSSB as baseline and calculates performance improvement values for each thread-aware birthmark in terms of the AUC metric. Specifically, it can be denoted as

$$\text{PerGain} = \frac{\text{AUC}_X - \text{AUC}_{\text{SCSSB}}}{\text{AUC}_{\text{SCSSB}}} \times 100\%, \tag{11}$$

where AUC_X and $\text{AUC}_{\text{SCSSB}}$ represent the AUC values of a thread-aware birthmark and SCSSB and $X \in \{\text{FPBirth}, \text{TOB}^{SA}, \text{TOB}^{SS}, \text{TreSB}\}$. As summarized in Table 3 regarding the PerGain values, FPBirth achieves 12.4%, 4.1% and 7.9% performance improvements with respect to URC, F-Measure and MCC metric, respectively, and consistently outperforms the other three thread-aware birthmarks.

Table 3. Quantitative comparison of thread-aware birthmarks with SCSSB regarding PerGain.

	TOB^{SA}	TOB^{SS}	TreSB	FPBirth
URC	2.5	2.0	9.4	12.4
F-Measure	1.9	1.0	3.9	4.1
MCC	2.3	1.7	6.7	7.9

6.3.2. *Comparative analysis on time cost*

FPBirth and other three birthmark-based detections mainly include trace capture, birthmark generation and similarity calculation. Considering that the experiments are conducted on the same set of execution traces, here we focus on comparing the time cost of FPBirth with others in terms of birthmark generation (Phase II) and similarity calculation (Phase III). Table 4 gives the average time cost of each

Table 4. Comparison of birthmarks over time cost (ms).

	SCSSB	TOB^{SA}	TOB^{SS}	TreSB	FPBirth
Phase II	103	103	103	102	1556
Phase III	0.1	0.1	20	0.02	9.9

birthmark. From the results, we can observe that the average time of FPBirth to generate birthmark is higher than others. The reason behind this is that other methods use k-gram directly to construct birthmarks, while FPBirth takes extra time to mine the frequent patterns that improves the birthmark's thread-aware ability. Since FPBirth constructs more representative frequent patterns, it takes a little more time (9.9 ms on average) for similarity calculation as well, which is still less than TOB^{SS} using maximum weighted dichotomy matching. Though it is more time-consuming, FPBirth is still significant and promising for multi-threaded program plagiarism detection for its better detection effectiveness. Our follow-up plan is to optimize the frequent pattern mining process to improve FPBirth's construction efficiency.

6.4. *Evaluation on pattern length*

As described in Sec. 4.2, the length of frequent patterns directly affects the validity of the constructed birthmark. Therefore, this section analyzes the impact of pattern length on detection performance. Figure 5(d) displays the AUC values of URC, F-Measure and MCC for plagiarism detection using FPBirth with different pattern lengths, where F-Measure slightly increases as the length increases, while URC and MCC suffer from a drop at length 3, but keep going up afterwards and reach to the best at length 5. Considering all three metrics, length 6 gives the best balance. This is why we choose $k = 6$ for k-gram slice and input sequence length for frequent pattern mining. In addition, given the length of the input sequences, frequent patterns after mining and reduction may still enjoy different pattern lengths ranging from 1 to 6. As discussed, patterns being too short may exist in different programs as common behaviors, which may not be able to differentiate a program from others and should be removed. From our results, pattern lengths ranging from 3 to 6 provide the best detection performance, which is what we've set up for our experiments.

7. Discussion

As FPBirth is extracted via frequent behavioral pattern mining over the execution traces captured from program runs, it suffers from the same limitation as other dynamic birthmarks in exhaustively covering all behaviors of a program, which thus brings performance concerns to the detection due to false positives. In other words, two independent programs, especially those with functional similarity, may handle certain inputs in the similar ways; if the provided inputs used to drive program executions occasionally cover a large portion of the similar parts (which actually constitute only a very small part of the whole program), a high similarity will be measured, thus causing false positives. To alleviate this problem, we may either use all the officially accompanied test cases (e.g. the programs from the

Parsec 3.0 benchmark come with test cases), or provide a lot of inputs varying types to drive a program executing different functional parts. A more reasonable way is to combine with test case generation techniques, and we take it as a future work.

