EAI Endorsed Transactions
on Security and Safety

Research Article EAI.EU

Binary Code Similarity Detection through LSTM and Siamese Neural Network

Zhengping Luo[1,*], Tao Hou[2], Xiangrong Zhou[3], Hui Zeng[3], Zhuo Lu[2]

[1]Department of Computer Science & Physics, Rider University, Lawrenceville, NJ 08648, USA.
[2]Computer Science Engineering and Electrical Engineering, University of South Florida, Tampa FL 33620, USA.
[3]Intelligent Automation Inc., Rockville MD 20855, USA.

Abstract

Given the fact that many software projects are closed-source, analyzing security-related vulnerabilities at the binary level is quintessential to protect computer systems from attacks of malware. Binary code similarity detection is a potential solution for detecting malware from the binaries generated by the processor. In this paper, we proposed a malware detection mechanism based on the binaries using machine learning techniques. Through utilizing the Recurrent Neural Network (RNN), more specifically Long Short-Term Memory (LSTM) network, we generate the uniformed feature embedding of each binary file and further take advantage of the Siamese Neural Network to compute the similarity measure of the extracted features. Therefore, the security risks of the software projects can be evaluated through the similarity measure of the corresponding binaries with existing trained malware. Our real-world experimental results demonstrate a convincing performance in distinguishing out the outliers, and achieved slightly better performance compared with existing state-of-the-art methods.

Received on 27 May 2021; accepted on 10 September 2021; published on 14 September 2021

Keywords: Malware detection, binary analysis, LSTM, Siamese Neural Network, similarity detection

doi:10.4108/eai.14-9-2021.170956

1. Introduction

Identification of bugs or other security-related vulnerabilities in software projects still remains to be a challenging problem in software security. Even though intensive work has been conducted by both the research and industry communities[1–3], the security of software projects still poses great concerns given that many software projects are used by millions or even billions of users. A small bug, such as the failure of checking buffer boundaries could lead to disastrous consequences [4, 5].

These kinds of vulnerabilities in software projects could be resulted from mistakes of trusted program developers, or from malicious attackers. Thanks to the intellectual property or security concerns, many of those software providers will not disclose their source code to the public. Therefore, from the users' perspective, analyzing the binary code when executing the software becomes their practical option to evaluate the security of software projects.

Binary code similarity detection could be applied to different scenarios, while malware detection is one of them, other applications include plagiarism detection[2] and vulnerability detection [6]. Extensive efforts [1, 7, 8] have been given to detect similar functions from binaries that generated from different platforms, such as x86, ARM and MIPS.

The general working flow of binary code similarity detection [7–9] focuses on extracting representative local feature vectors for each node in the control flow graphs (CFGs) that disassembled from binary code. Various statistical features and block features are proposed to represent the graph blocks [7, 9]. Then these local feature vectors are encoded into embeddings such that further similarity comparison techniques could be applied to perform the detection or comparison. One important type of the similarity comparison method is the graph-based matching

[*]Corresponding author. Email: zhengpingluo@mail.usf.edu

algorithms. For example, using bipartite matching algorithm to calculate the similarity of two CFGs [7].

However, graph matching-based approaches have two shortcomings according to [10]: (i) It is less flexible for the similarity functions approximated through graph matching techniques to adapt to other applications; (ii) The graph matching algorithms suffer from low efficiency, thus leads to the inefficiency of the similarity detection process. These two shortcomings make the graph matching-based method challenging to be applied in a broader perspective. There are also some deep neural network-based methods have been proposed to embed the local features [1, 6, 10, 11]. However, most of them involves training a deep graph neural network or sematic-aware neural network, which usually complicates the problem especially in terms of computation cost.

In binary code similarity detection, the first step is to extract local features and encode them into embeddings. Given the shortcomings of graph matching-based approaches and deep neural network-based methods, in this paper, we propose a new embedding method based on long short-term memory (LSTM) recurrent neural network (RNN), which is widely applied to sequence classification [12] and pattern-based feature selection/-classification [13].

The sequence of local block features can be fed into an LSTM recurrent neural network. Through multiple rounds of training, the final cell state of the LSTM recurrent neural network will learn much of the information from all the previous trained input and output pairs, which could be utilized as the final embeddings for the binaries and be further analyzed using classification techniques. As we found in our experimental results that LSTM network-based methods often could achieve similar performance with less training and computation cost.

For the similarity comparison, the traditional methods including Euclidean distance comparison and classification methods focus on the statistical characteristics of two inputs. However, the embedding we extracted from graph-based embedding methods or neural network-based methods often contain much information that traditional methods fall short of capturing. To overcome this weakness and fully capture the information, we employ Siamese neural network [14] to perform the similarity measurement. The main advantage of the Siamese neural network is it employs a unique structure to learn the embedding such that the semantic similarity could be learned thus placing the same classes of the input close together.

