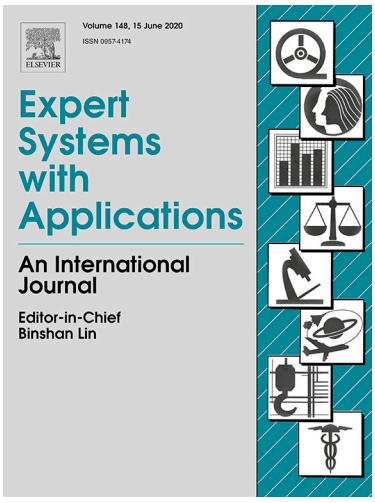
Journal Pre-proofs



BinDeep: A Deep Learning Approach to Binary Code Similarity Detection

Donghai Tian, Xiaoqi Jia, Rui Ma, Shuke Liu, Wenjing Liu, Changzhen Hu

PII: S0957-4174(20)31033-2
DOI: https://doi.org/10.1016/j.eswa.2020.114348
Reference: ESWA 114348

To appear in: *Expert Systems with Applications*

Received Date: 6 December 2019
Accepted Date: 17 November 2020

Please cite this article as: Tian, D., Jia, X., Ma, R., Liu, S., Liu, W., Hu, C., BinDeep: A Deep Learning Approach to Binary Code Similarity Detection, *Expert Systems with Applications* (2020), doi: https://doi.org/10.1016/j.eswa.2020.114348

BinDeep: A Deep Learning Approach to Binary Code Similarity Detection

Donghai Tian[a,b], Xiaoqi Jia[c], Rui Ma[a,*], Shuke Liu[a], Wenjing Liu[a], Changzhen Hu[a]

[a] *Beijing Key Laboratory of Software Security Engineering Technique, Beijing Institute of Technology, Beijing 100081, China*
[b] *Shanxi Military and Civilian Integration Software Engineering Technology Research Center, North Univeity of China, Taiyuan 030051, China*
[c] *Key Laboratory of Network Assessment Technology, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China*

Abstract

Binary code similarity detection (BCSD) plays an important role in malware analysis and vulnerability discovery. Existing methods mainly rely on the expert's knowledge for the BCSD, which may not be reliable in some cases. More importantly, the detection accuracy (or performance) of these methods are not so satisfied. To address these issues, we propose BinDeep, a deep learning approach for binary code similarity detection. This method firstly extracts the instruction sequence from the binary function and then uses the instruction embedding model to vectorize the instruction features. Next, BinDeep applies a Recurrent Neural Network (RNN) deep learning model to identify the specific types of two functions for later comparison. According to the type information, BinDeep selects the corresponding deep learning model for similarity comparison. Specifically, BinDeep uses the Siamese neural networks, which combine the LSTM and CNN to measure the similarities of two target functions. Different from the traditional deep learning model, our hybrid model takes advantage of the CNN spatial structure learning and the LSTM sequence learning. The evaluation shows that our approach can achieve good BCSD between cross-architecture, cross-compiler,

*Corresponding author

Email addresses: donghaitad@gmail.com (Donghai Tian), jiaxiaoqi@iie.ac.cn (Xiaoqi Jia), mary@bit.edu.cn (Rui Ma), whesen@qq.com (Shuke Liu), lw_jane@163.com (Wenjing Liu), chzhoo@bit.edu.cn (Changzhen Hu)

cross-optimization, and cross-version binary code.

Keywords: binary code, deep learning, similarity comparison, siamese neural network, LSTM, CNN

1. Introduction

With the rapid development of information technology, a large number of software are widely used in personal computers and IOT devices with different CPU architectures. As a result, more and more people are paying attention to software security. Malicious developers may insert malicious code into their software. For instance, some venders exploit the malware technique for commercial software promotion. Even if the software is developed by a trusted developer, it may contain one or more security vulnerabilities due to a programming mistake. Considering the intellectual property protection, most of software developers will not provide the source code of their product for security analysis.

To address these issues, a number of binary code analysis methods have been proposed in recent years. In this paper, we focus on the method for binary code similarity detection (BCSD), which is useful for malware analysis and vulnerability discovery. By conducting the similarity comparison to the known binary functions, we can identify the corresponding vulnerabilities (or malicious functions) in the different binary code.

The same source code can be compiled with different compilers and different optimization levels, targeting at different CPU architectures. As a consequence, the target compiled code will be syntactically different between cross-compiler, cross-optimization, and cross-architecture binaries. Due to the syntactic difference, it is difficult to recognize the similar functions between different binary code.

Previous solutions to deal with the binary code similarity problem can be divided into two categories: static analysis and dynamic analysis. In general, these solutions suffer from the following limitations: 1) most of the existing methods heavily rely on the engineered syntactic features (e.g., control-flow graph) of the binary code for similarity comparison, 2) some methods incur considerable performance cost due to adopting the time consuming mechanisms (e.g., graph matching and emulation), 3) some methods cannot well compare the binary code similarity across different CPU architectures and different program versions.

To address these problems, in this paper, we present a novel binary similarity detection framework, BinDeep, by utilizing the deep learning techniques. Different from previous solutions, our method does not need any manually selected feature for the similarity computation.

The basic idea of our approach is to leverage the siamese neural network to measure the similarity of binary functions. To obtain the input for the neural network, we first disassemble the binary code and extract the instruction sequence as the features. Next, we utilize the classical natural language processing model to convert the instruction sequences into the vectors. Considering different comparison scenario, we use different siamese neural network for similarity measurement. For this purpose, we apply a deep learning model to identify the specific types of two functions to be compared. Unlike the conventional Siamese network structure, we make use of the hybrid network structure, which combines CNN and LSTM network to measure the binary function similarity. Since CNN can extract the local spatial features while LSTM is capable of extracting sequential features automatically, the hybrid neural network model could improve the similariy detection.

We have implemented our approach based on IDA Pro (Hex-Rays, 2018) and Keras (keras team, 2019). To evaluate the effectiveness of our method, we prepare a custom dataset, which consists of more than 47 million function pairs. The experiments show that our solution can identify similar and dissimilar function pairs effectively on cross-architecture, cross-compiler, cross-optimization, and cross-version binaries.

In summary, we make the following contributions:

- We propose a novel deep learning based solution for binary code similarity detection. This model utilizes the hybrid siamese neural network to measure the binary code similarity.
- We use the instruction embedding model to vectorize the extracted instructions. To identify the types of functions to be compared, we apply a deep learning classification model.
- We conduct extensive experiments to evaluate our approach. The experimental results show that BinDeep can achieve an average precision, recall, and F1 Score of 97.07%, 98.88%, and 97.97% respectively for BCSD.

