Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2023.046595

ARTICLE

Binary Program Vulnerability Mining Based on Neural Network

Zhenhui Li[1], **Shuangping Xing**[1], **Lin Yu**[1], **Huiping Li**[1], **Fan Zhou**[1], **Guangqiang Yin**[1], **Xikai Tang**[2] **and Zhiguo Wang**[1,*]

[1]School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu, 611731, China

[2]School of Electrical and Computer Engineering, University of Waterloo, Waterloo, N2L 3G1, Canada

*Corresponding Author: Zhiguo Wang. Email: zgwang@uestc.edu.cn

Received: 08 October 2023 Accepted: 15 December 2023 Published: 27 February 2024

ABSTRACT

Software security analysts typically only have access to the executable program and cannot directly access the source code of the program. This poses significant challenges to security analysis. While it is crucial to identify vulnerabilities in such non-source code programs, there exists a limited set of generalized tools due to the low versatility of current vulnerability mining methods. However, these tools suffer from some shortcomings. In terms of targeted fuzzing, the path searching for target points is not streamlined enough, and the completely random testing leads to an excessively large search space. Additionally, when it comes to code similarity analysis, there are issues with incomplete code feature extraction, which may result in information loss. In this paper, we propose a cross-platform and cross-architecture approach to exploit vulnerabilities using neural network obfuscation techniques. By leveraging the Angr framework, a deobfuscation technique is introduced, along with the adoption of a VEX-IR-based intermediate language conversion method. This combination allows for the unified handling of binary programs across various architectures, compilers, and compilation options. Subsequently, binary programs are processed to extract multi-level spatial features using a combination of a skip-gram model with self-attention mechanism and a bidirectional Long Short-Term Memory (LSTM) network. Finally, the graph embedding network is utilized to evaluate the similarity of program functionalities. Based on these similarity scores, a target function is determined, and symbolic execution is applied to solve the target function. The solved content serves as the initial seed for targeted fuzzing. The binary program is processed by using the de-obfuscation technique and intermediate language transformation method, and then the similarity of program functions is evaluated by using a graph embedding network, and symbolic execution is performed based on these similarity scores. This approach facilitates cross-architecture analysis of executable programs without their source codes and concurrently reduces the risk of symbolic execution path explosion.

KEYWORDS

Vulnerability mining; de-obfuscation; neural network; graph embedding network; symbolic execution

1 Introduction

In light of the growing ubiquity of smart devices, data, and device securities have attracted increasing attention in recent years [1–5]. Imperceptible vulnerabilities emerge as code is created, and undetected vulnerabilities are a major hazard [6–8]. It is relatively difficult to analyze binary programs directly with today's technology. There are two main reasons: one is the huge number of instructions the logic is complex, and the other is the lack of corresponding semantic and type information of binary code [9]. Programs from various sources may involve different source languages, different compilation optimization options, and different architecture issues, which make the security analysis work very difficult. The American fuzzy lop (AFL) represents a significant advancement in fuzzing, but enhancing the coverage and accuracy of fuzzing remains a critical challenge. The preferred way to improve the coverage is symbolic execution [10], but it is vulnerable to path explosion. While the Angr framework [11] offers commendable symbolic execution capabilities, it does not effectively counteract the path explosion risk, especially with many invalid paths. In consideration of the problem of low discovering efficiency in current vulnerability fuzzing discovery methods, Zhang and Xi intended to explore the binary program vulnerability feature extraction and vulnerability discovery guiding strategy generation methods [12].

The lack of explicit semantic and type information in binary code increases the likelihood of security vulnerabilities. Hackers can exploit these ambiguities to launch sophisticated attacks that compromise system integrity and user data. Binary vulnerability analysis has no security impact on the source program and does not obtain user privacy information, so it is an important means to analyze the vulnerability of binary programs. However, the current binary vulnerability analysis tools are mostly for a specific type of vulnerability mining analysis, it is difficult to reuse these technologies for other types of vulnerabilities and other architectures. Solving the path explosion problem of symbolic execution, improving the efficiency and performance of fuzzing, and increasing its cross-architecture applicability are of great importance to the development of binary vulnerability analysis technology, and of great significance to the practical application security of science and technology. To solve the above problems, Han et al. proposed a binary vulnerability mining technology based on neural network feature fusion [13].

There are various binary mining techniques to mine vulnerabilities [14,15]. For example, automated testing based on object code search, blot analysis, and fuzzing are the main mining technologies, among which fuzzy technology is becoming the mainstream technology [16].

We propose a method that bridges the strengths of symbolic execution and fuzzing, introducing a function similarity estimation to avoid path explosion. Fuzzing, by processing randomly generated data (which could manifest as long strings, random negative numbers, floating numbers, large numbers, special strings, etc.), simulates program operations and monitors its real-time performance. Certain anomalies, such as assertion failures or program crashes, may flag vulnerabilities, e.g., memory leaks or buffer overflows [17]. Currently, most research is focused on improving the fuzzing engine. Little attention is given to the test cases, however, their main focus is on the fuzzing seed mutation algorithm [18]. Therefore, the research on the generation of the initial seed was still under-explored.

We believe that enhancing the seed generation process could greatly benefit fuzzing. Our proposed system leverages neural networks to analyze the similarities of functions. Function similarity analysis, which compares vulnerabilities in functions to pinpoint potentially high-risk ones, seeks to execute symbolic execution on these select functions, thereby avoiding path explosion [19]. We carry out symbolic execution based on this similarity score and convert the solution to an initial fuzzing seed to facilitate directional fuzzing. This system enables cross-architecture analysis of binary programs without their source code while mitigating the risk of path explosion. Our primary contributions are summaries as follows:

- A novel graph embedding technique for code similarity analysis. We first present a basic block embedding method to capture context information about the fundamental function of the block. This information is then mapped to a block-based embedding vector and encoded by a structure2vec network. After several iterations, a fusion of the vertex features of basic blocks and the program function characteristics is achieved. By computing the cosine distance between these vectors, functions can be effectively compared and assessed. Therefore, potential vulnerabilities in the program can be revealed.
- A seed generation technique is proposed based on function similarity detection. According to the vulnerability information Angr's symbolic execution strategy is used to solve the target function, and the solution is used as the initialization seed to perform fuzzing. This method not only addresses the path explosion issue but also satisfies fuzzing's high coverage requirements for test cases.

The rest of this paper is structured as follows. Section 2 presents related work on vulnerability analysis. In Section 3, we give the implementation method and main means. Section 4 shows the various experiments done by our method on the dataset, giving the real experimental steps and results. Finally, Section 5 gives the conclusions and our future work.

2 Related Work/Literature Review

2.1 Vulnerability Mining Technology

Binary vulnerability analysis techniques encompass methods such as target code search-based automated testing, taint analysis [20,21], fuzzing [22], and symbolic execution [23]. As computational capabilities expand, researchers are exploring alternative strategies for vulnerability detection, with symbolic execution emerging as particularly efficient.

Symbolic execution emphasizes monitoring the program control flow, boasting superior code coverage [24]. Concurrently, fuzzing remains a prominent technique in vulnerability detection. By injecting randomized or anomalous inputs, it scrutinizes anomalous behaviors, thereby exposing potential vulnerabilities and software attack surfaces. Taint analysis, on the other hand, tracks and evaluates information flow, ensuring that strategies governing this flow are properly executed by tracing data paths marked as tainted [25].

Whether fuzzing tests or symbolic execution, they are often used for some specific types of vulnerabilities, but it is a trend to combine excellent and mature vulnerability mining and analysis technology to form a general analysis tool.

