Mathematical Biosciences and Engineering

http://www.aimspress.com/journal/MBE

MBE, 18(4): 4528–4551.
DOI: 10.3934/mbe.2021230
Received: 12 March 2021
Accepted: 10 May 2021
Published: 25 May 2021

Research article

Cross-platform binary code similarity detection based on NMT and graph embedding

Xiaodong Zhu[1,*], **Liehui Jiang**[1] **and Zeng Chen**[2]

[1] State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China

[2] National Key Laboratory of Science and Technology on Blind Signal Processing, Chengdu, 610000, China

* **Correspondence:** Email: zxd10@tsinghua.org.cn.

Abstract: Cross-platform binary code similarity detection is determining whether a pair of binary functions coming from different platforms are similar, and plays an important role in many areas. Traditional methods focus on using platform-independent characteristic strands intersecting or control flow graph (CFG) matching to compute the similarity and have shortages in terms of efficiency and scalability. The existing deep-learning-based methods improve the efficiency but have a low accuracy and still using manually constructed features. Aiming at these problems, a cross-platform binary code similarity detection method based on neural machine translation (NMT) and graph embedding is proposed in this manuscript. We train an NMT model and a graph embedding model to automatically extract two parts of semantics of the binary code and represent it as a high-dimension vector, named an embedding. Then the similarity of two binary functions can be measured by the distance between their corresponding embeddings. We implement a prototype named SimInspector. Our comparative experiment result shows that SimInspector outperforms the state-of-the-art approach, Gemini, by about 6% with respect to similarity detection accuracy, and maintains a good efficiency.

Keywords: binary code similarity detection; deep learning; graph embedding; neural machine translation

1. Introduction

Nowadays, billions of IoT devices are ubiquitous and bring many conveniences to our daily life. However, security risks come along with the conveniences. One of the main reasons is that due to code reuse and sharing, a single vulnerability in source code may spread across hundreds or more IoT devices that have diverse hardware architectures and software platforms [1]. However, it is difficult to

obtain the source code from the many IoT device companies. We can only use binary analysis as a feasible approach. As an important binary analysis approach, binary code similarity detection (BCSD) can analyze the similarity of binary programs without any access to the source code and figure out the spreading of vulnerabilities among programs.

BCSD is determining whether two given binary functions are similar [2], and plays an vital role in many different fields such as plagiarism detection [3–5], patch analysis [6], malware analysis [7–9] and vulnerability discovery [10–12]. The most common scenario is detecting the similarity of a pair of binary functions coming from different platforms (such as x86, ARM, or MIPS). In recent years, focusing on cross-platform BCSD researchers have performed a lot of studies, which have two main directions. One direction is to represent the program with different platform-independent strands and measure the similarity of two given programs using the intersection of their corresponding strand sets (such as code slices [13–15], tracelets [16], tokens [17, 18], and expressions [19–21]). Another direction is to extract different robust platform-independent features of control flow graph (CFG) vertices from the binary code and determining the similarity by graph matching [1,10,22–24]. On the one hand, code representations in these methods (characteristic strands and CFG vertex features) did not take the semantics of code into account, which results in a low accuracy. On the other hand, the matching process (especially the CFG matching) is too expensive, which leads to a low efficiency.

With the good performance of deep learning (DL) in natural language processing [25,26], computer vision [27, 28] and many other fields, researchers have proposed some similarity detection methods using neural networks [2, 11, 12]. One of the classic methods is Gemini proposed by Xu et al. in 2017 [12]. Gemini firstly extracts some numerical characteristics (such as the number of constants, the number of instructions and the number of calls) of the basic block (BB) as a vector, and combines the vector with the CFG generating an attributed control flow graph (ACFG). Then, different from the traditional graph matching methods, Gemini uses an neural network to encode the ACFG into a high-dimension vector, called a graph embedding, and compute the similarity between two given functions using the distance between their corresponding graph embeddings. This method improves the detection efficiency and has good scalability. However, the numerical characteristics are manually selected and do not take the code semantics into account. These characteristics of the same function might be different under different platforms, which might lead to low detection accuracy.

Natural language processing (NLP) is a rich area focused on processing text of various natural languages, and shares a lot of analogical topics with our cross-platform binary code analysis task, such as semantics extraction, summarization, and classification. Neural machine translation (NMT) is a new proposed method for dealing with NLP problems. When translating the source language to the target language, NMT firstly extracts the semantics of the source language and encodes it into a sentence embedding, and then uses the sentence embedding and the previous outputs to predict the next word of the target language. For binary code similarity detection, we can use the idea of NMT to extract the semantic feature of basic blocks coming from different platforms.

Based on the commonalities of NLP and binary analysis, we use the idea and techniques of NLP to process binary code, and propose a cross-platform binary code similarity detection method based on NMT and graph embedding in this manuscript. Instead of using those manually selected numerical characteristics of basic blocks, this method uses NMT inspired basic block embedding model to encode the semantics of basic blocks into high-dimension vectors, and takes them as the vertex attributes of the ACFG. Then an ACFG embedding model is used to transform the ACFG to a graph embedding,

and the similarity of two given binary functions is measured by the distance of their corresponding graph embeddings.

2. Related works

The main challenge of cross-platform binary code similarity detection is choosing a proper code representation that not only can eliminate the influence of different instruction sets under different platforms to ensure the detection accuracy, but also facilitates efficient detection. According to code representations and matching algorithms, the existing BCSD methods can be classified into three categories: characteristic strands based methods, graph matching based methods and machine learning based methods. This section will review these methods.

2.1. Characteristic strands based BCSD

Researchers have always been searching a type of characteristic strands that can represent a binary function. N-gram [17] and N-perms [18] is two of the earliest methods that perform this research. They use binary sequences or mnemonics to represent the binary codes, instead of extracting the semantics. These methods cannot torrent the influence to the order of the instructions caused by different compilers, thus have a poor compatibility. BinHunt [19] and iBinHunt [20] use symbolic execution and theorem prover to check the semantic equivalence between two basic blocks. These two methods need to perform binary analysis to extract the equation and conduct equivalence check, thus have a pretty high computing complexity and cannot be applied to large-scale binary analysis.

In 2014, David et al. [16] proposed that the path strands consist of some sequential, brief paths generated in the process of function running can be used to represent the binary functions. After that, they use a simple rewriting engine to match the registers and memory addresses between path strands based on alignment constraints and date-based constraint solving. At last, wavelets matching is performed to compare the equivalency of binary functions. However, the dynamic debugging of some embedded firmware is difficult, thus these running time features are hard to capture. Aiming at this problem, they proposed a statistics-based method and implemented a similarity detection tool name Esh [13] in 2016. In this method, binary code is sliced into short comparable fragments and the semantic similarity between fragments is defined, which is then elevated to function similarity using statistical reasoning. Esh uses semantic features of binary programs thus can eliminate the influence comes from compilers and slight modifications to the code. But semantic comparisons require a lot of computation, causing a low efficiency. To improve the detection efficiency, David et al. [14] optimized the semantic strands using a compiler optimizer, making them more concise and standardized, thus reduced the computation complexity of semantic comparison and improved the efficiency of similarity analysis to a certain extent.

Instead of using features of the whole function for comparison, Pewny et al. [21] used a smaller comparative granularity and proposed a binary code bug searching method based on semantic signatures. Their method extracts the semantics of given segment of binary program through symbolic simplification, and then uses Tree Edit Distance (TED) as a code similarity measure at the basic block level. This method can eliminate the influence of the syntax but has a poor robustness to the changes of architectures and compilers. Thus this method cannot be applied to large-scale cross-platform binary bug searching.

2.2. Graph matching based BCSD

With the deepening of code analysis, researchers found that the Control Flow Graph of program contains rich function information, which can be used as the characteristics of binary functions. Flake et al. [24] suggested that some compilation optimizations (such as instruction rearrangement and register allocation changes) can be eliminated by matching CFG functions, but this approach relies on precise graph matching and has a high computational complexity. BinDiff [23] and BinSlayer [22] use a similarity measure based on the isomorphism relationship between CFGs to check the similarity of two binary files. Since these two methods are not designed specifically for vulnerability detection, the vulnerability detection result is not good enough, and the computational complexity of graph isomorphism algorithm is still high.

