Journal on Internet of Things
DOI: 10.32604/jiot.2022.019469
Article

Tech Science Press

FirmVulSeeker—BERT and Siamese Network-Based Vulnerability Search for Embedded Device Firmware Images

Yingchao Yu*, Shuitao Gan and Xiaojun Qin

State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, Jiangsu, 214083, China
*Corresponding Author: Yingchao Yu. Email: yuych830305@163.com
Received: 15 November 2021; Accepted: 15 February 2022

Abstract: In recent years, with the development of the natural language processing (NLP) technologies, security analyst began to use NLP directly on assembly codes which were disassembled from binary executables in order to examine binary similarity, achieved great progress. However, we found that the existing frameworks often ignored the complex internal structure of instructions and didn't fully consider the long-term dependencies of instructions. In this paper, we propose firmVulSeeker—a vulnerability search tool for embedded firmware images, based on BERT and Siamese network. It first builds a BERT MLM task to observe and learn the semantics of different instructions in their context in a very large unlabeled binary corpus. Then, a finetune mode based on Siamese network is constructed to guide training and matching semantically similar functions using the knowledge learned from the first stage. Finally, it will use a function embedding generated from the fine-tuned model to search in the targeted corpus and find the most similar function which will be confirmed whether it's a real vulnerability manually. We evaluate the accuracy, robustness, scalability and vulnerability search capability of firmVulSeeker. Results show that it can greatly improve the accuracy of matching semantically similar functions, and can successfully find more real vulnerabilities in real-world firmware than other tools.

Keywords: Embedded device firmware; vulnerability search; BERT; siamese network

1 Introduction

In recent years, the open-source communities have been expanded rapidly, accelerating the design and development of the software, also the spread of vulnerabilities. Especially in the field of embedded devices, due to the cost, device manufacturers usually reuse large number of third-party components (from open-source community, historical codebase, the third-party companies, etc.) to smoothly port the software programs on mature x86 terminal devices to platforms such as ARM or MIPS. All of these will lead to the same manufacturer's different types of products and even different models of different products are likely to be influenced by the same vulnerability in the reused component, while the user who uses the product (even security analyst) has no clear knowledge about the internal relationships

of the product's firmware, which undoubtedly aggravates the spread of the potential vulnerabilities. Therefore, how to use know vulnerabilities to detect whether there is a homologous vulnerability in other devices' firmware across platforms becomes more and more important [1]. However, it is a very challenging task. Compared with the vulnerability detection on common desktop applications, it faces the following challenges [2]: 1. The source code of the firmware is usually unavailable. 2. The ISA, the compiler and the optimization level used on the firmware are usually different, so given the same source code, it will get different binary executables whose syntax and structure characteristics are different and we can't easily get the knowledge about which compiler and optimization level it used. To address this challenge, cross-architecture binary firmware vulnerability search has been received more and more attention in recent years, and many well-performed methods have been proposed [3–7].

Existing methods feed artificially designed features to deep learning models, such as Gemini [3], Vulseeker [4] etc., or use some representation learning model to automatically general a vector representation for similarity comparison, such as InnerEye [5], SAFE [6], OrderMatters [7], etc. Especially in recent years, researchers began to use NLP technologies to process assembly language directly, because there are many similar characteristics shared by natural language and assembly code. For example, InnerEye [5] and Asm2Vec [8] both use word2vec [9] model as a preprocessing model to generate instruction embedding, on top of which LSTM [10] or RNN [11] will be used to generate basic block embedding or function embedding to represent basic block or function semantics. However, as explained in Section 2.2, existing methods usually ignore the complex internal structure of instructions, which just use the statistical characteristics of the instructions (e.g., Geimini treats the constant number as one feature) or treat the entire instruction as a token (e.g., InnerEye, SAFE, etc.) or just consider some simple instruction format (e.g., Asm2Vec treats an instruction as an opcode plus a combination of up to two operands) or ignore the position of the instruction (e.g., OrderMaters), therefore, they can't model the dependencies within and between instructions especially the long-range dependencies, further can't better capture the semantic information of the binary code (basic block or function). On the other hand, existing methods are mostly supervised learning, whose performance seriously depend on the quality of the training data. But in the binary similarity detection especially in the embedded firmware vulnerability detection, collecting a large, representative and balanced training data set is very difficult, and there may be overfitting problem in supervised learning [12].