In addition, we dynamically instrument the program binaries to capture the system call traces. Despite that the dynamic instrument tool PIN supports the collection of program execution traces on multiple platforms (including Windows, Linux and macOS), FPBirth is not applicable to cross-platform plagiarism detection, as the system calls under different operating systems are different and there exists no definite correspondence between them. For those programs that do not involve any system calls or have very few system calls, FPBirth will not work either. Moreover, obfuscation techniques that perform deletion, modification or substitution to system calls are likely to frustrate our method. However, it is worth noting that system calls being the only way an application uses to request services from the operating system's kernel, are difficult to delete and tamper with in a large scale. Therefore, it is rather troublesome and expensive to successfully enforce such kind of obfuscations, which are very likely to introduce subtle concurrent problems for multi-threaded programs if not properly handled.

8. Conclusion

This paper proposes a new dynamic thread-aware birthmark FPBirth to detect the multi-threaded program plagiarism. More specifically, we first explore dynamic monitoring to capture multiple execution traces with respect to system calls for each multi-threaded program under a specified input, and then leverage Apriori algorithm to mine frequent patterns to formulate our dynamic birthmark, which can not only depict the program's behavioral semantics, but also resist the changes and perturbations over execution traces caused by the thread scheduling in multi-threaded programs. Using FPBirth, we design a multi-threaded program plagiarism detection system. The experimental results based on a public software plagiarism sample set demonstrate that the developed system integrating our proposed birthmark FPBirth outperforms alternative dynamic birthmark approaches in multi-threaded plagiarism detection.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (61702414), the Natural Science Basic Research Program of Shaanxi (2018JQ6078, 2020GY-010), the Science and Technology of Xi'an (2019218114GXRC017CG018-GXYD17.16), the International Science and Technology Cooperation Program of Shaanxi (2018KW-049, 2019KW-008) and the Key Research and Development Program of Shaanxi (2019ZDLGY07-08).

References

1. K. Chen, P. Liu and Y. Zhang, Achieving accuracy and scalability simultaneously in detecting application clones on android markets, in *Proc. Int. Conf. Software Engineering*, 2014, pp. 175–186.
2. L. Luo, Y. Fu, D. Wu, S. Zhu and P. Liu, Repackage-proofing android apps, *46th Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks*, 2016, pp. 550–561.
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*, 2012, pp. 317–326.
4. H. Tamada, M. Nakamura, A. Monden and K.-I. Matsumoto, Design and evaluation of birthmarks for detecting theft of Java programs, *IASTED Conf. Software Engineering*, 2004, pp. 569–574.
5. G. Myles and C. Collberg, Detecting software theft via whole program path birthmarks, *Int. Conf. Information Security*, 2004, pp. 404–415.
6. Z. Tian, T. Liu, Q.-h. Zheng, F. Tong, D. Wu, S. Zhu and K. Chen, Software plagiarism detection: A survey, *J. Cyber Secur.***1**(3) (2016) 52–76.
7. 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. Softw. Eng.* **41**(12) (2015) 1217–1235.
8. F. Zhang, D. Wu, P. Liu and S. Zhu, Program logic based software plagiarism detection, *IEEE Int. Symp. Software Reliability Engineering*, 2014, pp. 66–77.
9. 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. Softw. Eng.* **41**(9) (2015) 925–943.
10. J. Ming, D. Xu, Y. Jiang and D. Wu, Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking, *26th USENIX Security Symp.*, 2017, pp. 253–270.
11. 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. Software Engineering*, 2011, pp. 756–765.
12. Z. Tian, T. Liu, Q. Zheng, E. Zhuang, M. Fan and Z. Yang, Reviving sequential program birthmarking for multithreaded software plagiarism detection, *IEEE Trans. Softw. Eng.* **44**(5) (2017) 491–511.
13. Z. Tian, T. Liu, Q. Zheng, M. Fan, E. Zhuang and Z. Yang, Exploiting thread-related system calls for plagiarism detection of multithreaded programs, *J. Syst. Softw.* **119** (2016) 136–148.
14. R. Agrawal and R. Srikant, Mining sequential patterns, in *Proc. Int. Conf. Data Engineering*, 1995, pp. 3–14.
15. 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*, 2014, pp. 304–313.
16. 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 Int. Symp. Foundations of Software Engineering*, 2014, pp. 389–400.
17. G. Myles and C. Collberg, *K*-gram based software birthmarks, in *Proc. ACM Symp. Applied Computing*, 2005, pp. 314–318.
18. J. Han, H. Cheng, D. Xin and X. Yan, Frequent pattern mining: Current status and future directions, *Data Min. Knowl. Discov.* **15**(1) (2007) 55–86.
19. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, Pin: Building customized program analysis tools with dynamic

instrumentation, in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2005, pp. 190–200.

