IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 9, 2020, accepted August 27, 2020, date of publication September 2, 2020, date of current version September 15, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3021184

Plagiarism Detection of Multi-Threaded Programs via Siamese Neural Networks

ZHENZHOU TIAN[1,2], QING WANG[1,2], CONG GAO[1,2], LINGWEI CHEN[3], AND DINGHAO WU[3], (Member, IEEE)

[1]School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an 710121, China
[2]Shaanxi Key Laboratory of Network Data Analysis and Intelligent Processing, Xi'an University of Posts and Telecommunications, Xi'an 710121, China
[3]College of Information Sciences and Technology, The Pennsylvania State University, University Park, PA 16802, USA

Corresponding author: Zhenzhou Tian (tianzhenzhou@xupt.edu.cn)

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

ABSTRACT Widespread intentional or unintentional software plagiarisms have posed serious threats to the healthy development of software industry. In order to detect such evolving software plagiarism, software dynamic birthmark techniques of better anti-obfuscation ability serve as one of the most promising methods. 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 multi-threaded program plagiarism detection. Some thread-aware birthmarking methods have been then proposed to address this issue, which nevertheless largely rely on manual feature engineering and empirical observations without any ground-truth training, and thus require domain knowledge, making them inflexible to be deployed in the wild. Inspired by the success of self-guided optimization using deep neural networks and their superior feature learning ability, in this article, we transform multiple execution traces for each multi-threaded program under a specified input to the plain feature matrix, and feed it to the deep learning framework to learn latent representation as thread-aware birthmark that enjoys better semantic richness and perturbation resistance; instead of empirically determining the plagiarism over direct birthmark similarity metric, we further build up sophisticated siamese neural networks to supervise birthmark construction, similarity measurement, and decision making. Integrating our proposed method, a system called *NeurMPD* is developed to perform **Neur**al network-based **M**ulti-threaded program **P**lagiarism **D**etection. The experimental results based on a public software plagiarism sample set demonstrate that NeurMPD copes better with multi-threaded plagiarism detection than alternative approaches.

INDEX TERMS Software plagiarism detection, multi-threaded programs, dynamic birthmark, semantic behaviors, deep learning, siamese neural network.

I. INTRODUCTION

Open-source software communities and social coding platforms, such as GitHub, Stack Overflow, and CodeShare, have been enjoying explosive growth for recent years. For example, GitHub is hosting more than 100 million software projects maintained by over 50 million registered developers in 2019 [12]. The worldwide accessibility to these highly interoperable and collaborative social coding environments has drastically reshaped the software programming ecosystem that allows the developers all around the globe to conveniently reuse code snippets and libraries or adapt existing ready-to-use projects during the process of software development [39], [44]. Unfortunately, such apparent benefits attract not only developers and researchers to legitimately study programming and understand software structure for extensions and comparisons, but also some individuals and companies to violate the open source license to illegally incorporate others' software code into their own commercial products for profit. To put it into perspective, the recent software

The associate editor coordinating the review of this manuscript and approving it for publication was Haider Abbas.

plagiarism incidents include the lawsuit against Verizon by Free Software Foundation for distributing Busybox in its FIOS wireless routers [3], the crisis of Skype's VOIP service for the violation of licensing terms of Joltid [2], the suspicion on APICloud for directly misappropriating dll files and source code from DCloud [1], and the scandal surrounding a Chinese startup's ''self-made'' web browser Redcore for plagiarizing the substantial code from Google Chrome [4]. Due to the openness of Android [9], application (app) plagiarism has become even more severe through repackaging [7] such that about 13% of apps hosted in third-party marketplaces are repackaged [8], [48]. In addition, some downstream software companies may integrate the software components delivered in binary form from upstream companies into their own products without awareness of possible license violations [14]. As a result, the detection of these widespread intentional or unintentional software plagiarisms is of major concern to both software companies and programmers in order to curb serious threats to the healthy development of software industry they pose.

In order to detect the evolving software plagiarism, different birthmarking techniques [17], [18], [27], [32], [38] have been developed in recent years. In these methods, software birthmark, which is a set of features of good invariability, 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 [32], [33], dynamic birthmarking techniques [17], [34], [38], [46] 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 and detection performance.

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 [36]. For instance, given an input, birthmarks constructed 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 [35]. In recent years, some dynamic birthmarking methods have been proposed to cope with multi-threaded program plagiarism detection [35], [36], [40], [47]. Despite the promising results accomplished, most of these methods have largely relied on manual feature engineering over execution traces to extend the traditional dynamic birthmarks and empirical observation to determine the plagiarism without any ground-truth training. Defining such features and detection models, nevertheless, requires domain knowledge *a priori*, making these methods faced with either weak universality or limitation of behavior understanding and representation learning in multiple threads, and thus cumbersome and inflexible to be deployed in the wild. To this end, it calls for a better paradigm to formulate dynamic thread-aware birthmarks for the multi-threaded programs and detect the plagiarism among them in an automatic way.

Deep neural networks (DNNs) have been widely adopted in a variety of machine-learning tasks, ranging from computer vision [22], speech recognition [15] to natural language processing [5], many of which have achieved state-of-the-art performance. More importantly, they leverage many layers of non-linearities to capture invariances from transformation in the raw input space [21], and have thus boosted the semantic richness and robustness for the learned representations [16]. Inspired by the success of this self-guided representation learning through DNNs, we would like to enable such a paradigm to automatically abstract the behaviors of the multi-threaded programs from their plain 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 deep learning framework [24] to learn the representation that encodes not only the semantics and structures of each execution trace but also the inherent relationships among them. These representations for the multi-threaded programs are powerful, flexible, and difference-tolerant because they are learned via a supervised metric-based approach without imposing any prior knowledge, and thus more task-specific to be capable of distinguishing one plagiarized program from the rest. Accordingly, these latent representations can be considered as our thread-aware birthmarks to thwart the impact of interleaving threads. Furthermore, in order to facilitate the supervised representation learning and proceed with automatic similarity measurement and decision making, we further devise the deep learning framework with siamese neural networks [21] to train the model with labeled pairs of multi-threaded programs, which enables the model to generalize successfully to the test programs. We develop a system called *NeurMPD* (i.e., Neural network-based Multi-threaded program Plagiarism Detection) integrating our proposed method. In summary, this article has the following major contributions:

- We explore a novel perspective of dynamic birthmark construction for multi-threaded programs, where we take advantage of superior feature learning ability of DNNs, transform multiple execution traces for each multi-threaded program under the same input to the plain feature matrix, and feed it to the deep learning framework to learn the latent representation as thread-aware birthmark. The proposed method is automatic and computationally tractable, while the constructed birthmarks can capture intrinsic properties of programs and tolerate differences among execution traces as well.
- Instead of empirically determining the plagiarism between the programs over direct birthmark similarity metric, we further build up siamese neural networks to supervise the birthmark construction, similarity measurement, and decision making, and then reuse the trained networks over multiple pairs of birthmarks under

different program inputs to approximate the final inference output without any retraining.
- Comprehensive experimental studies on a public software plagiarism sample set are conducted to demonstrate that our developed plagiarism detection system NeurMPD can achieve the state-of-the-art results, which also outperforms alternative baselines.