In this paper, a BERT MLM [13] and Siamese network [14] based vulnerability search tool for embedded firmware images is proposed, which is named firmVulSeeker. The core idea is to train a BERT MLM task to observe and learn the semantics of different instructions in their context firstly. At this stage, it takes the original instruction sequence without the need to label the training samples, so it is an unsupervised learning. Then, a fine-tune model based on Siamese network is constructed, utilizing the knowledge from the first stage to guide training and matching semantically similar functions. Finally, given a vulnerability function, to search it in the target corpus. The contributions are as follows:

1. We propose firmVulSeeker—a three-staged embedded device firmware vulnerability search tool based on BERT and Siamese network. In the first stage, we will use a BERT MLM task to train the original instruction sequences disassembled from binary function to learn the semantic information of the instructions and the dependencies among instructions. In the second stage, we construct a fine-tune model based on Siamese network to find-tune the pretrained model from the first stage. In the third stage, we search the vulnerability function in the target corpus based on its function embedding generated from the model generated from the second phase.

2. We implement firmVulSeeker and evaluate its accuracy, robustness and scalability. The results show that firmVulSeeker is superior to SAFE and Gemini in the accuracy of the evaluation dataset in this paper, and firmVulSeeker can deal with the situation across architecture, across optimization options and across projects. Also, when we use the firmVulSeeker pretrain model to other similar tools, they can significantly improve their function similarity matching accuracy.
3. We evaluate the vulnerability search capability of the firmVulSeeker on real-world embedded device firmware images with 14 CVEs (Common Vulnerability & Exposures) and the results show that it can successfully search 13 CVE vulnerability functions in the 70 embedded device firmware image datasets, and the accuracy is higher than SAFE and Gemini.

2 Problem Statement

2.1 Embedded Device Firmware Vulnerability Based on Binary Code Similarity Analysis

Given a binary vulnerability function of interest (affected by a CVE, such as a binary function containing the Heartbleed vulnerability [15]), at function granularity, we hope to analyze a very large binary function corpus (a large number of binary functions disassembled from binary executables which are extracted from different embedded device firmware images with firmware extraction tools such as Binwalk), identify semantically equivalent or similar candidates (suspicious vulnerable functions) quickly and accurately, and then confirm whether they are vulnerable or not by artificial or static or dynamic means. Here, we name the interested binary vulnerability as query and the binary function corpus as target corpus. Therefore, the problem is translated to find in the target corpus the most similar function as the query function (which means binary code similarity analysis). This process is shown in Fig. 1.

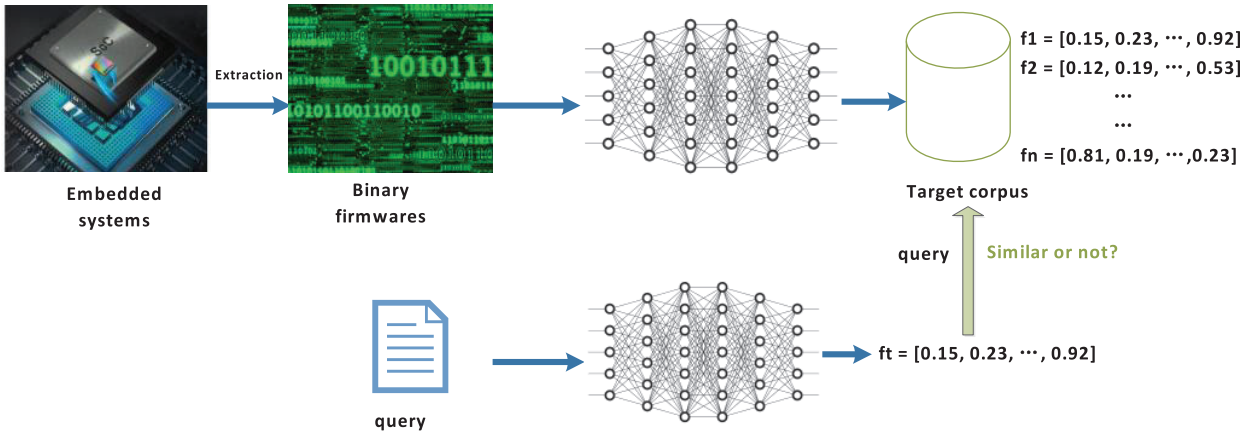

Figure 1: embedded device firmware vulnerability search overview

2.2 Summary of Existing Binary Similarity Analysis Methods Based on NLP and Their Shortcomings

Because of many similar characteristics shared by natural language and assembly code, in recent years, researchers have begun to use NLP technology directly on assembly code to analyze binary code similarity. Tab. 1 summarizes and compares the existing popular binary similarity analysis methods based on NLP, with respect to token granularity, context range, encoding model and whether instruction internal structure is considered.

