International Journal on Semantic Web and Information Systems
Volume 19 • Issue 1

Binary Vulnerability Similarity Detection Based on Function Parameter Dependency

Bing Xia, State Key Laboratory of Mathematical Engineering and Advanced Computing, China
Wenbo Liu, ZhongYuan University of Technology, China*
https://orcid.org/0009-0001-2747-2564
Qudong He, Zhongyuan University of Technology, China
Fudong Liu, State Key Laboratory of Mathematical Engineering and Advanced Computing, China
Jianmin Pang, State Key Laboratory of Mathematical Engineering and Advanced Computing, China
RuiNan Yang, Zhongyuan University of Technology, China
JiaBin Yin, Zhongyuan University of Technology, China
YunXiang Ge, Zhongyuan University of Technology, China

ABSTRACT

Many existing works compute the binary vulnerability similarity based on binary procedure, which has coarse detection granularity and cannot locate the vulnerability trigger position accurately, and have a higher false positive rate, so a new binary vulnerability similarity detection method based on function parameter dependency in hazard API is proposed. First, convert the instructions of different architectures into an intermediate language, and use the compiler with a back-end optimizer to optimize and normalize the binary procedure. Then, locate the hazard API that appears in the binary procedure, and perform the function parameters dependency analysis to generate a set of parameter slices on the hazard API. Experiments show that the method has a higher recall rate (up to 14.3% better than the baseline model) in real-world scenarios, and not only locates the triggering position of the vulnerability but also identifies the fixed vulnerability.

KEYWORDS

Binary Similarity, Neural Networks, Parameter Slice, Vulnerability Detection

INTRODUCTION

Open-source codes accelerate the development of software systems, and once vulnerable found in classic open-source codes, it may cause large-scale network security problems. According to the "2022 Open Source Security and Risk Analysis Report" provided by Synopsys, by 2022, 97% of code bases contain open source components, and 81% of code bases contain at least one vulnerability. Using the unpatched cloned code to attack will have a serious impact on the software system (Yang., 2019). Due to the need to protect trade secrets or intellectual property rights, the software is usually

DOI: 10.4018/IJSWIS.322392 *Corresponding Author

deployed in binary form. So detecting open-source vulnerabilities in binary files is critical to the security of the software supply chain.

At present, the binary code similarity technology based on the binary procedure has been widely used in binary vulnerability detection. The main idea is to compare the semantic information of the binary code with the semantic information of the binary code carrying the binary vulnerability (Xia et al., 2022). Because the binary vulnerabilities are typically caused by calling hazard API and its incomplete parameter constraints, a vulnerability is triggered not the whole binary procedure code but the part of the code, so the existing binary vulnerability detection based on code similarity of the binary procedure has a larger detection granularity, lead to higher false positive rate in the real-world application scenario. Therefore, we propose a binary vulnerability similarity detection method based on hazard API function union which is composed of hazard API name and API parameter slices, which detects vulnerabilities with finer granularity and effectively reduces the false alarm rate.

The contributions of this paper are as follows:

1. A novel vulnerability representation for binary procedures. This representation is based on hazard API name and API parameter slices which can fine-grained locates the triggering position of the vulnerability.
2. This representation can be used in a variety of instruction architectures such as X86, ARM, and MIPS. Different architectures of binary procedure are promoted to intermediate language and use back-end optimizer to optimize and normalize the binary procedure, minimizing the semantic gap of architectures.
3. ComFU, a complete binary code similarity vulnerability detection system. Experiment results show that it provides better accuracy than previous state-of-the-art systems and has an ability to identify whether the vulnerability patch is fixed.

BACKGROUND

To quickly discover the known vulnerabilities in the binary code, binary code similarity technology came into being. The main method is to use the neural network to obtain the corresponding semantic information of the binary vulnerability and the target, and compare whether they are similar (Xia et al., 2022). According to the content of the comparison, the existing binary code similarity methods can be divided into two types based on procedure matching and patching.

David (David et al., 2017) and Zheng (Zheng et al., 2021), used the representation method based on LLVM(Low-Level Virtual Machine, LLVM) intermediate language (Intermediate Representation, IR) to alleviate the problem of comparing the same source procedure code in different architectures. Based on LLVM IR, the XLIR (Gui et al., 2022) scheme establishes a mapping between procedure source code and function binary code, which effectively alleviates the instruction difference of the same source procedure code in different architectures. David et al. (2014), Xu et al. (2017), Alrabaee et al. (2015), and Qiu et al. (2015) propose to introduce instruction attributes into the control flow graph CFG(Control Flow Graph, CFG), which enriches the function semantics. The above schemes are all oriented to functions as the comparison object. Since vulnerabilities are usually triggered by very few internal codes, there is a problem of coarse detection granularity, resulting in a high false positive rate of vulnerability detection. MVP (Xiao et al., 2020) adopts the vulnerability signature and patch signature scheme and captures the generation and repair of vulnerabilities through the patch signature. The BinXray (Xu et al., 2020) scheme signifies a set of basic block trajectories within a function and applies patch semantics to design trajectory similarity to identify whether the target program is patched. The patch-based method mainly utilizes patch update information, but the software version update contains a large amount of information, so it is difficult to locate the patch, which limits the application scenarios of the above solution.

Usually, the code that causes a vulnerability is not the entire function code but a part of the function code, and the patch for the vulnerability is usually to replace a function (such as the "sprintf" or "strcpy" hazard API function) or add constraints to parameters (such as a judgment condition). Taking the vulnerability CVE-2018-14714 shown in Figure 1 as an example, Figure 1(a) shows the binary procedure containing the vulnerability, and the vulnerability is fixed in Figure 1(b). Specifically, the repair solution is to replace the hazard API called "sprintf" with the "snprintf" function that limits the length of the formatted variable and increases the number of copies of the parameter 0x40 constraint string as judging condition. The current binary function-based similarity scheme needs to compare the entire binary function, as shown in Figure 1(d), that is, the entire binary function is used to represent the vulnerability. Since the difference between the binary code before and after the vulnerability repair is small, the vulnerability detection scheme based on the binary function is easy to cause a false positives. This paper believes that false positives are mainly caused by the following reasons: 1. The analysis granularity based on binary functions is too coarse, and it is part of the code that causes the vulnerability. 2. The implementation of the vulnerability patch is to modify the hazard function, and the execution logic modification of the binary function is rare and difficult to capture. 3. Based on the binary function similarity calculation, it is impossible to describe the data dependency and constraint relationship, and it is difficult to capture the loopholes triggered by unconstrained data, which is the main reason for the high false positive rate.