20. X. Wang, Y.-C. Jhi, S. Zhu and P. Liu, Detecting software theft via system call based birthmarks, *Annual Computer Security Applications Conf.*, 2009, pp. 149–158.
21. X. Xie, F. Liu, B. Lu and L. Chen, A software birthmark based on weighted k-gram, *2010 IEEE Int. Conf. Intelligent Computing and Intelligent Systems*, 2010, pp. 400–405.
22. B. W. Matthews, Comparison of the predicted and observed secondary structure of T4 phage lysozyme, *Biochim. Biophys. Acta (BBA)-Protein Struct.* **405**(2) (1975) 442–451.
23. A. A. Pandit and G. Toksha, Review of plagiarism detection technique in source code, *Int. Conf. Intelligent Computing and Smart Communication*, 2020, pp. 393–405.
24. 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*, 2006, pp. 872–881.
25. L. Prechelt, G. Malpohl and M. Philippsen, Finding plagiarisms among a set of programs with JPlag, *J. Univers. Comput. Sci.* **8**(11) (2002) 1016.
26. G. Cosma and M. Joy, An approach to source-code plagiarism detection and investigation using latent semantic analysis, *IEEE Trans. Comput.* **61**(3) (2011) 379–394.
27. S. Choi, H. Park, H.-i. Lim and T. Han, A static API birthmark for Windows binary executables, *J. Syst. Softw.* **82**(5) (2009) 862–873.
28. H.-i. Lim, H. Park, S. Choi and T. Han, A method for detecting the theft of Java programs through analysis of the control flow information, *Inf. Softw. Technol.* **51**(9) (2009) 1338–1350.
29. H.-i. Lim and H. Taisook, Analyzing stack flows to compare Java programs, *IEICE Trans. Inf. Syst.* **95**(2) (2012) 565–576.
30. H. Park, H.-i. Lim, S. Choi and T. Han, Detecting common modules in Java packages based on static object trace birthmark, *Comput. J.* **54**(1) (2011) 108–124.
31. 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 Trans. Softw. Eng.* **43**(12) (2017) 1157–1177.
32. Y. David, N. Partush and E. Yahav, Statistical similarity of binaries, in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2016, pp. 266–280.
33. Y. David, N. Partush and E. Yahav, Similarity of binaries through re-optimization, in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2017, pp. 79–94.
34. S. H. Ding, B. C. Fung and P. Charland, Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization, *IEEE Symp. Security and Privacy*, 2019, pp. 472–489.
35. Q. Le and T. Mikolov, Distributed representations of sentences and documents, *Int. Conf. Machine Learning*, 2014, pp. 1188–1196.
36. X. Xu, C. Liu, Q. Feng, H. Yin, L. Song and D. Song, Neural network-based graph embedding for cross-platform binary code similarity detection, in *Proc. ACM SIGSAC Conf. Computer and Communications Security*, 2017, pp. 363–376.
37. F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng and Z. Zhang, Neural machine translation inspired binary code similarity comparison beyond function pairs, in *Proc. 2019 Network and Distributed Systems Security Symp.*, 2019, pp. 1–15.
38. 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 & Mobile Networks*, 2014, pp. 25–36.
39. D. Schuler, V. Dallmeier and C. Lindig, A dynamic birthmark for Java, in *Proc. IEEE/ACM Int. Conf. Automated Software Engineering*, 2007, pp. 274–283.

40. X. Wang, Y.-C. Jhi, S. Zhu and P. Liu, Behavior based software theft detection, in *Proc. ACM Conf. Computer and Communications Security*, 2009, pp. 280–290.
41. P. P. F. Chan, L. C. K. Hui and S.-M. Yiu, Heap graph based software theft detection, *IEEE Trans. Inf. Forensics Secur.* **8**(1) (2013) 101–110.
42. D.-K. Chae, S.-W. Kim, S.-J. Cho and Y. Kim, Effective and efficient detection of software theft via dynamic API authority vectors, *J. Syst. Softw.* **110** (2015) 1–9.