Our main contributions can be summarized as follows:

- We proposed an LSTM recurrent neural network-based model to embed local feature vectors of CFGs disassembled from binaries, which captures the general information of all the historical blocks and the dependence information, also makes the following similarity detection process more efficient.
- We employed Siamese neural network to perform the similarity measurement in binary code similarity detection, which learns the embedded semantic information and demonstrates multiple advantages compared with existing traditional similarity detection methods such as Euclidean distance.
- Comprehensive experiments are conducted based on real-world collected binary dataset and the experimental results validated our mechanism has a slightly better performance with existing methods such as graph matching-based methods in terms of detection accuracy and less computation cost compared with existing deep neural network-based methods.

The related work are stated in Section 2. In Section 3, we elaborate the details of our designed framework of binary code similarity detection based on LSTM recurrent neural networks and Siamese neural networks. Experimental results and analysis are given in Section 4 and Section 5 concludes the paper.

2. Background and Related Work

In this section, we give the background information of the LSTM networks, Siamese neural networks and the related works of binary code similarity detection.

2.1. LSTM Recurrent Neural Networks

One of the main advantages of recurrent neural networks is they have loops within their structure that allowing information to persist. However, ordinary RNN models have vanishing gradients and exploding gradients problems[15], LSTM offers a decent solution to this two problems through using constant error carousels, which could protect and control the cell state. As a special kind of RNN, LSTM has the capability of learning long-term dependencies. The basic form of LSTM consists of a chain of repeated neural networks. The structure of the repeated module is shown as in Fig. 1. In the module, x_t is the input at time t and h_t denotes the corresponding output value. C_{t-1} represents the cell state, which runs straight down the entire LSTM model.

At the first part of the LSTM modules, it aims to decide what kind of information in the cell state from previous time will be deleted through a Sigmoid layer, mathematically it is shown as:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right), \qquad (1)$$

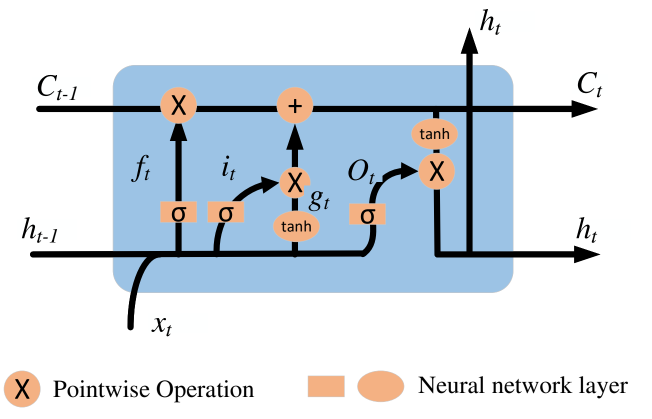

Figure 1. The repeated module in a standard LSTM networks

in which W_f, b_f is the parameters of the corresponding neural network layer (consists of a bunch of Sigmoid functions) and h_{t-1} is the previous output.

The following part of LSTM module is to decide what kind of new information we need to add to the cell state. The operations within the repeated model could be shown mathematically as:

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i), \\ g_t &= \tanh(W_g \cdot [h_{t-1}, x_t] + b_g). \end{aligned} \tag{2}$$

Similarly, W_i, W_g, b_i, b_g are the corresponding neural network layer parameters.

After the two operations on the cell state, the new cell state is updated through a pointwise operation from f_t, i_t and g_t through:

$$C_t = f_t * C_{t-1} + i_t * g_t, \tag{3}$$

which includes both the added information from current time and also compromised some historical information. The output of the LSTM is denoted as:

$$h_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) * \tanh(C_t), \tag{4}$$

in which $\tanh(C_t)$ is used to push the cell state values to be between -1 and 1.

LSTM networks have been widely applied to various real-world applications and achieved state-of-the-art performances such as speech recognition [16], handwriting recognition [17]. In [18], LSTM networks are used to process localized features and perform the classification to detect stealthy malware. In this paper, we employed LSTM to extract general information from local features of each block in disassembled CFGs of binaries and further use the information for similarity detection.

2.2. Siamese Neural Networks

Siamese neural networks were first introduced by Bromley and LeCun in the early 1990s, in which Siamese neural networks are designed for verification of signatures written on a pen-input tablet [19]. It has been successfully applied in image recognition [14], gait recognition [20].

Siamese neural networks consist of a class of neural network architectures, in which they have two or more identical sub-networks, each of them has the same configuration in terms of the parameters and weights. In the training process, the parameter is updated concurrently across all sub-networks. Their values will be mirrored thus all the parameter values across different sub-networks will still be the same. This kind of architecture can be used to compare the similarity of different inputs.

Siamese neural networks have some impressing advantages compared to tradition neural networks [21, 22]. For example, Siamese neural networks are more robust to class imbalance as new classes of data can be added to the network without training the whole model again. Siamese networks focus on learning embedding (in the deeper layer), further the same classes/categories of the inputs are placed close together, which we denote it as semantic similarity. Another important advantage is that Siamese networks can be easily trained using standard machine learning techniques on pairs sampled from source data and provide a competitive approach that does not rely upon domain-specific knowledge.