The rest of this article is organized as follows. Section II introduces the problem statement. Section III presents our proposed method in detail. Section IV systematically evaluates the effectiveness of our developed system NeurMPD in plagiarism detection of multi-threaded programs. Section V discusses the related work. Finally, Section VI concludes.

II. PROBLEM STATEMENT

In this section, we first define the software plagiarism detection problem before diving into the technical details for NeurMPD in the following section. Table 1 shows some important notations used in this article. Given two programs *plaintiff* and *defendant* where plaintiff refers to the original program and defendant refers to the suspect program, the goal of software plagiarism detection is to determine whether the defendant is a copy of the plaintiff. To this end, software birthmarks are typically extracted from programs in an either static or dynamic fashion to measure the similarity between plaintiff and defendant [36], [38]. However, as aforementioned, these traditional static and dynamic birthmarks are ineffective to deal with multi-threaded program plagiarism due to its non-deterministic thread scheduling under a fixed input. Thread-aware software birthmarking is hence in need to address this challenge.

TABLE 1. Notations.

Notation	Description
p	A plaintiff multi-threaded program
q	A defendant multi-threaded program
I	An input for program
ζ	A thread schedule
$f(p, I, \zeta)$	An attribute set from p with input I and schedule ζ
ε	A threshold to determine the plagiarism
s	An execution trace
n	Number of execution traces for a program under an input
e	A system call
m	Number of system calls in an execution trace
u	Number of inputs for a program
$\mathbf{X}_p^I$	A plain feature matrix of p under input I
$\mathbf{e}$	An embedding space of system call learned by CNN
d	Dimension of $\mathbf{e}$
k	Dimension of LSTM hidden layer
$\mathbf{x}_p^I$	A birthmark of p under input I
$\varphi_\theta(\cdot)$	Similarity metric using MLP weighted by θ
$\mathcal{D}$	Training sample pairs
$\mathcal{Y}, y$	Ground truth

Specifically, given two multi-threaded programs p and q, an input I and a thread schedule ζ to p and q, a thread-aware dynamic software birthmark can be defined as a set of attributes $f(p, I, \zeta)$ extracted from program p when executing p with the input I and schedule ζ if and only if both of the following conditions are satisfied [37]:

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

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, the similarity between the original program p's birthmark and the suspect program q's birthmark $\text{sim}(f(p, I, \zeta), f(q, I, \zeta))$ is generally measured to determine the plagiarism. The higher the similarity, the more possible the suspect program q copies code from the original program p. Built upon the similarity, a threshold ε is accordingly set up to obtain the final results:

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

Such an empirical and simple inference procedure over direct similarity metric is error-prone and difficult to generalize to other unknown datasets. In this article, we leverage siamese neural networks [21] to train the model with a large number of ground-truth sample pairs, while the trained model is further used to determine the plagiarism through ensemble.

III. PROPOSED METHOD

In this section, we present the details of NeurMPD that how we construct thread-aware birthmarks for multi-threaded programs over their execution traces and how we elaborate siamese neural networks over such constructed dynamic birthmarks for plagiarism detection. The overview of NeurMPD is shown in Figure 1, which consists of the training and detection (testing) phases with steps of execution trace extraction, thread-aware birthmark construction, model training using siamese neural networks, and plagiarism detection through ensemble.

A. EXECUTION TRACE EXTRACTION

The thread interleaving in multi-threaded programs leads to changes in the program execution traces, an example of which is illustrated in Figure 2. To capture such unique behaviors and semantics 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 learn the latent representation over execution traces to formulate birthmark.

Specifically, execution trace for a multi-threaded program is a sequence of system calls related to program and thread operations, such as thread and process management (e.g., creation, join and termination, capability setting and getting), thread synchronization, signal manipulating, thread and process priority setting, etc., while system calls are essential for user applications to request the kernel services of the

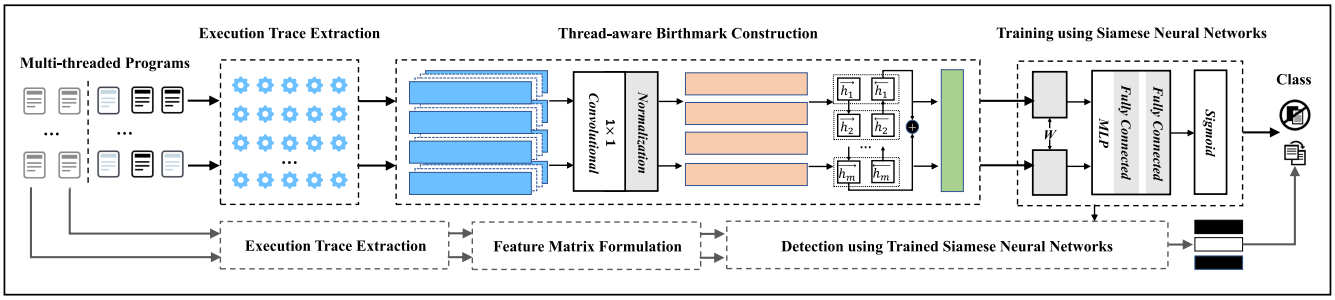


FIGURE 1. The overview of system NeurMPD integrating execution trace extraction, thread-aware birthmark construction, similarity measurement, and decision making chained by siamese neural networks.

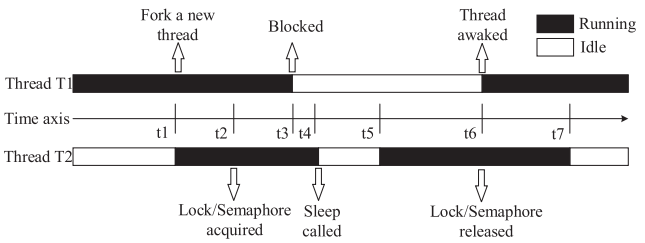


FIGURE 2. Execution examples of a multi-threaded program [35].

operating system and thus difficult to delete, replace and tamper with. In our work, we run a dynamic profiler to capture those execution traces for multi-threaded programs. With the help of the Pin dynamic instrumentation framework and the use of the interface functions *PIN_AddSyscallEntryFunction* and *PIN_AddSyscallExitFunction*, monitoring and analysis codes are implanted before and after the system call invoking positions respectively to capture the relevant system call information during the execution of the program. The form of each system call is specified as <ID of the thread where the call occurs, system call number, system call name, specific parameter, return value>. Accordingly, a system call sequence is formulated by the system call numbers.

However, the raw execution traces are not applicable for direct thread-aware birthmark construction. First, those system calls that fail to correctly reflect the program's behaviors [25] 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 the captured execution traces before fed to birthmark construction.

B. THREAD-AWARE BIRTHMARK CONSTRUCTION

Given a set of extracted execution traces for each multi-threaded program under the same input, it is promising to utilize sequence learning [31] to learn the embedding for each execution trace, and then explore max-pooling, averaging or concatenation to aggregate the embeddings of all execution traces as the final representation (i.e., birthmark) for each program. However, such an implementation may not significantly capture the variations and associations among different execution traces. In order to learn the representation that encodes not only the semantics and structures of each execution trace but also the inherent relationships among them, we transform the multiple execution traces for each multi-threaded program under the same input to the plain feature matrix, and feed it to the deep learning framework to learn the latent representation as thread-aware birthmark.

