# Plagiarism Detection of Multi-threaded Programs using Frequent Behavioral Pattern Mining

Qing Wang[1,2], Zhenzhou Tian[1,2*], Cong Gao[1,2], Lingwei Chen[3]
[1]School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an, China
[2]Shaanxi Key Laboratory of Network Data Analysis and Intelligent Processing, Xi'an, China
[3]College of Information Sciences and Technology, Pennsylvania State University, PA, USA
*Corresponding: tianzhenzhou@xupt.edu.cn

***Abstract*—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 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 cope better with multi-threaded plagiarism detection than alternative approaches.**

***Index Terms*—Software plagiarism, Dynamic birthmark, Multi-threaded program, Frequent pattern**

## I. 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 [2] such that about 13% of apps hosted in third-party marketplaces are repackaged [19], which poses serious threats to the healthy development of software industry.

In order to detect the evolving software plagiarism, different birthmarking techniques [10], [7], [12], [15], [4] 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 [12], [15], [4] 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 [13]. 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 [11]. Two dynamic birthmarking methods (i.e., thread-related system call birthmark (TreSB) [11] and thread-oblivious birthmark (TOB) [13]) have been proposed, yet they still suffer from either weak universality or limitation of overall behavior understanding in multiple threads.

To address the aforementioned 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 elaborate Apriori 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. The contributions of this paper are summarized as follows:

DOI reference number: 10.18293/SEKE2020-135.

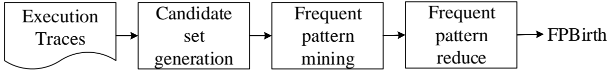



Figure 1: Basic flow of FPBirth extraction.

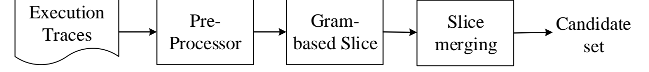



Figure 2: Basic process of pattern candidate set generation.

- 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.

## II. Problem Statement

In this section, we first define the software plagiarism detection problem. Given two multi-threaded programs $p$ and $q$, an input $I$ and a thread schedule $s$ to $p$ and $q$, a thread-aware dynamic software birthmark can be defined as a set of characteristics $f(p, I, s)$ extracted from program $p$ when executing $p$ with the input $I$ and schedule $s$ if and only if both of the following conditions are satisfied [14]:

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

Obviously, this is an abstract guideline 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, instead of enforcing exact birthmark matching, 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 further set up a threshold $\varepsilon$ to obtain the final results:

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

## III. 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 Figure 1.

### A. 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 Figure 2.

*1) Pre-Processor:* The pre-processor is to prune the captured execution traces, consisting 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 [5], which should be considered noises to be filtered out using their return values. Second, those system calls that are invoked randomly may perturb the execution traces, which should be also removed. For example, *futex*, providing a way to keep the thread blocked until certain conditions are met, can be only called when the expected blocking time is long enough; another kind of system calls that are responsible for memory management, such as *mmap* and *brk*, may be invoked only when a particular chunk of memory is involved.

*2) Gram-based Slice:* Due to its simplicity and scalability, $k$-gram model [8] 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 $s = (e_1, e_2, \cdots, e_n)$, a series of subsequences split by $k$-gram can be defined as $\mathrm{grams}(s, k) = \{g_i | g_i = (e_i, e_{i+1}, \cdots, e_{i+k})\}$ $(1 \leq i \leq n-k+1)$. 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.

*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 $\mathrm{CanSet}_p^I = \bigcup_{i=1}^{m} \mathrm{grams}(s_i, k)$ where $s_i$ is $p$'s $i$th execution trace under input $I$ and $m$ is the number of execution traces.

### B. Frequent Pattern Mining

Frequent pattern mining is an important research topic in data mining [3], which searches for recurring relationships in a given data set 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 [1] 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 $\sigma$) are generated by joining frequent $(t-1)$-itemsets with itself until no new frequent itemsets are identified. In this way, given a candidate set $\text{CanSet}_p^I$, the generated frequent pattern set over it can be defined as $\text{FreSet}_p^I = \{f_i | \text{count}(f_i) \geq \sigma, 1 \leq i \leq l\}$ where $f_i$ is $i$th frequent pattern in $\text{CanSet}_p^I$, and $l$ is the number of frequent patterns in $\text{FreSet}_p^I$.

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.

### C. Frequent Pattern Reduction

Using frequent pattern mining over $\text{CanSet}_p^I$, we may generate the frequent pattern set $\text{FreSet}_p^I$ 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 $\text{FreSet}_p^I$, FPBirth can be defined as $\text{FPBirth}_p^I = \{\langle f_i, \text{sup}(f_i)\rangle | f_i \in \text{FreSet}_p^I\}$ where $f_i$ is $i$th frequent pattern in $\text{FreSet}_p^I$, and $\text{sup}(f_i)$ is the frequency of pattern $f_i$ (i.e., support count).