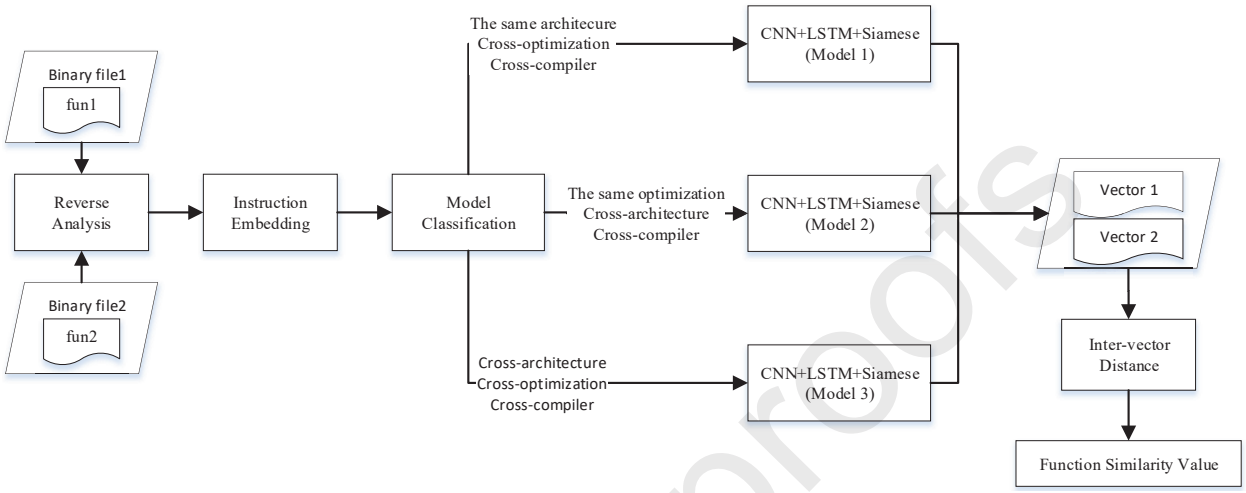


Figure 1: The Similarity Detection Framework of BinDeep. The input to the framework is two binary functions. The output is the similarity value of these two functions.

2. Approach

2.1. Problem Statement

The key task of our study is to judge whether the two functions I_1, I_2 from different binary code are similar or not. If the two functions are the compiled results from the same original source code, they are similar. Otherwise, they are dissimilar. It is worth noting that the target binaries may be compiled using different compilers with different optimization levels, and they may also come from different CPU architectures and different program versions. Due to the complications arising from different compilation, it is a non-trivial task to achieve accurate similarity comparison for two binary functions.

2.2. The Framework of BinDeep

As shown in Figure 1, the framework of BinDeep can be divided into three stages. In the first stage, we exploit IDA Pro to analyze the binary code statically. After disassembling the instructions, we get an instruction sequence for each function in the binary code. For the convenience of later using the deep learning model, we leverage a NLP model to perform the instruction embedding. By doing so, the instruction sequences are converted into the vectors.

In the second stage, we make use of a deep learning model to identify the CPU architectures and optimization levels of target binary functions. According to the identified function types, we will select a proper model for the binary code similarity detection. The main advantage of this stage is that we can improve the similarity detection with pertinence. Thanks to this stage, different comparison scenarios will result in adopting different comparison models in the next stage.

In the third stage, we utilize the Siamese neural networks to detect the binary code similarity. There are three Siamese network models, and each one corresponds to a different comparison scenario. For simplicity, all these three Siamese neural networks have the same structures, but their network parameters are not identical. Different from the traditional Siamese structure, we combine the CNN and LSTM models for the neural network construction. After all these networks are well trained, they can convert the similar (or dissimilar) binary functions into similar (or dissimilar) vectors. By computing the distance between two binary functions, we can get their similarity value. If the value is smaller than the predefined threshold, we think the two binary functions are similar. Otherwise, they are dissimilar.

2.3. Feature extraction and processing

In previous approaches, most of them rely on the prior knowledge for feature extraction. In our solution, we just utilize instruction sequence as the features. For this purchase, we exploit IDA Pro to dissemble the binary code and then get an instruction sequence for each function. For simplicity, the internal control flow of a function is not considered. Similar to the recent studies (Massarelli et al., 2019; Zuo et al., 2019), we make use of the NLP (Nature Language Processing) model to build our instruction embedding. In general, an instruction can be divided into two parts: one opcode and one (or more) operand(s). The number of opcode type is limited while the representation of the operands changes a lot across different computing scenes. To address this problem, a straightforward method is to use only instruction opcodes as tokens for instruction embedding, omitting instruction operands. However, doing so will result in the information loss. In fact, instruction operands contain important semantic information for similarity comparison.

To keep the operand information, the normalization processing is need. Different from the recent method (Massarelli et al., 2019; Zuo et al., 2019), we propose a simple but effective way to normalize the instruction operands. Specifically, we classify the common operands into 8 different categories:

Table 1: Instruction normalization examples

Original instruction		Normalized instruction	
push	ebp	push	TypeOne
mov	ebp, esp	mov	TypeOne, TypeOne
sub	esp, 18h	sub	TypeOne, TypeFive
mov	dword ptr [esp+74h], 0	mov	TypeThree, TypeFive
lea	eax, [ebp+esi*2+0]	lea	TypeOne, TypeFour
cmp	eax, 210h	cmp	TypeOne, TypeFive
jl	loc_804B271	jl	TypeSeven
mov	eax, 1	mov	TypeOne, TypeFive
call	_exit	call	TypeSix

General Register, Direct Memory Reference, Memory Ref [Base Reg + Index Reg], Memory Reg [Base Reg + Index Reg + Displacement], Immediate Value, Immediate Far Address, Immediate Near Address and Other Type. Table 1 shows the examples of instruction normalization for the x86 architecture.

After the raw instructions are normalized, the next step is to perform the instruction embedding. There are two common methods for the embedding: one-hot encoding (Wikipedia, 2018) and word2vec (Gensim, 2018). The one-hot encoding is very simple to represent an instruction, but it cannot capture the relevance of two similar instructions. On the contrary, the word2vec method is capable of converting similar instructions to similar vectors. For example, by using word2vec, the add and sub instructions will be converted into the similar vectors. The word2vec contains two different models: CBOW (Continuous Bag of Words) and skip-gram. Compared with the CBOW model, the skip-gram model can achieve better performance on a large dataset. Therefore, we make use of the skip-gram model to build our instruction embedding.