1) FEATURE MATRIX FORMULATION

Given a set of pre-processed execution traces, we denote a multi-threaded program under the input I to be of the form $\text{TraSet}_p^I = \{s_1, s_2, \cdots, s_n\}$ of n execution traces, where each execution trace $s = (e_1, e_2, \cdots, e_m)$ contains m system call numbers. Therefore, the plain feature matrix $\mathbf{X}_p^I \in \mathbb{R}^{n\times m}$ can be formulated as follows:

$$\mathbf{X}_p^I = \begin{bmatrix} e_{11}, e_{12}, e_{13}, \cdots, e_{1m} \\ e_{21}, e_{22}, e_{23}, \cdots, e_{2m} \\ e_{31}, e_{32}, e_{33}, \cdots, e_{3m} \\ \vdots \\ e_{n1}, e_{n2}, e_{n3}, \cdots, e_{nm} \end{bmatrix} \tag{2}$$

Note that, the lengths of the extracted execution traces for the same program p under the same input I may be different due to the thread interleaving, while obviously, the lengths of the extracted execution traces vary in different programs with different inputs and different execution times. Accordingly, we only intercept the first m system calls from n executions to obtain our initial feature matrix. For the case that the number of the captured system calls in the execution trace is less than m, we zero-pad that execution trace to the right so that its length can be extended to m.

Considering that each element in the plain feature matrix $\mathbf{X}_p^I$ specifies an individual system call number, we need to map each of them into a d-dimensional embedding vector before proceeding with representation learning for birthmark construction. Here we shift the word-context concept in a text corpus into execution traces, and utilize computationally efficient skip-gram [26] to learn the embedding for each system call, which is applied on execution traces to minimize the loss of observing a system call's neighborhood (within a window w) conditioned on its current embedding [43]. The objective function of skip-gram can be defined as [29]:

$$\arg\min_{\phi} \sum_{-w\le j\le w, i\ne j} -\log p(e_{i+j}|\phi(e_i)), \tag{3}$$

where $\phi(e_i)$ is the current embedding of e_i. After embedding each system call, $\mathbf{X}_p^I$ can be further transformed to an $n \times m \times d$-dimensional space.

2) LATENT REPRESENTATION LEARNING FOR BIRTHMARK

As advanced neural network structures, both convolutional neural network (CNN) [24] and Long Short-Term Memory (LSTM) [13] have achieved great success in learning salient features for classification tasks, where CNN is known to capture the local correlations while LSTM excels in modeling the sequential dependency. Therefore, taking the generated plain feature matrices from the previous section as inputs, we devise a deep learning framework leveraging the crafty architecture of CNN and LSTM to learn the latent representations for programs, which serves as our thread-aware birthmarks.

First, we take advantage of CNN's capability of learning local correlations with shared weights, and feed each multi-threaded program's plain feature matrix to the convolutional layer to aggregate the system call information in the same column but different rows (i.e., different execution traces) and thus formulate a new single execution trace of higher-level concept for sequence modeling in the next step. Specifically, given $\mathbf{X}_p^I \in \mathbb{R}^{n\times m\times d}$ (n can be also interpreted as number of channels for the input), the designed CNN builds up a pair of convolutional layer and normalization layer; in the convolutional layer, the raw feature matrix gets convoluted by 1×1 kernel, and then abstracted to $\widehat{\mathbf{X}_p^I}$ of 1 channel with dimensionality reduction to $m \times d$.

Afterwards, the resulting feature matrix $\widehat{\mathbf{X}_p^I} \in \mathbb{R}^{m\times d}$ from CNN (i.e., an execution trace with m abstracted system calls in a new d-dimensional embedding space), is then passed through LSTM to learn the sequential dependency and output the final representation. A LSTM is an architecture designed for recurrent neural network to address the vanishing/exploding gradient issue [31]. The designed LSTM reads the input execution trace $(\mathbf{e}_1, \mathbf{e}_2, \cdots, \mathbf{e}_m)$ through the hidden layer function $\mathcal{H}$ so that each hidden layer vector $\mathbf{h}_t$ at timestep t can be denoted as

$$\mathbf{h}_t = \mathcal{H}(\mathbf{e}_t, \mathbf{h}_{t-1}), \quad \mathbf{h}_t \in \mathbb{R}^k \tag{4}$$

where $\mathcal{H}$ is implemented using memory cells to store information, which can be formulated as the following composite functions [13]:

$$\mathbf{i}_t = \sigma(\mathbf{W}_{ei}\mathbf{e}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{W}_{ci}\mathbf{c}_{t-1} + \mathbf{b}_i) \tag{5}$$
$$\mathbf{f}_t = \sigma(\mathbf{W}_{ef}\mathbf{e}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{W}_{cf}\mathbf{c}_{t-1} + \mathbf{b}_f) \tag{6}$$
$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tanh(\mathbf{W}_{ec}\mathbf{e}_t + \mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c) \tag{7}$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_{eo}\mathbf{e}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{W}_{co}\mathbf{c}_{t-1} + \mathbf{b}_o) \tag{8}$$
$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t) \tag{9}$$

where σ is the logistic sigmoid function, $\mathbf{i}_t$, $\mathbf{f}_t$, $\mathbf{o}_t$, $\mathbf{c}_t$ are the input gate, forget gate, output gate, and cell activation vectors respectively, $\mathbf{W}$s are the weight matrices, $\mathbf{b}$s are the bias vectors, and $\circ$ is the point-wise product between two vectors. In order to learn both the forward and backward sequential dependency and global contextual information in the execution trace, we utilize bidirectional LSTM (BiLSTM) so that hidden layer vector $\mathbf{h}_t$ at timestep t can be concatenated as

$$\mathbf{h}_t = [\overrightarrow{\mathbf{h}_t}; \overleftarrow{\mathbf{h}_t}]. \tag{10}$$

After forward and backward reading the execution trace $(\mathbf{e}_1, \mathbf{e}_2, \cdots, \mathbf{e}_m)$, the concatenation of the last two hidden states $[\overrightarrow{\mathbf{h}_m}; \overleftarrow{\mathbf{h}_m}]$ acts as the program's thread-aware birthmark $\mathbf{x}_p^I \in \mathbb{R}^{2k}$ under a specified input I.

C. TRAINING USING SIAMESE NEURAL NETWORKS

The representation learning process presented in the previous section is a complete forward propagation, which still requires a backpropagation optimization to update the parameters and thus learn useful and predictive representations for birthmark construction. Also, as discussed in Section I and Section II, traditional software plagiarism inference procedure [35], [36], [40], [47] simply measures the similarities between pairs of birthmarks and empirically analyzing such similarities to determine the plagiarism, which lacks necessary training and requires significant domain knowledge, and is hence error-prone and infeasible to be deployed in the practical use. To address this issue, we elaborate sophisticated siamese neural networks [21], [30] to supervise the representation learning, similarity measurement, and decision making over ground truth.

Siamese networks [6] have been widely used to leverage similarities of input sample pairs for a variety of tasks (e.g., one-shot image recognition, image matching, etc.). They are generally applicable to the scenarios that the number of categories is relatively large and the number of samples in each category is small, while especially feasible to address the learning issues that the number of categories is unknown and in a changing state. More specifically, a siamese neural network consists of twin networks with the same structure and shared weights, which reads different inputs, maps them to the target space respectively, and then uses the distance function to join them for similarity measurement; in the training phase, some optimization strategy is adopted to evaluate the loss between the output and the corresponding ground truth.