Based on the above observations, this paper proposes a binary vulnerability detection ComFU (Compare Function Union, ComFU) scheme based on hazard API parameter dependencies. This scheme extracts the name of the hazard API and instructions that are data-dependent on the parameters used by the hazard API. As shown in Figure 1(c), the ComFU scheme analyzes each parameter of the hazard API "sprintf" function in the binary function sub_1A26C (such as the variable "v5") data dependencies (such as the instruction in line 2 of (c)). ComFU's fine-grained comparison scheme not only captures the core code of vulnerabilities but also focuses on parameter dependencies that contain less execution logic. In this way, the parameter constraints clarify the dependencies between variables, so that the ComFU solution filters out a large number of instructions that are not related to vulnerability representations, reducing the false positive rate.

Other hazard APIs (such as "strcpy" "system" "memcpy", etc.) are similar to the "sprintf" function. ComFU can also obtain the key information of vulnerabilities and the dependencies between instructions by extracting parameter data-dependent instructions.

Figure 1. Vulnerability of CVE-2018-14714

```
1 char *__fastcall sub_1A26C(const char *a1)
2 {
3   char v5[64]; // [sp+0h] [bp-58h] BYREF
4   sprintf(v5, "/tmp/%s", a1);
5   if ( !strcmp(a1, "syscmd.sh") )
6   {
7     if ( byte_6D16C )
8     {
9       sprintf(&byte_6D16C, "%s > /tmp... ", &byte_6D16C);
  ...
```

(a) Binary function that contain vulnerabilities

```
1 int __fastcall sub_290C8(const char *a1)
2 {
3   char s[80]; // [sp+100h] [bp-50h] BYREF
4   snprintf(s, 0x40u, "/tmp/%s", a1);
5   if ( !strcmp(a1, "syscmd.sh") )
6   {
7     if ( !byte_B0D2C )
8       return f_write_string(...);
9     snprintf((char *)v4, 0x100u, "%s > /tmp...", &byte_B0D2C);
  ...
```

(b) Vulnerability-fixed binary functions

```
1 char *__fastcall sub_1A26C(const char *a1)
2   char v5[64]; // [sp+0h] [bp-58h] BYREF
3   sprintf(v5, "/tmp/%s", a1);
```

(c) Vulnerabilities that the function union scheme needs to match

```
1 char *__fastcall sub_1A26C(const char *a1)
2 {
3   char v5[64]; // [sp+0h] [bp-58h] BYREF
4   sprintf(v5, "/tmp/%s", a1);
5   if ( !strcmp(a1, "syscmd.sh") )
6   {
7     if ( byte_6D16C )
8     {
9       sprintf(&byte_6D16C, "%s > /tmp/...", &byte_6D16C);
  ...
```

(d) Vulnerabilities that require matching for binary function similarity schemes

SOLUTION

The binary files mentioned in this paper are executable files generated by compiling the source code. The binary code is the bytecode that can be directly executed by the CPU after the compiler compiles the source code. Hazard APIs are functions that may cause program vulnerabilities due to improper use. So, this paper only discusses binary vulnerabilities that have incomplete parameter constraints of hazard API and cause problems for devices.

The binary vulnerability similarity detection method based on function parameter dependency is shown in Figure 2. Given an unknown binary file, the hazard API name and parameter slice are extracted after processing, which is called hazard API Function Union. Then the semantic vector is extracted from the parameter slice using the neural network model.

First, obtain the hazard API name and parameter slice semantic vector of the hazard API Function Union from the vulnerability library and the unknown binary file. If Function Union with the same hazard API name, then compute the distance between the two-parameter slices. Once the threshold is exceeded, and finally, obtain the vulnerability report of the unknown binary file.

Extraction of Hazard API Union

Function Union (for short FU): $FU = \left(FuncName, ParamSli\right)$, Where *FuncName* is the API name and *ParamSli* is the parameter slice. Parameter slicing is a set of instructions that form a data dependency relationship with a hazard API in a binary procedure.

Given a binary file, obtain all hazard APIs used in each binary procedure and the instructions related to the processing of hazard API parameters. The processing required for each binary procedure is shown in Figure 3.

Data Preprocessing

Even the same software will generate different binary codes under different system architectures, different compilers and different optimization levels. Direct analysis of decompiled assembly instructions will bring great errors. Data preprocessing refers to converting instructions of different architectures into a unified intermediate language and then using the compiler back-end optimizer to optimize binary procedures.

If a binary file is not processed, the bytecode that can be directly executed by the CPU is directly obtained. The machine code translation process can convert the bytecode into assembly instructions. However, different CPUs of different architectures use different assembly instruction systems. The optimization of the compiler will also change the calculation order of the generated assembly instructions.