In the aspect of efficiency improvement, Pewny et al. [1] starts with the matching algorithms and use semantic hash to match the vulnerability signatures. This method takes a segment of binary code (basic blocks and the control flow conversion between them), which is similar to a bug instance, as a bug signature. Then both the bug signature and target program are transformed into intermediate representations (IR) using VEX-IR. After random inputs are fed to the IRs, input-output pairs can be obtained to represent the semantics of the program (this process is named "sampling" in their paper). At last, semantic hash is used to measure the similarity between the bug signature and target program. This method extracts the bug code as the feature, which is only applicable to these bugs with obvious features thus has a poor versatility. Besides, although the semantic feature is used, the sampled feature is only an implementation of the semantics, and it is difficult to ensure the coverage. So random errors might be introduced. By contrast, discovRE [10] looks at the graph itself that needs to be matched and uses a prefilter to filter the CFG to improve the matching efficiency. This method indeed improved the efficiency, but this kind of filtering is not reliable because it introduced a lot of false positive results, which leads to a relative low accuracy.

2.3. Machine learning based BCSD

With the wide applications of machine learning, researchers have tried to use machine learning methods to solve the problem of BCSD. Genius [11] transforms the CFG into a numerical vector to perform similarity detection. Specifically, it extracts ACFG from the binary functions and use an unsupervised learning method to learn high-level classifications. Based on these high-level classifications, the ACFGs can be encoded into characteristic vectors in high-dimension space. At last, locality sensitive hashing (LSH) can be used to search similar functions efficiently. This method transforms graph matching into distance computing of vectors, thus greatly improved the matching efficiency. But the process of transforming from ACFGs to high-dimension vectors still relies on graph matching algorithms to compute the similarity between the target function and the binary function codebooks. Gemini [12] replaced the codebook checking with deep learning, which finally solved the graph matching problem to some extent, but the attributions selection of ACFG in this method limited the accuracy of detection.

3. Preliminary

In order to make the proposed method more comprehensible and present some foundations of this method, this section mainly introduces preliminary works in the following three aspects: firstly, some word representing methods are introduced in Section 3.1; secondly, some fundamentals about NMT and some neural networks commonly used to implement the NMT model are presented in Section 3.2; lastly, the main idea of graph embedding and one of its typical methods called Structure2vec are depicted in Section 3.3.

3.1. Word representations

In the field of machine translation, before the text is feed to the translation model, the words in the text need to be represented as a numerical form. Classic word representation methods can be classified into two categories: word representation methods using external resources and without external resources. Among word representation methods using external resources, the most popular method is WordNet [29]. WordNet relies on an external vocabulary knowledge corpus, which encodes the definition, synonyms, etyma and derivatives of the given word. Currently, the WordNet corpus for English has over 150,000 words and 100,000 synsets. However, the corpus for instructions is currently nonexistent, so WordNet is inappropriate for instruction embedding task.

Word representation methods without external resources mainly include One Hot Encoding, Term Frequency-Inverse Document Frequency (TF-IDF), and Co-occurrence Matrixes. Among them, the One Hot Encoding is a simple word representation and is the basis for many other well-known representations. Assume that there is a vocabulary consists of V words. The i-th word in the vocabulary w_i can be represented as a V-dimension vector [0, 0, ..., 0, 1, 0, ..., 0, 0], whose i-th element is 1 and the others are 0. This representation has many disadvantages. For example, this representation does not encode the relationship between words in any way and completely ignores the context of the word. For large vocabularies, this approach becomes very ineffective.

TF-IDF is a frequency-based approach that takes into account the frequency with which words appears in the corpus. This is a word representation that indicates the importance of a particular word in the given document. Intuitively, the higher the frequency of the word, the more important that word will be in the document. But just counting frequencies does not work because words like "this" and "is" are very frequent, but they do not carry a lot of information. IF-IDF takes this into account and assigns zero values to these common words.

Unlike One Hot Encoding, Co-occurrence Matrix encodes the contextual information of words, but requires maintaining a $V \times V$ matrix that grows polynomially with the size of vocabulary. Thus maintaining such a matrix is costly. Besides, expanding the context window to sizes greater than 1 is not simple either. All the above-mentioned shortcomings prompted us to investigate more robust and scalable learning methods for representing word semantics.

Word2vec [30] is a distributed word representation learning technology proposed recently and is currently used as a feature engineering technique for many NLP tasks (e.g., machine translation, chatbots and image caption generation). It can automatically learn the semantics of a word by checking its surrounding words (i.e., the context), without any human intervention. Specifically, we try to predict contextual words by the neural network based on some given word (and vice versa), which forces the neural network to learn good word embeddings. Compared with the above methods, Word2vec

has many advantages. Firstly, it does not rely on any external resources. Secondly, the representation vector generated by Word2vec is independent to the size of the vocabulary. Thirdly, it is a distributed representation and its representation depends on the activation state of all elements in the vector, which gives it a more powerful representation ability.

There are two mature algorithms for Word2vec word embedding: the skip-gram model and the CBOW model. Among them, skip-gram model proposed by Mikolov et al. [31] is widely welcomed due to its high efficiency and low memory usage compared with other models. According to the test by Mikolov [30] and Pennington [32], skip-gram performs better in semantic tasks, while CBOW performs better in syntactic tasks.

Because our goal is to extract the semantics of basic blocks, eliminating syntactic changes caused by different architectures and compile configurations, we select skip-gram as the word embedding algorithm. In Section 4.2, skip-gram algorithm is implemented to encode instructions under different platforms to numerical vectors.

3.2. RNN, LSTM, BiLSTM, and LSTM-based NMT

When dealing with language translation tasks, NMT typically adopts an Encoder-Decoder model to accomplish the mapping from source language to target language, as is shown in Figure 1. In the Encoder-Decoder model, the encoder reads the words of the source language, denoted as $\mathbf{x} = (x_1, x_2, ..., x_I)$, and encodes them into hidden states, denoted as $\mathbf{h} = (h_1, h_2, ..., h_I)$. The sentence of source language is represented as $\mathbf{c}$. In contrast, the decoder reads vector $\mathbf{c}$ and the previous output sequence $\{y_1, ..., y_{t-1}\}$ and predicts the next word of the target language y_t using 3.1.

$$p(\mathbf{y}) = \prod_{t=1}^{T} p(y_t|\{y_1, ..., y_{t-1}\}, \mathbf{c}) \tag{3.1}$$

where $\mathbf{y} = (y_1, y_2, ..., y_T)$.

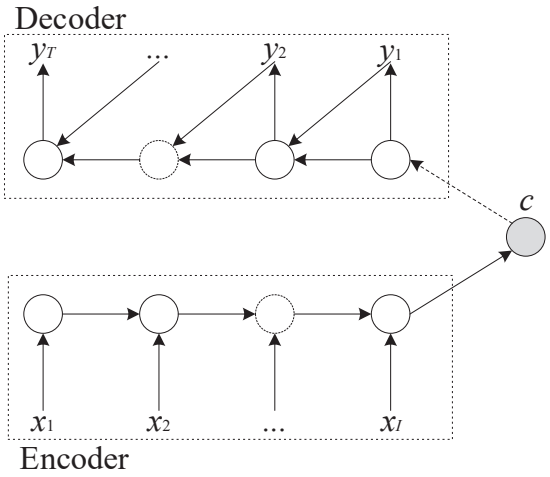


Figure 1. Encoder-Decoder model.

Specifically, the Encoder-Decoder model is often implemented by Recurrent Neural Networks (RNNs). RNN is a type of deep neural network, and can be applied to transform the word embeddings of a sentence into a sentence embedding. RNN maintains a state variable to capture the patterns

in sequential data. RNN shares the same set of parameters over time and for every input in the sequence, RNN updates its parameters. When RNNs are used to implement the Encoder-Decoder model, the relationship between the input x, the hidden state h, the sentence representation c and the output y is shown in 3.2 to 3.4.

$$h_t = f(x_t, h_{t-1}) \tag{3.2}$$

$$\mathbf{c} = q(\{h_1, h_2, ..., h_I\}) \tag{3.3}$$

$$p(y_t|\{y_1, ..., y_{t-1}, \mathbf{c}\}) = g(y_{t-1}, s_t, \mathbf{c}) \tag{3.4}$$

where f, q and g are nonlinear functions, s_t is the hidden state of RNN and can be calculated using 3.5.

$$s_t = f(s_{t-1}, y_{t-1}, \mathbf{c}) \tag{3.5}$$

However, the Encoder-Decoder model implemented by standard RNNs cannot process long sentences. Because standard RNNs can maintain only one hidden state (as shown in Figure 2(a)) thus have vanishing gradient problem. Currently, Encoder-Decoder models with better performance are usually implemented by Long Short Term Memory RNNs (LSTMs) [33]. LSTM is a special RNN, and was developed for covering the shortage of standard RNN in capturing the long-term memory. Compared with standard RNN maintaining only one hidden state, LSTM has more parameters and provides better control over which memories are stored and which are discarded within the given time steps (as shown in Figure 2(b)). It can be seen that the most obvious difference between Figure 2(a) and Figure 2(b) is that a LSTM has three gates, the input-gate, the forget-gate and the output-gate. In Figure 2(b), the x_t, c_t and h_t is respectively the current input, cell state and hidden state, and c_{t-1}, h_{t-1} is respectively the cell state and hidden state of the previous time step.