The basic idea of the skip-gram model is to utilize the context information (i.e., a sliding windows) to learn word embeddings on a text stream. For each word, the model will initially set a one-hot encoding vector, and then it gets trained when going over each sliding window. The key point of the model is

to figure out the probability P of an arbitrary word w_k[1] in a sliding window C_t[2] given the embedding $\vec{w}_t$[3] of the current word w_t. For this purpose, the *softmax* function is used as follows:

$$P\left(w_k|w_t\right) = \frac{\exp\left(\vec{w}_t^T \vec{w}_k\right)}{\sum_{w_i \in C_t} \exp\left(\vec{w}_t^T \vec{w}_i\right)}$$

where $\vec{w}_k$ and $\vec{w}_i$ are the embeddings of words w_k and w_i, $\vec{w}_t^T \vec{w}_i$ is the similarity of two words w_t and w_i. To train the model on a sequence of T words, we utilize stochastic gradient descent to minimize the log-likelihood objective function $J\left(w\right)$ defined as follows:

$$J\left(w\right) = -\sum_{t=1}^{T} \sum_{w_k \in C_t} \log P\left(w_k|w_t\right)$$

Figure 2 illustrates the examples of the instruction embedding. The input of this model is an instruction, the output is a 300-dimensinal vector. In other words, the instruction sequence will be represented as multiple 300-dimensinal vectors. Section 3.4 shows the effect of different embedding dimension on the experiments.

2.4. Model Classification

After the instruction sequence of functions are vectorized, the next step is to train the deep learning based comparison model so that the similarities of two functions can be well measured. Previous methods (Liu et al., 2018; Massarelli et al., 2019) use only one single deep learning model for similarity detection between cross-architecture, cross-compiler, cross-optimization, and cross-version binary code. Considering heterogeneous comparison targets, using one deep learning model may not achieve good detection accuracy in some cases. To deal with this issue, we adopt different deep learning

[1] A sentence W consists of several words w_k, and it can be represented as $W = \left(w_1, ..., w_n\right), w_k \in R, 1 \leqslant k \leqslant n$, n refers to the number of words in a sentence.

[2] A sliding window C is a series of small part of a sentence W, and it can be represented as $C = \left(c_1, ..., c_m\right), c_i = \left(w_j, ..., w_{j+T-1}\right), 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n - T + 1, w_j \in R$, m refers to the number of fixed windows, n refers to the number of words in a sentence W, and T refers to the size of a fixed window. In practice, for each word w_j, we use a floating-point number for representation, and each one has the same accuracy.

[3] $\vec{w}_t \in R^L, L$ refers to the vector dimension.

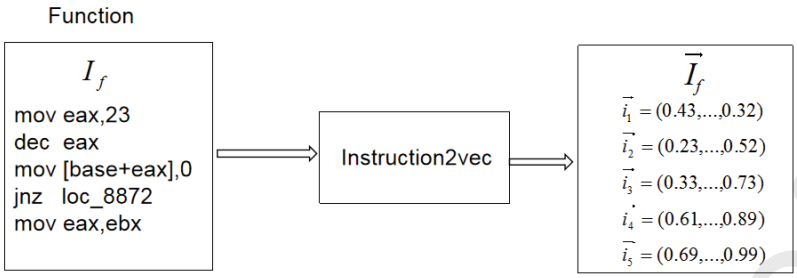


Figure 2: The process of converting assembly instructions into embedding vectors

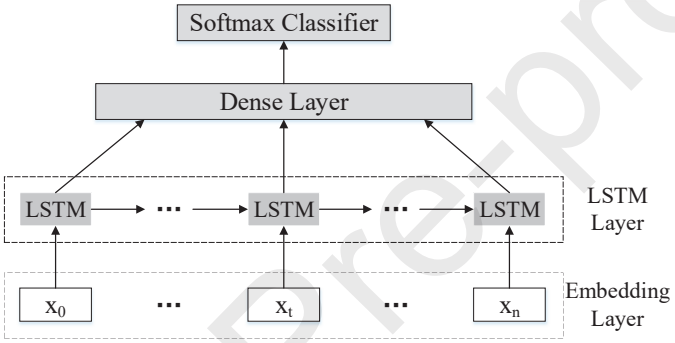


Figure 3: Neural network architecture for model classification. The input to the network is a binary function. The output is the identification type of this function.

models for different comparison scenarios. In other words, we use a tailored model for each comparison scenario. Specifically, we classify the comparison scenarios into three categories: the same CPU architectures with different compilation, the same optimization levels with different CPU architectures, and different CPU architectures with different compilation and optimization. Accordingly, we use three different Siamese neural network models for these similarity comparison. As mentioned previously, these models have the same network structures.

To select a proper neural network model, we need to identify the specific types of two functions to be compared. For this purpose, we make use of a RNN-based classifier. The classifier consists of two models: one model to identify the CPU architecture (x86/x64/arm) and the other model to identify the optimization level (O0/O1/O2/O3). In each hidden layer of the RNN model, the output of the next moment is determined by the output

of the current moment and the input of the next moment. Thanks to this recycling design, the RNN model can well handle the sequential information such as sentences in texts. Since the instruction sequences are similar to the sentences, the RNN model is suitable for the classification task to identify the CPU architecture and optimization level. Although the RNN model is powerful for processing the sequential data, it is hard to train this model due to the back propagation gradient diffusion or gradient explosion. To address this problem, we make use of the LSTM model.

Figure 3 illustrates the architecture of our LSTM-based neural network for the compiling optimization level identification. This neural network consists of three layers. The first layer is the embedding layer. By reusing the pre-trained weights from the word2vec model, this layer can map the instruction sequence to 300 dimensional vectors. The input to this layer is the instruction embedding vectors $\vec{I} = \left(\vec{i}_1, ..., \vec{i}_n\right), \vec{i}_j \in R, 1 \leqslant j \leqslant n, n = 1000$, and the output of this layer is another instruction embedding vectors $\vec{l} = \left(\vec{l}_1, ..., \vec{l}_m\right), \vec{l}_j \in R^L, 1 \leqslant j \leqslant m, m = 1000, L = 300$. The second layer is a LSTM layer, and its output dimension is 50.