## IV. FPBirth-based Software Plagiarism Detection

Using FPBirth, we can effectively and dynamically birthmark a multi-threaded program under a specified input. However, a 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 $\{I_1, I_2, \cdots, I_d\}$, we accordingly generate a set of FPBirth birthmark pairs for $p$ and $q$, which can be denoted as $\{(\text{FPBirth}_p^{I_1}, \text{FPBirth}_q^{I_1}), \cdots, (\text{FPBirth}_p^{I_d}, \text{FPBirth}_q^{I_d})\}$. 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:

$$\text{sim}\left(p_f, q_f\right) = \sum_{i=1}^{d} \text{sim}\left(\text{FPBirth}_p^{I_i}, \text{FPBirth}_q^{I_i}\right) \Big/ d \quad (2)$$

Based on $\text{sim}\left(p_f, q_f\right)$ and Eq. (1), we can obtain the final plagiarism detection results, where the threshold $\varepsilon$ is adjustable for different sample data set. 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 $[0, 1]$.

Table I: Benchmark multi-threaded programs

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

## V. Experimental Results and Analysis

### A. Experimental Setup

We evaluate the effectiveness of our proposed detection system over FPBirth on a public software plagiarism sample set [11], 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 I. 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 and TOB.

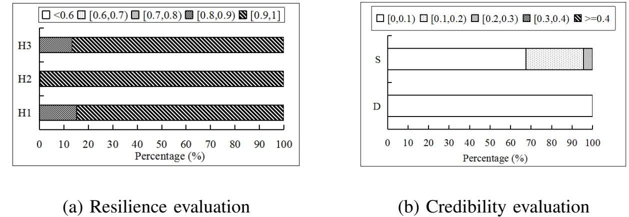
### B. FPBirth Evaluation

With these settings, we mainly evaluate the resilience and credibility of the thread-aware birthmark FPBirth [12], which can be described as follows [7]:

- *Resilience.* Let $p$ be a program and $q$ be a copy of $p$ generated by applying semantics-preserving code transformations $\tau$. A birthmark is resilient to $\tau$ if $\operatorname{sim}(p_f, q_f) \geq 1 - \varepsilon$.
- *Credibility.* Let $p$ and $q$ be independently developed programs. A birthmark is credible if it can differentiate the two programs, that is $\operatorname{sim}(p_f, q_f) < \varepsilon$.

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.

*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 with respect to the similarity distribution under three different obfuscations (H1, H2 and H3) are illustrated in Figure 3(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. From the results, we can observe that most of the comparison pairs




(a) Resilience evaluation (b) Credibility evaluation

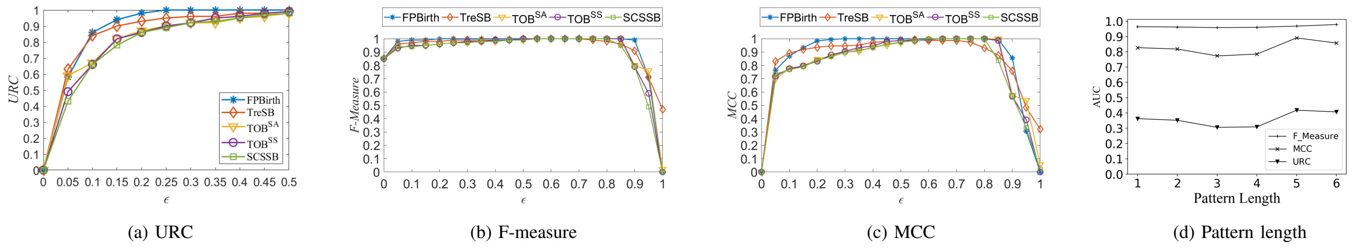
Figure 3: FPBirth Evaluation.

enforce a similarity higher than 0.9; this indicates that FPBirth birthmark enjoys an excellent resistance to the obfuscation strategies involved in this public data set.

*2) Credibility Evaluation:* In this experiment, the programs independently developed are selected from the data set to evaluate the credibility of FPBirth. More specifically, the selected experiment instances include 6 multi-threaded compression/decompression software, 7 web browsers, and 5 audio player software. We use FPBirth to birthmark the software and then calculate the similarity between them. Figure 3(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. From the results, 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 similarity 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 browsers use WebKit layout engines. Overall, FPBirth performs well in differentiating independently developed software.

### C. Comparisons with Traditional Birthmark Techniques

*1) Comparative Analysis on Detection Effect:* In this section, we compare FPBirth with TreSB [11] and TOB [13], two traditional thread-aware birthmark techniques, and SCSSB [16], a dynamic birthmark technique also based on system



(a) URC (b) F-measure (c) MCC (d) Pattern length

Figure 4: Comparative analysis on detection performance and pattern length.

calls. To quantitatively validate the effectiveness of different methods, we use `URC` (union of resilience and credibility) [17], `F-Measure`, `MCC` (matthews correlation coefficient) [6], and `AUC` (area under the curve) as the performance measures.

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

$$\text{URC} = 2 \times \frac{R \times C}{R + C} \tag{3}$$

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 $\varepsilon$. We set the effective value range of the threshold as 0-0.5, that is, $1-\varepsilon \geq \varepsilon$. Figure 4(a) shows the comparison 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:

$$\text{sim}(p_f, q_f) = \begin{cases} \geq \varepsilon & q \text{ is a copy of } p \\ < \varepsilon & q \text{ is not a copy of } p \end{cases} \tag{4}$$

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

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

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:

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

Figure 4(b) and Figure 4(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 MCC. Table II 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.