Table 1: Summary of existing binary analysis methods based on NLP

Methods	Tokens	Context	Encoding	Internal structure
Asm2Vec [8]	opcode, operand	control flow	PV-DM [16]	partially
InnerEye [5]	entire instruction	slide window	word2vec	no
MIRROR [17]	opcode, operand	slide window	NMT [18]	partially
SAFE [6]	entire instruction	slide window	skip-gram [19]	no
DeepBinDiff [12]	opcode, operand	control flow	derived word2vec	partially
OrderMatters [7]	entire instruction	control flow	BERT [13]	no

From Tab. 1, we find that the existing methods have more or less the following shortcomings in their embedding generation process:

First, existing methods ignore the complex internal formats of instructions or only consider some simple format of instructions. InnerEye [5], SAFE [6] and OrderMatters [7] all treat entire instruction as a token, ignoring the internal structure of the instruction. Although Asm2Vec [8], DeepBinDiff [12] and MIRROR [17] separate the opcode and operands in the instruction as different tokens, they only consider a simple internal format, for example, Asm2Vec [8] only consider up to two operands while in x86 assembly code, the number of operands can range from 0 to 3, and can be register, memory location expression, immediate constants, string symbols, and so on.

Second, existing methods are limited in modeling long-range dependencies. As Tab. 1 shows, existing methods mostly use word2vec or its derived model (such as PV-DM model used in Asm2Vec) to train the tokens within the specified slide windows. Other methods extract the instruction sequence in the CFG by means of random walk, such as Asm2Vec and DeepBinDiff. However, due to compiler optimizations, there may some noise in the context information on the control flow, which can't truly reflect the actual dependency relations between instrucitons.

To solve the above shortcomings in the existing methods, we take the original assembly instruction sequence of the function (disassembled from binary executable) as input, introduce a more fine-grained strategy to decompose the instructions (the decomposed parts are called tokens), consider the position information of tokens between and within the instructions, and then create a BERT MLM task to learn the tokens in order to get the instruction semantics and function semantics with the target of the function embedding generated by the model can fully represent the semantics of the original function in mind.

3 Design

3.1 Overview

Fig. 2 shows the overview of firmVulSeeker, which consists of three phases.

The first stage is the pretraining stage. In this paper, we use a BERT MLM task to pretrain the assembly instruction sequence to capture the complex internal semantics of instructions and the long-range dependency relations of different instructions. At this stage, we will feed the original assembly instruction sequence to the model and the output will be instruction embedding sequency. Compared to SAFE, a more fine-grained strategy is used: we take one instruction as a sentence, and break it down into basic tokens. Take the instruction "mov eax, [esp + 4]" as example, we will break it down to seven tokens: "mov", "eax", "[","esp","+","4", "]".

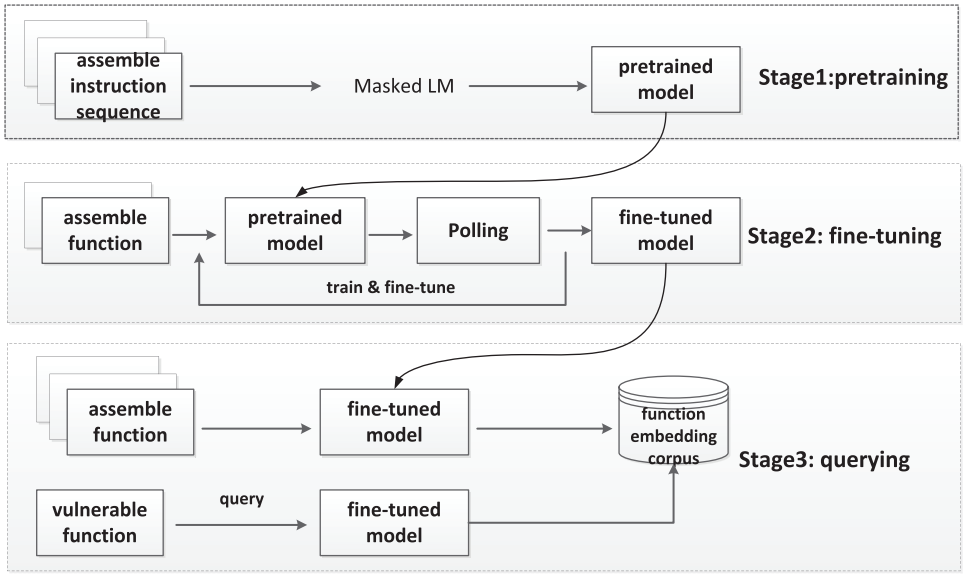

Figure 2: firmVulSeeker overview

The second stage is the fine-tuning stage, aiming to further refine the semantics learned from the first stage for different downstream tasks. Inspired by SBERT [19], we add a pooling layer to the pretrained model from the first stage to derive a fixed-size embedding as a function embedding. By fine-tuning the pooling layer and the pretraining model, the function embedding was calculated and the similarity between the two function was calculated by calculating the cosine similarity between their embeddings [21].