2.2 Word2vec

Word2vec is widely utilized in natural language processing and offers capabilities like text summarization, sentiment analysis, and entity recognition [26,27]. Its strengths lie in autonomously discerning word meanings without necessitating human intervention. Characteristics such as autonomy, flexibility, and potent expressiveness further underscore its utility. It finds application in instruction embedding tasks, generating instructional embedding vectors autonomously.

Two primary learning modes characterize Word2vec: The continuous bag-of-words (CBOW) model, which learns from context to a central point [28], and the skip-gram model, proposed by Mikolov, which learns from a central point to its surrounding context [29]. The skip-gram model stands out for its superior coding efficiency and adeptness at capturing contextual semantics. In contrast, CBOW, while differing significantly from skip-gram, monitors internal vocabulary changes and analyzes external vocabulary to get accurate results. Notably, the skip-gram model is particularly suited for translating basic block instructions into vectors suitable for program processing [30]. No matter what kind of learning mode, the current Word2vec technology has been well applied in natural language processing to help complete code similarity analysis.

2.3 Code Similarity Analysis

Traditional binary code similarity analysis typically focuses on feature extraction from codes, followed by comparisons with known vulnerability codes [31–35]. Given that rules are formulated manually, rule-based matching struggles with intricate binary code similarity challenges. With the fast development of deep learning [36,37], utilizing deep learning methods to compute code similarity has attracted extensive attention from researchers [38]. For example, in 2017, David et al. [39] introduced GitZ, underpinned by semantic matching accuracy. GitZ splits the complex binary code into several independent parts and converts them into standards-compliant lattices. Following this, it employs simple rules for similar code retrieval.

Integrating deep learning into code similarity assessment can improve retrieval precision. In contrast to conventional approaches, neural networks considerably reduce the dependence on manual feature selection and have achieved state-of-the-performance in many anomaly detection and vulnerability detection tasks [40–42]. For instance, Genius [43] leverages a graph embedding network for code similarity analysis, which in turn, employs a structure2vec network [44] for embedding vector computation, with subsequent processes handled by a siamese network [45]. These works can do a good job in code similarity analysis, and using deep learning for program analysis is a very promising field.

3 Method

We now introduce the details of our proposed method. First, we load the program intended for de-obfuscation, excising the control flow fattening to restore the program's real control flow. Following this, we undertake control flow graph (CFG) structure optimization to retrieve the valid control flow. In the subsequent step, we transcribe the binary code into VEX-IR code. This serves to neutralize disparities across instruction sets while preserving the program's intent and functionality.

We introduce a similarity assessment technique tailored for the pre-processed data. We first use a basic block embedding method to extract comprehensive context information. This extracted data is subsequently transformed into a block-centric embedding vector and gets encoded through a structure2vec network. After several iterations, the fusion of the vertex features of the basic block and the program function features is completed. By computing the cosine distance between these vectors, we can efficiently compare and analyze functions, thereby revealing potential vulnerabilities in the program.

At last, according to the vulnerability information, Angr's symbolic execution strategy is used to solve the target function, and the solution is used as the initialization seed to perform fuzzing. This method solves the issue of path explosion, and it also meets the high coverage requirements of fuzzing for the test cases. The overall framework of the proposed method is depicted in Fig. 1.

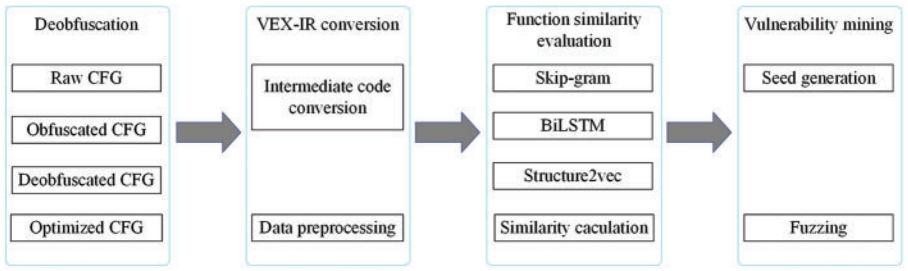


Figure 1: The framework of the proposed method

3.1 Preprocessing

3.1.1 De-Obfuscation and CFG Structure Optimization

The binary code is often obfuscated to prevent it from being reversely analyzed. When analyzing the obfuscated program, there are many interference parts in the control flow graph, which will affect the analysis results and efficiency. Taking the Ollvm obfuscation as an example, its obfuscation methods include a false block, a control flow expansion, an instruction expansion, and a basic block separation. De-obfuscation can remove control flow and redundant blocks, and restore the real CFG of the code. The specific algorithm principle of anti-obfuscation for Ollvm is as follows:

1. Using simulated execution and fuzz technology, fake blocks are found and marked.

2. Culling control flow blocks using the basic block signature.

3. Marking the remaining blocks as real blocks, and finding corresponding relationships by simulating the function execution.

4. According to the correspondence, reconstructing the CFG.

According to the principle of anti-obfuscation, we propose a universal de-obfuscation method that is not limited to the low level virtual machine (LLVM) compilation architecture. The approximate process is summarized as follows:

1. Utilizing Angr's solution engine to explore the branch path of the program, setting comparison conditions, matching features of the real branch, and obtaining the address of the branch by changing the symbol value of the input branch.

2. Once the call command is received, it should be returned immediately by the hook.

3. Patching all redundant blocks as no operation (NOP) instructions.

4. If no branch is found, replace the last instruction with a jump instruction, and jump to the next step immediately.

5. By adjusting the instructions to meet the jump requirements, a new branch can be generated from the original real block. Then we can easily make it jump to the branch that meets the requirements.

In addition, the CFG structure of the de-obfuscated code should be optimized to restore the coherence of the code making the subsequent comparison and code analysis performed better. The optimization methods have the following steps:

1. Basic block integration. Traverse all basic blocks in the function f, filter out those basic blocks b with only one exit, and find their successor basic blocks b'. Then we treat these two basic blocks as the same basic block, and merge b and b'. In addition, if the end of b is a jump instruction, we delete these two basic blocks together.

2. Improving embedded calls. Suppose cb is the called function, c is the calling function, |c| represents the length of the calling function, and |cb| represents the length of the called function. When the called function meets the following conditions, cb can be embedded and called. First, instructions in cb only belong to its function realization; second, the ratio of |cb| to |c| is less than 0.75. The second condition is to prevent the semantic characteristics of the called function from overriding the semantic features of the calling function.

3.1.2 VEX-IR Intermediate Code Conversion

Different compilation platforms represent different forms of the instruction sets, the binary code of different platforms cannot be analyzed in the same way. To achieve cross-architecture analysis, we argue that it is necessary to convert the de-obfuscated and CFG-optimized binary code into intermediate code. Converting binary codes under different architectures into an intermediate language form can eliminate 5 the differences in the instruction sets and keep the meaning and function of the program unchanged. We convert the binary code into VEX-IR code through pyvex of the Angr framework, and then use the Angr framework to extract the VEX-IR code and optimize it accordingly.

The VEX-IR converted code cannot be directly used for basic block embedding, as it contains many simple operators, such as =, (), and so on. It contains many low-frequency words such as numerical constants, memory addresses, and special strings, affecting the performance and reliability of the similarity estimation, and introducing many uncertain words. This issue may lead to out-of-vocabulary (OOV). We replace low-frequency instructions with predefined labels, and the corresponding relationships between low-frequency instructions and labels are shown in Table 1. Therefore, these instructions have to be preprocessed:

1. Operators are mainly used to increase the readability of the codes, and they have little effect on the code analysis. Therefore, they can be removed and identifiers are separated by spaces.

2. Eliminate the extra overhead caused by low-frequency instructions by establishing a matching table, as shown in Table 1.