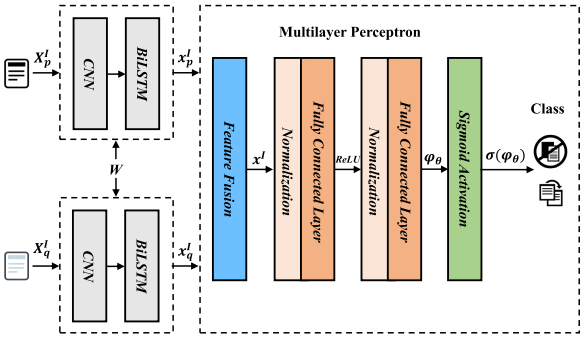


FIGURE 3. The structure of siamese neural networks for multi-threaded program plagiarism detection, including twin networks consisting of CNN and BiLSTM, feature fusion, and multilayer perceptron for similarity metric.

Therefore, such a network of symmetric structure guarantees that two similar inputs will be mapped by their respective networks to similar feature space, while distinct inputs can be effectively differentiated.

In this article, our designed model, which is displayed in Figure 3, is a deep siamese neural network, where each twin network accepts two plain feature matrices $\mathbf{X}_p^I$ and $\mathbf{X}_q^I$, and successively passes them through CNN and BiLSTM for representation learning to obtain birthmarks $\mathbf{x}_p^I$ and $\mathbf{x}_q^I$ respectively, which has been detailed in Section III-B. Typically, in the top conjoining layer, similarity metric is directly computed over like and unlike pairs to update the energy [10]. Differently, in our work, we perform feature fusion over birthmarks $\mathbf{x}_p^I$ and $\mathbf{x}_q^I$, and use a multilayer perceptron [30] with multiple fully-connected layers to learn the similarity metric before a sigmoid activation that maps onto the interval [0, 1] to facilitate minimizing cross-entropy loss. Thus, the fused feature space can be presented as the concatenation of $\mathbf{x}_p^I$ and $\mathbf{x}_q^I$ as follows:

$$\mathbf{x}^I = [\mathbf{x}_p^I;\ \mathbf{x}_q^I], \quad \mathbf{x}^I \in \mathbb{R}^{4k} \tag{11}$$

and the similarity metric φ is learned by:

$$\varphi_\theta(\mathbf{x}_p^I, \mathbf{x}_q^I) = \mathrm{MLP}_\theta(\mathbf{x}^I). \tag{12}$$

The θ are additional parameters that are learned by the network during training. Let $(\mathbf{X}_p^I, \mathbf{X}_q^I) \in \mathcal{D}$ represent the training sample pairs, and $y \in \mathcal{Y}$ denote the ground truth label where $y = 0$ implies that q is not a copy of p, while $y = 1$ specifies that q is a copy of p. The cross-entropy loss on our binary classifier can be formulated as:

$$\mathcal{L} = \sum_{(\mathbf{X}_p^I, \mathbf{X}_q^I) \in \mathcal{D}} -y \log(P) - (1-y)\log(1-P) \tag{13}$$

$$\text{s.t. } P = \sigma(\varphi_\theta(\mathbf{x}_p^I, \mathbf{x}_q^I)) \tag{14}$$

where σ is sigmoid activation function. All the parameters can be efficiently updated using some gradient descent optimization algorithms (e.g., Adam [19]). From the training procedure, we can see that the final output of the siamese neural networks is the probability for the input sample pair $(\mathbf{X}_p^I, \mathbf{X}_q^I)$ that program p is plagiarized by program q. Considering that this is a binary classification problem, we can simply use 0.5 or other well-defined thresholds shown in Eq. (1) to make the decision.

Our designed siamese neural networks for supervised thread-aware birthmark construction and software plagiarism detection yield some significant benefits: (1) the learning process does not rely upon domain knowledge by instead exploiting deep learning frameworks; and (2) the neural networks are easily trained using standard optimization techniques on paired samples yet the learned representations are task-specific and difference-tolerant which can be flexibly generalized to the unseen data.

D. DETECTION THROUGH ENSEMBLE

In the previous section, a siamese neural network is presented to train our software plagiarism detection model in a supervised setting. Based on the trained neural networks with updated weights, we can effectively and automatically birthmark a multi-threaded program and output the plagiarism probability (or prediction label) between plaintiff and defendant programs under a specified input. However, a birthmark merely abstracts part of the semantics and behaviors of the program under a single input, resting with 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 birthmarks to thoroughly represent the semantics and behaviors of the program. Given a plaintiff program p, a defendant program q, and a set of inputs $\{I_1, I_2, \cdots, I_u\}$, we can generate a set of plain feature matrix pairs for p and q, which can be denoted as $\{(\mathbf{X}_p^{I_1}, \mathbf{X}_q^{I_1}), (\mathbf{X}_p^{I_2}, \mathbf{X}_q^{I_2}) \cdots, (\mathbf{X}_p^{I_u}, \mathbf{X}_q^{I_u})\}$; passing these plain feature matrix pairs through the trained siamese neural networks, we can accordingly obtain the corresponding plagiarism probabilities $\{\sigma(\varphi_\theta(\mathbf{x}_p^{I_1}, \mathbf{x}_q^{I_1})), \sigma(\varphi_\theta(\mathbf{x}_p^{I_2}, \mathbf{x}_q^{I_2})) \cdots, \sigma(\varphi_\theta(\mathbf{x}_p^{I_u}, \mathbf{x}_q^{I_u}))\}$. Instead of determining the plagiarism using a single pair of execution trace inputs, we utilize a bagging-like ensemble to average the predictions over all the individual outputs, and take their mean value to approximate the inference result between p and q, which can be denoted as follows:

$$\mathrm{sim}\left(p_f, q_f\right) = \sum_{i=1}^{u} \sigma(\varphi_\theta(\mathbf{x}_p^{I_i}, \mathbf{x}_q^{I_i})) \Big/ u \tag{15}$$

Based on $\mathrm{sim}\left(p_f, q_f\right)$ and Eq. (1), we can obtain the final plagiarism detection results, where the threshold ε can be fine-tuned for best detection performance; trained by the ground

TABLE 2. Benchmark multi-threaded programs.

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

truth, the specified ε may work for plagiarism detection over other datasets as well.

IV. 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 system NeurMPD which integrates the above proposed method in multi-thread program plagiarism detection.

A. EXPERIMENTAL SETUP

a: DATASETS

We evaluate the effectiveness of our proposed detection system NeurMPD on a public software plagiarism sample set [35], including 234 versions of 35 mature multi-threaded programs implemented in C or Java. These different versions are derived from a series of obfuscations as follows:

- Use relatively weak obfuscations provided by different compilers, optimization levels and debugging options (gcc, llvm, o0-os, -g) to generate the target code.
- Apply professional obfuscation tools, including Sandmark, Zelix KlassMaster, Allatori, DashO, Jshrink, ProGuard and RetroGuard, to construct strong obfuscated programs.
- Exploit packing tool UPX (the ultimate packer for executables) to process the target code.