Each LSTM unit contains the following equations:

$$
\begin{aligned}
i_t &= \text{sigmoid}\left(W_i x_t + U_i h_{t-1} + b_i\right) && i_t \in R^{50}, b_i \in R^{50} \\
f_t &= \text{sigmoid}\left(W_f x_t + U_f h_{t-1} + b_f\right) && f_t \in R^{50}, b_f \in R^{50} \\
\widetilde{c}_t &= \tanh\left(W_c x_t + U_c h_{t-1} + b_c\right) && \widetilde{c}_t \in R^{50}, b_c \in R^{50} \\
c_t &= i_t \odot \widetilde{c}_t + f_t \odot c_{t-1} && c_t \in R^{50} \\
o_t &= \text{sigmoid}\left(W_o x_t + U_o h_{t-1} + b_o\right) && o_t \in R^{50}, b_o \in R^{50} \\
h_t &= o_t \odot \tanh\left(c_t\right) && h_t \in R^{50}
\end{aligned}
$$

In the above equations,c_t, i_t, f_t, o_t denote the memory state, input, forget, output gates at time $t \in \{1, \ldots, T\}$, $\odot$ operator is the point-wise multiplication, W, U are the weight matrices, and b is the bias parameter.

The final layer is a dense output layer with four neural units, and its output is the classification result. To facilitate the multi-classification, we utilize the softmax as the activation function. For training this neural network, we use the categorical_crossentropy as our loss function, and apply adam as the optimizer. To prevent the trained neural network getting overfitting, we use dropout and set its rate to 0.5.

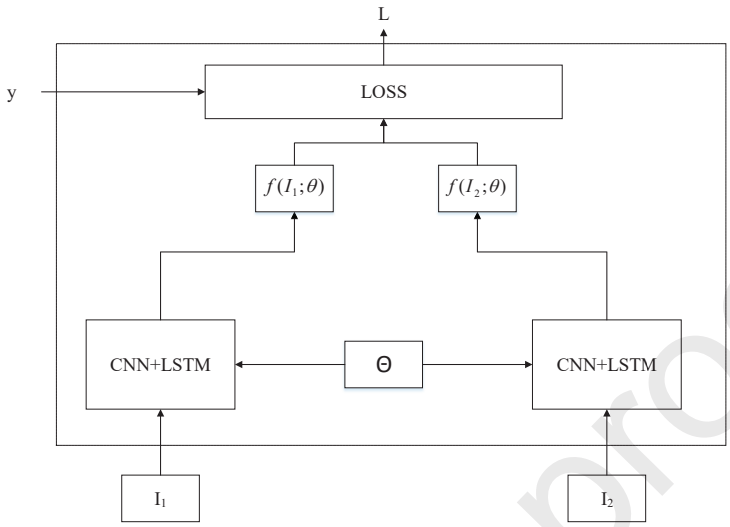


Figure 4: Siamese network architecture for similarity comparison. The input to the network is two binary functions. The output is the similarity comparison result.

2.5. Similarity Comparison

After the types of two binary functions are identified, we select the corresponding neural network model for computing the similarity. Motivated by the recent method on assessing semantic similarity between sentences (Mueller and Thyagarajan, 2016), we leverage the Siamese network for measuring the similarity of two binary functions. The basic idea of the Siamese network is to use two identical mapping function for converting the two inputs into two fixed-dimensional vectors. By comparing the distance of these two vectors, we can figure out the similarity of two original inputs. As shown in Figure 4, the Siamese network architecture consists of two identical embedding neural networks. Different from the standard methods, we combine the CNN and LSTM models to construct the neural networks. The CNN model is good at spatial structure learning, while the LSTM model excels at sequence learning. Our hybrid model takes both advantages of these two models so that the deep learning-based similarity detection could be improved. For ease of representation, we use the term hybrid model and CLSTM (i.e., CNN+LSTM) model interchangeably.

Figure 5 illustrates the structure of the hybrid neural network. It consists of 5 layers. The first embedding layer is used for mapping the instruction sequence to a fixed-dimensional vector. This layer can be represented by

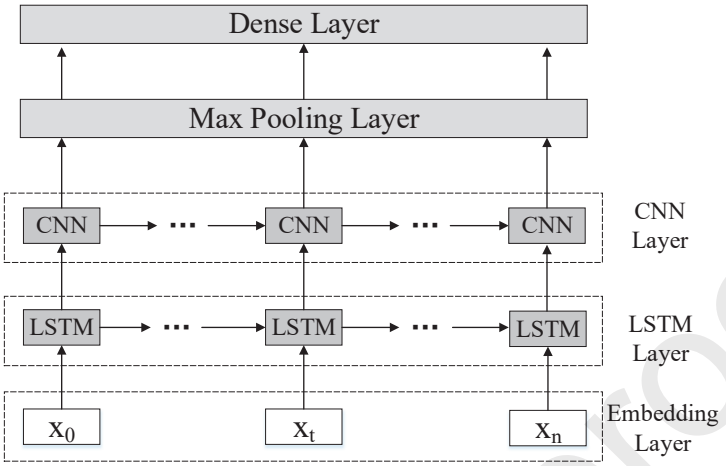


Figure 5: Hybrid neural network architecture

a mapping function $\boldsymbol{f} : X \rightarrow \vec{X}$, where X is the instruction vector, and $\vec{X}$ is the mapping vector. $X = (x_1, ..., x_{1000}), x_j \in R, 1 \leqslant j \leqslant 1000; \vec{X} = (\vec{x}_1, ..., \vec{x}_{1000}), \vec{x}_j \in R^L, 1 \leqslant j \leqslant 1000, L = 300$. This layer reuses the pre-trained weights from the word2vec model. The second layer is a LSTM layer, and its output dimension is 20. The output of this layer can be expressed as follows:

$$H = LSTM\left(\vec{X}\right)_{W_1, b_1}$$

where $LSTM$ represents the mathematical operations performed at LSTM units, W_1 and b_1 are their weight and bias parameters. $H = (h_1, ..., h_{1000}), h_j \in R^L, 1 \leqslant j \leqslant 1000, L = 20$. The third layer is a CNN layer, which contains 300 one-dimensional convolution filters[4]. The length of the 1D convolution window is set to 5, and the convolution stride is set to one. The CNN layer is aimed at extracting local features from the embedding layer. The output of this layer can be represented as follows:

$$A = \sigma\left(\mathrm{Conv}\left(H\right)_{W_2, b_2}\right)$$

where σ is the ReLU function, Conv represents the convolution operation, W_2 and b_2 are the weight and bias parameters of the convolutional filters. $A = (a_1, ..., a_m), a_i \in R^L, 1 \leqslant i \leqslant m, L = 996, m = 300$. The fourth layer is