The training process of Siamese neural networks involves training a pairwise model, thus traditional entropy loss cannot be used in this case. Triplet loss is a usual loss function used in Siamese neural networks, in which a baseline (anchor) input is compared to a positive (truthy) input and a negative (false) input. The optimization objective of the training is to minimize the distance between the baseline input and the positive input while maximizing the distance between the baseline input and the negative input, as shown in Fig.2.

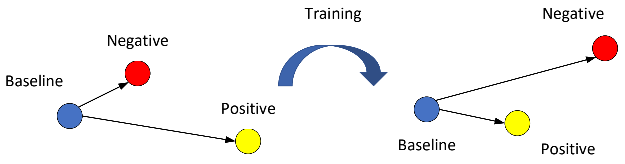

Figure 2. The Triplet loss minimizes the distance between the baseline and positive inputs while maximizing the distance between the baseline and the negative inputs.

Mathematically the triplet loss can be formulated as:

$$L(a, p, n) = \frac{1}{2}\{max(0, m + D^2(a, p) - D^2(a, n))\}, \tag{5}$$

in which $D(u, v) = |u - v|_2$, and m is the desirable distance for dissimilar pair (p,n).

Another typical type of loss function is the contrastive loss. The logic behind contrastive loss is similar to the triplet loss. Contrastive loss is used to learn embedding in which two similar points have a smaller Euclidean distance while two dissimilar points have a large Euclidean distance. Given an input training pair (x_1, x_2), we have the label: $y = 0$ if $(x1, x2)$ is similar and 1 if (x1,x2) is dissimilar pair. The optimization objective in terms of contrastive loss can be written as:

$$min\ L(x_1, x_2) = \frac{1}{2}(1-y)D^2 + \frac{1}{2}y\ max\{0, m-D\}^2, \quad (6)$$

in which D is the Euclidean distance of the outputs from the two networks given inputs x_1, x_2.

Given the advantages of Siamese Neural Networks, the embeddings we obtained through LSTM neural networks could be fed into the Siamese Neural Network to perform similarity measurement such that the embedding information could be fully considered.

2.3. Binary Code Similarity Detection

Binary code similarity detection is quite challenging while also rewarding. It is challenging because the binaries generated from software projects could vary enormously due to the diversity of processors, compilers, optimization level options, and platforms. It is rewarding because we could evaluate the security of the software without the source code, which often is unavailable.

Many of existing works regarding to binary code similarity detection follows a similar logic [7, 10]. It first recovers CFGs from the binaries and then based on the statistical and structural information of the CFGs to perform graph matching. To achieve scalability and high accuracy concurrently, graph-embedding based methods such as [9] focused on learning an indexable feature representation from the CFGs. Based on the embeddings of test binaries and the embeddings of known bugs or benchmarks, a similarity score could be measured, further paves the way for evaluating the security vulnerabilities.

A basic block distance-based method, named as discovRE, is proposed in [7]. The general idea behind discovRE is to calculate similarity between functions based on CFGs disassembled from binaries. However, this process is computationally expensive if not being processed appropriately. To minimize the computation cost, discovRE employs an efficient pre-filter to identify a small set of candidate functions and then use them to search for similar functions in the binaries that need to be evaluated.

Another type of methods is the Neural network-based methods [9–11], which focuses on using neural networks to embedd CFGs. The main advantage of this type of methods is they could alleviate the limitations of graph matching-based approaches, such as high computation cost. Further the embeddings can be used for malware detection or classification.

Grieco et al [6]. proposed a vulnerability discovery tool, named VDiscover. They combined static and dynamic analysis with machine learning techniques to predict the vulnerabilities among binaries. The basic idea of the approach is that it applies both static and dynamic analysis to extract features from a large-scale of binary programs. These features are further studied and used to predict vulnerabilities by machine learning techniques.

In this paper, we propose a new embedding method based on LSTM networks, which has the advantage of containing the general information of all the previous inputs and the dependency information. Thus, the feature vectors of each block within the CFGs could be processed and embedded into an indexable representation. Our experimental results show that this method is computationally efficient compared with the existing graph neural network-based methods[10].

In addition, our method of using Siamese Neural Network to measure the similarity based on the cell state of the LSTM network is different from existing studies [14, 19, 20]. To the best of our knowledge, we are the first to use the cell state of LSTM in CFG embedding and use Siamese Neural Network to measure the similarity of such embedding, which has the advantage of measuring the semantic similarity of the overall local feature information of CFGs.

3. Binary Code Similarity Detection based on LSTM and Siamese Neural Network

We elaborate the motivation and the proposed binary code similarity detection framework using LSTM and Siamese neural network in this section.