The third stage is the querying stage, aiming to find in the target corpus the most similar function to the vulnerability function, and confirm whether it's really a vulnerability through manual analysis or other methods. The target corpus is constructed as follows: get the firmware, extract the binary executables from the firmware, disassemble the binary executables, get all the functions, feed them into the fine-tuned model to generate the function embeddings, and store them in the dataset finally.

In the following, we will introduce the design details of firmVulSeeker, including its input representation, the pretraining model, the fine-tuning model and more.

3.2 Input Representation

Given a function f (assembly instruction sequence), we will prepare the model input x, consisting of 3 types of token sequence with the same size n:

Instruction sequence: $x_f = \{mov,\ eax, +, \ldots\}^n$, generated by tokenizing the assembly instructions in the function. As described in Section 3.1, all symbols that appear in the assembly instruction sequence are treated as tokens, with the purpose to remain key information about the syntax and semantics of assembly instructions.

Inter-instruction position sequence: $x_{it} = \{1,\ 1,\ 1,\ 1,\ 2, \ldots\}^n$, a sequence of integers encoding the position of each instruction in the function. The opcode and all operands within a single instruction share the same integer value. The reason of introducing this position sequence is that it's critical for inferring binary semantics, for example, exchanging the two instructions "mov eax, [rbp-0x2c]" and "add eax, 1" will get different semantics.

Intra-instruction position sequence: $x_{in} = \{1,\ 2,\ 3,\ 4,\ 1, \ldots\}^n$, a sequence of integers encoding the position of opcode and operands within a single instruction. The reason of introducing this position sequence is that exchanging the two operands in one instruction can significantly change the semantics of the instruction. For example, when we change "eax" and "[rbp-0x2c]" in "mov eax, [rbp-0x2c]", we will get different semantics.

Since a BERT MLM task requires a vector as input, we use one-hot encoding to generate embedding vector for each token in these three input sequences, and sum them up to get a single embedding vector as the input to the model. Take the i-th token (e.g., x_i) as an example, the input representation is as follows:

$$E_i = E_f\left(x_{f_i}\right) + E_{it}\left(x_{it_i}\right) + E_{in}\left(x_{in_i}\right)$$

Here, x_{f_i}, x_{it_i} and x_{in_i} represents the instruction code sequence, the inter-instruction position sequence and the inner-instruction position sequence of the i-th token in the function assembly code and $E_f\left(x_{f_i}\right)$, $E_{it}\left(x_{it_i}\right)$ and $E_{in}\left(x_{in_i}\right)$ represents their embedding vector generated by one-hot encoding [22–23] respectively.

3.3 Pretraining Based on BERT MLM

In this paper, we will use a BERT MLM task to perform pretraining in order to help learning specific instruction semantics and its context dependencies, which are critical for many binary analysis tasks. MLM task requires the model to predict randomly masked words, that is, given the context, to predict the masked words, which forces the model to learn the dependencies between the masked words and their surrounding words, thus train the model to understand the semantics of the statement. At the same time, MLM task is an unsupervised learning process that doesn't need any manual labelling. Therefore, some other pretraining samples can be used to further improve the accuracy of model training, while avoiding overfitting.

Masking Strategy. As shown in Fig. 4, we follow the random masking strategy of the original BERT model to mask the tokens to be predicted. Specifically, it selects 15% tokens to deal with as follows: 1) randomly select 80% of them to be masked; 2) replace 10% with any other token than it; 3) 10% remains unchanged. Masking means to use a special token (e.g.,<MASK>) to replace the selected tokens. Then, the model will predict what the masked tokens are based on the remaining 85% un-masked tokens. What to be masked in this paper can be opcode or operand of an instruction or the entire instruction, or even a sequence of multiple instructions, as long as a fixed percentage of tokens are selected.

Training objectives. The pretraining model f_p takes the input representation described above as input: $(E_1, E_2, \ldots, E_i, \ldots, E_n)$, $i \in P$, where, we use E_i to represent the embedding of the i-th token (e.g., x_i) to be masked. In this paper, we select the token x_{f_i} in the instruction code token sequence x_f to be masked, and we use P to represent a set of positions to be masked. Then, the model will predict the masked tokens: $\left\{\hat{x}_{f_i}, i \in P\right\} = f_p\left(E_1, E_2, \ldots, E_i, \ldots, E_n\right)$. Let f_p be parameterized by θ, the training objective of f_p is thus to search for θ that minimizes the cross-entropy losses between the predicted tokens and the actual tokens, that is:

$$arg\ \min_{\theta} \sum_{i=1}^{|P|} \left(-x_{f_i} \log\left(\hat{x}_{f_i}\right)\right) \tag{1}$$