Figure 2. Binary vulnerability detection process

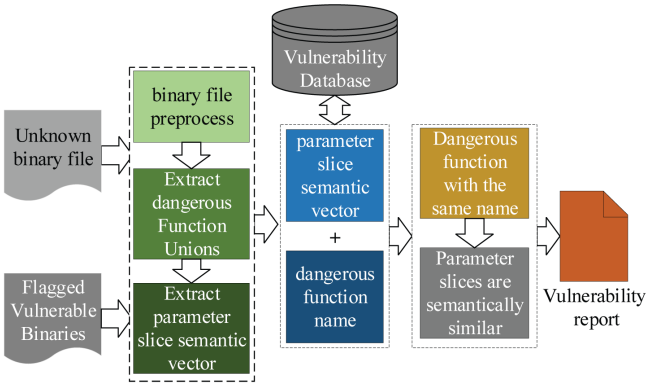
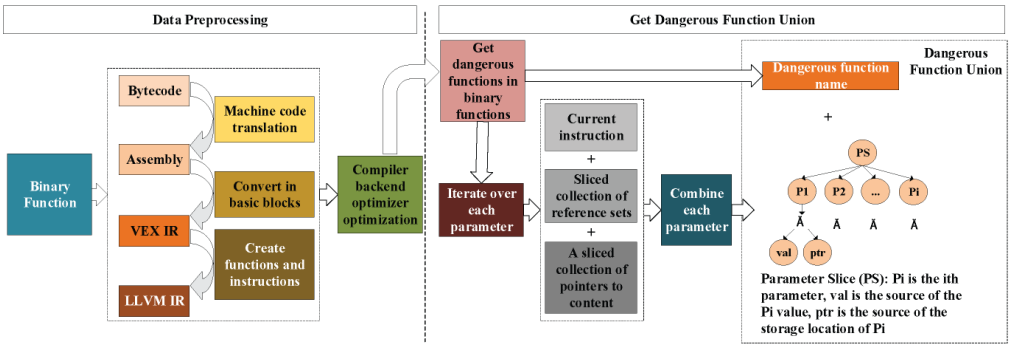


Figure 3. The acquisition process of the hazard API union



VEX IR has excellent compatibility with Various CPU architectures and operating systems. ComFU relies on the VEX binary translation engine of the PyVEX (Shoshitaishvili et al., 2015) tool. Using VEX IR as a transition can well shield the information of the binary file running environment. However, the design of VEX IR instruction is to show the impact of each machine's instruction on the machine, which makes its data dependency analysis very complex. To solve this problem, we leverage (David et al., 2017) converting VEX IR into LLVM IR. LLVM IR is an intermediate language used by the backend of the clang compiler and has SSA (Static Single Assignment Form) attribute. It can also directly use the backend of the clang compiler to optimize the code.

Extracting function parameter dependencies needs to determine the parameters and return values of the function, and recreate the function call instruction during the conversion process. The corresponding parameter transfer instruction can be determined through the calling convention, such as the x86 architecture of Intel __cdecl, which specifies that stacks are used to transfer parameters and EAX registers store return values. AMD64 generally uses SystemV x64 as the calling convention, which specifies that RDI, RSI, RDX, RCX, R8, and R9 transfer parameters, and the stack is used to transfer parameters if there are too many parameters.

The conversion process of a function call is shown in Table 1. According to the calling convention, the push statement in the assembly instruction is a parameter passing instruction, which means to store data into the stack and move up the top pointer of the stack. After converting VEX IR, offset=24 means the top pointer of the stack. By detecting the change of the pointer, three parameters are added to the Call instruction, namely t18, t17, and t14.

Different optimization strategies may be used when compiling binary files. Using the compiler back-end optimizer to re-optimize the obtained LLVM IR instructions can ensure that all binary files ultimately use the same optimization rules, and effectively solve the semantic gap caused by compilation options of different architectures and different optimization levels. We adopt O3 optimization, including most optimization strategies, such as the optimization of code branches, constants, and expressions. The role of re-optimization is described in detail in the document (David et al., 2017). The conversion process of instructions is shown in Table 1. The three parts represent assembly instructions, VEX IR and optimized LLVM IR respectively. It can be seen that the optimized LLVM IR instructions will be very compact.

Get Hazard API Function Union

As shown in Figure 2, by analyzing the existing vulnerabilities, a list of dangerous functions can be obtained. Traverses each normalized binary procedure and locates the hazard APIs in all binary procedures. After that, the FU is extracted from each hazard API in the binary procedure.

Table 1. Function call conversion process

Assembly Instruction	.text:080487DF push [ebp+var_C] .text:080487E2 push [ebp+var_10] .text:080487E5 push offset aLld ; "%lld\n" .text:080487EA call _printf
VEX IR	t44 = GET:I32(offset=24) t43 = Sub32(t44,0x00000004) t18 = t43 PUT(offset=24) = t18 eax=Call(printf,ob0.t18,ob0.t17,ob0.t14)
LLVM IR	%.50 = load i32, i32* %.38, align 4 %.53 = tail call i32 @"00dbe45702b24.Eprintf.3"(i32%.50, i32%.51, i32 134516102)

The extraction of hazard API consortia needs to obtain the API name and parameter slice of the hazard API in a binary procedure. The extraction process of the parameter slice is to trace the parameters of the hazard API in the binary procedure and generate a slice set. The algorithm (David et al., 2020) for generating slice sets of each parameter in binary procedures are designed as follows.

Assuming that the hazard function has a parameter *A*, for the instruction *n* assigned to the variable $A = {}^{*}B + C$. The slice set of variable *A* in $prede(n)$ is $S_{prede(n),A}$, Specifically expressed as:

$$S_{prede(n),A=\{n\}} \cup \left(S_{prede(n),V_1} \ldots \cup S_{prede(n),V_i} \ldots \forall V_i \in refs(n)\right) \cup \left(S_{prede(n),P_1} \ldots \cup S_{prede(n),P_i} \ldots \forall P_i \in points(n)\right) \quad (1)$$

wherein, the parameter *A* of the function is the starting variable of slice generation, and $prede(n)$ represents the set of subsequent instructions of instruction *n*, $S_{prede(n),V_i}$ represents the slice set of variables V_i in $prede(n)$.

$refs(n)$ is the read variable in instruction *n* (e.g. {*B, C*}), $points(n)$ is a set of addresses that can be pointed to at the instruction n (e.g. $\left\{{}^{*}B\right\}$), If there is an instruction in $prede(n)$ to write the contents pointed to by the pointer type variable in the instruction *n*, the variable storing the contents pointed to by the pointer type is the address that can be pointed to at the instruction *n*. The pointer type variable refers to a variable storing a memory address.

It can be seen from Formula (1) that the slice set of a parameter is the union of the current instruction and the two instruction sets, i.e. the current instruction *n*; The slice set referencing the base variable in instruction *n*; A pointer type variable points to a collection of slices of content. For example, the basic variable referenced in the *aaf = load aac* instruction is *aac*, and the instruction *aac = inttoptr aae* with *aac* as the value is finally obtained upward. The basic variable referenced in the instruction *aac = inttoptr aae* is *aae*, from which the instruction with *aae* as the value (*aae = add aad, -176*) and the instruction with *aae* pointing to the content assignment (*store aaa, aae*) can be obtained.