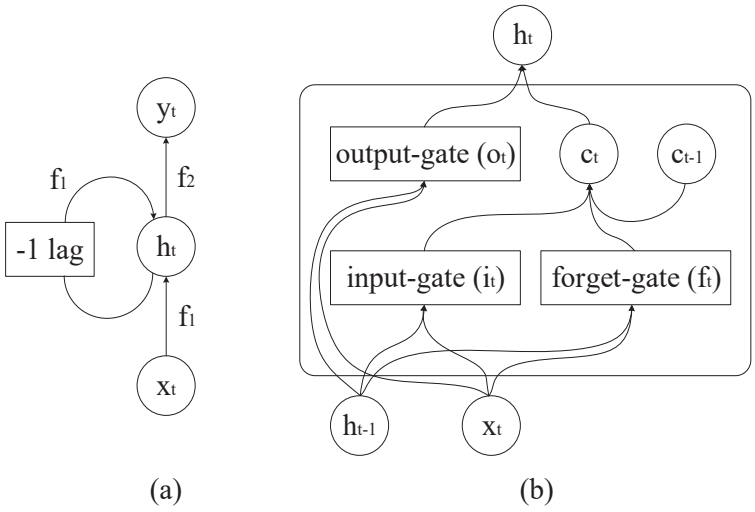


Figure 2. Structure of standard RNN and LSTM.

In text analysis, LSTM regards every sentence as a sequence of words with internal structures (i.e., the dependence and associativity of words). It incrementally encodes the semantic vector of the sentence, from left to right, word for word. In every time step, one word is encoded and the result is used to "refresh" the word dependency embedded to the semantic vector. When this process reaches the end of the sentence, all the words and their dependencies have been embedded to the semantic

vector. Thus, the semantic vector can be regarded as the representation of the whole sentence and the last semantic vector is the sentence embedding.

Recently, Bi-directional Long Short-Term Memory (BiLSTM) [34] is proposed to capture the backward information to further improve the prediction performance. BiLSTM is a combination of two independent LSTM in essence. One learns information from beginning to end, while another learns backward.

In out manuscripts, the main idea of the encoder model is used to achieve the mapping from basic blocks (sequences of instructions) to BB embeddings. In the basic block embedding task, instructions are regarded as words and basic blocks are regarded as sentences. Then an encoder implemented by BiLSTM is used to encode basic blocks under different platforms into BB embeddings. Details are presented in Section 4.3.

3.3. Graph embedding and structure2vec

Graph embedding is to find a mapping function from the graph to a low-dimension vector space. This mapping function can automatically learn the characteristics of the graph instead of manually extracting and the embedding vector generated by this map facilitates the following process of other algorithms.

Assume that an ACFG can be represented as $g = (V, E)$, where V and E respectively represent the sets of vertices and edges. Besides, each vertex v in V has an attribute x_v corresponding to the basic block in ACFG. The graph embedding model firstly calculates a p-dimension characteristic vector for each vertex v ($v \in V$). Then all the characteristic vectors are aggregated and generate the embedding vector μ_g of graph g, as shown in 3.6.

$$\mu_g := A_{v \in V}(\mu_v) \tag{3.6}$$

where A is an aggregating function (summing or averaging), thus 3.6 can be selected as 3.7.

$$\mu_g = \sum\nolimits_{v \in V}(\mu_v) \tag{3.7}$$

Structure2vec [35] is a type of graph embeddings that pays more attention to the structural similarity of the vertices. Its main idea comes from graphical model inference algorithms where vertex features are recursively aggregated based on the topological structure of the graph. After several recursive steps, the network will generate a new feature representation (or embedding) for each vertex which takes into account both graph characteristics and long-range interaction between vertex features. Assume that the set of adjacent vertices of vertex v in graph g is donated as $N(v)$. The Structure2vec network will firstly initialize the embedding of each vertex to 0 and update them at each iteration using 3.8.

$$\mu_v^{(t+1)} = F(x_v, \sum_{u \in N(v)} \mu_u^{(t)}), \forall v \in V \tag{3.8}$$

where x_v is the vertex feature of v, V is the set of vertices in graph g, $\mu_u^{(t)}$ is the embedding of u at the t-th iteration step, $\mu_v^{(t+1)}$ is the embedding of v at the $(t+1)$-th iteration step, and F is a genetic nonlinear mapping.

From the update formula we can see that the embedding process has following properties:

- The update process of the embeddings is carried out in a synchronous manner based on the graph topology. Only after the update for all the embeddings from the previous round has completed, a new round of traversal will be started.

- The feature of each vertex is propagated to other vertices by the nonlinear propagation function F.
- The more times the iterative update is performed, the farther away the vertex features are propagated to distant vertices and get aggregated nonlinearly at distant vertices.
- If the update process terminates after T iterations, each vertex embedding $\mu_v^{(T)}$ will contain information about its T-hop adjacent vertices, which is determined by both the graph topology and the involved vertex features.

Because the basic Structure2vec network is designed for classification problems, each input graph needs a ground truth label to indicate which "class" it belongs to. While ACFG embedding task is not a classification problem. Thus in the proposed method, a variant of Structure2vec network is used to achieve the embedding of ACFGs, as detailed in Section 4.4.

4. Proposed method and implementation

In this section, main ideas of the proposed method are introduced. Firstly, an overview of the proposed method is given out in Section 4.1. Subsequently, the embedding approaches of instructions, basic blocks and the ACFG are presented in Section 4.2,4.3, and 4.4. At last, an implementation of the proposed method named SimInspector is introduced in Section 4.5.

4.1. Solution overview

A given binary function Q consists of some basic blocks and their relationships which can be represented as a CFG. The research aims at finding a function from a set of functions coming from different platforms (such as x86, ARM, and MIPS) that is semantically similar to Q.

The proposed method extracts two parts of semantics. One is the semantics of basic blocks and the other is that of the CFG. The basic block semantics is extracted using a NMT model implemented by BiLSTMs. This model regards every instruction as a word, every basic block as a sentence (a sequence of words). Then, using the idea of the Encoder of the Encoder-Decoder model introduced in Section 3.2, the NMT model encodes the basic blocks under different platforms into vectors with a fixed dimension (named BB embeddings). The NMT model is trained to satisfy the requirement that the embeddings of basic blocks with similar semantics are close to each other, and vice versa. The semantics in the CFG is extracted using an ACFG embedding model implemented by a customized Structure2vec network. The ACFG embedding model takes an ACFG as input and aggregates the attributes of the vertices (BB embeddings) to a graph embedding based on the topology of the ACFG, thus combines the basic block semantics and the control flow transition relationships. Similar to the NMT model, the ACFG embedding model is trained to satisfy the requirement that the embeddings of ACFGs with similar semantics are close to each other, and vice versa.

Specifically, the proposed method mainly consists of four stages: instruction embedding, basic block embedding, ACFG embedding and similarity computing, which will be detailedly introduced in the following sections.

4.2. Instruction embedding

Because the NMT model only accept numerical vectors as inputs, the instructions in the blocks need to be encoded into a numerical representation before extracting the semantics of basic blocks using NMT model.

As introduced in Section 3.1, we use skip-gram model to perform our instruction embedding task. The skip-gram model uses a neural network to learn the word embeddings. In training, a sliding window is designed to cover the text flow. The size of the sliding window m is defined as the number of words that are considered as context on one side. Thus, the number of words in the sliding window is $2m + 1$. For a given word w_i, the structured data is constructed in the form of $[\ldots, (w_i, w_{i-m}), \ldots, (w_i, w_{i-1}), (w_i, w_{i+1}), \ldots, (w_i, w_{i+m}), \ldots]$, where $m + 1 \leq i \leq N - m$, and N is the number of words with practical meanings.