The data statistics are summarized in Table 2, where Column #Ver gives the number of versions for each multi-threaded program including the original and its obfuscated ones; Column Size lists the number of kilobytes for the largest version, with its version number listed in Column Version; the programs include six compression/decompression software, five audio players, ten web browsers, four Java programs from the JavaG benchmark, and ten programs from the PARSEC 3.0 benchmark.

b: PARAMETER SETTING

The parameter setting to implement our model for evaluation is specified as: under each input, $n = 4$ for the number of execution traces extracted and $m = 256$ for the number of system calls in each execution trace; $d = 50$ for the dimension of each system call embedding space; $k = 100$ for the dimension of LSTM hidden layer space. Some other fine-tuned hyperparameters set for siamese neural network training are detailed in Table 3. We leverage the EarlyStopping mechanism for model learning, where training is stopped when the accuracy rate no longer rises, so as to avoid over-fitting, non-convergence and other problems. As for the baselines, we compare our approach with two multi-threaded program plagiarism detection methods TreSB [35] and TOB [36], and system call-based dynamic birthmark technique SCSSB [41].

TABLE 3. Setting of model parameters.

Parameter	Value	Parameter	Value
embedding_dim	50	max_sequence_length	1024
lstm_hidden_dim	100	dense_dim	256
lstm_drop_rate	0.17	dense_drop_rate	0.25
activation_func	relu	validation_split_ratio	0.2

c: DATA SAMPLING FOR TRAINING AND TESTING

In the model training stage, 7,830 pairs of program samples are used, including 7,647 positive pairs and 183 negative pairs. Since our classification is binary and the positive and negative samples are not balanced, the over-sampling algorithm SMOTE is used to generate new samples by interpolation method, which solves the training data imbalance problem. Specifically, It employs k-nearest neighbor method for a minority sample (k value needs to be specified in advance) to find the k nearest minority samples that are close in the feature space, which is measured using Euclidean distance, and then one of these neighbors is randomly selected to facilitate new sample generation in the way that:

$$x_{new} = x_i + \left(\hat{x}_i - x_i\right) \times \delta \tag{16}$$

where x_i is the nearest neighbor, and $\delta \in [0, 1]$ is a random number. The minority samples generated by SMOTE are easy to overlap with the surrounding majority samples such that it is difficult to perform classification. To address this issue, data cleaning techniques can be integrated with SMOTE to deal with overlapping samples after oversampling. Such

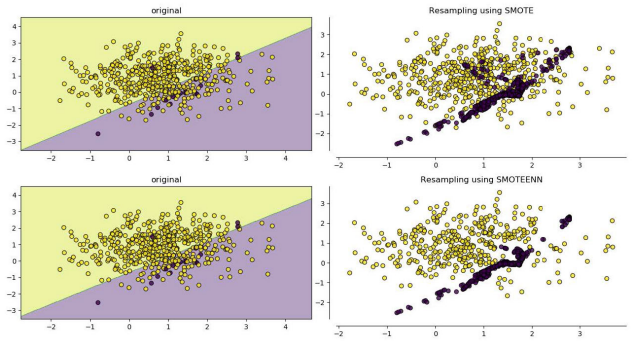


FIGURE 4. Data oversampling using SMOTE and SMOTEENN.

major pipelines include "SMOTE + ENN" and "SMOTE + Tomek", while "SMOTE + ENN" usually manages to remove more overlapping samples than "SMOTE + Tomek". To put it into perspective, we create an example dataset that accords with our actual data, where the number of samples is 20,000 and the ratio of positive and negative samples is 30:1, and we use SMOTE and SMOTEENN (short for "SMOTE + ENN") to over-sample the negative data (and further clean the overlapping samples for SMOTEENN). The oversampling results are illustrated in Figure 4: the original data is displayed on the left side, while the upper right is the samples generated by SMOTE and the lower right is the data generated by SMOTEENN. Clearly, SMOTEENN not only increases the number of the negative samples, but also removes those that overlap with the positive, which benefits our experimental evaluations. Therefore, we adopt SMOTEENN to over-sample the negative program pairs, such that we prepare 15,290 pairs of samples (7,647 positive and 7,643 negative) for model training, where sample pairs are randomly split as 80% for training and 20% for testing.

B. EVALUATION OF NeurMPD

With aforementioned experimental setup, we first evaluate the effectiveness of our developed plagiarism detection system NeurMPD by classification performance, and resilience and credibility.

1) EVALUATION ON CLASSIFICATION PERFORMANCE

To quantitatively validate the classification effectiveness of NeurMPD, we use `Recall`, `Precision`, `F-Measure`, and `ROC` (receiver operating characteristic) as the performance measures. 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:

$$\text{sim}(p_f, q_f) = \begin{cases} \geq \varepsilon & q \text{ is a copy of } p \\ < \varepsilon & q \text{ is not a copy of } p \end{cases} \tag{17}$$

Accordingly, given true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN), Recall and Precision can be defined as follows:

$$Recall = \frac{TP}{TP + FN} \tag{18}$$

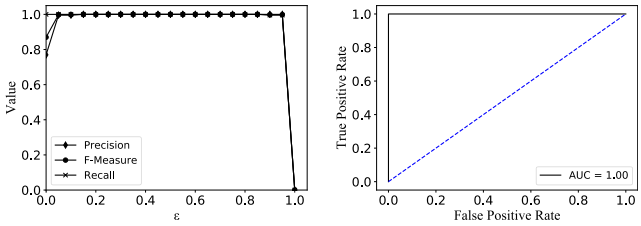
$$Precision = \frac{TP}{TP + FP} \tag{19}$$

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

$$\text{F-Measure} = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{20}$$

To obtain ROC curve that is a powerful tool to study the generalization performance of the classifier from the perspective of threshold selection, we change the value of threshold ε from 0 to the maximum according to the prediction results. In this process, two values true positive rate (TPR) and false positive rate (FPR) are further calculated to facilitate ROC curve formulation.

Based on the model parameter settings described in Table 3, the experimental results under different threshold values are illustrated in Figure 5(a) and Figure 5(b). From the results, we can observe that when $\varepsilon \geq 1$, NeurMPD succeeds in retaining all of our classification measures (i.e., Recall, Precision, and F-Measure) at 99%, while false negative rate (FNR = 1 − Recall) achieves less than 1%. This implies that (1) most of plagiarism pairs and independently developed pairs are both correctly classified; and (2) the similarity metric for plagiarism pairs is extremely close to 1 while the similarity metric for independently developed pairs is approaching to 0. The reason behind this is that the latent representations learned by our designed deep learning framework can significantly capture the semantics and behaviors of the multi-threaded programs, while supervised learning with siamese neural networks further enhance the birthmarks to be task-specific. In this respect, the impact of thread interleaving in multi-threaded programs can be effectively alleviated.

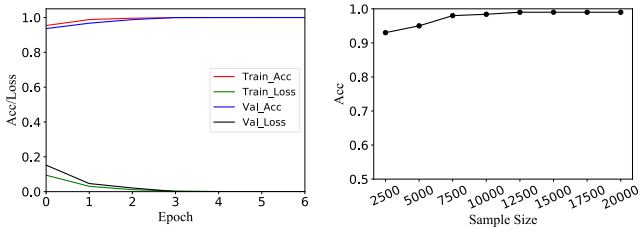


(a) Classification measures (b) ROC curve

FIGURE 5. Evaluation on classification performance (a) Recall, Precision, and F-Measure and (b) ROC curve under different thresholds.