Table 1: Low-frequency instruction matching rules

Low-frequency instructions	Label
Numerical constants	Num
Memory addresses	Mem
Special strings	Str
Unrecognized function address	Func

The original binary program needs to be de-obfuscated and then the CFG structure can be optimized. We convert them into VEX-IR code and clean instructions before inputting them into the code similarity evaluation network.

3.2 Code Similarity Evaluation for Multi-Layer Data Embedding

An unavoidable difficulty in binary code similarity analysis is that programs of different architectures cannot be compared directly. As the architecture changes, the format of the instruction set and binary code will also change dramatically. Therefore, simply comparing the similarity of the program syntax to evaluate the similarity of the code is not feasible. Other features of the program need to be compared. The principle of binary code similarity analysis is based on the similarity comparison between features of functions, while natural language processing (NLP) techniques can perform similarity comparisons on words in text [46]. Therefore, we resort to the idea of NLP to solve the problem of binary code similarity analysis.

The basic block of programs is composed of multiple instructions, and each instruction consists of an operator and several parameters. For example, {*push eax, num*} is an instruction, where push is the opcode, *eax* and *num* are the operands. Therefore, each instruction of the program can be regarded as a word, and then the instructions are encoded into instruction vectors by distributed word encoding. We utilize skip-gram to convert instructions of the basic block into feature vectors, thus the basic block can be regarded as a sentence composed of phrases. We introduce the bidirectional long short term memory (BiLSTM) model [47,48] to add the basic block function feature vector with an additional basic block semantic information vector to form a basic block embedding vector with basic block attributes. These basic block embedding vectors represent the attributed control flow graph (ACFG). ACFG contains useful information on the basic block function and also integrates the semantic information of the context into the feature vector. As a result, our program can be employed for programs on different architectures.

After obtaining the ACFG vector of each basic block, the structure2vec network is used to convert the ACFG vector into a more complex graph, thereby obtaining a high-dimensional feature vector. The feature vector is used for code similarity evaluation. Fig. 2 shows the workflow of code similarity evaluation.

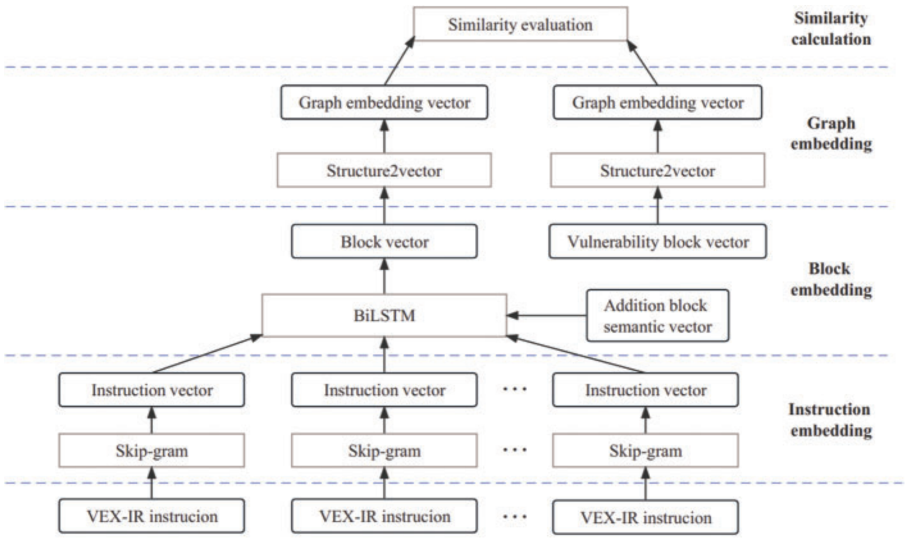


Figure 2: The framework of similarity evaluation model

3.2.1 Instruction Embedding

Skip-gram model connects the intermediate vocabulary with the entire text by controlling the size of the sliding window. However, it will increase the computational overhead and training time. The instruction embedding of programs only considers the semantic information and context of basic blocks. Therefore, we improve the model by using intermediate words to predict context and keywords, to increase the contextual connection of basic blocks and the utilization of semantic information with less training overhead. The improved skip-gram model is shown in Fig. 3.

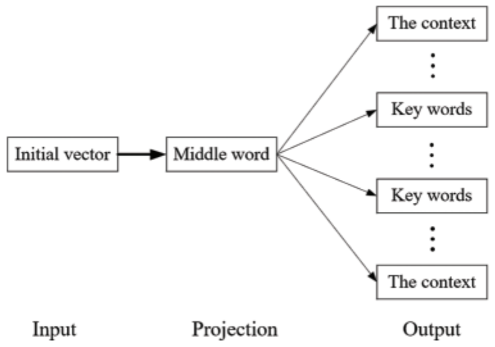


Figure 3: The framework of the skip-gram model

By constructing a sliding window of size m, all the instructions of the basic block are mapped to a specific position, and the number of instructions in each position is $2m + 1$. According to the given instruction, instructions of the basic block are constructed as:

$$\left[\ldots, \left(\omega_i, \omega_{(i-m)}\right), \ldots, \left(\omega_i, \omega_{(i-1)}\right), \left(\omega_i, \omega_{(i+m)}\right), \ldots, \left(\omega_i, \omega_{(i+m)}\right)\right] \tag{1}$$

where $m + 1 \le i \le N - m$, and N is the total number of instructions with practical significance.

The loss function of the improved skip-gram model is defined as:

$$Loss = \frac{1}{|V|} \sum_{i=1}^{|V|} \sum_{0<|m|<c} \log p \left(\omega_{i+m} \middle| \omega_i\right) + g\left(\omega_{i+m}, \omega_i\right) \tag{2}$$

The structure of the instruction embedding model based on skip-gram is shown in Fig. 4.

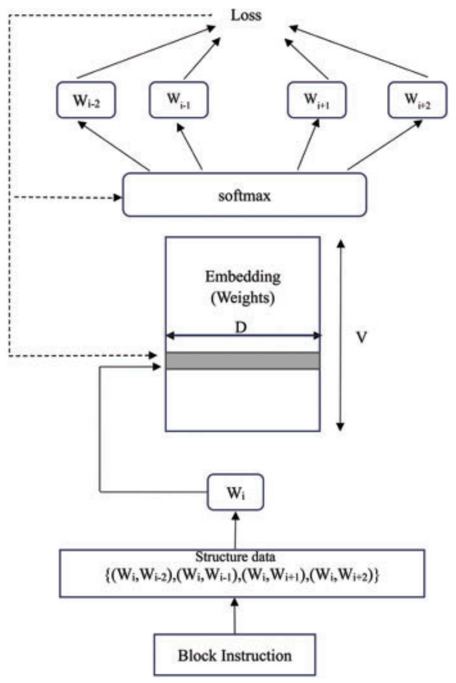


Figure 4: Instruction embedding model based on skip-gram

3.2.2 Block Embedding

We adopt BiLSTM to extract the contextual semantic information carried by the basic block to improve the integrity of the basic block information. As shown in Fig. 5, e_i^f $(i = 1, 2, \ldots, n)$ is an instruction embedding vector, which is used to construct a basic block. e_i^f is inserted into the LSTM$_f$ in the correct order, and then inserted reversely into the LSTM$_b$. The two components of LSTM, h_T^f and h_1^b, are fused as the basic block embedding vector h^{comb} generated by the BiLSTM model. This embedding vector contains all the useful semantic information of the basic block.