Then the structured data are fed to the neural network to learn the word embeddings. A $V \times D$ matrix, named embedding space or embedding layer, is designed to save the word embeddings, where V is the size of the vocabulary, and D is the dimension of the embeddings. The embedding layer is followed by a softmax layer, whose weights are $D \times V$ and bias size is V. Every word are represented as a one hot encoding vector with size V. Assume that the i-th input is denoted as x_i, and its corresponding embedding is z_i, and its corresponding output is y_i. For every input x_i, z_i can be found from the corresponding embedding layer and the predicted output $\hat{y}_i$ can be calculated using 4.1 and 4.2.

$$logit(x_i) = z_i W + b \tag{4.1}$$

$$\hat{y}_i = softmax(logit(x_i)) \tag{4.2}$$

where *logit* is the nonstandardized score (logits), $\hat{y}_i$ is the predicted output with size V, W is the weights matrix with size $D \times V$, b is the $V \times 1$ bias vector, and $softmax$ is the activation function. Now the loss of the given data point (x_i, y_i) can be calculated using negative log likelihood loss function as 4.3.

$$J(\theta) = -(1/N - 2m) \sum_{i=m+1}^{N-m} \sum_{j \neq i \wedge j = i-m}^{i+m} logit(x_n)_{w_j} - \log\Big(\sum_{w_k \in vocabulary} \exp(logit(x_n)_{w_k}) \Big) \tag{4.3}$$

The loss function can be minimalized using Stochastic Gradient Descent (SGD) in training. However, we can see from 4.3 that every time the formula is used to calculate the loss of a word, the logits of all the words in the vocabulary need to be calculated. It is very costly and needs an effective approximation. In our experiments, we use negative sampling to simplify the calculation and improve the efficiency. Therefore, the loss function can be simplified to 4.4.

$$J(\theta) = -(1/N - 2m) \sum_{i=m+1}^{N-m} \sum_{j \neq i \wedge j = i-m}^{i+m} \log(\sigma(logit(x_n)_{w_j})) + \sum_{q=1}^{k} \mathrm{E}_{w_q \sim vocabulary - (w_i, w_j)} \log(\sigma(-logit(x_n)_{w_q})) \tag{4.4}$$

where σ is the sigmoid activation function and $\sigma(x) = 1/(1 + exp(-x))$, w_j is a contextual word of w_i and w_q is a noncontextual word of w_i.

Because of the simple structure of the network and the usage of layered softmax, the model can be trained on desktop machines at billions of words per hour. In addition, the word embeddings learning process is completely unsupervised.

4.3. Basic block embedding

After the instructions encoding, a BiLSTM-based basic block embedding model is used to capture the first part of the semantics, the semantics of the basic blocks.

InnerEye-BB [36] uses an LSTM as the basic block embedding model, which is not good enough for resisting the changes of the instruction order. Because LSTM sequentially encodes the information of each word, only forward dependencies can be captured, instead of backward dependencies of words. However, the variation of instruction order is very common in programs under different platforms. Because different architectures, compilers, optimization options, as well as possible code confusions can all lead to changes of instruction order. Therefore, a better model is needed to overcome this problem.

In this paper, we use BiLSTM to capture both the forward and the backward information of instruction sequences to overcome this problem. BiLSTM consists of a forward LSTM and a backward LSTM, in which the forward one captures the front-to-back information, and the backward one captures the back-to-front information. In Figure 3, $\mathbf{e}_i$ ($i = 1, 2, ..., T$) are instruction embeddings of a basic block. On the left side, they are fed to the forward LSTM_f in order, while on the right side, they are fed to the backward LSTM_b in reversed order. The last hidden state of these two LSTMs $\mathbf{h}_1^b$ and $\mathbf{h}_T^f$ are spliced together as the final hidden state, which contains both the forward information and backward information of the basic block.

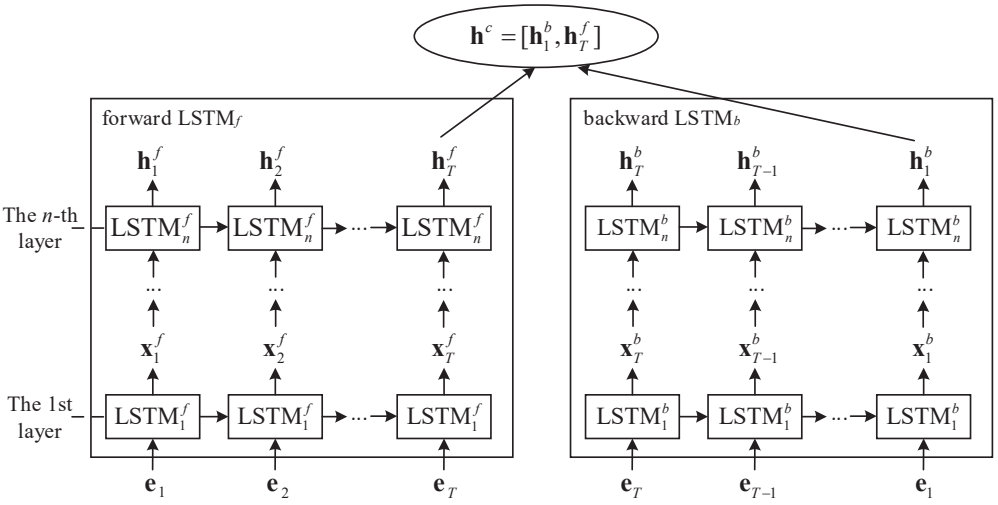


Figure 3. Structure of BiLSTM.

In the generation process of BB embeddings, each LSTM unit receives an input sequentially at each time step (for the first layer, the input is an instruction embedding), thus accumulating and transmitting more and more information. When the last instruction embedding is reached, the last LSTM unit at the last layer provides a semantic representation of the inputs. Specifically, assume that the input basic blocks B can be represented as a sequence of instruction embeddings $(\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_T)$. The specific process of basic block embedding generation is as follows. After basic block B is fed to each part of the BiLSTMs respectively, all the LSTM units update their hidden states in each time step.

Take the forward LSTM as example, in each time step, each LSTM unit processes an input vector, which is the input embedding or the result of the previous step. As is introduced in Section 3.2, each LSTM unit consists of four parts: the memory state c, the output-gate o, the input-gate i and the forget-

gate *f*. The input-gate controls the content to be written to the memory while the forget-gate controls the content to be removed from the memory. For example, the LSTM units in the first layer update their hidden states at time step *t* using 4.5 to 4.10.

$$\mathbf{i}_t = sigmoid(\mathbf{W}_i\mathbf{e}_t + \mathbf{U}_i\mathbf{x}_{t-1} + \mathbf{v}_i) \tag{4.5}$$

$$\mathbf{f}_t = sigmoid(\mathbf{W}_f\mathbf{e}_t + \mathbf{U}_f\mathbf{x}_{t-1} + \mathbf{v}_f) \tag{4.6}$$

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c\mathbf{e}_t + \mathbf{U}_c\mathbf{x}_{t-1} + \mathbf{v}_c) \tag{4.7}$$

$$\mathbf{c}_t = \mathbf{i}_t \odot \tilde{\mathbf{c}}_t + \mathbf{f}_t \odot \tilde{\mathbf{c}}_t \tag{4.8}$$

$$\mathbf{o}_t = sigmoid(\mathbf{W}_o\mathbf{e}_t + \mathbf{U}_o\mathbf{x}_{t-1} + \mathbf{v}_o) \tag{4.9}$$

$$\mathbf{x}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{4.10}$$

where $\odot$ is the Hadamard product (the product of elements), $\mathbf{W}_i$, $\mathbf{W}_f$, $\mathbf{W}_c$, $\mathbf{W}_o$, $\mathbf{U}_i$, $\mathbf{U}_f$, $\mathbf{U}_c$, $\mathbf{U}_o$ are weights matrixes and $\mathbf{v}_i$, $\mathbf{v}_f$, $\mathbf{v}_c$, $\mathbf{v}_o$ are bias vectors. These parameters are all learned during the training. In the last time step T, the last hidden state of the last layer provides a vector $\mathbf{h}_T^f$, which contains the forward information of the inputted basic block B.

For the backward LSTM, the process is almost same, except for the reverse order input (as shown in Figure 3). Similarly, the backward LSTM also provides a vector $\mathbf{h}_1^b$, which contains the backward information of the inputted basic block B. The final hidden state $\mathbf{h}^c$ is the splicing of $\mathbf{h}_1^b$ and $\mathbf{h}_T^f$. $\mathbf{h}^c$ is the embedding of the inputted basic block B.

For the training of the model, we use Siamese architecture [37] to automatically learn the parameters of the network and the BiLSTMs on both sides are completely same, as is shown in Figure 4. This Siamese network takes the instruction embedding sets of two basic blocks B_1 and B_2 as inputs and the output is the similarity of the input basic blocks. The goal of training is to make the embeddings generated by this network satisfy the following requirement: the embeddings of similar basic blocks are as close as possible to each other, while that of dissimilar blocks are as far as possible to each other.