[4]1D-Convolution processing is usually used in NLP.

a Max pooling layer. It is used to simplify the extracted features. The size of the max pooling windows is set to 2, and the stride is also set to 2. For simplicity, the max pooling operation can be represented as the expression: $\widetilde{A} = Max\left(A\right)$. $\widetilde{A} = \left(\widetilde{a_1}, ..., \widetilde{a_m}\right), \widetilde{a_i} \in R^L, 1 \leqslant i \leqslant m, L = 498, m = 300$. The final layer is a dense layer, which can help connecting the local features. Its output can be represented as follows:

$$O = \sigma\left(W_3 \cdot \widetilde{A} + b_3\right)$$

where $\cdot$ represents the dot product operation, W_3 and b_3 are the weight and bias parameters of this layer. $O = \left(o_1, ..., o_m\right), o_i \in R, 1 \leqslant i \leqslant m, m = 300$.

The inputs to the Siamese network are two binary functions, namely I_1 and I_2. These two functions may be compiled from different CPU architectures, compilers, optimization levels, and program versions. The input length is set to 1000. If the function contains less than 1000 instructions, we use the nop instructions as paddings. On the other hand, if the function contians more than 1000 instructions, we will truncate the tail instructions. The outputs of the embedding layers of the Siamese network are the two embedding vectors, namely f(I_1, θ) and f(I_2, θ), where f [5] represents the hybrid network structure, and θ represents the parameters of this network structure. We assume the embedding dimension is m. Additionally, there is an indicator input y to the Siamese network, indicating whether the two input are similar or not. Precisely, if y is equal to 1, it indicates the two binary functions are similar, if y is 0, it indicates the two binary functions are dissimilar.

To define the loss function of the network, we leverage the contrastive loss function (Hadsell et al., 2006). The basic idea of this loss function is to maximize the distance between two dissimilar inputs, but to minimize the distance between two similar inputs. For this purpose, the loss function is defined as follows:

$$\mathrm{L}(\theta) = \mathrm{A}verage\{y \cdot D(I_1, I_2) + (1 - y) \cdot \max(0, 1 - D(I_1, I_2))\}$$

[5] $f : (I, \theta) \rightarrow \vec{I}$ is a parameterized function that takes a binary function $I = \left(i_1, ..., i_{1000}\right), i_j \in R, 1 \leqslant j \leqslant 1000$, as inputs, and outputs an embedding vector $\vec{I} = \left(\vec{i}_1, ..., \vec{i}_{300}\right), \vec{i}_j \in R, 1 \leqslant j \leqslant 300$.

$$D(I_1, I_2) = \sum_{k=1}^{m} |\mathrm{f}(\mathrm{i}_{1k}, \theta_k) - \mathrm{f}(\mathrm{i}_{2k}, \theta_k)|$$

Where D(I_1,I_2) denotes a Manhattan distance between two binary functions. Training the Siamese network is to find the parameters θ to minimize the loss function. To this end, we make use of the Adam with standard back propagation algorithm.

After the Siamese network is well trained, we can infer two statements. When two binary functions are similar (i.e., y=1), their Manhattan distance should be close to zero so that loss value will be minimal. When two binary functions are dissimilar (i.e., y=0), their Manhattan distance should be close to one[6], and the loss function value will still be minimal.

3. Evaluation

Our experiments are carried out on a Dell T360 Server equipped with two Intel Xeon E5-2603 V4 CPUs, 16GB memory, 2TB hard drives, and one NVIDIA Tesla P100 12GB GPU card. We implement a plug-in of the tool IDA Pro 7.0 to extract an instruction sequence from each binary function. These network models are implemented in TensorFlow-1.8 (Abadi et al., 2016) and Keras-2.2 (keras team, 2019). All these networks are well trained within 50 epochs.

3.1. Dataset

To prepare the dataset, we select 6 popular Linux packages, including coreutils, findutils, diffutils, sg3utils, and util-linux,. After getting the package source code, we use three CPU architectures (x86, x86-64 and ARM) and two compilers (gcc and clang) with four optimization levels (O0, O1, O2, and O3) to compile each program. For x86 and x86-64 architectures, the compilers are allowed to use the extended instruction set (e.g., MMX and SSE). If the two binary functions are compiled from the same source code, they are matched. Otherwise, they are not matched.

To facilitate the supervised learning, the proportions of positive and negative samples (i.e., pairs of matched and unmatched functions) are relatively

[6]The minimal margin distance between two dissimilar functions is set to one according to our experiments.

Table 2: Description of sample types

Sample type	Number	Remarks
Cross-compiler function pairs	855136	Only the compilers are different
Cross-optimization function pairs	855136	Only the optimization levels are different
Cross-version function pairs	501028	Only the function versions are different
Cross-architecture function pairs	1282704	Only the CPU architectures are different
Mixed function pairs	1235136	The Compilers, optimization levels,versions, and architectures may be all different

balanced. To get the positive training samples, we use the unstripped information in the binary code to identify the matched functions in the different compiled files. To get the negative training samples, we randomly select functions in the different binary files with the different function names. The labels for the positive samples and negative samples are ones and zeros.

In total, we obtain 4729140 samples. As shown in the Table 2, these samples can be divided into 5 categories: cross-compiler, cross-optimization, cross-version, cross-architecture, and mixed function pairs. To evaluate the effectiveness of our method on unseen binary code, the whole dataset is split into three disjoint subsets for training, validation and testing. We set the proportion of these three subsets to 4:1:1. The debug symbol information are all stripped in these samples.

3.2. Evaluation metrics

In order to evaluate the performance of our method, we make use of the standard metrics: accuracy, precision, recall, F1, and TPR, which are defined as follows:

Table 3: Results under different instruction features

Instruction feature	Accuracy	Precision	Recall	F1 Score	FPR
Opcode	0.9763	0.9654	0.9725	0.9689	0.0215
Opcode + Operand	0.9851	0.9705	0.9913	0.9808	0.0187

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{4}$$

$$FPR = \frac{FP}{FP + TN} \tag{5}$$

In the above formulas, The True Positive (TP) represents the number of correctly identified matched function pairs. The False Positive (FP) refers to the number of wrongly identified function pairs when the deep learning model identifies the unmatched function pairs as matched. The True Negative (TN) represents the number of correctly identified unmatched function pairs. The False Negative (FN) refers to the number of wrongly identified unmatched function pairs. Accuracy refers to the percentage of function pairs that are identified correctly. Precision measures the percentage of matched function pairs that are correctly labeled. Recall represents the ability to identify matched function pairs correctly. FPR measures the percentage of unmatched function pairs that are incorrectly labeled as matched ones. F1 score refers to the harmonic mean of Precision and Recall.