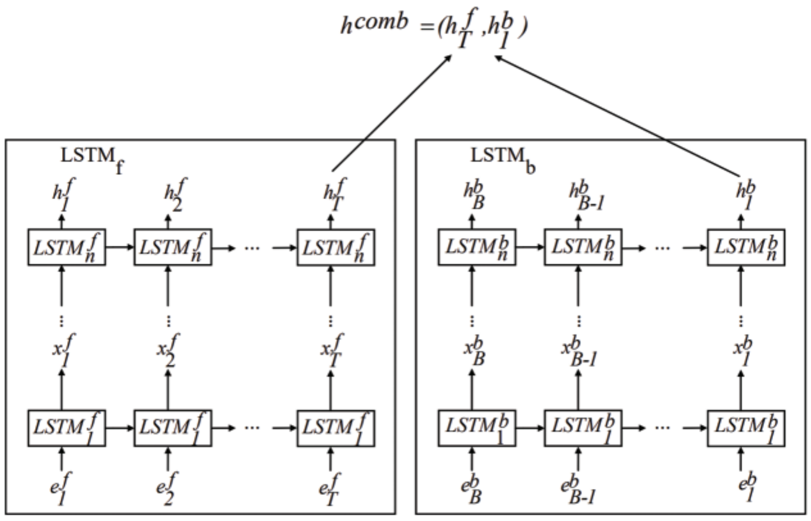


Figure 5: The workflow of the BiLSTM

To improve the overall performance of the model, we introduce a self-attention network after the BiLSTM network, which can effectively transfer information while reducing redundant embedding vectors.

3.2.3 Similarity Calculation Model

Neural networks are adopted to calculate the similarity between the functions in the program and the functions in the vulnerability library. This similarity is used to determine whether the program contains high-risk functions. We convert the binary program into an ACFG, such that each node has features related to the basic block not only the source code, but also the relationship between them can be readily obtained.

Let the ACFG be a function as $g = (V, E)$, where V represents the eigenvector of the vertices of the program basic block, and E represents the set of eigenvectors of the calling block and the called basic block of the program.

Each basic block vertex vector v in V has a property x_v identical to the rest of the vertex vectors in ACFG. The feature vectors representing the functions of the basic blocks are generated in the p-dimensional space composed of the basic blocks $v\,(v \in V)$, and then the vertex feature vectors of all the basic blocks are aggregated. At last, the embedding vector μ_g' of ACFG is generated as follows:

$$\mu_g' := M_{v \in V}\,(\mu_v) \tag{3}$$

Based on the structure2vec network, we proposed the ACFG embedding vector generation method and its basic block vertex vector mapping relationship as:

$$G(x_v, \sum_{u \in E(v)} \mu_u) = \tanh\left(W_1 x_v + \sigma\left(\sum_{u \in E(v)} \mu_u\right)\right) \tag{4}$$

Among them, x_v is a d-dimensional vector, representing the features of the ACFG nodes, W_1 is a $d \times p$ parameter matrix, and p is the embedding dimension of ACFG. σ represents a fully connected network with n layers:

$$\sigma\,(l) = P_1 \times Relu\,(P_2 \times \cdots Relu\,(P_n l)) \tag{5}$$

where $P_i\,(i = 1, 2, \cdots, n)$ is a $p \times p$ parameter matrix.

The structure2vec network proposed in this work adopts recurrent neural (RNN) networks as its backbone. We use the attention network to fuse the feature vector generated by the previous network unit to further reduce the redundant part in the feature vector. The structure of the structure2vec network proposed in this paper is shown in Fig. 6. We measure the similarity by calculating the cosine distance between the ACFG vector corresponding to the input binary program and the ACFG vector corresponding to the high-risk vulnerability function in the sample library. We design the structure2vec network as a twin structure and then calculate the similarity of the input vectors to obtain the similarity score. The twin architecture is shown in Fig. 7.

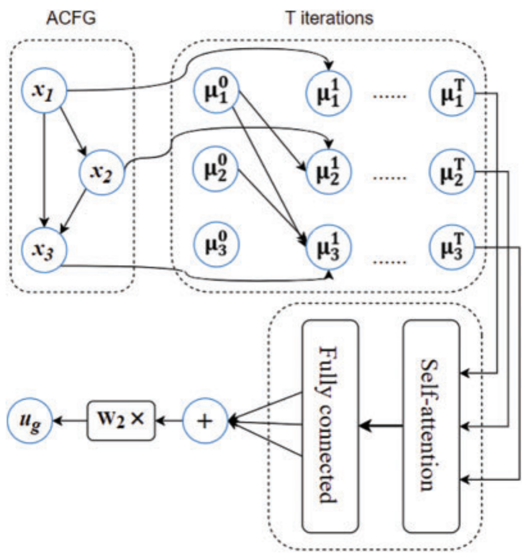


Figure 6: Structure2vec with self-attention

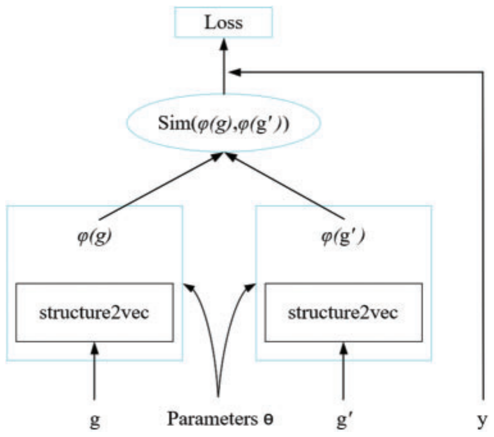


Figure 7: The proposed twin architecture based on the structure2vec network

The two structure2vec networks take the g and g' of ACFG as input during the training process. We obtain the graph embedding vectors $\varphi\ (g)$ and $\varphi\ (g')$ by the embedding algorithm. Then, the similarity between them is obtained by calculating the cosine distance:

$$Sim\,(g, g') = \cos\,(\phi\ (g), \phi\ (g')) = \frac{\sum_{i=1}^{n} < \phi\ (g), \phi\ (g') >}{\sqrt{\sum_{i=1}^{n} \|\phi\ (g)\|} \cdot \sqrt{\sum_{i=1}^{n} \|\phi\ (g')\|}} \tag{6}$$

where n represents for vector dimension, and $\phi\ (g)$ is the vectors in ACFG.

If the similarity score is greater than or equal to the threshold, we output the function name and the address in the memory of the program. Then we use symbolic execution to solve the function, and the solution result can be converted into the initial seed of the fuzzing and input into the fuzzing system for vulnerability mining. Subsequent seed mutations will also be performed based on the generated initial seed.

The biggest challenge of symbolic execution is the path explosion issue. If the program can generate test cases with a high coverage rate under the premise of as little path exploration as possible, then the risk of path explosion can be greatly alleviated. To solve this issue, we first determine the functions in the program that may have potential vulnerabilities. We then conduct fuzzing for these functions. We proposed the simnetseed-based system to increase the speed of fuzzing by increasing the effective coverage of the path in fuzzing, and to solve the high-risk vulnerability functions detected in the program by symbolic execution.

4 Experiments

4.1 Vulnerability Database and Dataset Construction

To identify potential security risks by the similarity of binary codes, it is necessary to establish a vulnerability sample library. By consulting the common vulnerabilities and exposures (CVE) website, a large number of CVE vulnerabilities can be obtained, and we divide them into two categories according to the source of these vulnerabilities.

1. Vulnerabilities obtained from source code. We collect the relevant function source code from the internet and use the arm-gcc compiler to convert it into binary functions under the advanced reduced instruction set computer machines (ARM) and microcomputer without interlocked pipeline stages (MIPS) architectures with different compilation optimization levels.

2. Vulnerabilities in IoT devices. To detect the security of binary functions, such as the security of firmware, unpacking tools such as binwalk and interactive disassembler professional (IDA Pro) can be used to detect the security of binary functions.

We de-obfuscate and optimize the CFG structure of the binary codes, converting them into VEX-IR intermediate codes and conducting data cleaning. At last, the ACFG embedding vectors are obtained through Word2vec and BiLSTM networks, which constitute the vulnerability sample library. We constructed a vulnerability sample library with a total of 1,266 samples.