We also report the training accuracy and loss with respect to different epochs for our designed siamese neural networks in Figure 6(a), and classification accuracy regarding different sample size in Figure 6(b). Figure 6(a) demonstrates that NeurMPD guarantees efficient convergence where the training accuracy and validation accuracy reach to the optimum, and the loss drops to a very low and stable value at the third epoch. From Figure 6(b), we can see that as the number of

sample pairs increases, the test accuracy increases as well. It is not difficult to understand that deep neural networks benefit from large data sets, while a smaller number of training sample pairs may easier enforce model over-fitting and underperforming. These encouraging classification performances and reasonable model parameter requirements show that our proposed NeurMPD can be applied in a realistic setting to detect the plagiarism of multi-threaded programs.



(a) Training accuracy and loss (b) Accuracy on sample size

FIGURE 6. The influence of epoch and sample size on NeurMPD: (a) training accuracy and loss; (b) test accuracy.

2) EVALUATION ON RESILIENCE AND CREDIBILITY

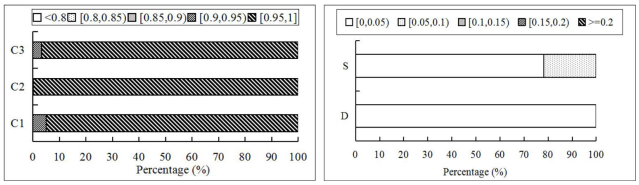
In this set of experiments, we mainly evaluate the resilience and credibility of our thread-aware birthmarks constructed in NeurMPD (i.e., detection capability of NeurMPD), which can be specified as [28]:

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

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

a: RESILIENCE

The benchmark program is taken as the plaintiff program p while the obfuscated program is considered as the defendant program q so that a series of plaintiff-defendant program pairs are formulated to evaluate the resilience of NeurMPD. The experimental results with respect to the similarity distribution under three different obfuscations (C1, C2 and C3) are illustrated in Figure 7(a), where C1 uses different compilers and optimizations (e.g., llvm, gcc, o0-oS) for weak obfuscation, C2 applies professional obfuscation tools (e.g., Zelix, ProGuard, Allatori, Jshrink) for strong obfuscation, and C3 uses UPX for packing. From the results, we can observe that most of the program pairs enforce a similarity higher than 0.95 and a few fall between 0.9 and 0.95; this indicates that NeurMPD enjoys an excellent resistance to the obfuscation strategies performed in this data set.



(a) Resilience evaluation (b) Credibility evaluation

FIGURE 7. Evaluation on resilience and credibility.

b: CREDIBILITY

The programs independently developed are selected from the data set to evaluate the credibility of NeurMPD. Specifically, the selected programs include 6 multi-threaded compression/decompression software, 10 web browsers, and 5 audio player software. Figure 7(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 0.05. This indicates that NeurMPD 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 program sample pairs with a similarity between 0.05 and 0.1 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 higher than others, since both browsers use WebKit layout engines. Overall, NeurMPD performs well in differentiating independently developed software.

C. COMPARISONS WITH OTHER DETECTION MODELS

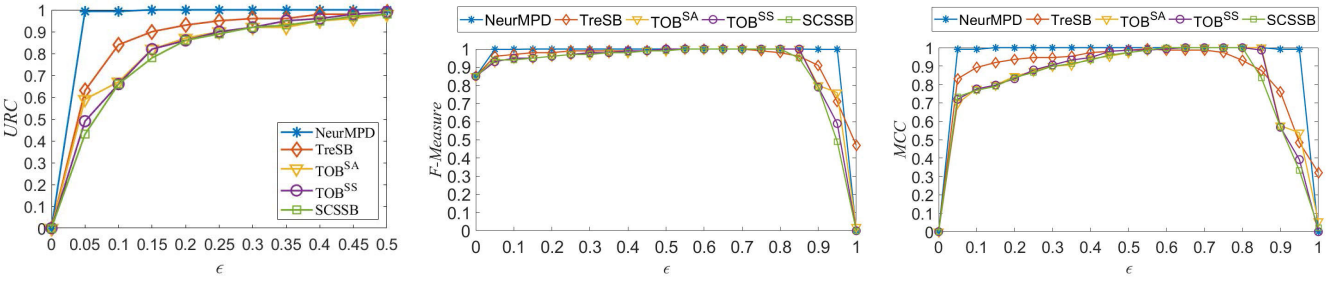
We also compare NeurMPD with two multi-threaded program plagiarism detection methods TreSB [35] and TOB [36] (TOB on slice aggregation (TOB^{SA}) and TOB on slice set (TOB^{SS}), and a system call-based dynamic birthmark method SCSSB [41] by detection performance and time cost.

1) COMPARATIVE ANALYSIS ON DETECTION PERFORMANCE

In this section, we measure the detection performance for each model using `URC` (union of resilience and credibility), `F-Measure`, `MCC` (matthews correlation coefficient), and `AUC` (area under the curve). URC is an indicator designed for comprehensively measuring the model in terms of resilience and credibility:

$$\text{URC} = 2 \times \frac{R \times C}{R + C} \tag{21}$$

where R represents the true positive rate (TPR), and C represents the true negative rate (TNR). The value of URC is between 0 and 1, and the higher the URC, the better the performance of the detection model. According to the criteria given in Eq. (1), the plagiarism detection result is decided



(a) URC (b) F-measure (c) MCC

FIGURE 8. Comparative analysis on detection performance with respect to URC, F-Measure and MCC.

by the threshold ε. We set the effective value range of the threshold as 0-0.5, that is, $1-\varepsilon \geq \varepsilon$. Figure 8(a) shows the comparison between NeurMPD and other techniques under different thresholds. As the blue line shows, NeurMPD performs better than the other three methods. To use F-Measure (defined above) and MCC for measuring, plagiarism detection decision is made according to Eq. (17). MCC is an evaluation metric that can be used to make a reasonable assessment of test effectiveness in the case of unbalanced positive and negative samples, which is denoted as:

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)\,(TP+FN)\,(TN+FP)\,(TN+FN)}} \tag{22}$$

The comparative results on F-Measure and MCC for NeurMPD and other birthmark techniques under different thresholds are respectively displayed in Figure 8(b) and Figure 8(c), where NeurMPD outperforms TreSB, TOB^{SA}, TOB^{SS}, and SCSSB in most measurements.

With the help of AUC, we can perform the quantitative analysis of the technical performance of each model with respect to URC, F-Measure, and MCC. Table 4 summarizes the specific AUC values of different measure metrics for each detection technique. It can be observed that all three AUC values of NeurMPD are higher than those of other detection methods, which indicates that NeurMPD yields a better advantage of coping with thread interleaving, and thus achieves better performance for multi-threaded program plagiarism detection.

TABLE 4. Quantitative comparison of detection techniques on AUC.

Metric	SCSSB	TOB^{SA}	TOB^{SS}	TreSB	NeurMPD
URC	0.394	0.404	0.402	0.431	**0.474**
F-Measure	0.916	0.933	0.925	0.952	**0.971**
MCC	0.820	0.839	0.834	0.875	**0.948**

2) COMPARATIVE ANALYSIS ON TIME COST

The three dynamic birthmark based detections (i.e., TreSB, TOB, and SCSSB) mainly include three phases: execution trace capture, birthmark generation, and similarity calculation, while our proposed method NeurMPD is also implemented in three phases: execution trace extraction, model training using siamese neural networks (including supervised birthmark construction and similarity measurement), and detection through ensemble. Considering that the experiments are conducted on the same set of execution traces, in this section, we focus on comparing the time cost of NeurMPD with others in the last two phases with respect to Phase II and Phase III. Table 5 gives the average time cost of each plagiarism detection method in Phase II and Phase III.