Table II: Comparison of birthmark techniques over AUC

| | SCSSB | TOBSA | TOBSS | 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 |

*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, in this section, we focus on comparing the time cost of FPBirth with others in terms of birthmark generation (Phase II) and similarity calculation (Phase III). Table III gives the average time cost of each birthmark. From the results, we can observe that the average time of FPBirth to generate birthmark is higher than other methods. 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 TOBSS using maximum weighted dichotomy matching. Though it is more time-consuming, FPBirth is still significant 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.

Table III: Comparison of of birthmarks over time cost (ms)

| | SCSSB | TOBSA | TOBSS | TreSB | FPBirth |
|---|---|---|---|---|---|
| Phase II | 103 | 103 | 103 | 102 | 1556 |
| Phase III | 0.1 | 0.1 | 20 | 0.02 | 9.9 |

### D. Evaluation on Pattern Length

As described in Section III-B, the length of frequent patterns directly affects the validity of the constructed birthmark. Therefore, this section specifically analyzes the impact of pattern length on the detection performance. Figure 4(d) displays the AUC values of URC, F-Measure and MCC for plagiarism detection using FPBirth with respect to different pattern lengths. We can see that 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 detection metrics, length 6 gives the best balance. This is the reason why we choose $k = 6$ for $k$-gram slice and the length of the input sequences 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 experimental results, pattern lengths ranging from 3 to 6 provide the best detection performance, which is what we've set up for our experiments.

## VI. Related Work

The existing software plagiarism detection work mainly falls into two categories: static birthmark and dynamic birthmark. For static birthmark, Xie et al. [17] introduced the weighted short sequence birthmark, DroidMoss [19] took hash value of bytecode fragments as birthmark, and ViewDroid [18] presented a functional view graph birthmark; For dynamic bithmark, Wang et al. [16] designed SCSSB (System Call Short Sequence Birthmark) and IDSCSB, and SUPB [9] constructed call sequence diagram of the program. These traditional dynamic birthmarks cannot well address the uncertainty caused by multi-threaded programs. Tian et al. [14] introduced the concept of thread-aware birthmark for the first time. Accordingly, two dynamic birthmarking methods TreSB [11] and TOB [13] were proposed to detect multi-threaded program plagiarism. Differently, our proposed FPBirth takes the frequent patterns from execution traces of multi-threaded programs as birthmark, which preserves the behavioral semantics and also improves the difference-tolerant ability.

## VII. 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 approaches in multi-threaded plagiarism detection.

## Acknowledgment

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] R. Agrawal, R. Srikant *et al.*, "Mining sequential patterns", in *icde*, vol. 95, 1995, pp. 3–14.

[2] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets", in *ICSE*, 2014.

[3] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions", *DMKD*, vol. 15, no. 1, pp. 55–86, 2007.

[4] 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 TSE*, vol. 41, no. 9, pp. 925–943, 2015.

[5] 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 *FSE*, 2014, pp. 389–400.

[6] B. W. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme", *BBA-Protein Structure*, 1975.

[7] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks", in *ISC*, 2004, pp. 404–415.

[8] G. Mylos and C. Collberg, "K-gram based software birthmarks", in *ACM symposium on Applied computing*, 2005, pp. 314–318.

[9] J. Park, D. Son, D. Kang, J. Choi, and G. Jeon, "Software similarity analysis based on dynamic stack usage patterns", in *RACS*, 2015, pp. 285–290.

[10] H. Tamada, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Design and evaluation of birthmarks for detecting theft of java programs", in *IASTED Conf. on Software Engineering*, 2004, pp. 569–574.

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

[12] Z. Tian, T. Liu, Q. Zheng, F. Tong, D. Wu, S. Zhu, and K. Chen, "Software plagiarism detection: a survey", *Journal of Cyber Security*, vol. 1, no. 3, pp. 52–76, 2016.

[13] Z. Tian, T. Liu, Q. Zheng, E. Zhuang, M. Fan, and Z. Yang, "Reviving sequential program birthmarking for multithreaded software plagiarism detection", *IEEE TSE*, vol. 44, no. 5, pp. 491–511, 2017.

[14] 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 *ICPC*, 2014, pp. 304–313.

[15] 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 Transactions on Software Engineering*, vol. 41, no. 12, pp. 1217–1235, 2015.

[16] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Detecting software theft via system call based birthmarks", in *2009 Annual Computer Security Applications Conference*. IEEE, 2009, pp. 149–158.

[17] X. Xie, F. Liu, B. Lu, and L. Chen, "A software birthmark based on weighted k-gram", in *2010 IEEE International Conference on Intelligent Computing and Intelligent Systems*, vol. 1. IEEE, 2010, pp. 400–405.

[18] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: Towards obfuscation-resilient mobile application repackaging detection", in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, 2014, pp. 25–36.

[19] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces", in *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM, 2012, pp. 317–326.