We adopt the method provided by InnerEye-BB to construct an appropriate dataset for model evaluation. Specifically, we use gcc and clang compilers and set three optimization levels of O1-O3 to compile several functions of OpenSSL (v1.0.1a and v1.0.1f) and Linux packages to obtain the dataset. Then the dataset is further divided into three parts training, validation, and testing. The details of the dataset are shown in Table 2.

Table 2: Dataset partition details

Level	Training set			Validation set			Test set		
	ARM	MIPS	Total	ARM	MIPS	Total	ARM	MIPS	Total
O1	34,156	32,235	66,391	3,029	3,649	6,678	4,368	4,345	8,713
O2	44,561	42,758	87,319	5,013	5,069	10,082	5,068	5,590	10,658
O3	46,138	44,782	90,920	5,390	5,379	10,769	6,000	5,899	11,899
Total	124,855	119,775	244,630	13,432	14,097	27,529	15,436	15,834	31,270

4.2 Evaluation

4.2.1 Influence of Hyper-Parameters

We train the similarity evaluation model for 10 epochs using the training set, and then use the test set to evaluate the accuracy and reliability of the model. In order to study the influence of the dimension of instruction embedding on similarity discrimination, we set the dimension of instruction embedding from {20, 40, 60, 80, 100}. The experimental results are shown in Table 3. It can be observed that increasing the number of dimensions will increase the area under curve (AUC), but the AUC is not simply determined by the dimension of instruction embedding. Therefore, we set the dimension of instruction embeddings to 60 in subsequent experiments.

Table 3: The influence of the dimension of instruction embedding

Dimension	20	40	60	80	100
AUC (%)	96.36	96.54	97.4	97.31	96.87

Then, we compared the AUC under different basic block embedding dimensions. As shown in Table 4, increasing the dimension will result in better performance. Considering that a higher embedding dimension will significantly increase the computational cost, we set the basic block embedding dimension to 10.

Table 4: The influence of the dimension of basic block embedding

Dimension	4	8	10	12	14
AUC (%)	96.28	96.57	97.38	97.40	97.41

4.2.2 Comparison with Other Methods

We compare the performance of the proposed model with InnerEyeBB [49], DeepBinDiff [50], and DeepWalk [51] on the validation set, and the experimental results are shown in Table 5.

Table 5: Comparison with other methods in feature extraction

Method	InnerEyeBB	DeepBinDiff	DeepWalk	Ours
AUC (%)	88.94	90.25	92.36	97.40

We verify the effectiveness of the proposed similarity evaluation method by comparing it with the Genius and Gemini models. The experimental results are shown in Table 6. It can be seen that the performance of the proposed method is better than that of the Gemini model. Compared with the unsupervised learning model Genius, it has achieved a great lead.

Table 6: Comparison with other methods in similarity evaluation

Method	Genius	Gemini	Ours
AUC (%)	79.83	91.78	95.05

4.2.3 Simnet-Seed Application Test

We integrated the method proposed in this work into the simnet-seed tool for practical application tests. The overall workflow of the simnet-seed tool for vulnerability detection is shown in Fig. 8.

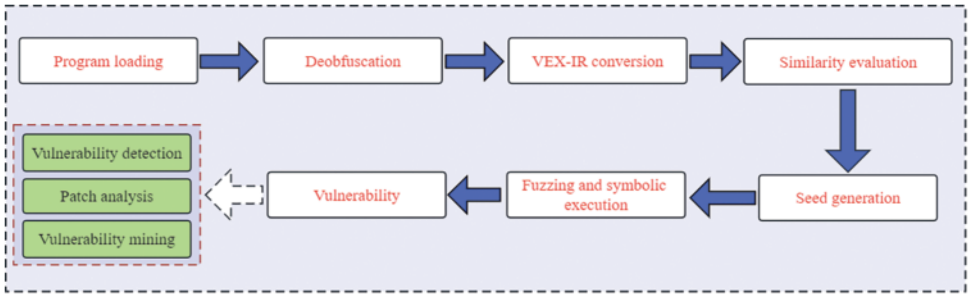
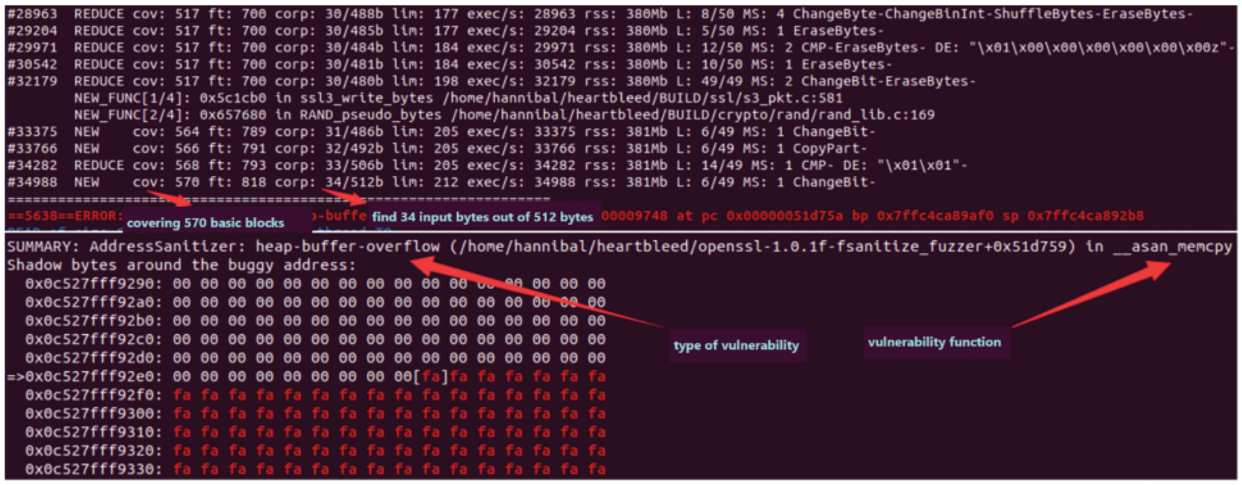