The slice set structure of the parameter is a tree structure as shown in Figure 4. If the parameter *source* is a memory variable (for example, a temporary variable will exist in the memory), the slice contains the source of the pointer (*aa = call malloc(ab)* in the Figure 4) and the source of the content at the address (*ae = add ad, -176*).

The slice set of each parameter is taken as a branch of the tree, and the parameter slice of the danger function is obtained after the combination. The FU in Figure 5 shows the final result of

parameter slicing. *PS* is an empty node, and *Pi* and its lower branches are the slice set of the ith parameter, as shown in Figure 4.

Vector Representation of Function Union

To facilitate the comparison of function union, we design a representation method of FU based on binary similarity and converts its function union into vectors. Because the API name can distinguish different functions, this method first matches the API name exactly. According to (1), the function parameter slice is a tree structure. Therefore, we first use the depth-first traversal algorithm of the tree to convert the parameter slice into a sequential structure and name it as a parameter sequence, and then uses NLP(Natural Language Processing) technology to obtain the semantic vector of the parameter slice. The semantic extraction process of the parameter slice is shown in Figure 5.

Word Expression

The first step to obtaining the semantics of the parameter sequence is to obtain the embedding vector of each word. That is, to map the semantics of each word to different dimensions of the vector. Instructions in parameter sequences cannot be directly embedded because it contains many simple operators (such as "i32", "()") and low-frequency words (i.e., large numeric constants, memory addresses, and binary procedure names) as well as variable names (such as *%r4, sl.1.reg2mem*, etc.). The main function of the operator is to improve the readability of the code. The value of low-frequency words is random and has little relationship with the function and semantics of the program, but it will expand the corpus and affect the learning effect of the model, so we delete it simply. The system uses tags to replace low-frequency words of the same type. The corresponding relationship between low-frequency word types and tags in the abstract rules is listed in Table 2. The function of variable names is to increase readability. To the number of words in the dictionary, the variable names in a function union are replaced uniformly, that is, in a function union, each variable name is recoded according to the number of variable occurrences, specifically *aaa*, *aab*, *aac* ...

To further reduce the size of the thesaurus, the LLVM IR instruction is segmented according to the structure described in (2). The *call* command may contain many params, only one is reserved in this article, and other parameters are deleted directly, for example, *aaa = call strncmp(aab, MEM,*

Figure 4. Slice of the first parameter of the "memcpy" function

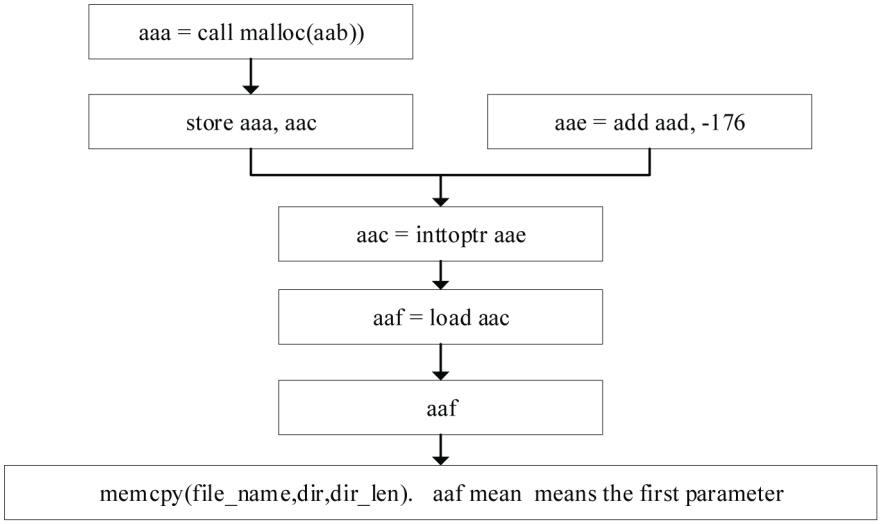


Table 2. Low-frequency vocabulary comparison table

Low-Frequency Vocabulary Types	Label
Large value constant	Num
memory address	MEM
binary procedure name	FUNC

3), this paper takes call as opcode, *"strncmp"* as param1, and variable *aab* as param2. The remaining variables (*MEM* and *3*) are deleted directly. Since the parameter slice contains the information of all parameters, the deletion here will not affect the semantic expression of the call instruction:

$$value = opcode\ param1, param2 \tag{2}$$

The *opcode* shown in (2) is the just *opcode* in the instruction, and *param1*, *param2,* and value are all operands in the instruction. It is necessary to train the *opcode* and the *operand* separately. Specifically, each *operand* or *opcode* is taken as a word, and the word in each function union is taken as a sentence, so Word2vec (Mikolov et al., 2013) models were trained respectively.

Semantic Representation of Parameter Sequence

After the embedding of each word is obtained, the embedding vector of each word in the parameter sequence needs to be convoluted in turn, and finally, a vector containing each instruction and its location information is obtained, which contains the semantics of the parameter sequence. As shown in Figure 5, the word embedding vectors of operands and opcodes are spliced according to (2), and the embedding vector of each instruction is I_i. To further describe the semantics of the parameter slice, this paper uses the Bi-LSTM (Zhou et al., 2016) (Bidirectional Long Short Term Memory) network to obtain the semantic vector O_i propagating forward and backward respectively. The semantic vector representation V of the parameter slice of the function union is obtained after the full connection layer fusion.

LSTM (Shi et al., 2015) (Long Short-Term Memory, LSTM) network is a kind of RNN (Zaremba et al., 2014) (Recurrent Neural Network). Due to its unique design structure, LSTM is suitable for processing and predicting important events with very long intervals and delays in time series. At the same time, forward and backward propagation two-way LSTM networks contain more order information, so the order of the parameter sequence has been learned that which is more important for the comparison of binary similarity.

Get Parameter Sequence Semantic Model

The process of obtaining function semantic vectors is shown in Figure 5. The Word2vec model with pre-trained instructions in the parameter sequence completes word embedding. After instruction splicing, the pre-trained neural network model obtains the semantic vector representation of the parameter slice. To generate the semantic embedding vector of the parameter sequence, it is also necessary to train the network described in Figure 5.