TABLE 5. Average time cost (ms) of detection techniques in Phase II and III.

	SCSSB	TOB^{SA}	TOB^{SS}	TreSB	NeurMPD
Phase II	103	103	103	102	1.22×10^7
Phase III	0.1	0.1	20	0.02	0.15

The results illustrate that the average time of NeurMPD in Phase II is significantly higher than others. The reason behind this is that in Phase II, other methods use k-gram directly over execution traces to construct birthmarks without any ground truth training, while NeurMPD feeds a large number of program pairs to train the deep siamese neural networks to not only construct thread-aware birthmarks, but also leverage multilayer perceptron to learn similarity metric. Such extra time cost leads to an advantage that NeurMPD can automatically generalize to other unknown multi-threaded program plagiarism detection without any retraining, while others may suffer from weak generalizability and dependency on domain knowledge. Since NeurMPD is built upon deep learning framework, it takes a little more time (0.15 ms on average) for decision making through ensemble, which is still far less than TOB^{SS} using maximum weighted dichotomy matching. Phase III further proves that once the model is trained, NeurMPD can effectively and efficiently determine the plagiarism among the programs. It is worth remarking that (1) our training for NeurMPD is conducted on CPU and its Phase II time can be significantly reduced by resorting to GPUs or TPUs, while such optimization is very limited to other models; and (2) the offline training for NeurMPD

is a one-time effort, while other models have to repeatedly perform the whole process to deal with different datasets. In this respect, though it is more time-consuming for training, NeurMPD is still significant and promising for multi-threaded program plagiarism detection for its better detection effectiveness and better feasibility in practice.

V. RELATED WORK

This article focuses on the plagiarism detection of multi-threaded programs. The recent research works in this field mostly belong to the category of software birthmark. The software birthmark was first proposed by Tamada *et al.* [32] and Collberg *et al.* [28]. Since then, software plagiarism detection works have mainly fallen into two categories: static birthmark and dynamic birthmark. For static birthmark, DroidMoss [48] took hash value of bytecode fragments as birthmark. Ko *et al.* [20] used k-gram algorithm to slice the decompiled bytecode, and used the generated short sequence set as a software birthmark. ViewDroid [45] presented a functional view graph birthmark. Cop [25] is a static birthmark technique based on strict semantic analysis, but it is difficult to analyze large-scale software. Generally, static birthmarks cannot resist obfuscation.

For dynamic birthmark, Myles *et al.* [28] first proposed the concept of dynamic software birthmark and designed WPP (Whole Program Path) birthmark. After that, Wang *et al.* [41] designed SCSSB (System Call Short Sequence Birthmark) and IDSCSB (Input Dependent System Call Subsequence Birthmark). These traditional dynamic birthmarks cannot well address the uncertainty caused by multi-threaded programs. Tian *et al.* [37] introduced the concept of thread-aware birthmark for the first time. Accordingly, two dynamic birthmarking methods TreSB (thread-related system call birthmark) [35] and TOB (thread-oblivious birthmark) based on slicing-merging [36] were proposed to detect multi-threaded program plagiarism. For a systematic introduction of software birthmark techniques, please refer to the work by Tian *et al.* [34]. Differently, our proposed NeurMPD leverages supervised siamese neural networks for thread-aware birthmark construction and plagiarism detection without imposing any prior knowledge, and thus more semantics-preserving and task-specific to generalize to unknown data.

Recently, deep learning and representation learning techniques are starting to be leveraged for binary code similarity detection. Asm2Vec [11] generated vector representations for assembly sequences by designing and training a representation learning model that is improved upon the PV-DM [23] model, and compared the vectors with cosine similarity. Xu *et al.* [42] designed a deep neural network based graph embedding model for processing program CFGs to vectors, followed with a siamese architecture and cosine similarity metric to achieve similarity detection. INNEREYE [49] trained a LSTM based neural network model by feeding in basic-block level instructions to obtain embedding vectors for basic blocks, based on which program-level similarity can also be detected. Different from these methods which process statically disassembled assembly instructions or CFGs, our method operates on dynamically captured execution traces. Besides, the similarity metric is also learned through a multi-layer perceptron in our model rather than adopting manually defined measures as the other works do.

VI. CONCLUSION

Existing thread-aware birthmarking techniques may suffer from weak generalizability and dependency on domain knowledge. To address this issue, we build up deep siamese neural networks to supervise thread-aware birthmark construction, similarity measurement, and plagiarism decision making. More specifically, we transform multiple execution traces for each multi-threaded program under a specified input to the plain feature matrix, and successively pass it through CNN and BiLSTM to learn latent representation as thread-aware birthmark that enjoys better semantic richness and perturbation resistance; afterwards, we further use a multilayer perceptron to learn the similarity metric, which is then fed to sigmoid activation to obtain the output. Integrating our proposed method, a system called NeurMPD is developed to detect plagiarism of multi-threaded programs. The experimental results based on a public software plagiarism sample set demonstrate that NeurMPD achieves encouraging detection effectiveness and excellent resilience and credibility, which outperforms other alternative techniques.

REFERENCES

[1] *China's Courts Pass Controversial Rulings on Open-Source Licencing*. Accessed: Aug. 31, 2020. [Online]. Available: https://www.lexology.com/library/detail.aspx?g=597bfc93-0e53-4ffb-8311-a8fe3129d7f
[2] *Joltid Vs. Skype: Is There a Workaround*. Accessed: Aug. 31, 2020. [Online]. Available: https://www.zdnet.com/article/joltid-vs-skype-is-there-a-workaround/
[3] *Open Source Compliance Trend*. Accessed: Aug. 31, 2020. [Online]. Available: http://sourceauditor.com/blog/open-source-compliance-trend/.
[4] *Redcore*. Accessed: Aug. 31, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Redcore
[5] D. Andor, C. Alberti, D. Weiss, A. Severyn, A. Presta, K. Ganchev, S. Petrov, and M. Collins, "Globally normalized transition-based neural networks," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics (Long Papers)*, vol. 1, 2016, pp. 2442–2452.
[6] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a, 'siamese' time delay neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 1994, pp. 737–744.
[7] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *Proc. 36th Int. Conf. Softw. Eng. ICSE*, 2014, pp. 175–186.
[8] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-Play scale," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 659–674.
[9] L. Chen, S. Hou, and Y. Ye, "SecureDroid: Enhancing security of machine learning-based detection against adversarial Android malware attacks," in *Proc. 33rd Annu. Comput. Secur. Appl. Conf.*, Dec. 2017, pp. 362–372.
[10] S. Chopra, R. Hadsell, and Y. LeCun, "Learning a similarity metric discriminatively, with application to face verification," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2005, pp. 539–546.
[11] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2 Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 472–489.