As shown in Fig. 3, f_p uses Transformer self-attention layer as encoder to generate context-aware embedding, so that the context information of every token can be embedded into the embedding of

each token in the entire input sequence, that is E_i. Specifically, each embedding of each token at each layer will participate in all the other tokens' embedding generation process, then aggregates at that layer, and finally updates its embedding at the next layer. That means the last embedding, learned after the last layer, encodes semantic information of each token in the entire input sequence. That's very different from the word2vec approach used in previous work, where once the training is completed, the token embedding is fixed. However, even the same token may have very different semantic in different contexts, which word2vec can't deal with but our model can. On top of the transformer, we use a softmax layer to output the probability of the masked token x_{f_i}.

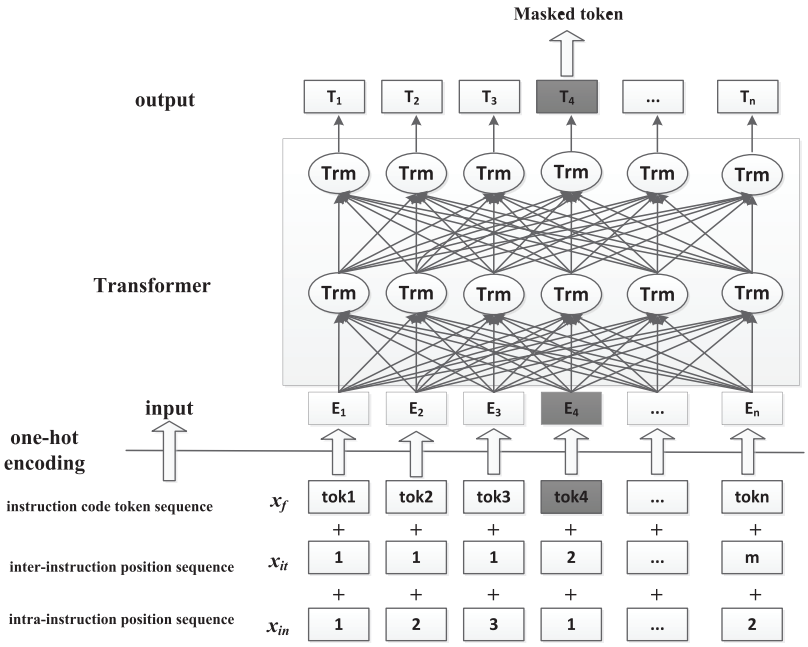

Figure 3: pretraining based on BERT MLM

3.4 fine-tuning Based on Siamese Network

As shown in Fig. 2, the second stage is to find-tune the model, whose task is: given a triplet (f_1, f_2, y) consisting of a pair of functions (f_1 and f_2) and their similarity label y(1 and -1 respectively indicate that these two functions are similar and dissimilar), to train the model to learn the similarity of two functions. There are two main operations in this stage: one for function embedding generation; one for fine-tuning the model according to the true label of the two functions.

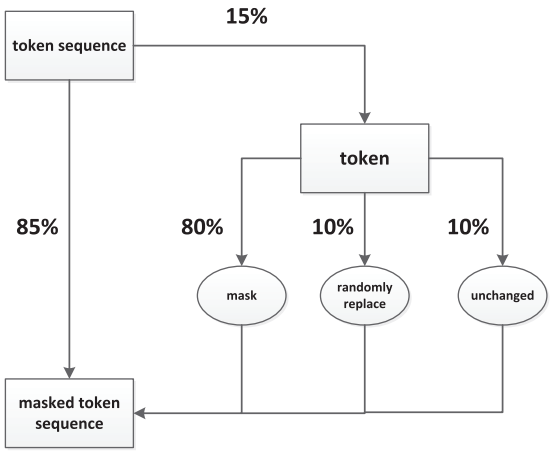

Figure 4: masking strategy

Function embedding generation. Inspired by SBERT [20], we add a pooling layer to the pretrained model outputted in the first stage to derive a fixed-size embedding as a function embedding. In terms of strategy, we select the MEANstrategy strategy recommended by SBERT model, taking the average of all the embedding vectors outputted in the first stage as its function embedding. Given the embedding vector sequence produced by the pretraining model $E_l = (E_{l1}, E_{l2}, \ldots, E_{ln})$, the function embedding generation formula is as follows:

$$u = \tanh\left(\frac{\sum_{i=1}^{n} E_{l,i}}{n} \cdot W_1\right) \cdot W_2 \tag{2}$$

Here, W_1 and W_2 are $n \times n$ and $n \times m$ matrix respectively.