3.1. Motivation

Existing methods of performing binary code similarity detection mainly focused on two perspective: graph matching-based methods and deep neural network-based methods. Although graph matching-based methods are widely discussed in binary code similarity detection[7, 9]. The shortcomings are also obvious [10]: (i) It is less flexible for the similarity functions approximated through graph matching techniques to adapt to other applications; (ii) The graph matching algorithms suffer from low efficiency, thus leads to the inefficiency of the similarity detection process.

Another perspective is the deep neural network-based methods [1, 6, 10, 11], which often involves training a deep graph neural network or semantic-aware neural network. This could complicate the problem especially in terms of computation cost when the input dimensions increase.

In the similarity measurement stage, the traditional methods including Euclidean distance comparison and classification methods focus on the statistical characteristics of two inputs. However, the embedding we extracted from graph-based embedding methods and neural network-based methods contain much information that traditional methods fall short of capturing. The embedding information of the binaries requires a more effective method to perform the similarity comparison.

Based on the above points identified, we propose an LSTM network-based embedding methods in this paper to improve the information representation in the embedding process, and we employ Siamese Neural Networks to compare the similarity between different binaries. This strategy does not require graphs, which avoids the computation cost concerns from graph computation.

The detailed architecture design using LSTM networks to perform embedding and Siamese neural networks to conduct similarity comparison is described in the following subsection.

3.2. Our Proposed Framework

Our proposed method includes three steps: local block feature extraction, LSTM network-based embedding and Siamese neural network-based similarity detection.

Local block feature extraction. To perform binary code similarity detection, we need to first extract corresponding characters from binary files and transform these characters into a matrix or vector representation. In this work, we use angr [23], a multi-architecture binary analysis toolkit, to disassemble the binaries and get the desired CFGs. Further we could perform embedding based on the CFGs.

angr is capable of analyzing binaries in both static and dynamic disassembling styles. Therefore there are two types of CFGs that can be disassembled: static CFGs (CFGFast) and dynamic CFGs (CFGEmulated). CFGFast uses static analysis to generate the CFGs of binaries. It is faster but might lose a small number of control-flow transitions which can only be resolved at execution time. CFGEmulated generates CFGs through symbolic execution, which is usually slower, especially for large binaries. In our experiment, we found that usually CFGFast achieves better performance in terms of computation cost.

The disassembled files recovered through the CFGFast operation of an example binary is shown as in the following, we disassembled the binary file in a depth-first order:

```
<CFGNode _init [11]>
0x80489c4: push ebp
0x80489c5: mov ebp, esp
0x80489c7: sub esp, 8
0x80489ca: call 0x8048c90
offsprings: 1
betweenness: 2.0495163141498604e-05
<CFGNode call_gmon_start [27]>
0x8048c90: push ebp
0x8048c91: mov ebp, esp
0x8048c93: push ebx
0x8048c94: push eax
0x8048c95: call 0x8048c9a
0x8048c9a: pop ebx
0x8048c9b: add ebx, 0x404a
0x8048ca1: mov eax, dword ptr [ebx + 0xac]
0x8048ca7: test eax, eax
0x8048ca9: je 0x8048cad
offsprings: 2
betweenness: 3.415860523583101e-05
```

From the above disassembled information, we can observe the detailed graph node, or block information including the operations, offspring and betweenness information, which can be used to describe the characteristics of the binaries. As the results show, we obtain the disassembled information of each CFG basic block (in a depth-first order), including the opcode and the statistic results of offspring number and betweenness count, which combined can be used to comprehensively describe the characteristics of the binaries.

Given a binary function, the disassembled blocks contain multiple attributes that could be utilized to characterize the corresponding binary function. The basic block attributes can be divided into two types according to their level [7, 10]: block-level attributes and inter-block level attributes. Block-level attributes includes No. of String Constants, No. of Numeric Constants, No. of Calls, No. of Transfer Instructions, No. of Data Transfer Instructions, No. of Logic Instructions, No. of Arithmetic Instructions, etc. Inter-block level attributes includes No. of offsprings, betweenness, etc.

Here we employ an intuitive statistical property, i.e., the total number of times a type of operation occurred, together with two inter-block attributes, to describe the block.

Given above statistical and structural attributes of the block, we form them into a 9-dimension feature vector for each block (a.k.a., graph node). In the following part we'll explain how to transform the local features of all blocks from a binary file into an indexable embedding.

LSTM network-based embedding. There are dozens, if not hundreds or thousands, of blocks could be disassembled given a binary file. Correspondingly we could obtain the same number of localized feature vectors. However, how to learn an indexable embedding that has the capability of reflecting the overall characteristics of the binary based on all these local features?

Existing methods include graph-embedding methods [7, 9] focuses on transform the CFGs into a graph, and then using graph matching-based method such as bipartite matching algorithm to calculate the similarity of two CFGs. Deep graph neural network-based methods [9–11] perform the embedding through deep neural networks. However, LSTM networks offers a practical mechanism to potentially store the related information of all historical inputs. Given a sequence of inputs, the input x_t at each time slot t will update the cell state C_t based on the learned parameters of the neural network layer.