The specific method is shown in Figure 6. Compile the source code of the same file and the same function with different compilers and different optimization levels, extract the corresponding hazard API union from the compiled binary procedure, set the label as 1, randomly extract different hazard API union, and set the corresponding label as 0.

The corresponding hazard API union calculates the semantic vector V through two bidirectional LSTM networks with public weights, and uses the cosine distance to calculate the similarity. Then

Figure 5. Parameter slice semantic vector extraction process

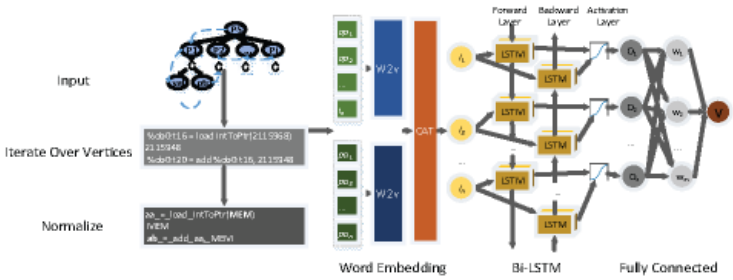
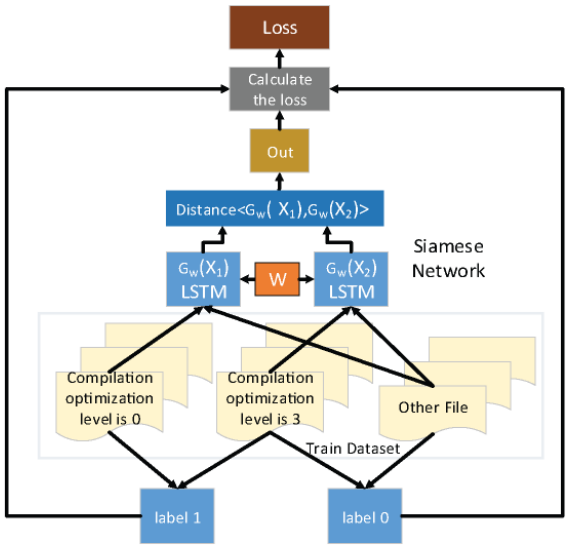


Figure 6. Neural network training process



uses the cross-entropy algorithm to calculate the loss between the comparison result and the label. And finally uses the optimizer Adam (Adaptive Moment Estimation) algorithm is optimized. After adjusting the model parameters, the stochastic gradient descent algorithm is used to continue iterating, and the ComFU model training is completed.

After the neural network in Figure 5 goes through the training process shown in Figure 6, the input parameter slice sequence can obtain the semantic vector *V* containing the complete semantics of the parameter slice of the FU.

Compare Function Semantic Vectors and Report Vulnerability

As shown in Figure 2, manual methods are used to mark the vulnerabilities existing in the binary file. The marked vulnerability binary file is processed to obtain the semantic vector of the API name and parameter slice of the hazard API FU, and the API name and semantic vector are stored in the vulnerability library.

Given an unknown binary file, extract a hazard API FU, compare them with the hazard API FUs in the vulnerability library, and finally form a vulnerability report of the binary file.

We believe that the hazard API FU is completely different if the hazard API name is different, so the parameter slice matching will only be performed if the hazard API name is the same.

Parameter slices cannot be directly compared. In this paper, the neural network trained in Figure 6 is used to encode the obtained parameter slices of the hazard API union, and the semantic embedding vector corresponding to the parameter slices is obtained. The semantic similarity of parameter slices can be expressed using the cosine of two vectors.

EXPERIMENTAL

This paper conducts experiments on the following platforms with 64-bit Ubuntu16.04, Intel Core E5- 2650 CPU, NVIDIA Tesla P100 GPU card, and 64GB memory, and the development language use python.

The evaluation is carried out from three aspects: model evaluation, learning model evaluation, and real-world application scenarios.

Binwalk (ReFirmLabs et al., 2023) is one of the most commonly used firmware analysis tools. In the process of firmware analysis, the tool can well identify the firmware compression format and file system, and then parse and extract the files.

GitZ converts binary functions into LLVM IR and uses the LLVM IR back-end optimizer for instruction re-optimization. Due to its early implementation, it does not use a neural networks for similarity comparison. This paper improves it, uses the Bi-LSTM network for training and comparison, and uses this model as a baseline.

Dataset 1: Compiled 30 open source projects with different compilers with *O2* and *O3* optimized levels to get X86, ARM, and MIPS architectures binaries. This paper randomly selects 10 binary files obtained from each open-source project, after that, 683379 function unions were extracted from 300 binary files. The function combination compiled with the same source code is used as a positive sample, and the function combination compiled with non-identical source code is randomly selected as a negative sample, which is used for model training.

Dataset 2: 10 groups of function unions are randomly selected from dataset 1, each group composed of 3 function unions compiled from the same source code. One is randomly selected from each group as the source and the rest are used as the target. Each source is combined with the function union of the group to obtain a positive sample, and the combination of other groups of functions is combined to obtain a negative sample. In the end, 20 pairs of positive samples and 180 pairs of negative samples were obtained as model evaluation datasets.

Dataset 3: Manually search for CVE entries, collect vulnerable binaries from firmware containing the CVE, and annotate the address where the vulnerability occurs. We collect 8 vulnerabilities from five vendors, including 6 router vulnerabilities and 2 camera vulnerabilities.

Dataset 4: Contains 8 firmware where the vulnerabilities in Dataset 3 are located. And 14 different versions of the firmware of the device where the vulnerability is located. After manual verification, 12 firmware contained vulnerabilities in the vulnerability library, and 2 firmware were versions after the vulnerabilities were repaired. Unzip and extract all the binary files in the firmware.

Dataset 5: Compile 5 open source projects with different compilers to get X86, ARM, and MIPS architectures, and binaries of O2 and O3 optimization levels. 10 binary files were randomly selected from the binary files obtained from each open-source project, and a total of 50 binary files were obtained, from which 82,443 binary procedures were extracted as the training and test sets of the comparative experiments.