Figure 8: The workflow of vulnerability mining

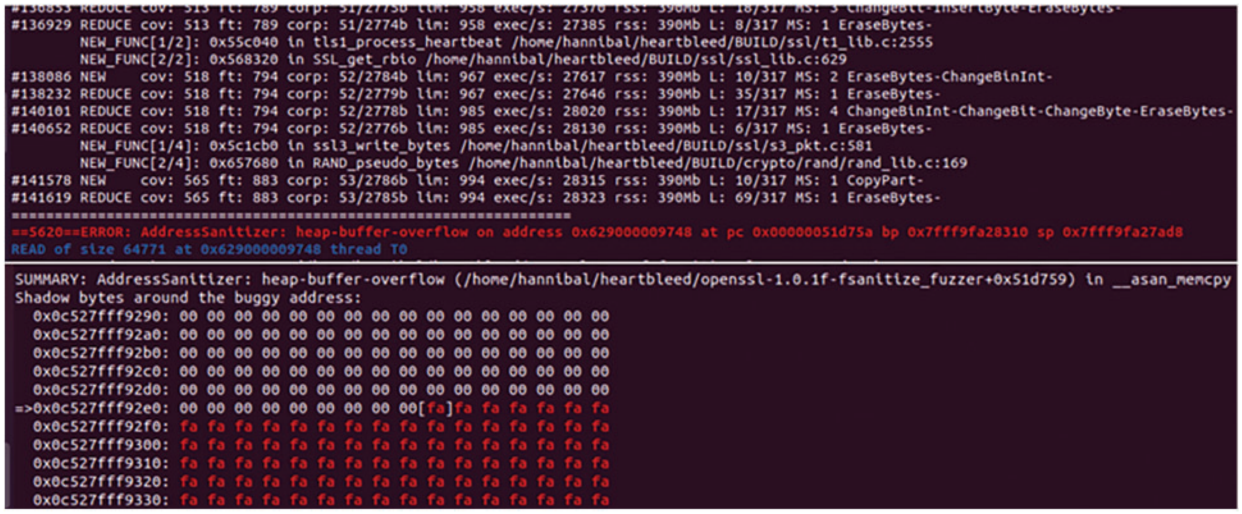
We first use simnet-seed to load the binary program, then de-obfuscate the binary program and optimize the CFG structure to restore the real control flow graph of the program. Then we convert the program language into the intermediate language to eliminate the influence caused by different architectures and compiled languages. Similarity matching is performed between the functions in the program and the functions in the vulnerability library. Finally, the symbolic execution is performed on the similarity score and the solution is converted into the initial seed of fuzzing. We take the CVE vulnerability (Heartbleed Vulnerability, CVE-2014-0160) as the case and the libfuzzer as the fuzzing tool to verify the proposed method. It can be seen from Figs. 9 and 10 that when using the seed generated by the simnet-seed tool as input, libfuzzer has covered 570 basic blocks of the program after 34,988 executions and found 34 abnormal input bytes in 512 bytes, and successfully found the asan_memcpy() function belonging to the heap overflow vulnerability. When the seeds generated by the simnet-seed tool are not used, libfuzzer has covered a total of 565 basic blocks of the program after 141,619 executions and found 53 input bytes out of 2,785 bytes before successfully finding the asan_memcpy() vulnerability. It can be seen that the seeds generated by the simnet-seed tool cover more basic blocks with fewer execution times.


covering 570 basic blocks
find 34 input bytes out of 512 bytes
type of vulnerability
vulnerability function

```
#28963  REDUCE cov: 517 ft: 700 corp: 30/488b lim: 177 exec/s: 28963 rss: 380Mb L: 8/50 MS: 4 ChangeByte-ChangeBinInt-ShuffleBytes-EraseBytes-
#29204  REDUCE cov: 517 ft: 700 corp: 30/485b lim: 177 exec/s: 29204 rss: 380Mb L: 5/50 MS: 1 EraseBytes-
#29971  REDUCE cov: 517 ft: 700 corp: 30/484b lim: 184 exec/s: 29971 rss: 380Mb L: 12/50 MS: 2 CMP-EraseBytes- DE: "\x01\x00\x00\x00\x00\x00\x00z"-
#30542  REDUCE cov: 517 ft: 700 corp: 30/481b lim: 184 exec/s: 30542 rss: 380Mb L: 10/50 MS: 1 EraseBytes-
#32179  REDUCE cov: 517 ft: 700 corp: 30/480b lim: 198 exec/s: 32179 rss: 380Mb L: 49/49 MS: 2 ChangeBit-EraseBytes-
        NEW_FUNC[1/4]: 0x5c1cb0 in ssl3_write_bytes /home/hannibal/heartbleed/BUILD/ssl/s3_pkt.c:581
        NEW_FUNC[2/4]: 0x657680 in RAND_pseudo_bytes /home/hannibal/heartbleed/BUILD/crypto/rand/rand_lib.c:169
#33375  NEW    cov: 564 ft: 789 corp: 31/486b lim: 205 exec/s: 33375 rss: 381Mb L: 6/49 MS: 1 ChangeBit-
#33766  NEW    cov: 566 ft: 791 corp: 32/492b lim: 205 exec/s: 33766 rss: 381Mb L: 6/49 MS: 1 CopyPart-
#34282  REDUCE cov: 568 ft: 793 corp: 33/506b lim: 205 exec/s: 34282 rss: 381Mb L: 14/49 MS: 1 CMP- DE: "\x01\x01"-
#34988  NEW    cov: 570 ft: 818 corp: 34/512b lim: 212 exec/s: 34988 rss: 381Mb L: 6/49 MS: 1 ChangeBit-
=================================================================
==5638==ERROR:                    -buffe                                00009748 at pc 0x00000051d75a bp 0x7ffc4ca89af0 sp 0x7ffc4ca892b8

SUMMARY: AddressSanitizer: heap-buffer-overflow (/home/hannibal/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x51d759) in __asan_memcpy
Shadow bytes around the buggy address:
  0x0c527fff9290: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c527fff92a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c527fff92b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c527fff92c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c527fff92d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c527fff92e0: 00 00 00 00 00 00 00 00 00[fa]fa fa fa fa fa fa
  0x0c527fff92f0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c527fff9300: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c527fff9310: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c527fff9320: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c527fff9330: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

Figure 9: The output of libfuzzer with seed generated by the simnet-seed tool



```
#136853 REDUCE cov: 513 ft: 789 corp: 51/2775b lim: 958 exec/s: 27370 rss: 390Mb L: 18/317 MS: 3 ChangeBit-InsertByte-EraseBytes-
#136929 REDUCE cov: 513 ft: 789 corp: 51/2774b lim: 958 exec/s: 27385 rss: 390Mb L: 8/317 MS: 1 EraseBytes-
        NEW_FUNC[1/2]: 0x55c040 in tls1_process_heartbeat /home/hannibal/heartbleed/BUILD/ssl/t1_lib.c:2555
        NEW_FUNC[2/2]: 0x568320 in SSL_get_rbio /home/hannibal/heartbleed/BUILD/ssl/ssl_lib.c:629
#138086 NEW    cov: 518 ft: 794 corp: 52/2784b lim: 967 exec/s: 27617 rss: 390Mb L: 10/317 MS: 2 EraseBytes-ChangeBinInt-
#138232 REDUCE cov: 518 ft: 794 corp: 52/2779b lim: 967 exec/s: 27646 rss: 390Mb L: 35/317 MS: 1 EraseBytes-
#140101 REDUCE cov: 518 ft: 794 corp: 52/2778b lim: 985 exec/s: 28020 rss: 390Mb L: 17/317 MS: 4 ChangeBinInt-ChangeBit-ChangeByte-EraseBytes-
#140652 REDUCE cov: 518 ft: 794 corp: 52/2776b lim: 985 exec/s: 28130 rss: 390Mb L: 6/317 MS: 1 EraseBytes-
        NEW_FUNC[1/4]: 0x5c1cb0 in ssl3_write_bytes /home/hannibal/heartbleed/BUILD/ssl/s3_pkt.c:581
        NEW_FUNC[2/4]: 0x657680 in RAND_pseudo_bytes /home/hannibal/heartbleed/BUILD/crypto/rand/rand_lib.c:169
#141578 NEW    cov: 565 ft: 883 corp: 53/2786b lim: 994 exec/s: 28315 rss: 390Mb L: 10/317 MS: 1 CopyPart-
#141619 REDUCE cov: 565 ft: 883 corp: 53/2785b lim: 994 exec/s: 28323 rss: 390Mb L: 69/317 MS: 1 EraseBytes-
=================================================================
==5620==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x629000009748 at pc 0x00000051d75a bp 0x7fff9fa28310 sp 0x7fff9fa27ad8
READ of size 64771 at 0x629000009748 thread T0

SUMMARY: AddressSanitizer: heap-buffer-overflow (/home/hannibal/heartbleed/openssl-1.0.1f-fsanitize_fuzzer+0x51d759) in __asan_memcpy
Shadow bytes around the buggy address:
  0x0c527fff9290: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c527fff92a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c527fff92b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c527fff92c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x0c527fff92d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x0c527fff92e0: 00 00 00 00 00 00 00 00 00[fa]fa fa fa fa fa fa
  0x0c527fff92f0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c527fff9300: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c527fff9310: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c527fff9320: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c527fff9330: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

Figure 10: The output of libfuzzer without seed generated by the simnet-seed tool

Finally, we combine the simnet-seed tool with the mainstream symbolic execution tool Driller, the fuzzing tool AFL++, and the hybrid fuzzing tool qsym to verify its effectiveness. We adopt five programs named Xpdf, TCPdump, Adobe Reader, Gimp, and Libxml2 as test samples, respectively, and record the time it takes to generate the first three crashes. The experimental results are shown in Table 7.