The update process at each time slot t includes two steps: the first step is to decrease or delete some unrelated information from the previous cell state through a "forget gate layer". Mathematically it is denoted as:

$$C_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) * C_{t-1} \tag{7}$$

In our local feature embedding scenario, this operation could be utilized to compromise the influence of some extreme blocks, and also, we have the capability of getting rid of redundant information through the training of the neural network layers.

The second step of the update process is to add information from each input to the cell state C_t, also known as the "input gate layer", which is shown as:

$$C_t = C_t + \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) * \tanh(W_g \cdot [h_{t-1}, x_t] + b_g). \tag{8}$$

After the two steps stated above, the information of interest with regard to the binary embedded in the features extracted at each local block will be able to get filtered and reflected by the cell state if trained appropriately. Therefore, after training the parameters within the LSTM network with the training dataset, a final cell state will be learned on each type of binaries. This cell state is learned over the entire training blocks. We use this cell state as the final indexable feature embedding of the corresponding binary file.

Siamese Neural Network-based similarity detection. Given the obtained indexable feature embeddings of binaries, we could apply various traditional methods such as Euclidean distance, neural networks to do similarity measure or classification. However, most of these traditional methods focus on the numeric features the embeddings presented while falling short of denoting the semantic information.

Provided with the multiple advantages of Siamese neural network [14] including: **a)** More robust to class imbalance; **b)** learning from semantic similarity of the embeddings; **c)** easily to be trained using standard optimization techniques. We design a Siamese neural network-based similarity detection mechanism to compare the embeddings from the test binaries with embeddings of existing known malwares or other trained binaries.

Our designed Siamese neural network-based binary analysis framework is shown in Fig.3. The architecture includes a sequence of convolutional layers, max pool layers, fully connected layers. Each of the neural network layer uses a single channel with filters of varying size and the stride is fixed to 1. We employ rectified linear Unit (ReLu) activation functions in each layer. The max pooling operation is assigned with a filter size of stride 2.

We have two inputs for the model, each of which is an embedding we learned through LSTM networks. We aim to minimize the distance of the outputs of each sub-model in the training process if the two inputs are from the same category, while maximizing the distance of the outputs of each sub-model if the two inputs are from different categories.

The units in the last convolutional layer are flattened into a single vector. A fully connected layer is followed after this convolutional layer. Then the induced distance is computed based on the output (h_1, h_2) of the previous fully connected layer, further it is given to a single sigmoid activation function layer for prediction. Mathematically, the prediction vector is shown as:

$$\mathbf{P}(x_1, x_2) = \sigma(\sum_j \alpha_j |h_1^{(j)} - h_2^{(j)}|), \tag{9}$$

in which x_1, x_2 are the two input embeddings. σ denotes the sigmoidal activation function. α_j are parameters learned by the model during the training process, which is used to weight the importance of the component-wise distance. The final layer induces a measure on the learned feature space of the previous hidden layer and scores the similarity between the two feature vectors.

Based on this architecture, we can first train the model on our known training dataset, then when new test binaries come in, we only need to compare the test input with the reference input (which comes from the trained dataset). Another big advantage of this architecture is that when we have new samples, we only require a very small number of samples to be stored in the dataset, using it as a reference, we can calculate the similarity for any new test instance with this type of samples [14].

In the learning process, we use M to denote the minibatch size and i is used to index the i_{th} minibatch. Let $\mathbf{y}(x_1^{(i)}, x_2^{(i)})$ be a vector of length M that contains the labels for the minibatch. We set $y(x_1^{(i)}, x_2^{(i)}) = 1$ whenever x_1 and x_2 are from the same category and $y(x_1^{(i)}, x_2^{(i)}) = 0$ if they are from different categories. Then the regularized cross-entropy objective on the binary

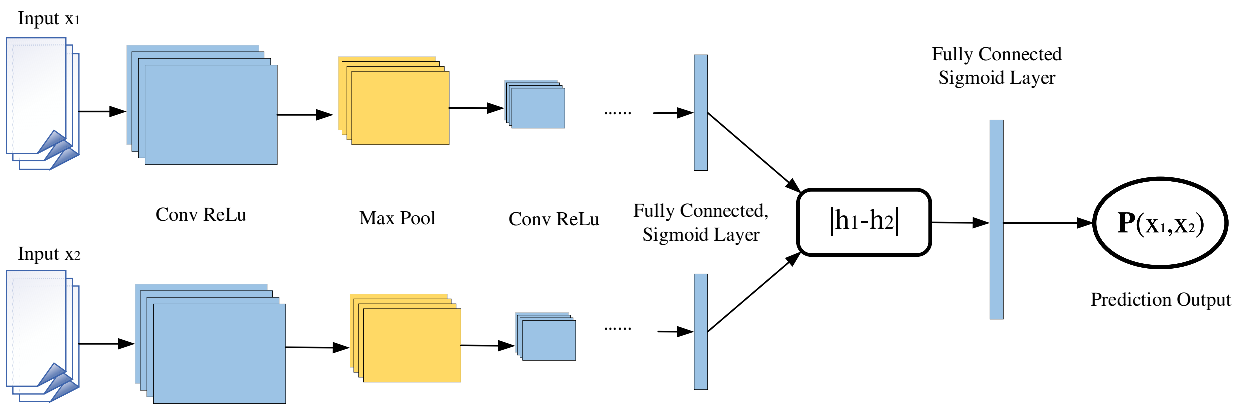

Figure 3. The architecture of Siamese Neural network-based binary code similarity detection.