The following concepts need to be explained before the experiment starts:

- **True Positive (TP):** Predict the positive class as the number of positive classes.
- **True Negative (TN):** Predict the negative class as the number of negative classes.
- **False Positive (FP):** Predict the negative class as the number of positive classes, that is, false positives.
- **False Negative (FN):** Predict the positive class as the number of negative classes, that is, false negatives.
- **Accuracy(acc):** $acc = \left(TP + TN\right) / \left(TP + FP + FN + TN\right)$
- **False Positive Rate (FPR):** $FPR = FP / \left(FP + FN\right)$
- **True Positive Rate (TPR):** $TPR = TP / \left(TP + FN\right)$

Model Evaluation

This paper evaluates and compares the models using accuracy and TPR, respectively.

First, Dataset 1 is divided into a training set, verification set, and test set according to 8:1:1, respectively, for training the ComFU model. Since the scale of binary function instructions is much larger than that of function unions, this paper reduces Dataset 1 to obtain Dataset 5, uses Dataset 5 to train GitZ, and compares the ROC curves of the two models after training.

The results are shown in Table 3. In terms of similarity matching, both models have high accuracy, which shows that using re-optimized LLVM IR instructions for similarity comparison works well. The accuracy of GitZ is slightly higher. The reason may be that the CFG-based detection mechanism contains more information, but the extracted CFG semantic information cannot be interpreted, and more messy information will cause higher false positives.

In the case that the accuracy rate of the two is not much different, this paper further compares the TPR of the two models. Use the GitZ model and the ComFU model to predict the data selected in Dataset 2, respectively, and the prediction results are sorted by similarity. Select K samples from the sorting results from each time to calculate TPR (recall rate).

As shown in Figure 7, when ComFU selects Top 40, the recall rate of the ComFU model has reached more than 90%, while the recall rate of the GitZ model at this time is only Less than 80%. GitZ predicts that all positive samples need to select the Top 180, while ComFU only needs to select the Top 60 to predict all positive samples.

Scenario Variable Assessment

Figure 8 depicts the impact of the dataset size on the model. As the dataset size increases, the accuracy of the model gradually increases. After the model size reaches 500,000, the increase in the dataset size has little effect on the accuracy of the model.

We tested the ROC curves of different models on the test set, the abscissa is the FPR, and the ordinate is the TPR. The model already has a higher TPR with a lower FPR. We also tested the effect of different learning models on ROC as shown in Table 4, learning model of LSTM adding and not adding the Attention (Bahdanau et al., 2014) mechanism has little effect on the model, and the LSTM model is better than the RNN model.

Real-World Application

We tested the model in real scenarios. Select a vulnerability function union in Dataset 3 to match all function unions collected in Dataset 4. For the convenience of description, they are called source

Table 3. Comparison of model accuracy

Scheme	acc(%)
GitZ	0.982
ComFU	0.973

Figure 7. The relationship between the number of selected samples and the recall rate

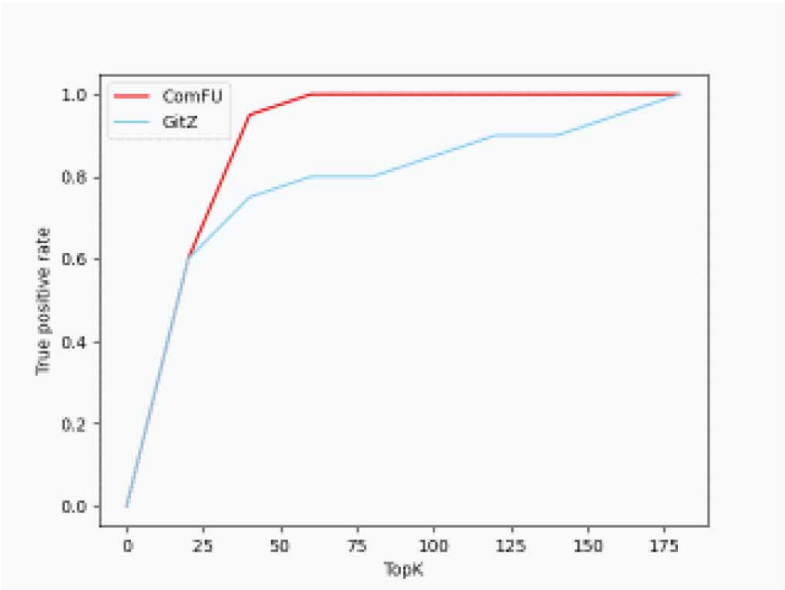
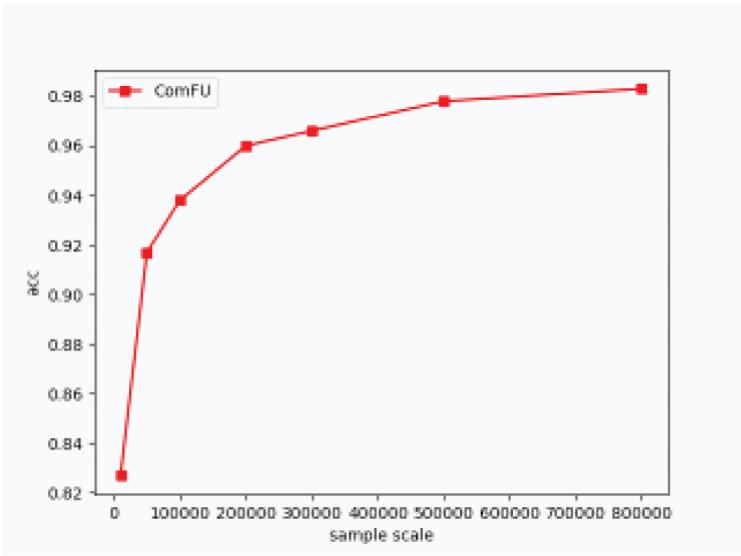


Figure 8. The effect of dataset size on the model



and target, respectively. After filtering, the target matching result shown in Figure 10 is obtained. The abscissa is the unique identifier of the target function union in the database, and the ordinate is the similarity between the target and source function unions. It can be seen that only a few points have high similarity with the vulnerability to be tested, and the ComFU model can filter dissimilar function unions on a large scale.