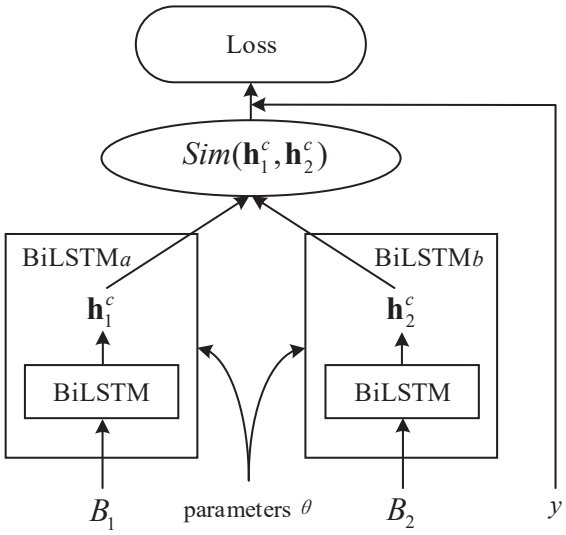


Figure 4. BiLSTM-based BB embedding model.

In the end, the Manhatton Distance ($\in [0, 1]$) is used to measure the similarity of basic blocks, as shown in 4.11.

$$Sim(B_1, B_2) = \exp(-\left\|\mathbf{h}_1^c - \mathbf{h}_2^c\right\|_1) \tag{4.11}$$

During the training, we use SGD to minimize the loss function in 4.12.

$$\min_{\mathbf{W}_i, \mathbf{W}_f, ..., \mathbf{V}_o} \sum_{i=1}^{N} (y_i - Sim(B_1^i, B_2^i))^2 \tag{4.12}$$

where y_i is the ground truth of the similarity of basic block pair $\left\langle B_1^i, B_2^i \right\rangle$ and N is the number of basic block pairs in the training set.

At last, once the AUC value converges, the training process can be terminated. The cross-platform basic block embedding model after training can encode the input binary basic block into an embedding, which contains the semantic information of the basic block.

4.4. ACFG embedding

After the basic block semantic extraction process in Section 4.3, the first part of semantics, semantics contained in the basic block is represented as an embedding vector. In this section, the second part of the semantics, the semantics of the CFG, is to be extracted. Before that, to combine the basic block semantics with the CFG semantics, basic block embeddings are attached to the CFG as vertices, generating an ACFG. Then, a graph embedding model customized from Structure2vec networks is proposed to encode the ACFG into a graph embedding to facilitate the similarity computing process.

As introduced in Section 3.3, basic Structure2vec network is designed for classification problems, each input graph needs a ground truth label to indicate which "class" it belongs to. Then the model is linked with a Softmax-layer, so that the entire model can be trained end-to-end by minimizing the cross-entropy loss. However, binary code similarity detection is not a classification problem, thus the Structure2vec network needs to be customized.

In the proposed method, the Structure2vec networks are embedded to a Siamese architecture, as shown in Figure 5. Both Structure2vec networks share the same set of parameters θ.

More specifically, the nonlinear mapping F in 3.8 is defined as 4.13.

$$F(x_v, \sum_{u \in N(v)} \mu_u) = (\mathbf{W}_1 x_v + \sigma(\sum_{u \in N(v)} \mu_u)) \tag{4.13}$$

where x_v is a d-dimension vector represents the feature of a vertex in ACFG (a basic block embedding), $\mathbf{W}_1$ is a $d \times p$ matrix, and p is the size of ACFG embeddings. To make the nonlinear transformation $\sigma(\cdot)$ more powerful, σ is defined as an n-layer full connected neural network shown in 4.14.

$$\sigma(l) = \underbrace{\mathbf{P}_1 \times \mathrm{ReLU}(\mathbf{P}_2 \times ...\mathrm{ReLU}(\mathbf{P}_n l))}_{n\text{layers}} \tag{4.14}$$

where $\mathbf{P}_i$ $(i = 1, 2, ..., n)$ is a $p \times p$ matrix, n is the depth of ACFG embeddings, ReLU is the rectified linear unit and $\mathrm{ReLU}(x) = \max\{0, 1\}$.

Each network of the Siamese respectively takes an ACFG g and g' as the input and outputs their graph embeddings $\phi(g)$ and $\phi(g')$. The Siamese network outputs the cosine distance between these two embeddings as the similarity of the corresponding ACFGs.

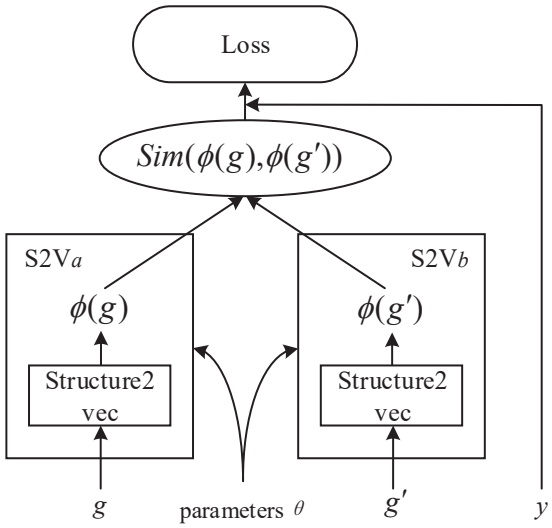


Figure 5. CNN-based ACFG embedding model.

Given a set of K pairs of ACFGs $\langle g_i, g_i' \rangle$ and the ground truth pairing information $y_i \in \{+1, -1\}$, where $y_i = +1$ indicates that g_i and g_i' are similar, or $y_i = -1$ otherwise. The output of Siamese network for each pair of ACFGs is defined as 4.15.

$$Sim(g, g') = cos(\phi(g), \phi(g')) = \frac{\langle \phi(g), \phi(g') \rangle}{||\phi(g)|| \cdot ||\phi(g')||} \tag{4.15}$$

where $\phi(g)$ is generated by Algorithm 1.

Algorithm 1: ACFG Embedding Generation

Input: ACFG $g =< V, E, \bar{x} >$
Initialize $\mu_v^{(0)} = \bar{0}$, for all $v \in V$
for $t = 1$ to T **do**
  for $v \in V$ **do**
    $l_v = \sum_{u \in N(v)} \mu_u^{(t-1)}$
    $\mu_v^{(t)} = \tanh(\mathbf{W}_1 x_v + \sigma(l_v))$
  end
end
return $\phi(g) := \mathbf{W}_2(\sum_{v \in V} \mu_v^{(T)})$

To train the parameters $\mathbf{W}_1$, $\mathbf{P}_1$, ..., $\mathbf{P}_n$, $\mathbf{W}_2$, we use SGD to minimize the loss function as 4.16.

$$\min_{\mathbf{W}_1, \mathbf{P}_1, ..., \mathbf{P}_n} \sum_{i=1}^{K} (Sim(g_i, g'_i) - y_i)^2 \tag{4.16}$$

Once the AUC of the network converges to a relatively stable value, the training is terminated. The network after training can transform ACFGs to graph embeddings that facilitate the similarity detections.

4.5. Implementation

Based on the proposed method, we implement a prototype system named SimInspector. SimInspector mainly consists of four modules, which are the preprocess module, the BB embedding model, the graph embedding model and the similarity computing module, as shown in Figure 6.

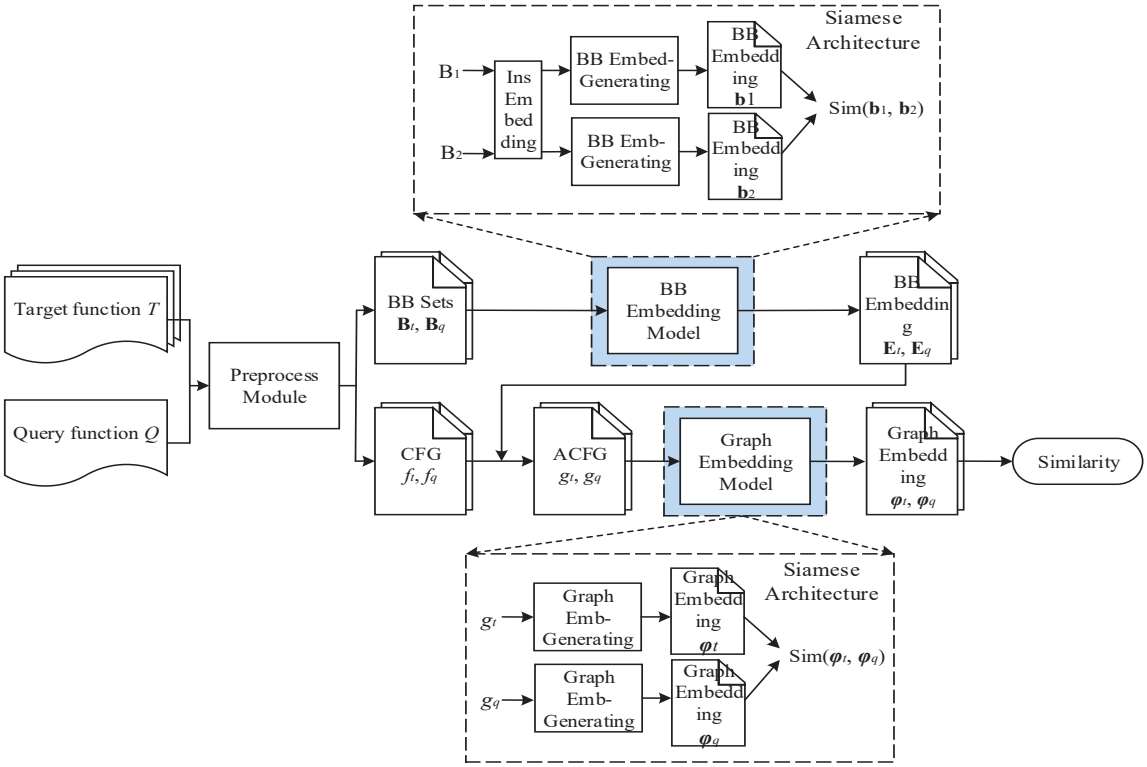


Figure 6. Overview of SimInspector. Siamese architecture is only used for the training of the models.