Table 7: Time-consuming comparison of generating crashes with and without simnet-seed

Programs	AFL++		QSYM		Driller	
	Without	With	Without	With	Without	With
Xpdf	45 min	32 min	35 min	28 min	38 min	20 min
TCPdump	20 h	14 h	13 h	8 h	15 h	10 h
Adobe reader	4 h	3 h	3 h	2 h	8 h	4 h
Gimp	3 h	1 h	1 h	45 min	2 h	57 min
Lixml2	5 h	3 h	4 h	2 h	2 h	75 min

It can be seen from Table 7 that when these three fuzzing tools are combined with the simnet-seed tool, the time-consuming to generate the first three crashes is significantly reduced, i.e., the effective coverage of the seeds generated by simnet-seed on these five programs is better than the seed generation and mutation strategies of the fuzzing tool. This also proves that the path exploration algorithm with dynamic seed priority and threat function priority can effectively improve the overall path exploration ability of fuzzing. We believe that the simnet-seed tool can significantly improve the mining efficiency of most programs, and its basic block function coverage is also beyond baselines, thus proving its reliability and effectiveness.

5 Conclusion

Aiming at the obfuscation problem of the program, we introduce and implement a de-obfuscation method, which can eliminate the influence of control flow flattening, and solve the problem of incomplete recovery results and redundant control flow graph in the obfuscation method of control flow flattening. Aiming at the problem of cross-architecture cross-compiler and compilation options, this paper introduces and implements an interlingua transformation method based on VEX-IR. By using label replacement to realize VEX-IR preprocessing, the learning efficiency of basic blocks based on interlingua is improved. We proposed a basic block embedding method based on bidirectional LSTM, which solves the problem of missing important information such as semantics when embedding basic blocks. Aiming at the measurement problem of binary code similarity, we proposed a binary function similarity analysis method based on graph embedding. Before the ACFG embedding vector is finally generated, the self-attention mechanism is introduced to improve the quality of the feature vector, and the important features of the basic block are integrated to solve the error of manual feature construction and the problem of semantic loss in function similarity analysis. We proposed a seed generation method based on function similarity detection, which solves the problem of symbolic execution path explosion and also meets the high coverage requirements of fuzzing for test cases. In the experiment, we used the simnet-seed tool for vulnerability mining and detection and verified the effectiveness of the proposed method in vulnerability detection and fuzzing speed optimization.

We built a professional analysis tool that security researchers can use to achieve cross-architecture analysis of source-free binaries. By generating more effective test cases and optimizing test objectives, the tool makes fuzzing more efficient, accelerates the discovery of potential vulnerabilities, and improves the efficiency of testing. Through the identification and analysis of key functions, the research is helpful to dig into the possible security risks in the software, improve the depth and breadth of safety analysis, and enhance the safety analysis. The cross-architecture analysis technology of the research increases the generality of the method, makes it applicable to various software systems with different architectures, and improves the flexibility of the research in practical applications. The research enhances the feasibility of fuzzing testing tools in large-scale projects, provides a set of innovative and practical methods for the field of software security, provides strong support for the improvement and performance enhancement of fuzzing testing tools, and provides a more reliable solution for practical applications.

Our proposed technique can mitigate the risk of path explosion without obtaining the source code, and protect the privacy and security of users to a certain extent.

Admittedly, our work has achieved good effectiveness, but there are still some limitations in the implementation process. This article is based on the VEX-IR intermediate language, but the basic block embedding vector generation involves modifying the LLVM backend to add ID to the basic block and then converting it to VEX-IR. This would be slower but complete, so consider changing the intermediate language type unification to LLVM-IR.

In the future, we plan to optimize the structure2vec model to build an n-layer fully connected neural network to improve the efficiency of message functions. In addition, we also consider using gated recurrent unit (GRU) networks to replace the original RNN networks, hoping to alleviate the problems of long-term memory and gradient disappearance more effectively.

Acknowledgement: Not applicable.

Funding Statement: The authors received no specific funding for this study.

Author Contributions: The authors confirm contribution to the paper as follows: study conception and design: Zhenhui Li, Shuangping Xing; data collection: Lin Yu, Huiping Li; analysis and interpretation of results: Fan Zhou, Guangqiang Yin, Xikai Tang; draft manuscript preparation: Zhiguo Wang. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: Not applicable.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] A. Kumar, K. Abhishek, M. R. Ghalib, A. Shankar and X. Cheng, "Intrusion detection and prevention system for an IoT environment," *Digital Communications and Networks*, vol. 8, no. 4, pp. 540–551, 2022.

[2] L. Qi, Y. Yang, X. Zhou, W. Rafique and J. Ma, "Fast anomaly identification based on multiaspect data streams for intelligent intrusion detection toward secure Industry 4.0," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 9, pp. 6503–6511, 2021.

[3] B. Weinger, J. Kim, A. Sim, M. Nakashima, N. Moustafa *et al.,* "Enhancing IoT anomaly detection performance for federated learning," *Digital Communications and Networks*, vol. 8, no. 3, pp. 314–323, 2022.

[4] J. Gallego-Madrid, R. Sanchez-Iborra, P. M. Ruiz and A. F. Skarmeta, "Machine learning-based zero-touch network and service management, A survey," *Digital Communications and Networks*, vol. 8, no. 2, pp. 105–123, 2022.

[5] H. Dai, J. Yu, M. Li, W. Wang, A. X. Liu *et al.,* "Bloom filter with noisy coding framework for multi-set membership testing," *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, pp. 6710–6724, 2022.

[6] S. Li, M. Cai, R. Edwards, Y. Sun and L. Jin, "Research on encoding and decoding of non-binary polar codes over GF (2m)," *Digital Communications and Networks*, vol. 8, no. 3, pp. 359–372, 2022.

[7] C. B. Sahin, "Semantic-based vulnerability detection by functional connectivity of gated graph sequence neural networks," *Soft Computing*, vol. 27, no. 9, pp. 5703–5719, 2023.

[8] H. Miura, T. Kimura, H. Aman and K. Hirata, "Game-theoretic approach to epidemic modeling of countermeasures against future malware evolution," *Computer Communications*, vol. 206, pp. 160–171, 2023.

[9] W. Su and H. Fei, "Survey of coverage-guided grey-box fuzzing," *Journal of Information Security Research*, vol. 8, pp. 643–655, 2022.

[10] K. Yang, Y. He, H. Ma and X. Wang, "Precise execution reachability analysis: Theory and application," *Journal of Software*, vol. 29, pp. 1–22, 2018.

[11] C. Song, X. Wang and W. Zhang, "Analysis and optimization of angr in dynamic software test application," *Computer Engineering and Science*, vol. 40, pp. 163–168, 2018.

[12] B. Zhang and Z. Xi, "A systematic review of binary program vulnerabilities feature extraction and discovery strategy generation methods," *Journal of Physics: Conference Series,* vol. 1827, no. 1, pp. 012090, 2021.

[13] W. Han, J. Pang and X. Zhou, "Binary vulnerability mining technology based on neural network feature fusion," in *2022 5th Int. Conf. on Advanced Electronic Materials, Computers and Software Engineering (AEMCSE)*, Wuhan, China, IEEE, pp. 257–261, 2021.

[14] X. Sun, Z. Ye, L. Bo, X. Wu, Y. Wei *et al.,* "Automatic software vulnerability assessment by extracting vulnerability elements," *Journal of Systems and Software*, vol. 204, pp. 111790, 2023.