classifier scenario takes the following form:

$$\begin{aligned}\mathcal{L}(x_1^{(i)}, x_2^{(i)}) = \ & \mathbf{y}(x_1^{(i)}, x_2^{(i)}) \ log \ \mathbf{P}(x_1^{(i)}, x_2^{(i)}) + \\ & (1 - \mathbf{y}(x_1^{(i)}, x_2^{(i)})) \ log \ (1 - \mathbf{P}(x_1^{(i)}, x_2^{(i)}) \ ) + \\ & \lambda^T |\mathbf{w}|^2,\end{aligned} \tag{10}$$

in which $\lambda^T |\mathbf{w}|^2$ is the regularization term.

In terms of the update policy, we employ the standard backpropagation algorithm to optimize the parameter values. Let η_j denotes the learning rate and μ_j the momentum. The update rule at epoch T can be formularized as:

$$\begin{aligned}\mathbf{W}_{kj}^{(T)}(x_1^{(i)}, x_1^{(i)}) &= \mathbf{W}_{kj}^{(T)} + \Delta\mathbf{W}_{kj}^{(T)}(x_1^{(i)}, x_1^{(i)}) + 2\lambda_j |W_{kj}|, \\ \Delta\mathbf{W}_{kj}^{(T)}(x_1^{(i)}, x_1^{(i)}) &= -\eta_j \nabla w_{kj}^{(T)} + \mu_j \Delta\mathbf{w}_{kj}^{(T-1)}.\end{aligned} \tag{11}$$

In which $\nabla w_{kj}^{(T)}$ denotes the partial derivative in terms of the weight between the j_{th} neuron in one layer and k_{th} neuron in the following layer.

In our following experiments, all the values of parameters within the convolutional neural network is assigned with a normal distribution of zero-mean and a standard deviation of 0.01. Bias is initialized with also a normal distribution of mean 0.5 and standard deviation 0.01. The initialization configuration is similar with those in [14] due to their similarity in terms of objective.

4. Experimental Results and Analysis

To validate the performance of our proposed framework, which using LSTM networks to embed the general features of binary files based on the local features of disassembled CFGs, and employing the Siamese neural network to perform the similarity detection, we experimented the framework on a dataset we collected by ourselves from real-world systems. The dataset includes 4000+ ELF malware executables. It contains 560 categories of binaries, and each category contains multiple binary files.

In the experiments, we implemented the framework based on TensorFlow v2.4.1 and Keras 2.4.3. The experiments are run on a Ubuntu 18.04 version. The framework is implemented using Python 3.6.9.

Local feature vector extraction. We first perform the disassembling operations on the binaries to obtain the corresponding CFGs. Then based on the graphs, we could extract the local block feature vectors, in our experiments, we build the local feature vectors as the following form:

[No. of string constants,
No. of numeric constants,
No. of arithmetic instructions,
No. of logic instructions,
No. of transfer instructions,
No. of calls,
No. of data transfer instructions,
No. of offspring,
the value of betweenness]ᵀ

This is intuitive and easy to obtain from the disassembled CFGs, and also is widely used in existing methods[7, 9].

Based on the assigned rule, we could easily obtain the numerical forms of the local feature vectors for each block in a binary file. An example numerical features of a sequence of local blocks is shown as follows:

```
             ......
4 0 0 0 1 1 1 2.0495163141498604e-05
1 10 1 0 0 1 2 2 3.415860523583101e-05
1 3 0 0 0 0 1 1 6.968355468109526e-05
0 1 0 0 0 1 0 1 0.00012570366726785811
1 6 0 0 0 0 2 2 6.0119145215062575e-05
```

```
0 2 0 0 0 0 0 1 4.9188391539596654e-05
0 1 0 0 0 1 0 1 0.00018172377985462097
1 8 0 0 0 0 3 2 6.968355468109526e-05
0 4 0 0 0 0 0 1 2.4594195769798327e-05
0 2 0 0 0 0 0 1 0.00011750560201125867
                ......
```

The local feature vectors extracted above represent both the block-level attributes information and inter-block level attributes information. As we can observe from local feature vectors, they are changing at each block. To obtain an indexable representation of the binary that contains multiple blocks, we feed this sequence of local feature vectors into the LSTM networks.

LSTM network-based embedding. To embed the obtained local feature vectors, we first need to truncate the blocks we have as we want to train each of the binary files on the same number of local features. However, the real-world binaries we collected are vary each other in terms of the length. We employ a strategy of choosing a fixed number of the blocks from each type of the binary file in a random way but without disrupting the order of the chosen sequence.