Firstly, SimInspector takes a pair of binary functions, the target function T and the query function Q, as inputs. Before extracting the semantics, a preprocess module is used to decompose function T and Q as basic block sets $\mathbf{B}_t$, $\mathbf{B}_q$ and CFG f_t, f_q. After that, the basic blocks in $\mathbf{B}_t$ and $\mathbf{B}_q$ are encoded into embeddings by the BB embedding model and their BB embeddings are respectively represented as $\mathbf{E}_t$ and $\mathbf{E}_q$, which are combined with f_t and f_q as vertex attributes and generate ACFG g_t and g_q. Then, g_t and g_q are encoded to embeddings φ_t and φ_q by the graph embedding model. At last, the similarity computing module computes the cosine distance of these two embeddings, $Sim(\varphi_t, \varphi_q)$, as the similarity of function T and Q.

5. Evaluation

In this section, we evaluate the performance of SimInspector. First, the experimental setups and the datasets are introduced in Section 5.1 and 5.2. Next, the accuracy of the BB embedding model and the ACFG embedding model are evaluated in Section 5.3. Then, the impact of some hyperparameters to these two models are tested in 5.4. Besides, the efficiency of the BB embedding model and the ACFG embedding model are evaluated in Section 5.5. At last, embeddings of some binary functions

are visualized to give an intuitive understanding of embeddings in Section 5.6.

5.1. Experimental setups

The proposed method is implemented in Python using the Keras [38] platform with TensorFlow [39] as backend. The experiments were performed on a server running the Ubuntu 18.04 operating system with a 64-bit 2.20 GHz Intel Xeon Silver 4114 CPU and 128GB RAM.

5.2. Datasets and training

The datasets used to evaluate the proposed method consist of two parts: Dataset I (as shown in Table 1) for evaluating the basic block embedding model and Dataset II (as shown in Table 2) for evaluating the ACFG embedding model.

Table 1. Basic block pairs in the training, validation and testing datasets.

	Training			Validation			Testing			Total		
	Sim	Dissim	Total	Sim	Dissim	Total	Sim	Dissim	Total	Sim	Dissim	Total
O1	35,416	35,223	70,639	3,902	3,946	7,848	4,368	4,354	8,722	43,686	43,523	87,209
O2	45,461	45,278	90,739	5,013	5,069	10,082	5,608	5,590	11,198	56,082	55,937	112,019
O3	48,613	48,472	97,085	5,390	5,397	10,787	6,000	5,988	11,988	60,003	59,857	119,860
Cross-opts	34,118	33,920	68,038	3,809	3,750	7,559	4,554	4,404	8,958	42,481	42,074	84,555
Total	163,608	162,893	326,501	18,114	18,162	36,276	20,530	20,336	40,866	202,252	201,391	403,643

Table 2. ACFGs in the training, validation and testing datasets.

	Training			Validation			Testing			Total		
	x86	ARM	Total	x86	ARM	Total	x86	ARM	Total	x86	ARM	Total
O0	12,906	12,993	25,899	1,613	1,624	3,237	1,614	1,625	3,239	16,133	16,242	32,375
O1	10,764	10,943	21,707	1,345	1,367	2,712	1,346	1,369	2,715	13,455	13,679	27,134
O2	10,708	10,821	21,529	1,338	1,352	2,690	1,340	1,354	2,694	13,386	13,527	26,913
O3	10,354	10,553	20,907	1,294	1,319	2,613	1,295	1,320	2,615	12,943	13,192	26,135
Total	44,732	45,310	90,042	5,590	5,662	11,252	5,595	5,668	11,263	55,917	56,640	112,557

Dataset I This dataset is used for the training, validation and testing of the basic block embedding model. It is provided by the authors of InnerEye-BB [36] (referred to as Dataset I in their paper) and includes 403,643 basic block pairs in total. Details are shown in Table 1.

Dataset II This dataset is used for the training, validation and testing of the ACFG embedding model. It consists of binaries compiled from source code and two binary functions are considered to be similar if they are compiled from the same source code function, or dissimilar otherwise. Specifically, we compiled several versions of OpenSSL (1.0.2q and 1.1.1-pre1) to different architectures (x86-64 and ARM) with different optimization operations (O0-O3) using GCC (v4.8.4 for x86 and v4.7.3 for ARM). Besides, we developed an ACFG extraction tool and implemented it in Python based on Angr [40]. This tool can extract ACFGs from given binary program with basic block embeddings as the attributions of ACFG vertices. Based on this, 112,557 ACFGs are generated. All the ACFGs are split into three disjoint subsets of functions for training, validation and testing respectively. During the split, it can be

guaranteed that no two binary functions compiled from the same source function are separated into two different sets, examining whether the pre-trained model can generalize to unseen functions. Details are shown in Table 2.

ACFG pairs are constructed when training the ACFG embedding model as follows: at each epoch, for each ACFG g in the training set, we randomly select one ACFG g_1 from the set of all ACFGs compiled from the same source function as g but at another optimization level, and one ACFG g_2 from the set of all other ACFGs in the training set at another optimization level. Thus two training samples are generated: $\langle g, g_1 \rangle$ with ground truth label +1 and $\langle g, g_2 \rangle$ with label -1. Since g_1 and g_2 are randomly selected for each g independently at each epoch, the training data often vary at different epochs. Besides, before fed to the training model, the data in each epoch is randomly shuffled to further improve the variety.

We train the models for 100 epochs respectively and measure the loss and AUC on the corresponding validation sets after each epoch. During the 100 epochs, we save the models achieving the best AUC on the validation sets.

5.3. Accuracy

In this section, we evaluate the accuracy of the basic block embedding model and ACFG embedding model.

5.3.1. Accuracy of basic block embedding model

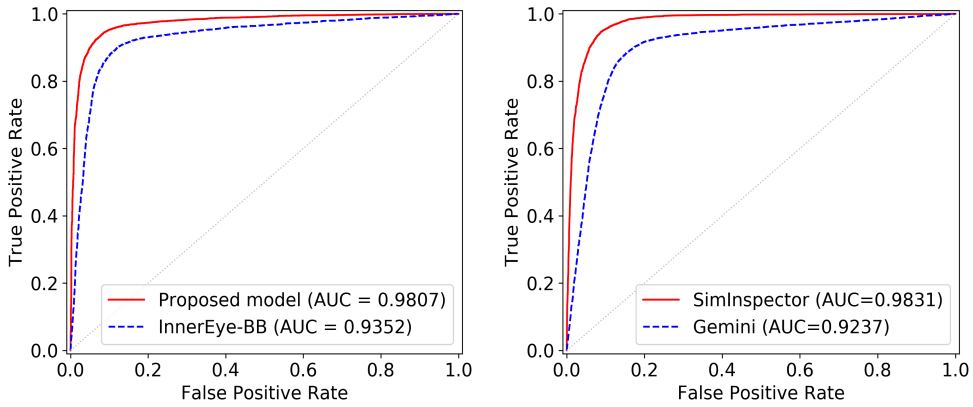
We evaluate the accuracy of the basic block embedding model using the testing dataset shown in Table 1. For comparison, InnerEye-BB [36] is also tested on the same dataset as a baseline approach. The ROC curves of the proposed model and InnerEye-BB is shown in Figure 7(a). The red line in the figure is ROC of the proposed model and the blue line is that of the baseline. We can observe that the ROC of the proposed model is closer to the left-hand border and top border and has a higher AUC, which means our model has a better accuracy.

5.3.2. Accuracy of SimInspector

To evaluate the accuracy of SimInspector, we construct ACFG pairs in testing dataset using the same method as constructing the training ACFG pairs. After that, SimInspector is evaluated on these ACFG pairs. For comparison, we take Gemini [12] as the baseline approach and evaluated its accuracy on the same testing dataset. The ROC curves of these two approaches are shown in Figure 7(b). The red line in the figure is ROC of the proposed method and the blue line is that of the baseline. We can see that the ROC of SimInspector is closer to the left-hand border and top border and has a higher AUC, which means our method has a better accuracy.