3.3. Effect of the feature processing

In general, we use the instruction sequence as the features. An instruction consists of one opcode and one (or more) operand(s). Some methods (HaddadPajouh et al., 2018) use the opcode as the feature, omitting the operands,

Table 4: Results under different embedding dimension in the LSTM model

Embedding dimension	Accuracy	Precision	Recall	F_1Score	FPR
100	0.9062	0.8964	0.8526	0.8739	0.0609
200	0.9107	0.9021	0.8599	0.8804	0.0578
300	0.9248	0.9208	0.8776	0.8986	0.0466

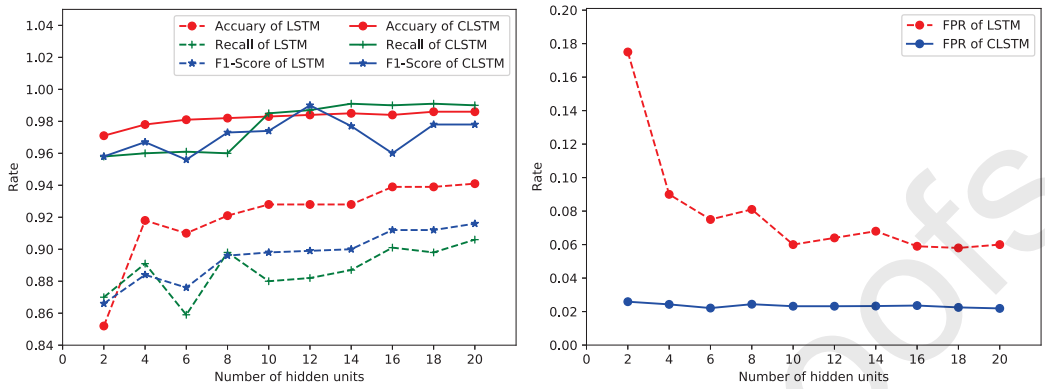
Table 5: Results under different embedding dimension in the CLSTM model

Embedding dimension	Accuracy	Precision	Recall	F_1Score	FPR
100	0.9820	0.9692	0.9844	0.9767	0.0195
200	0.9812	0.9690	0.9825	0.9757	0.0196
300	0.9843	0.9707	0.9888	0.9797	0.0185

while our method uses the whole instruction as the feature. To compare the effect of the different feature processing on the similarity identification, we conduct the corresponding experiments. As shown in the Table 3, our method is better than the approaches that only use the opcode as features in various evaluation metrics, including accuracy, precision, recall, F1 score, and FPR. The main reason is that the whole instruction contains more information than the opcode.

3.4. Effect of the embedding dimension

In this part, we explore the effect of the embedding dimension on the identification results. For this purpose, we analyze the performance of the LSTM and CLSTM neural network models for the mixed function pairs with different embedding dimension. Table 4 shows the identification metrics of the LSTM model. When the embedding dimension is increased, the identification result will be better. For the CLSTM model shown in the Table 5, when the embedding dimension is changed from 100 to 300, the identification result is relatively stable. These experiments show the CLSTM model is more robust than the LSTM model on the embedding dimension setting.



(a) (b)

Figure 6: Results under different hidden unites in the LSTM and CLSTM models

3.5. Effect of the number of hidden unites

To examine whether the number of hidden unites affects the identification results, we carry out the experiments under different number of hidden unites in the neural network from 2 to 20. Figure 6a and 6b show the results on accuracy, recall, F1 score, and FPR when using the LSTM and CLSTM neural network models. In general, as the number of hidden unites increase, the accuracy and recall of these models are increased, the FPR are decreased. In the LSTM model, the identification metrics are similar when setting the number of hidden unites to 16, 18, and 20. Considering the more hidden unites will result in more computing cost, we think setting the unite number to 16 is a good compromise between effectiveness and efficiency. In the CLSTM model, when the number of hidden unites is 18, the various evaluation metrics are optimal.

3.6. Effect of adding the classification model

Previous methods only use a single neural network model to measure the similarities of two functions. Different from these methods, we first utilize the LSTM based classification model to identify the function types and then select the proper neural network model for the similarity measurement. To show the effeteness of the classification model, we conduct a set of tests. Table 6 and Table 7 show the comparison results of the LSTM and CLSTM neural network models when the classification model is enabled/disabled. With help of the classification model, the identification accuracy, precision,

Table 6: Results of the LSTM and classification model + LSTM

Model	Accuracy	Precision	Recall	F_1Score	FPR
LSTM	0.8830	0.8788	0.8493	0.8637	0.0908
Classification model + LSTM	0.9263	0.9128	0.8746	0.8932	0.0515

Table 7: Results of the CLSTM and classification model + CLSTM

Model	Accuracy	Precision	Recall	F_1Score	FPR
CLSTM	0.9206	0.9023	0.8829	0.8924	0.0591
Classification model + CLSTM	0.9844	0.9702	0.9895	0.9797	0.0187

recall, and F1 score are all improved, and the FPR is decreased in the LSTM and CLSTM models. In particular, the accuracy, precision, recall, and F1 score of the CLSTM model are increased by 6.38%, 6.79%, 10.66% and 8.73%, the FPR of the CLSTM model is decreased by 4.04%. The main reason for the effectiveness of adding the classification model is that we can utilize more targeted neural network model for the similarity detection.