Fine-tuning the model. After generating function embedding, we construct a Siamese network [14] to update its weight parameters according to the cosine similarity [21] between the two function embeddings and their real similarity label. We show the progress in Fig. 5.

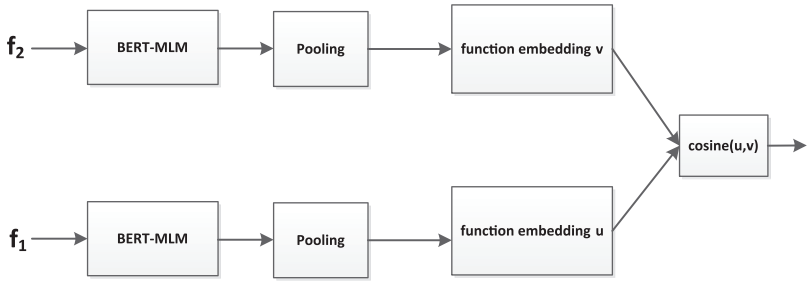

Figure 5: Fine-tuning model based on Siamese network

Formally, given K function pairs $\langle f_i, f_i^{'} \rangle$ and its real-word similarity label $y_i \in \{1, -1\}$ (1 means f_i is semantically similar to $f_i^{'}$, -1 means dissimilar). For each function pair, the output of the Siamese network is:

$$sim\left(f_i, f_i'\right) = cos\left(u, v\right) = \frac{\sum_{i=1}^{n}\left(u\left[i\right] \cdot v\left[i\right]\right)}{\sqrt{\sum_{i=1}^{n} u\left[i\right]} \cdot \sqrt{\sum_{i=1}^{n} v\left[i\right]}} \tag{3}$$

Here, u and v are respectively the embedding of the function f_1 and f_2, $u[i]$ and $v[i]$ are the i-th component of the vector u and v, respectively. Therefore, the objective of the fine-tuning is to minimize the distance between the cosine similarity of the two function and their real-world label:

$$\min_{W_1,\ldots,W_2} \sum_{i=1}^{K} \left(sim\left(f_i,\ f_i'\right) - y_i\right)^2 \tag{4}$$

As in Gemini, we choose backpropagation gradient descent method to solve the formula (4), and recursively calculate the gradient of the parameters according to the generated function embedding. Because the pretrained model is part of formula (4), so, the parameters of the pretrained model can be updated during fine-tuning. Finally, once a good performance (such as using AUC as a metric) is achieved, we will terminate the fine-tuning progress. And the resulting model can be used to transform a function into a valid function embedding for similarity detection.

3.5 Querying with a CVE Vulnerability Function

As shown in Fig. 6, the target of the querying stage is to search in the target corpus the most similarity function with the CVE vulnerability as query function and finally confirm whether it's indeed a vulnerability. To do this: Firstly, we will use Binwalk to extract all the binary executables from the embedded device firmware images, disassembled them to extract all the binary functions as a set of target functions. Secondly, we feed all the functions in the target corpus to the resulting model from the second stage to generate all their embeddings, and store them in the target corpus. Thirdly, we compile the CVE vulnerability function, feed it to the model and generate the function embedding as the query embedding u. Fourthly, we calculate the similarity between u and all the function embedding *vs.* in the target corpus, and ranking them. Finally, we manually confirm whether the most similar function is indeed a vulnerability function.

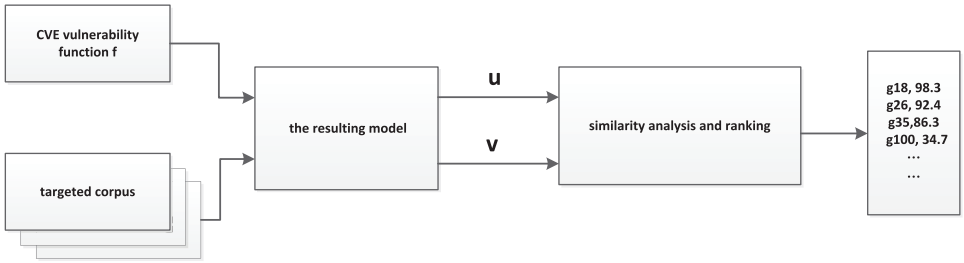

Figure 6: Querying with a CVE vulnerability function

4 Experiment and Evaluation

4.1 Experiment Settings

Experiment environment. We conduct all the experiments on a Linux Desktop Machine, Intel(R) Core (TM) i7-8700K CPU @ 3.70GH, with 12 virtual cores, 16GB RAM, 1 GeForce RTX 2080-Ti GPU, Ubuntu 16.04, CUDA 10.0, CUDNN 7.6.5, Python 3.7, PyTorch 1.6.0, GCC 7.3.0, and IDA Pro 7.5.