5.4. Hyperparameters

In this section, the impact of different hyperparameters to the models are evaluated, including the number of training epochs, embedding dimensions, and embedding depths.



(a) Results of BB embedding.

(b) Results of ACFG embedding.

Figure 7. ROC curves of the proposed models and their baselines. Corresponding AUC values are presented in the legend.

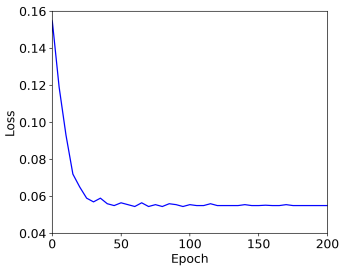
5.4.1. Hyperparameters of basic block embedding model

Instruction embedding dimension We first evaluate the the impact of different instruction embedding dimensions to the model. We set the instruction embedding dimension as 50, 100 and 150, and their corresponding AUC values are 96.36%, 97.25% and 97.31%. We can see from the results that increasing the embedding dimension yields higher performance but the AUC values corresponding to the embedding dimension higher than 100 are close to each other. Because higher dimensions lead to higher computational costs, a dimension of 100 is a good trade-off between effectiveness and efficiency.

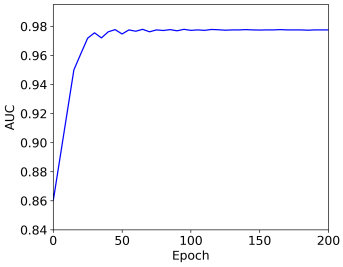
Training epochs We train the model for 200 epochs and evaluate the model over the validation set every 5 epochs for the loss and the AUC. The results are shown in Figure 8(a) and 8(b). It can be seen from the figure that the loss value decreases quickly and almost stays stable after 25 epochs; and the AUC value steadily increases and is stabilize at the end of epoch 25. Therefore, it can be concluded that the model can be quickly trained to achieve reasonably good performance.

Basic block embedding dimension We vary the basic block embedding dimensions and evaluate the corresponding AUC values. Results are shown in Figure 8(c). It can be seen that the AUC value increases with the increase of embedding dimension. But the AUC value corresponding to the embedding dimension higher than 32 are close to each other. Since a higher embedding dimension leads to higher computational costs (requires longer training time and evaluation time), we conclude that a moderate dimension of 32 is a good trade-off between performance and efficiency.

Embedding depth We change the number of layers of each BiLSTM and evaluate the corresponding AUC values. Results are shown in Figure 8(d). It can be seen that the BiLSTM with 2 and 3 layers outperform the network with a single layer, and the AUC values for the network with 2 and 3 layers are close to each other. Since adding more layers increases the computational costs and does no help much on the accuracy, we choose the embedding depth as 2.



(a) Loss vs. no. of epochs.



(b) AUC vs. no. of epochs.

Dimemsion	AUC value (%)
8	96.16
16	97.03
32	97.79
64	97.87
128	97.92

(c) AUC vs. emb-dims

Depth	AUC value (%)
1	95.35
2	97.82
3	97.96

(d) AUC vs. emb-depths

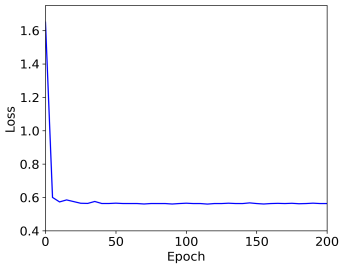
Figure 8. Stability of the BB embedding model with respect to different hyperparameters. Figure 8(a) and 8(b) are evaluated on the validation dataset of Dataset I, and others are evaluated on the testing dataset of Dataset I.

5.4.2. Hyperparameters of ACFG embedding model

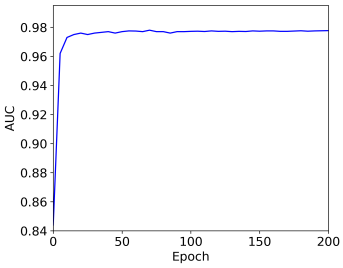
Training Epochs We train the ACFG embedding model for 200 epochs and evaluate the model every 5 epochs for the loss and AUC. Results are shown in Figure 9(a) and 9(b). We observe that the loss drops to a low value after 10 training epochs and then almost remains the same. Similarly, the AUC increases to a high value after 10 training epochs and remains almost the same. Therefore, we conclude that the model can be quickly trained to achieve reasonably good performance (after 10 epochs).

Embedding dimension We vary the dimensions of the ACFG embeddings and evaluate the corresponding AUC values. Results are shown in Figure 9(c). We can see from the results that higher embedding dimension leads to higher AUC values. But after the dimension achieved 64, the corresponding AUC values are close to each other. Therefore, taking efficiency into account, 64 is a proper dimension for the ACFG embeddings.

Embedding depth Similarly, we change the number of network layers in ACFG embedding model to evaluate the effect of embedding depth. It can be seen from Figure 9(d) that the model with 2 layers and 3 layers have extremely close AUC values and both outperform that of the model with only single layer. Thus 2 is a proper embedding depth for the ACFG embedding.



(a) Loss vs. no. of epochs.



(b) AUC vs. no. of epochs.

Dimemsion	AUC value (%)
8	96.54
16	97.37
32	97.82
64	98.11
128	98.19
256	98.21

(c) AUC vs. emb-dims

Depth	AUC value (%)
1	96.75
2	98.22
3	98.23

(d) AUC vs. emb-depths

Figure 9. Stability of the ACFG embedding model with respect to different hyperparameters. Figure 9(a) and 9(b) are evaluated on the validation dataset of Dataset II, and others are evaluated on the testing dataset of Dataset II.

5.5. *Efficiency*

In this section we evaluate the efficiency of SimInspector, including the training time of instruction embedding model, BB embedding model, and ACFG embedding model; the time of BB embedding generation, ACFG extraction, and ACFG embedding generation.

5.5.1. Training time

Instruction embedding model The training time of the instruction embedding model is linear to the number of epochs and the corpus size. We use Dataset I, containing 6,199,651 instructions in total, as the corpus to train the instruction embedding model. The corpus contains 49,760 distinct instructions which form a vocabulary. Earlier we have shown the proper embedding dimension for the instruction embedding model to achieve a pretty good performance is 100. Therefore, we use 10^{-5} as down sampling rate and set the parameter mini-word-count as zero, the instruction embedding dimension as 100, and trained the model for 100 epochs. The training time is about 70.26 seconds. Therefore, the instruction embedding model can be trained in a very short period of time.

Basic block embedding model Based on the hyperparameter evaluations of the BB embedding model in Section 5.4, we set the BB embedding dimension as 32, the number of network layers as 2, and trained the model with 25 epochs. It takes about 1 hour ($153.26 \times 25 = 3831.50$ seconds) to train the model to achieve a good performance.

ACFG embedding model Based on the hyperparameter evaluations of the ACFG embedding model in Section 5.4, we set the ACFG embedding dimension as 64, the number of network layers as 2, and trained the model with 10 epochs. It requires about 2 hours and 20 minutes ($837.29 \times 10 = 8372.9$ seconds) to train the model to achieve a good performance.

5.5.2. Testing time

Basic block embedding generation We use the BB embedding model which has 2 network layers to transform the basic blocks in the testing dataset of Dataset I into 32-dimension embeddings and evaluate the embedding generation time. This dataset contains 43,710 blocks for x86 and 39,353 blocks for ARM (83,063 blocks in total) and the whole generation process takes about 3905.96 seconds, which means that it requires about 47.02 ms in average to generate the embedding for a basic block.

ACFG extraction We use the ACFG extraction tool introduced in Section 5.2 to extract ACFGs from binary functions in the testing dataset of Dataset II which contains 5,595 x86 functions and 5,668 ARM functions (11,263 functions in total). The attribute for the vertices of the ACFGs is the basic block embeddings generated by the BB embedding model, and the dimension of these embeddings is 32. The extraction process takes about 31087.59 seconds, which means that it requires about 2.76 seconds in average to extract the ACFG of a binary function. Notice that this time contains both the BB embedding generation time and the ACFG extraction time.

ACFG embedding generation We use the ACFG embedding model which has 2 network layers to transform the ACFGs in the testing dataset of Dataset II into 64-dimension embeddings and evaluate

the embedding generation time. It takes about 314.58 seconds to generate the embeddings of all the 11,263 ACFGs, which means that generating an embedding for single ACFG requires about 27.93 ms in average.