3.7. Effect of the neural network structure

To explore the effect of the network structure on the similarity measurement, we carry out a set of experiments in different scenarios. As mentioned previously, the dataset can be divided into 5 different categories, which are corresponding to different comparison scenarios. Regarding the performance on the mixed dataset, Table 8 shows the evaluation results of the CNN, LSTM and CLSTM neural network models. In these models, the embedding dimension is set to 300, and the number of hidden unites is set to 18. From this table, we can see the CLSTM model has obviously better performance than the CNN and LSTM models. Table 9, Table 11, and Table 10 show the performance results on the cross-architecture, cross-compiler and the cross-optimization datasets respectively. Similarly, the CLSTM model has better performance in these experiments. For the performance on the cross-version dataset, Table 12 illustrates the evaluation result. In this evaluation, the dataset consists of 6 versions of the GNU Core Utilities, including the lat-

Table 8: Comparison results of LSTM, CNN and CLSTM models on the mixed dataset

Model	Accuracy	Precision	Recall	F1 Score	FPR
LSTM	0.9228	0.9110	0.8808	0.8956	0.0534
CNN	0.9029	0.8824	0.8699	0.8761	0.0900
CLSTM	0.9868	0.9700	0.9911	0.9804	0.0189

Table 9: Comparison results of LSTM, CNN and CLSTM models on the cross-architecture dataset

Model	Accuracy	Precision	Recall	F1 Score	FPR
LSTM	0.9204	0.9025	0.8871	0.8947	0.0619
CNN	0.9093	0.8939	0.8649	0.8791	0.0897
CLSTM	0.9800	0.9690	0.9859	0.9804	0.0245

Table 10: Comparison results of LSTM, CNN and CLSTM models on the cross-optimization dataset

Model	Accuracy	Precision	Recall	F1 Score	FPR
LSTM	0.9304	0.9179	0.8812	0.8991	0.0459
CNN	0.8998	0.9149	0.8679	0.8907	0.0718
CLSTM	0.9913	0.9860	0.9955	0.9769	0.0123

Table 11: Comparison results of LSTM, CNN and CLSTM models on the cross-compiler dataset

Model	Accuracy	Precision	Recall	F1 Score	FPR
LSTM	0.9069	0.8959	0.8798	0.8878	0.0733
CNN	0.9023	0.9031	0.8699	0.8861	0.0721
CLSTM	0.9727	0.9312	0.9956	0.9623	0.0296

Table 12: Comparison results of LSTM, CNN and CLSTM on the cross-version dataset

Model	Version	Accuracy	Precision	Recall	F1 Score	FPR
LSTM	5.0	0.6424	0.6576	0.6411	0.6599	0.2904
	6.0	0.7241	0.7267	0.7190	0.7254	0.2888
	7.0	0.7409	0.7399	0.7536	0.7404	0.2565
	8.0	0.8133	0.8402	0.8209	0.8265	0.1409
	8.30	0.9888	0.9735	0.9743	0.9738	0.0377
CNN	5.0	0.6289	0.6304	0.6253	0.6296	0.3104
	6.0	0.6920	0.7156	0.6889	0.7036	0.2818
	7.0	0.7165	0.7085	0.7212	0.7125	0.2765
	8.0	0.7787	0.7792	0.7867	0.7789	0.1963
	8.30	0.9798	0.9774	0.9659	0.9716	0.0584
CLSTM	5.0	0.7135	0.7103	0.7036	0.7069	0.1221
	6.0	0.7868	0.7834	0.7765	0.7799	0.1325
	7.0	0.8154	0.8053	0.8175	0.8113	0.0899
	8.0	0.8732	0.8942	0.8766	0.8854	0.0321
	8.30	0.9960	0.9925	0.9995	0.9908	0.0075

est version 8.31. This evaluation also demonstrates the CLSTM model is superior to the CNN and LSTM models.

4. Discussions

Similar to the previous studies (Liu et al., 2018; Massarelli et al., 2019; Shalev and Partush, 2018; Xu et al., 2017; Zuo et al., 2019), our method is limited to cope with the obfuscated binary code. Before applying our approach, the deobfuscation procedure is needed to first extract the internal logic from the obfuscated code. To this end, we could leverage the recent deobfuscation techniques (Yadegari et al., 2015; Xu et al., 2018). We plan to explore the combination of our current method and the deobfuscation technique as our future work.

To further improve the detection accuracy of our method, a potential solution is to use different neural network sturctures for different comparison scenarios. For example, we could apply the BiLSTM model for BCSD across different architectures, and use the CLSTM model for BCSD across different

compilers. In addition, we may consider more comparison scenarios, which will correspond to more network models. To measure the similarity distance, we could explore a different method. For example, we may utilize Hamming distance of static binary features (Taheri et al., 2020a) for BCSD. Considering the recent data poising attacks on machine learning models, we may leverage the existing work (Taheri et al., 2020b) to implement the defense.

5. Related Work

Static Analysis. The basic idea of static analysis methods is to analyze the program code statically without executing it. David et al. (David and Yahav, 2014) propose a tracelet matching method for computing similarity between functions. Eschweiler et al. (Eschweiler et al., 2016) utilize the numeric features to identify the potential candidate functions, and then exploit the structural features for similarity computation. Chandramohan et al. (Chandramohan et al., 2016) present a selective inlining technique to capture function semantics and use this technique for binary search in different CPU architectures and operating systems. Shalev et al. (Shalev and Partush, 2018) employ a machine learning method for BCSD. For cross-architecture vulnerability search in binary firmware, Zhao et al. (Zhao et al., 2019) propose a novel solution based on kNN-SVM and attributed control flow graph. Wang et al. (Wang et al., 2019) develop a staged firmware function similarity analysis approach, which considers the invocation relations as important features. Feng et al. (Feng et al., 2016) present a Graph-based method for bug search across different architectures. By converting the CFGs into the numeric feature vectors, this approach can achieve real-time search. To improve the detection performance, Xu et al. (Xu et al., 2017) propose a novel neural network-based approach for BCSD. Recently, Liu et al. (Liu et al., 2018) make use of a deep neural network (DNN) to cope with the challenges of BCSD. For the final similarity measurement, this approach still relies on the manually selected inter-function features. Massarelli et al. (Massarelli et al., 2019) propose a solution to generate function embeddings based on a self-attentive neural network. These embeddings can easily used for computing binary similarity. Zuo et al. (Zuo et al., 2019) make use of NLP techniques to resolve the code equivalence problem and code containment problem. In industry, BinDiff (zynamics, 2018) is a popular tool to identify similar functions in different binaries. The main advantages of static analysis methods are the efficiency, simplicity, and scalability. Our method belongs

to this category. Different from the previous methods, our approach does not require the prior knowledge to extract syntactic features for BCSD.

Dynamic Analysis. Compared with the static analysis methods, dynamic analysis methods need to run the program code in the execution environment. Pewny et al. (Pewny et al., 2015) propose a dynamic approach to identify vulnerabilities cross-architecture. This method uses the I/O behavior of basic blocks for similarity measurement. Egele et al. (Egele et al., 2014) present a novel a novel dynamic equivalence testing primitive to obtain the semantics of a function. This approach can effectively identify the similar binaries with significant syntactic differences. Jhi et al. (Jhi et al., 2015) develop a emulation-based method for software plagiarism detection. Wang et al. (Wang and Wu, 2017) propose an in-memory fuzzing method for binary code similarity analysis. Similarly, Hu et al. (Hu et al., 2018) combine the emulation technique and Longest Common Subsequence (LCS) algorithm to detect binary clone functions. In general, the dynamic analysis methods will introduce considerable performance cost. Consequently, they may not be widely applied in some cases. In addition, most of the existing dynamic analysis methods cannot well handle the binaries across different architectures for complication.