[15] C. Caballero-Gil, R. Alvarez, C. Hernandez-Goya and J. Molina-Gil, "Research on smart-locks cybersecurity and vulnerabilities," *Wireless Networks,* 2023. https://doi.org/10.1007/s11276-023-03376-8

[16] Z. Ren, H. Zhen, J. Zhang, W. Wang, T. Feng *et al.,* "A review of fuzzing techniques," *Journal of Computer Research and Development*, vol. 58, pp. 2021-05-944, 2021.

[17] Y. Yang, X. Yang, M. Heidari, M. A. Khan, G. Srivastava *et al.,* "Astream: Data-stream-driven scalable anomaly detection with accuracy guarantee in IIoT environment," *IEEE Transactions on Network Science and Engineering*, vol. 10, pp. 3007–3016, 2022.

[18] Q. Zhang and Y. Ma, "Research on optimization of afl-fuzzer seed mutation strategy," *Modern Information Technology*, vol. 5, pp. 142–145, 2021.

[19] X. Wang, L. Yang, L. Ma, Y. Mu, J. Shi *et al.,* "Function similarity detection and lineage analysis for cross-architecture malware," *Journal of Army Engineering University of PLA*, vol. 1, pp. 36–47, 2022.

[20] Y. Ren, Y. Zhang and C. Ai, "Survey on taint analysis technology," *Journal of Computer Applications*, vol. 39, pp. 2302–2309, 2019.

[21] W. Jie, Q. Chen, J. Wang, A. S. Voundi Koe, J. Li *et al.,* "A novel extended multimodal ai framework towards vulnerability detection in smart contracts," *Information Sciences*, vol. 636, pp. 118907, 2023.

[22] W. Wang, Z. Chen, Z. Zheng and H. Wang, "An adaptive fuzzing method based on transformer and protocol similarity mutation," *Computers and Security*, vol. 129, pp. 103197, 2023.

[23] S. Baradaran, M. Heidari, A. Kamali and M. Mouzarani, "A unit-based symbolic execution method for detecting memory corruption vulnerabilities in executable codes," *International Journal of Information Security*, vol. 22, pp. 1277–1290, 2023.

[24] Q. Wang and L. Song, "Fuzzing test based on potential of seed mutation," *Science Technology and Engineering*, vol. 20, pp. 3656–3661, 2020.

[25] L. Wang, F. Li, L. Li and X. Feng, "Principle and practice of taint analysis," *Journal of Software*, vol. 28, pp. 860–882, 2017.

[26] K. W. Church, "Word2vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.

[27] A. Jaffe, Y. Kluger, O. Lindenbaum, J. Patsenker, E. Peterfreund *et al.,* "The spectral underpinning of word2vec," *Frontiers in Applied Mathematics and Statistics*, vol. 6, pp. 593406, 2020.

[28] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in Neural Information Processing Systems*, vol. 26, pp. 3111–3119, 2013.

[29] T. Mikolov, K. Chen, G. Corrado and J. Dean, "Efficient estimation of word representations in vector space," arXiv:1301.3781, 2013.

[30] I. Sutskever, O. Vinyals and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in Neural Information Processing Systems*, vol. 27, pp. 3104–3112, 2014.

[31] D. Kim, E. Kim, S. K. Cha, S. Son and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1661–1682, 2023.

[32] J. Guo, B. Zhao, H. Liu, D. Leng, Y. An *et al.,* "DeepDual-SD: Deep dual attribute-aware embedding for binary code similarity detection," *International Journal of Computational Intelligence Systems*, vol. 16, no. 1, pp. 35, 2023.

[33] D. Zheng, Z. Ran, Z. Liu, L. Li and L. Tian, "An efficient bar code image recognition algorithm for sorting system," *Computers, Materials & Continua*, vol. 64, no. 3, pp. 1885–1895, 2020.

[34] Y. Peng, X. Li, J. Song, Y. Luo, S. Hu *et al.,* "Verification mechanism to obtain an elaborate answer span in machine reading comprehension," *Neurocomputing*, vol. 466, pp. 80–91, 2021.

[35] X. Tang, D. Zheng, G. S. Kebede, Z. Li, X. Li *et al.,* "An automatic segmentation framework of quasi-periodic time series through graph structure," *Applied Intelligence*, vol. 53, no. 20, pp. 23482–23499, 2023.

[36] Y. Jia, B. Liu, W. Dou, X. Xu, X. Zhou *et al.,* "CroApp: A CNN-based resource optimization approach in edge computing environment," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 9, pp. 6300–6307, 2022.

[37] L. Hu, S. Bi, Q. Liu, Y. Jiang and C. Chen, "Intelligent reflecting surface aided covert wireless communication exploiting deep reinforcement learning," *Wireless Networks*, vol. 29, no. 2, pp. 877–889, 2023.

[38] M. K. Somesula, S. K. Mothku and A. Kotte, "Deep reinforcement learning mechanism for deadline-aware cache placement in device-to-device mobile edge networks," *Wireless Networks*, vol. 29, no. 2, pp. 569–588, 2023.

[39] Y. David, N. Partush and E. Yahav, "Similarity of binaries through re-optimization," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Barcelona, Spain, pp. 79–94, 2017.

[40] C. Park, J. Lee, Y. Kim, J. Park and H. Kim, "An enhanced AI-based network intrusion detection system using generative adversarial networks," *IEEE Internet of Things Journal*, vol. 10, no. 3, pp. 2330–2345, 2023.

[41] X. Sun, H. Wang, X. Fu, H. Qin, M. Jiang *et al.,* "Substring-searchable attribute-based encryption and its application for iot devices," *Digital Communications and Networks*, vol. 7, no. 2, pp. 277–283, 2021.

[42] Y. Miao, X. Bai, Y. Cao, Y. Liu, F. Dai *et al.,* "A novel short-term traffic prediction model based on SVD and ARIMA with blockchain in Industrial Internet of Things," *IEEE Internet of Things Journal*, vol. 10, no. 24, pp. 21217–21226, 2023.

[43] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa *et al.,* "Scalable graph-based bug search for firmware images," in *ACM SIGSAC Conf. on Computer and Communications Security*, Vienna, Austria, pp. 480–491, 2016.

[44] H. Dai, B. Dai and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Int. Conf. on Machine Learning*, New York, NY, USA, pp. 2702–2711, 2016.

[45] Y. Li and Y. Zhou, "Similar text matching based on siamese network and char-word vector combination," *Computer Systems and Applications*, vol. 31, pp. 295–302, 2022.

[46] W. Haining, "Development of natural language processing technology," *ZTE Technology Journal*, vol. 28, pp. 59–64, 2022.

[47] X. Xu, L. Huang and M. Gong, “Short-term traffic flow prediction based on combined model of convolutional neural network and bidirectional long-term memory network,” *Industrial Instrumentation and Automation*, vol. 1, pp. 13–18, 2020.

[48] L. Kong, G. Li, W. Rafique, S. Shen, Q. He *et al.,* “Time-aware missing healthcare data prediction based on ARIMA model,” *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2022. https://doi.org/10.1109/TCBB.2022.3205064

[49] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng *et al.,* “Neural machine translation inspired binary code similarity comparison beyond function pairs,” arXiv:1808.04706, 2018.

[50] Y. Duan, X. Li, J. Wang and H. Yin, “DeepBinDiff: Learning program-wide code representations for binary diffing,” in *Network and Distributed System Security Symp.*, San Diego, CA, USA, 2020.

[51] B. Perozzi, R. Al-Rfou and S. Skiena, “Deepwalk, online learning of social representations,” in *ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, New York, NY, USA, pp. 701–710, 2014.