Figure 9. Comparison of matching effects of different models

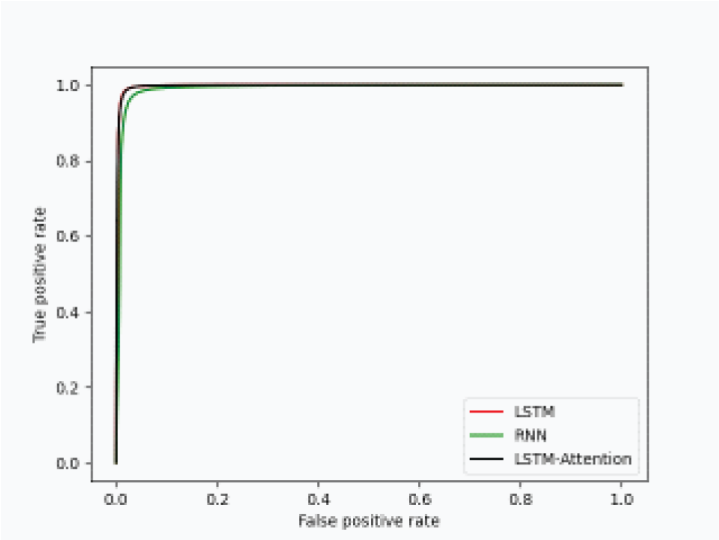
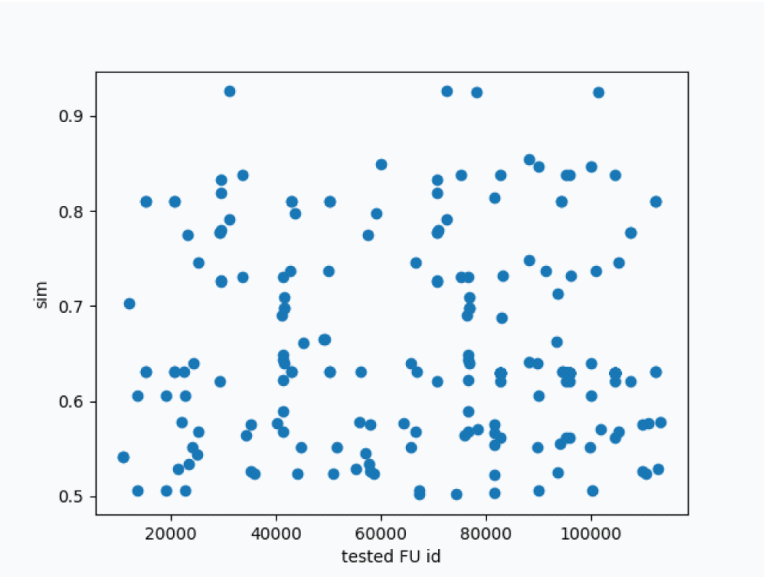


Table 4. Relationship between model type and accuracy

Model	acc
RNN	0.957
Bi-LSTM	0.973
Bi-LSTM-Attention	0.970

Figure 10. Results of a vulnerability comparison



The objective function union described in Figure 10 is sorted according to the similarity from high to low, and the two most similar points to the vulnerability function union are found. The decompilation result is shown in Figure 11. The left picture shows the function union matched in the target and the source from the same binary file, and the right picture shows another version from the target that is similar to the source function union. Although the variable names and other functions called in the left and right pictures have changed, there is no difference in the use of the vulnerability function "sprintf". It is confirmed that the vulnerability does exist.

Further, we evaluate the usability of the model in real-world scenarios.

Dataset 3 is used as the vulnerability database, and Dataset 4 is used as the target database. The function union obtained from the target library is compared with the function union in the vulnerability library, and the comparison results are sorted from high to low in similarity, and the top K target function unions similar to the vulnerability function union are taken as TOPK.

Since different versions of firmware may have very few updated files, so will contain binary files compiled from the same source code, and the model matching results will report multiple similar function unions in these different binary files. To remove this error, in this paper When calculating TOPK, similar function unions in multiple different firmware will only be counted once.

MTOPK means that when the matching result is TOPK, all function complexes similar to the vulnerability function complex have been matched, and -1 means that all results have not been found when TOPK is 20. Patched indicates whether the target library contains the patch for the vulnerability, and MPath indicates whether the patch is included in the result when TOPK is set to 20.

The matching results of the vulnerabilities are shown in Table 5. Among the 8 vulnerabilities in the experiment, number 7 of them found all binary files containing similar vulnerabilities within the *TOP5*. Moreover, the three vulnerabilities matched all similar vulnerabilities in TOP1, which means that these vulnerabilities have no false positives. It proves that the false positive rate of ComFU is very low when detecting similar vulnerabilities.

To prove that ComFU has a certain ability to identify patches, Dataset 4 includes firmware with 2 vulnerabilities (CVE-2017-8411 and CVE-2018-1156) that have been fixed. In Table 5, the similar vulnerabilities of the two vulnerabilities are matched, and the binary files containing the patch are all identified. Because ComFU first compares FuncName when performing similarity comparison, if the patch repair method is to modify the API name, such patches must be able to distinguish. However, the vulnerability database does not have enough data to fully prove that other patches can also be effectively detected.