Baselines and model parameters. In this paper, we select Gemini and SAFE as the baseline to compare with our firmVulSeeker, because SAFE is the state-of-the-art binary function similarity analysis tool and Gemini has provided its firmware vulnerability capability which can be usually baseline to compare with other tools. For the sake of fair comparison, we set the embedding dimension to 128 and the maximum number of instructions to 150 as in SAFE. We also set the layers 12, the headers 8 and the hidden embedding 128 in the basic BERT parameters.

Datasets. In our evaluation, we collect 3 datasets: (1) Dataset I for training and evaluating the accuracy of the model; (2) a vulnerability dataset (Dataset III); (3) Dataset II for evaluating the performance of the firmVulSeeker on real-word cases.

- **Dataset I**. This dataset is used for training and baseline comparison. It consists of binaries compiled from source code, so that we have ground truth. As shown in Tab. 2, we collect 4 (but different versions) software projects widely used in embedded device firmware, including: coreutils-6.5[1], coreutils-6.7, bintuils-2.30[2], busybox-1-21-stable[3],OpenSSL_1_0_1f[4] and OpenSSL_1_0_1u. We compile them into 5 architectures (x86_32, x86_64, arm_32, mips_32 and mipseb_32) using gcc-7.3.0[5] with 4 optimization levels (O0-O3), resulting 4, 140 binary files (considering openssl is often found in embedded firmware as libssl.so and libcrypto.so, so we compile openssl into libssl.so and libcrypto.so.) containing 1,493,950 functions.
  - **Dataset partitioning**. We strictly separate the dataset into different function sets to be used in pretraining, fine-tuning and evaluation, although in theory pretraining can be done on a large-scale dataset including functions for fine-tuning. In terms of dataset for fine-tuning, we set the ratio of training set and testing set as 1:4, which is just opposite to the common 4:1, to prove the generalization capability of our model from a small number of training samples to large number of unseen testing samples to alleviate the possibility of overfitting.
  - **Similar/dissimilar function pairs set**. In this paper, we didn't strip the symbols in the compilation process, so we use the symbol as index to search in the dataset, and treat two functions with the same function name as similar, others are dissimilar. Moreover, we keep the training and testing function pairs strictly non-duplicated by ensuring the functions that appear in training function pairs not appear in the testing set anymore. And we set the ratio between similar and dissimilar function pairs in the training set as 1:5 for fine-tuning, which follows the actual distribution of similar/dissimilar functions (in practice, the proportion of dissimilar functions is larger than similar functions).
- **Dataset II (vulnerability dataset)**. This dataset contains known CVE vulnerabilities collected from CVE website[6]. We use "openssl" and "busybox" (because the two projects are widely used in the embedded firmware) as index to search from the CVE website and select the items which clearly describe where are their vulnerable functions. In the resulting dataset, we exclude the items containing "openssl" but not related with the OpenSSL project (e.g., associated with PHP, OpenSSH, Ruby and so on), and finally we get the remaining 52 CVEs meet our requirements, based on which we built our vulnerability dataset.
- **Dataset III (firmware dataset)**. In order to evaluate the vulnerability search capability on the real-world firmware sets, we collect firmware images from the official sites of D-Link,

[1] https://www.gnu.org/software/coreutils/
[2] https://www.gnu.org/software/binutils/
[3] https://git.busybox.net/busybox
[4] https://github.com/openssl/openssl.git
[5] https://gcc.gnu.org/
[6] https://cve.mitre.org/

Netgear, Cisco, Tenda and TP_Link, and select out the firmware images whose filesystem and corresponding binary files and library function can be extracted by binwalk. Finally, we get 70 firmware images, mainly some small home router or camera firmware, and the number of arm_32 (little-endian), mips_32 (big-endian) and mips_32 (little-endian) architecture is 36, 23 and 11, respectively.

Table 2: Dataset I

Project	Version	Compiler	Optimization levels	Architecture	Functions(#)
OpenSSL	1_0_1f	gcc-7.3.0	O0-O3	x86_32, x86_64, arm_32, mips_32, mipseb_32	124,090
	1_0_1u				123,673
coreutils	6.5				337,823
	6.7				337,823
bintuils	2.30				505,503
busybox	1–21				65,038
Total functions(#)					1,493,950

4.2 Evaluation

We will evaluate firmVulSeeker in terms of accuracy compared to the state-of-the-art tools, the robustness of the model, the scalability of the model and the vulnerability search capabilities on the real-world firmware set.