We form the embedding problem as a regression problem, in which the input is the feature vector at time t, while the output is the feature vector at time $t + 1$. However, to make the prediction process could take more of the past time slots into consideration, we employ multiple previous time slots as the input to predict the following feature vector of the block. We found that if too many previous time slots are considered, then the prediction output will be less reflective regarding the variation and the number of training samples will decrease dramatically. There is a tradeoff balance between the flexibility and accuracy.

In the experiment, we employ 5 previous feature vectors as the input to train the model, and we set the dropout value as 0.1, and the training epoch is set as 100. At each prediction. The LSTM blocks or neurons we adopt is set as 16. In Fig. 4, we shown the relationship of the averaged Mean Squared Error (MSE) with the training epochs of three representative binaries, from which we can observe that when the training epochs approaching 70, the averaged MSEs for all three binaries are become stable. Thus, we could output the hidden cell state vectors as the embeddings for those binary files.

Siamese neural network-based similarity detection. After we obtained the embeddings of the binaries, we have the indexable representation of each binary file. Thus, the remaining work is to feed them into a Siamese neural network and perform the similarity detection.

In this group of experiments, we focused on classifying a test binary to the pre-trained binary categories. We chose 10 categories of the binaries that have the largest number of binaries from our collected dataset as the training categories. As some categories include only a very small number of binaries, thus we exclude them in our experiment such that we have enough binaries under each category that could be divided into a training set and a testing set. For the testing binaries, we'll include both the binaries from the training categories and from those categories that are not included in the training process except specially specified.

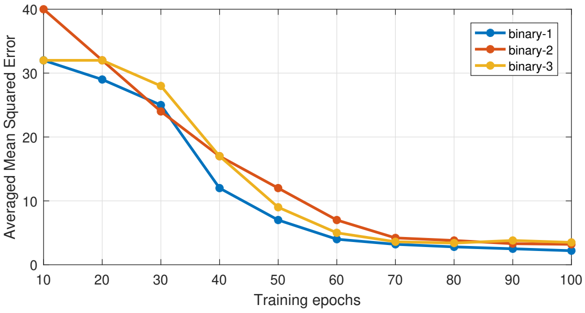

Figure 4. The relationship of the training epochs with the averaged MSE.

Since in a Siamese neural network, the weights from both sub-networks are supposed to be identical with each other, thus we use only one model and feed two inputs in succession. The optimization process or the backpropagation is conducted after we calculate the loss for two inputs. We build the Siamese neural network includes 3 fully connected convolutional layers with a structure of 16 × 32 × 16, and the dropout value of 0.1.

We first show the experimental results of the training performance using only test data from the categories that we have trained on. We define a measure named as detection accuracy to describe the performance, which is defined mathematically as:

$$\text{Detection Accuracy} = \frac{\text{\# of test binaries rightly classified}}{\text{\# of total test binaries}} \qquad (12)$$

We use half of the binaries within each category as the training data and the other half as the testing data, the relationship of the detection accuracy with the training epochs is shown in Fig.5. From the results we know that after training the Siamese neural network with at least 60 epochs, we can achieve the detection accuracy of around 90%, which validates the performance of the framework.

We also compared our proposed framework with existing methods as shown in [7, 10], which are representative methods to detect malware. [10] focuses on a graph embedding network to convert the graph into embeddings for binary functions, while [7] focuses on the maximum common subgraph isomorphism to

Table 1. The comparison of our proposed LSTM+Siamese neural network-based framework with existing works [7, 10].

	Our proposed method	Xu et al.[10]	Eschweiler et al.[7]
Detection Accuracy	0.85	0.85	0.83
Relative time cost for training	1.00	1.24	0.78
Relative time cost for testing	1.00	1.15	0.67

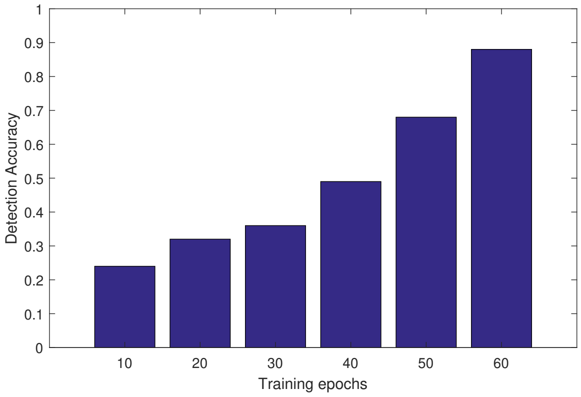

Figure 5. The relationship of detection accuracy with training epochs

measure the structural similarity between two different binaries.