5.6. Visualization of embeddings

In this section, we visualize the embeddings of ACFGs to understand the effectiveness of SimInspector. We randomly select 5 source functions from OpenSSL version 1.1.1-pre1 (including "cms_main", "do_passwd", "speed_main", "verify_chain", and "SM4_encrypt") and generate the embeddings of the corresponding binary functions compiled with different architectures (x86 and ARM) and optimization levels (O0 to O3). After that, we use a tool named t-SNE [41] to project the high-dimension embeddings to 2-D plane. Then we plot the points in Figure 10 and different source functions are presented in different colors. It can be seen from the figure that the embeddings of binary functions compiled from the same source function are close to each other, while that of binary functions compiled from different source functions are far from each other. Therefore, this visualization illustrates that SimInspector can preserve the information of the source functions into embeddings regardless of the target architectures and the optimization levels.

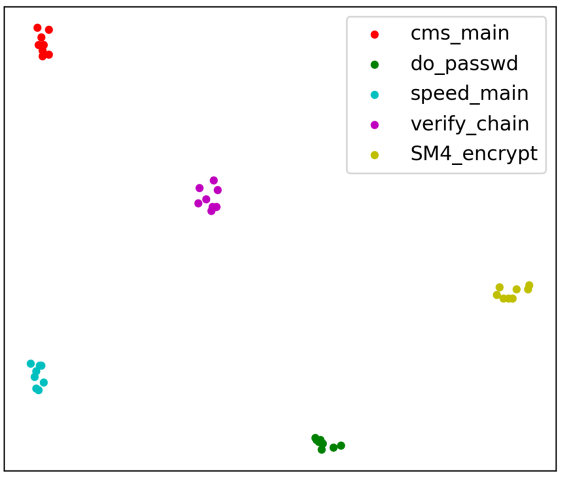


Figure 10. Visualization of the embeddings of different functions using t-SNE. Each color indicates one source function.

6. Conclusion

In this manuscript, we present a cross-platform binary code similarity detection method based on NMT and graph embedding. We implement a prototype called SimInspector and our extensive evaluation shows that SimInspector further improves the similarity detection accuracy compared with the state-of-the-art approach and has a pretty good efficiency as well. Our research shows that deep learning can be well used to accomplish binary analysis tasks.

Conflict of interest

All authors declare no conflicts of interest in this paper.

References

1. J. Pewny, B. Garmany, R. Gawlik, C. Rossow, T. Holz, Cross-architecture bug search in binary executables, in *2015 IEEE Symposium on Security and Privacy*, (2015), 709–724.
2. B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, et al., αdiff: cross-version binary code similarity detection with DNN, in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, (2018), 667–678.
3. L. Luo, J. Ming, D. Wu, P. Liu, S. Zhu, Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection, in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, (2014), 389–400.
4. L. Luo, J. Ming, D. Wu, P. Liu, S. Zhu, Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection, *IEEE Trans. Software Eng.*, **43** (2017), 1157–1177.
5. A. Sæbjørnsen, J. Willcock, T. Panas, D. J. Quinlan, Z. Su, Detecting code clones in binary executables, in *Proceedings of the 18th International Symposium on Software Testing and Analysis*, (2009), 117–128.
6. Z. Xu, B. Chen, M. Chandramohan, Y. Liu, F. Song, SPAIN: security patch analysis for binaries towards understanding the pain and pills, in *Proceedings of the 39th International Conference on Software Engineering*, (2017), 462–472.
7. U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, E. Kirda, Scalable, behavior-based malware clustering, in *Proceedings of the Network and Distributed System Security Symposium*, (2009).
8. X. Hu, T. Chiueh, K. G. Shin, Large-scale malware indexing using function-call graphs, in *Proceedings of the 2009 ACM Conference on Computer and Communications Security*, (2009), 611–620.
9. J. Jang, M. Woo, D. Brumley, Towards automatic software lineage inference, in *Proceedings of the 22th USENIX Security Symposium*, (2013), 81–96.
10. S. Eschweiler, K. Yakdan, E. Gerhards-Padilla, discovre: Efficient cross-architecture identification of bugs in binary code, in *23rd Annual Network and Distributed System Security Symposium*, (2016).
11. Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, H. Yin, Scalable graph-based bug search for firmware images, in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, (2016), 480—491.
12. X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, D. Song, 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*, (2017), 363—376.

13. Y. David, N. Partush, E. Yahav, Statistical similarity of binaries, in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, (2016), 266—280.

14. Y. David, N. Partush, E. Yahav, Similarity of binaries through re-optimization, in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, (2017), 79—94.

15. Y. David, N. Partush, E. Yahav, Firmup: Precise static detection of common vulnerabilities in firmware, in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, (2018), 392—404.

16. Y. David, E. Yahav, Tracelet-based code search in executables, in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, (2014), 349—360.

17. B. Lu, F. Liu, X. Ge, B. Liu, X. Luo, A software birthmark based on dynamic opcode n-gram, in *Proceedings of the 1st IEEE International Conference on Semantic Computing*, (2007), 37–44.

18. W. M. Khoo, A. Mycroft, R. J. Anderson, Rendezvous: A search engine for binary code, in *Proceedings of the 10th Working Conference on Mining Software Repositories*, (2013), 329–338.

19. D. Gao, M. K. Reiter, D. X. Song, Binhunt: Automatically finding semantic differences in binary programs, in *10th International Conference on Information and Communications Security*, (2008), 238–255.

20. J. Ming, M. Pan, D. Gao, ibinhunt: Binary hunting with inter-procedural control flow, in *15th International Conference on Information Security and Cryptology*, (2012), 92–109.

21. J. Pewny, F. Schuster, L. Bernhard, T. Holz, C. Rossow, Leveraging semantic signatures for bug search in binary programs, in *Proceedings of the 30th Annual Computer Security Applications Conference*, (2014), 406—415.

22. M. Bourquin, A. King and E. Robbins, Binslayer: accurate comparison of binary executables, in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, (2013), 4:1–4:10.

23. T. Dullien, R. Rolles, Graph-based comparison of executable objects (english version), in *SSTIC*, **5** (2005).

24. H. Flake, Structural comparison of executable objects, in *Proceddings of the 2004 SIDAR Workshop on Detection of Intrusions and Malware & Vulnerability Assessment*, (2004), 161–173.

25. J. Devlin, M. Chang, K. Lee, K. Toutanova, BERT: pre-training of deep bidirectional transformers for language understanding, in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, (2019), 4171–4186.

26. P. Liu, X. Qiu, X. Huang, Recurrent neural network for text classification with multi-task learning, in *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, (2016), 2873–2879.

27. F. Schroff, D. Kalenichenko, J. Philbin, Facenet: A unified embedding for face recognition and clustering, in *IEEE Conference on Computer Vision and Pattern Recognition*, (2015), 815–823.

28. H. O. Song, Y. Xiang, S. Jegelka, S. Savarese, Deep metric learning via lifted structured feature embedding, in *2016 IEEE Conference on Computer Vision and Pattern Recognition*, (2016), 4004–4012.
29. C. Fellbaum, *WordNet -– An Electronical Lexical Database*, (1998).
30. T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, in *Advances in Neural Information Processing Systems*, (2013), 3111–3119.
31. T. Mikolov, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, in *Proceedings of the 1st International Conference on Learning Representations*, (2013).
32. J. Pennington, R. Socher, C. D. Manning, Glove: Global vectors for word representation, in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, (2014), 1532–1543.
33. S. Hochreiter, J. Schmidhuber, Long short-term memory, *Neural Comput.*, **9** (1997), 1735—1780.
34. A. Graves, Supervised Sequence Labelling with Recurrent Neural Networks, *Stud. Comput. Intell.* **385** (2012).
35. H. Dai, B. Dai, L. Song, Discriminative embeddings of latent variable models for structured data, in *Proceedings of the 33rd International Conference on Machine Learning*, (2016), 2702–2711.
36. F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, Z. Zhang, Neural machine translation inspired binary code similarity comparison beyond function pairs, in *26th Annual Network and Distributed System Security Symposium*, (2019).
37. J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, R. Shah, Signature verification using a siamese time delay neural network, in *Advances in Neural Information Processing Systems*, (1993), 737–744.
38. F. Chollet, Keras: The Python Deep Learning library, (2018).
39. M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, et al., Tensorflow: A system for large-scale machine learning, in *12th USENIX Symposium on Operating Systems Design and Implementation*, (2016), 265–283.
40. F. Wang, Y. Shoshitaishvili, Angr - the next generation of binary analysis, in *IEEE Cybersecurity Development*, (2017), 8–9.
41. L. van der Maaten, G. Hinton, Visualizing data using t-sne, *J. Mach. Learn. Res.*, **9** (2008), 2579–2605.

AIMS Press