6. Conclusion

In this paper, we present BinDeep, a novel deep learning based solution for binary code similarity detection. We exploit IDA Pro to extract instruction sequence as features for target functions. To vectorize these features, we leverage a classical NLP model. Next, we apply a RNN-based model to identify the types of target functions. Based on these function type information, we select the corresponding Siamese neural network model for the similarity measurement. Compared to the previous work, we combine the CNN and LSTM models to construct the Siamese neural network. The evaluation results show that BinDeep can achieve an average detection accuracy of 98.43% for BCSD across different architectures, compilers, and optimization levels.

Acknowledgement. We would like to thank Dr. Wenmao Liu from NSFOCUS for his constructive comments. This work was supported in part by Strategic Priority Research Program of Chinese Academy of Sciences (No.XDC02010900), National Key Research and Development Program of

China (No.2016QY04W0903), Beijing Municipal Science and Technology Commission (No.Z191100007119010) and National Natural Science Foundation of China (No.61772078 and No.61602035), CCF-NSFOCUS Kun-Peng Scientific Research Foundation, and Open Found of Shanxi Military and Civilian Integration Software Engineering Technology Research Center.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X., 2016. Tensorflow: A system for large-scale machine learning, in: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA. pp. 265–283.

Chandramohan, M., Xue, Y., Xu, Z., Liu, Y., Cho, C.Y., Tan, H.B.K., 2016. Bingo: Cross-architecture cross-os binary search, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA. pp. 678–689.

David, Y., Yahav, E., 2014. Tracelet-based code search in executables, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM. pp. 349–360.

Egele, M., Woo, M., Chapman, P., Brumley, D., 2014. Blanket execution: Dynamic similarity testing for program binaries and components, in: Proceedings of the 23rd USENIX Conference on Security Symposium, USENIX Association, Berkeley, CA, USA. pp. 303–317.

Eschweiler, S., Yakdan, K., Gerhards-Padilla, E., 2016. discovre: Efficient cross-architecture identification of bugs in binary code, in: Proceedings of the 2016 Network and Distributed Systems Security Symposium (NDSS).

Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., Yin, H., 2016. Scalable graph-based bug search for firmware images, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ACM, New York, NY, USA. pp. 480–491.

Gensim, 2018. Word2vec embeddings. http://radimrehurek.com/gensim/models/word2vec.html.

HaddadPajouh, H., Dehghantanha, A., Khayami, R., Choo, K.K.R., 2018. A deep recurrent neural network based approach for internet of things malware threat hunting. Future Generation Computer Systems 85, 88 – 96.

Hadsell, R., Chopra, S., LeCun, Y., 2006. Dimensionality reduction by learning an invariant mapping, in: 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06), pp. 1735–1742.

Hex-Rays, 2018. Ida pro disassembler and debugger. https://www.hex-rays.com/products/ida/index.shtml.

Hu, Y., Zhang, Y., Li, J., Wang, H., Li, B., Gu, D., 2018. Binmatch: A semantics-based hybrid approach on binary code clone analysis, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 104–114.

Jhi, Y., Jia, X., Wang, X., Zhu, S., Liu, P., Wu, D., 2015. Program characterization using runtime values and its application to software plagiarism detection. IEEE Trans. Software Eng. 41, 925–943.

Liu, B., Huo, W., Zhang, C., Li, W., Li, F., Piao, A., 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, ACM. pp. 667–678.

Massarelli, L., Di Luna, G.A., Petroni, F., Baldoni, R., Querzoni, L., 2019. Safe: Self-attentive function embeddings for binary similarity, in: Perdisci, R., Maurice, C., Giacinto, G., Almgren, M. (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment, Springer International Publishing, Cham. pp. 309–329.

Mueller, J., Thyagarajan, A., 2016. Siamese recurrent architectures for learning sentence similarity, in: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI Press. pp. 2786–2792.

Pewny, J., Garmany, B., Gawlik, R., Rossow, C., Holz, T., 2015. Cross-architecture bug search in binary executables, in: 2015 IEEE Symposium on Security and Privacy, pp. 709–724.

Shalev, N., Partush, N., 2018. Binary similarity detection using machine learning, in: Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, ACM, New York, NY, USA. pp. 42–47.

Taheri, R., Ghahramani, M., Javidan, R., Shojafar, M., Pooranian, Z., Conti, M., 2020a. Similarity-based android malware detection using hamming distance of static binary features. Future Generation Computer Systems 105, 230 – 247.

Taheri, R., Javidan, R., Shojafar, M., Vinod, P., Conti, M., 2020b. Can machine learning model with static features be fooled: an adversarial machine learning approach. Cluster Computing .

keras team, 2019. Keras: The python deep learning library. https://keras.io/.

Wang, S., Wu, D., 2017. In-memory fuzzing for binary code similarity analysis, in: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, pp. 319–330.

Wang, Y., Shen, J., Lin, J., Lou, R., 2019. Staged method of code similarity analysis for firmware vulnerability detection. IEEE Access 7, 14171–14185.

Wikipedia, 2018. One-hot. https://en.wikipedia.org/wiki/One-hot.

Xu, D., Ming, J., Fu, Y., Wu, D., 2018. Vmhunt: A verifiable approach to partially-virtualized binary code simplification, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, ACM. pp. 442–458.

Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., 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, pp. 363–376.

Yadegari, B., Johannesmeyer, B., Whitely, B., Debray, S., 2015. A generic approach to automatic deobfuscation of executable code, in: 2015 IEEE Symposium on Security and Privacy, pp. 674–691.

Zhao, D., Lin, H., Ran, L., Han, M., Tian, J., Lu, L., Xiong, S., Xiang, J., 2019. Cvsksa: cross-architecture vulnerability search in firmware based on knn-svm and attributed control flow graph. Software Quality Journal .

Zuo, F., Li, X., Young, P., Luo, L., Zeng, Q., Zhang, Z., 2019. Neural machine translation inspired binary code similarity comparison beyond function pairs, in: Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS).

zynamics, 2018. Bindiff. http://www.zynamics.com/bindiff.html.