We compared the performance and relative time cost of the three methods and the results are shown in the Table 1. Both the time cost for training and testing are compared with our proposed method in a relative way, which means our proposed method is set as a benchmark with the value of 1. From the comparison results we know that our method shows a higher training efficiency while still maintaining similar detection accuracy performance with method [10]. It is worth noting that here our detection accuracy is 85%, as the testing data also includes binaries from other categories of the dataset, which is differ from the previous 90% of the detection accuracy, in which the testing data is from the same categories of the training data.

Compared with [7], our method showed a slightly better performance though suffer from the training of two neural networks which cost a little bit more time. Compared with our proposed LSTM + Siamese neural network combination, the methods in [10] require to train a deep graph embedding neural network, while the methods in [7] require a graph similarity algorithm based on the maximum common subgraph isomorphism.

5. Conclusion

Binary code similarity detection plays an important role in evaluating the security of a software project that are closed-source. It also offers many other applications such as plagiarism and malware detection.

In this paper, we proposed an LSTM neural network-based method to obtain an indexable embedding from the dissembled control flow graphs of binary files. Also, we employed Siamese neural network to conduct the similarity comparison of two embeddings due to that Siamese neural network has the capability of learn the semantic information embedded in the inputs. Our experimental results show a promising performance compared with existing methods both in terms of detection accuracy and computation cost.

In our future work, we will focus on extracting more representative local features from the disassembled blocks of the binaries. Also, how to combine the LSTM network and Siamese Neural Network to make them work closely to reduce the training complexity will be one of another future work.

Acknowledgement. This material is based upon work supported by the U.S. Department of Energy, Office of Science under Award Number DE-SC0018476. Zhengping Luo was with University of South Florida when participating in this work.

References

[1] Liu, B., Huo, W., Zhang, C., Li, W., Li, F., Piao, A. and Zou, W. (2018) αdiff: cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*: 667–678.

[2] Luo, L., Ming, J., Wu, D., Liu, P. and Zhu, S. (2017) Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering* **43**(12): 1157–1177.

[3] Yu, Z., Cao, R., Tang, Q., Nie, S., Huang, J. and Wu, S. (2020) Order matters: semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, **34**: 1145–1152.

[4] Jaramillo, L. (2018) Malware detection and mitigation techniques: lessons learned from mirai ddos attack. *Journal of Information Systems Engineering & Management* **3**(3): 19.

[5] Chen, L., Ye, Y. and Bourlai, T. (2017) Adversarial machine learning in malware detection: Arms race between evasion attack and defense. In *2017 European Intelligence and Security Informatics Conference (EISIC)* (IEEE): 99–106.

[6] Grieco, G., Grinblat, G.L., Uzal, L., Rawat, S., Feist, J. and Mounier, L. (2016) Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*: 85–96.

[7] Eschweiler, S., Yakdan, K. and Gerhards-Padilla, E. (2016) discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, **52**: 58–79.

[8] Pewny, J., Garmany, B., Gawlik, R., Rossow, C. and Holz, T. (2015) Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy* (IEEE): 709–724.

[9] Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B. and Yin, H. (2016) Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*: 480–491.

[10] Xu, X., Liu, C., Feng, Q., Yin, H., Song, L. and Song, D. (2017) Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*: 363–376.

[11] Shin, E.C.R., Song, D. and Moazzezi, R. (2015) Recognizing functions in binaries with neural networks. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*: 611–626.

[12] Graves, A., Fernández, S., Gomez, F. and Schmidhuber, J. (2006) Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*: 369–376.

[13] Lafferty, J., McCallum, A. and Pereira, F.C. (2001) Conditional random fields: Probabilistic models for segmenting and labeling sequence data .

[14] Koch, G., Zemel, R. and Salakhutdinov, R. (2015) Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop* (Lille), **2**.

[15] Staudemeyer, R.C. and Morris, E.R. (2019) Understanding lstm–a tutorial into long short-term memory recurrent neural networks. *arXiv preprint arXiv:1909.09586* .

[16] Graves, A. (2013) Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* .

[17] Graves, A., Liwicki, M., Fernández, S., Bertolami, R., Bunke, H. and Schmidhuber, J. (2008) A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence* **31**(5): 855–868.

[18] Shukla, S., Kolhe, G., PD, S.M. and Rafatirad, S. (2019) Stealthy malware detection using rnn-based automated localized feature extraction and classifier. In *2019 IEEE 31st international conference on tools with artificial intelligence (ICTAI)* (IEEE): 590–597.

[19] Bromley, J., Guyon, I., LeCun, Y., Säckinger, E. and Shah, R. (1994) Signature verification using a" siamese" time delay neural network. *Advances in neural information processing systems* : 737–737.

[20] Zhang, C., Liu, W., Ma, H. and Fu, H. (2016) Siamese neural network based gait recognition for human identification. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (IEEE): 2832–2836.

[21] Chicco, D. (2021) Siamese neural networks: An overview. *Artificial Neural Networks* : 73–94.

[22] Berlemont, S., Lefebvre, G., Duffner, S. and Garcia, C. (2018) Class-balanced siamese neural networks. *Neurocomputing* **273**: 47–56.

[23] https://angr.io/.