[12] Y. Fan, Y. Zhang, S. Hou, L. Chen, Y. Ye, C. Shi, L. Zhao, and S. Xu, "iDev: Enhancing social coding security by cross-platform user identification between GitHub and stack overflow," in *Proc. 28th Int. Joint Conf. Artif. Intell.*, 2019, pp. 2272–2278.
[13] A. Graves, "Generating sequences with recurrent neural networks," 2013, *arXiv:1308.0850*. [Online]. Available: https://arxiv.org/abs/1308.0850
[14] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proc. 8th Work. Conf. Mining Softw. Repositories MSR*, 2011, pp. 63–72.
[15] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-R. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Nov. 2012.
[16] S.-J. Huang, J.-W. Zhao, and Z.-Y. Liu, "Cost-effective training of deep CNNs with active model adaptation," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2018, pp. 1580–1588.
[17] 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.*, vol. 41, no. 9, pp. 925–943, Sep. 2015.
[18] 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. 33rd Int. Conf. Softw. Eng. ICSE*, 2011, pp. 756–765.
[19] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," in *Proc. Int. Conf. Learn. Represent.*, 2015, pp. 1–15.
[20] J. Ko, H. Shim, D. Kim, Y.-S. Jeong, S.-J. Cho, M. Park, S. Han, and S. B. Kim, "Measuring similarity of Android applications via reversing and K-gram birthmarking," in *Proc. Res. Adapt. Convergent Syst. RACS*, 2013, pp. 336–341.
[21] G. Koch, R. Zemel, and R. Salakhutdinov, "Siamese neural networks for one-shot image recognition," in *Proc. ICML Deep Learn. Workshop*, vol. 2, 2015, pp. 1–8.
[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[23] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 1188–1196.
[24] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015.
[25] 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. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng. FSE*, 2014, pp. 389–400.
[26] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. ICLR (Workshop Poster)*, 2013, pp. 1–12.
[27] J. Ming, F. Zhang, D. Wu, P. Liu, and S. Zhu, "Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection," *IEEE Trans. Rel.*, vol. 65, no. 4, pp. 1647–1664, Dec. 2016.
[28] G. Myles and C. Collberg, "Detecting software theft via whole program path birthmarks," in *Proc. Int. Conf. Inf. Secur.* Berlin, Germany: Springer, 2004, pp. 404–415.
[29] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online learning of social representations," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining KDD*, 2014, pp. 701–710.
[30] V. G. Satorras and J. B. Estrach, "Few-shot learning with graph neural networks," in *Proc. Int. Conf. Learn. Represent.*, 2018, pp. 1–13.
[31] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. neural Inf. Process. Syst.*, 2014, pp. 3104–3112.
[32] H. Tamada, M. Nakamura, A. Monden, and K.-I. Matsumoto, "Design and evaluation of birthmarks for detecting theft of java programs," in *Proc. IASTED Conf. Softw. Eng.*, 2004, pp. 569–574.
[33] H. Tamada, "Java birthmarks–detecting the software Theft–," *IEICE Trans. Inf. Syst.*, vol. E88-D, no. 9, pp. 2148–2158, Sep. 2005.
[34] Z. Tian, T. Liu, Q.-H. Zheng, F. Tong, D. Wu, S. Zhu, and K. Chen, "Software plagiarism detection: A survey," *J. Cyber Secur.*, vol. 1, no. 3, pp. 52–76, 2016.
[35] 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.*, vol. 119, pp. 136–148, Sep. 2016.
[36] 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.*, vol. 44, no. 5, pp. 491–511, May 2018.
[37] 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. 22nd Int. Conf. Program Comprehension ICPC*, 2014, pp. 304–313.
[38] 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.*, vol. 41, no. 12, pp. 1217–1235, Dec. 2015.
[39] B. Vasilescu, V. Filkov, and A. Serebrenik, "StackOverflow and GitHub: Associations between software development and crowdsourced knowledge," in *Proc. Int. Conf. Social Comput.*, Sep. 2013, pp. 188–195.
[40] Q. Wang, Z. Tian, C. Gao, and L. Chen, "Plagiarism detection of multi-threaded programs using frequent behavioral pattern mining," in *Proc. 32nd Int. Conf. Softw. Eng. Knowl. Eng.*, 2020, p. 1.
[41] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Detecting software theft via system call based birthmarks," in *Proc. Annu. Comput. Secur. Appl. Conf.*, Dec. 2009, pp. 149–158.
[42] 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. Comput. Commun. Secur.*, Oct. 2017, pp. 363–376.
[43] Y. Ye, S. Hou, L. Chen, J. Lei, W. Wan, J. Wang, Q. Xiong, and F. Shao, "Out-of-sample node representation learning for heterogeneous graph in real-time Android malware detection," in *Proc. 28 Int. Joint Conf. Artif. Intell.*, Aug. 2019, pp. 4150–4156.
[44] Y. Ye, S. Hou, L. Chen, X. Li, L. Zhao, S. Xu, J. Wang, and Q. Xiong, "ICSD: An automatic system for insecure code snippet detection in stack overflow over heterogeneous information network," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 542–552.
[45] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection," in *Proc. ACM Conf. Secur. Pivacy Wreless Mbile Netw. WiSec*, 2014, pp. 25–36.
[46] F. Zhang, D. Wu, P. Liu, and S. Zhu, "Program logic based software plagiarism detection," in *Proc. IEEE 25th Int. Symp. Softw. Rel. Eng.*, Nov. 2014, pp. 66–77.
[47] T. Zhenzhou, W. Ningning, W. Qing, G. Cong, L. Ting, and Z. Qinghua, "Plagiarism detection of multi-threaded programs by mining behavioral motifs," *J. Comput. Res. Develop.*, vol. 57, no. 1, pp. 202–213, 2020.
[48] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party Android marketplaces," in *Proc. 2nd ACM Conf. Data Appl. Secur. Privacy CODASKY*, 2012, pp. 317–326.
[49] 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. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.

ZHENZHOU TIAN was born in Shandong, China, in 1987. He received the B.S. and Ph.D. degrees in computer science and technology from Xi'an Jiaotong University, China, in 2010 and 2016, respectively. He is currently a Lecturer with the School of Computer Science and Technology, Xi'an University of Posts and Telecommunications. His research interests include software and system security, program similarity analysis, and software behavior analysis.

QING WANG was born in Shaanxi, China, in 1995. She is currently pursuing the master's degree with the Xi'an University of Posts and Telecommunications. Her research interests include software plagiarism detection and deep learning-based program analysis.

CONG GAO was born in Shaanxi, China, in 1985. He received the B.S. and Ph.D. degrees in computer science and technology from Xidian University, China, in 2008 and 2015, respectively. He is currently a Lecturer with the School of Computer Science and Technology, Xi'an University of Posts and Telecommunications. His research interests include information security, artificial intelligence, and network computing services.

LINGWEI CHEN received the Ph.D. degree in computer science from West Virginia University, in 2019. He is currently a Postdoctoral Scholar with the College of Information Sciences and Technology, Pennsylvania State University. Prior to that, he was a Software Engineer with the Software Development Center, Agricultural Bank of China. He also has internship experience at Tencent and Yahoo! for research and development. His research interests include machine learning and cybersecurity.

DINGHAO WU (Member, IEEE) received the Ph.D. degree from Princeton University, in 2005. He is currently an Associate Professor with the College of Information Sciences and Technology, Pennsylvania State University. His research interests include software systems, including software security, software analysis and verification, software engineering, and programming languages. Prior to joining Penn State, he was a Research Engineer with the Microsoft with the Center for Software Excellence and the Windows Azure Division.