Vulnerability number CVE-2013-4659 has a similar function combination in the vulnerability library, but it has not been detected because the trigger position of the CVE-2013-4659 vulnerability is not within a binary function. The current method only traces the hazard function parameter

Figure 11. Comparison of matching results

```
...
v18 = inet_addr(&byte_121B8[v17 + 32]);
v19 = (unsigned __int8)v16[169];
v16 += 480;
sprintf(
  &v31,
  "<%d>%s>%s>%s>%d>%s>%s>%d",
  3,
  ……
  &byte_121B8[v17 + 120],
  &byte_121B8[v17 + 64],
  v19);
v20 = strlen((const char *)&v27);
...
```

(a)Vulnerability decompilation results in the vulnerability database

```
...
v12 = inet_addr(&searchRouterInfo[480 * v11 + 32]);
sprintf(
  v24,
  "<%d>%s>%s>%s>%d>%s>%s>%d",
  3,
  ...
  &searchRouterInfo[480 * v11 + 120],
  &searchRouterInfo[480 * v11 + 64],
  *v10);
v13 = strlen(v24);
...
```

(b) Matched vulnerability decompilation result

Table 5. Statistics of vulnerability matching results

Vulnerability ID	MTOPK	Patched	MPath
CVE-2013-4659	-1	No	No
CVE-2016-6277	1	No	No
CVE-2017-8411	1	Yes	No
CVE-2018-1156	3	Yes	No
CVE-2018-14712	4	No	No
CVE-2020-13395	1	No	No
CVE-2020-13395	3	No	No
EDB-44001	1	No	No

dependencies of a single binary function, exploring the vulnerability detection mechanism caused by cross-functions, and alleviating the path explosion problem caused by cross-function variable tracing is the next research work of this paper.

Limitation

The above experiments demonstrate the usability of ComFU in terms of model accuracy and vulnerability matching accuracy. Experiments have proved that compared with the same type of methods, the ComFU model has a lower false positive rate, and has a lower false positive and false negative rate for similar vulnerabilities in actual vulnerability detection scenarios. However, since the extraction of the function union is in the binary function, the detection accuracy of some vulnerabilities is low. However, since the extraction of the function union is within the binary function, the effect on complex cross-function vulnerabilities is poor. At the same time, the extraction of function unions depends on function calls, and it is powerless to detect vulnerabilities without dangerous functions.

CONCLUSION

The trigger of a software vulnerability is not the whole binary function code, but a segment of it. Based on this observation, we study the similarity detection of vulnerabilities caused by the unconstrained use of hazard APIs and parameters, and proposes a binary vulnerability similarity detection model ComFU. Our experiment shows that the accuracy of the model is up to 97%, and the recall rate is 14.3% higher than the baseline model. In the real scene vulnerability experiment, the model is sensitive to the change of vulnerability repair and can reduce the false alarm rate of binary vulnerability detection. In our experiments, we also found that the current method only tracks the hazard API parameter dependence of a single binary function. Therefore, it is the next step to explore the vulnerability detection mechanism caused by cross-function and alleviate the path explosion problem caused by cross-function variable tracking.

ACKNOWLEDGMENT

Project supported by the Advanced Industrial Internet Security Platform Program of Zhijiang Laboratory (No.2018FD0ZX01), the National Natural Science Foundation of China (Nos.61802435 and 61802433).

REFERENCES

Alrabaee, S., Shirani, P., Wang, L., & Debbabi, M. (2015). SIGMA: A Semantic Integrated Graph Matching Approach for identifying reused functions in binary code. *Digital Investigation*, *12*(Supplement 1), S61–S71. doi:10.1016/j.diin.2015.01.011

Bahdanau, D., Cho, K., & Bengio, Y. (2014). *Neural Machine Translation by Jointly Learning to Align and Translate.* CoRR, abs/1409.0473.

David, Y., & Yahav, E. (2014). Tracelet-based code search in executables. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. doi:10.1145/2594291.2594343

David, Y., Alon, U., & Yahav, E. (2020). Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages, 4*, 1-28. doi:10.1145/3428293

David, Y., Partush, N., & Yahav, E. (2017). Similarity of binaries through re-optimization. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* doi:10.1145/3062341.3062387

Gui, Y., Wan, Y., Zhang, H., Huang, H., Sui, Y., Xu, G., Shao, Z., & Jin, H. (2022). Cross-Language Binary-Source Code Matching with Intermediate Representations. *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER),* 601-612. doi:10.1109/SANER53432.2022.00077

Mikolov, T., Chen, K., Corrado, G. S., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *International Conference on Learning Representations*.

Pewny, J., Garmany, B., Gawlik, R., Rossow, C., & Holz, T. (2015). Cross-architecture bug search in binary executables. *IT - Information Technology, 59*, 83 - 91.

Qiu, J., Su, X., & Ma, P. (2015). Library functions identification in binary code by using graph isomorphism testings. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER),* 261-270.

ReFirmLabs. (n.d.). *ReFirmLabs/binwalk: Firmware analysis tool*. GitHub. Retrieved January 31, 2023, from https://github.com/ReFirmLabs/binwalk

Shi, X., Chen, Z., Wang, H., Yeung, D., Wong, W., & Woo, W. (2015). *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting*. NIPS.

Shoshitaishvili, Y., Wang, R., Hauser, C., Krügel, C., & Vigna, G. (2015). Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. *Network and Distributed System Security Symposium*. doi:10.14722/ndss.2015.23294

Xia, B., Pang, J. M., Zhou, X., & Shan, Z. (2022). Research progress of binary code similarity search. *Computer Applications (Nottingham)*, (4), 985–998.

Xiao, Y., Chen, B., Yu, C., Xu, Z., Yuan, Z., Li, F., Liu, B., Liu, Y., Huo, W., Zou, W., & Shi, W. (2020). MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. *USENIX Security Symposium*.

Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., & Song, D. X. (2017). Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. doi:10.1145/3133956.3134018

Xu, Y., Xu, Z., Chen, B., Song, F., Liu, Y., & Liu, T. (2020). Patch based vulnerability matching for binary programs. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. doi:10.1145/3395363.3397361

Yang, X. (2019). *Binary vulnerability code clone detection based on semantic learning* [Master's thesis]. Tsinghua University. https://kns.cnki.net/KCMS/detail/detail.aspx?dbname=CMFD202101&filename=1020829305.nh

Zaremba, W., Sutskever, I., & Vinyals, O. (2014). *Recurrent Neural Network Regularization*. ArXiv, abs/1409.2329.

Zheng, J. Y., Pang, J. M., Zhou, X., & Wang, J. (2021). Enhanced Binary Vulnerability Mining Based on Constraint Derivation. *Computer Science*, (3), 320–326.

Zhou, P., Shi, W., Tian, J., Qi, Z., Li, B., Hao, H., & Xu, B. (2016). Attention-Based Bidirectional Long Short-Term Memory Networks for Relation Classification. *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (*Volume 2*: Short Papers).* doi:10.18653/v1/P16-2034