4.2.1 Accuracy

As shown in Section 4.1, Gemini and SAFE are selected as baseline tools for comparison with firmVulSeeker. Gemini is the first deep learning-based firmware vulnerability search for embedded firmware and is often used as the baseline tool for comparison. SAFE is the state-of-the-art tool for binary code similarity analysis in term of function granularity. As in Gemini and SAFE, we select the area under ROC(Receiver Operating Characteristic) curve (AUC) as metric to quantify the accuracy of firmVulSeeker, the higher the AUC score, the better the accuracy of the model.

Fig. 7 shows the average AUC score when matching functions on different architectures with irmVulSeeker, SAFE and Gemini, where Fig. 7a shows the comparison results about the actual test results on our evaluation dataset and Fig. 7b shows the comparison results about the reported results in their papers.

SAFE and Gemini reported their AUC score on their evaluation sets as 0.990 and 0.971 respectively in their papers, as shown in Fig. 7b. However, on our evaluation datasets, the AUC score of firmVulSeeker, SAFE and Gemini is 0.982, 0.971 and 0.871, respectively, as shown in Fig. 7a. We can see that firmVulSeeker is a little better than SAFE and far better than Gemini, thanks to its automated feature learning process and its pretraining model can extract instruction semantics more accurately, therefore the semantic features learned can better represent the function code than SAFE and Gemini. The training and testing set in Gemini all come from the same dataset compiled from OpenSSL with the same compiler, which is prone to overfitting, that's the reason contributed to the poor performance on the evaluation dataset provided in this paper.

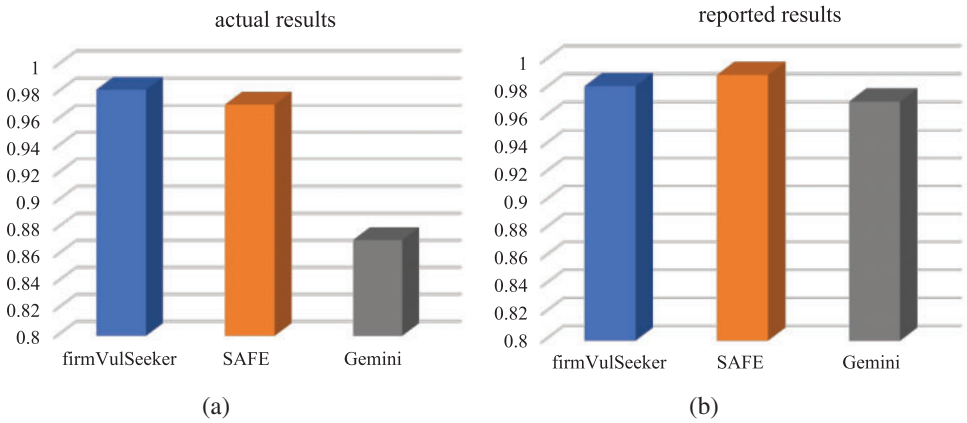

Figure 7: Actual results *vs.* reported results

4.2.2 Robustness

We discuss the robustness of the firmVulSeeker from three angels: across architectures, across optimization levels and dealing with new projects. The results are show in Fig. 8.

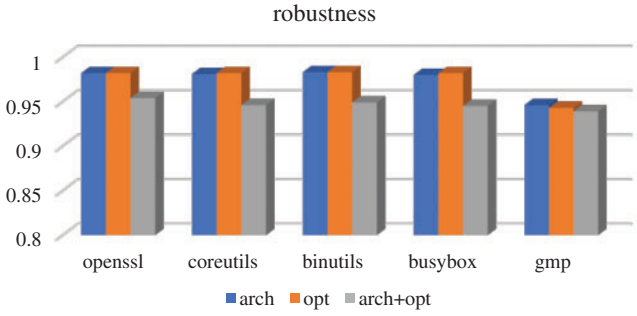

Figure 8: Robustness of the model

The entries "arch", "opt" and "arch + opt" in Fig. 8 respectively means we only consider cross-architecture, cross-optimization levels and consider both architecture and optimization. All the projects except "gmp"[7] in Fig. 8 participate in the pre-training and fine-tuning process. "gmp" is a new project used to test the capability of the model to deal with new, previously unseen functions.

As you can see, firmVulSeeker performs best on all the projects when considering optimization levels only, with an AUC up to 0.983. The reason behind this is the syntax of the two functions from different optimization levels but same architecture doesn't change significantly (i.e., the names of the opcode and operands remain unchanged). The AUC score of only considering cross-architecture is slightly lower, while the AUC score of both considering architecture and optimization levels lowest, only 0.945%, 3.8% lower than that of only considering optimization levels. The "gmp" project has the worst results because it didn't participate in the pretraining or fine-tuning process, however, even in its worst case (arch + opt), the AUC score is still 0.939, only 4.47% lower than the optimal case of the other projects. This result shows that firmVulSeeker can find semantically similar function pairs on a completely new dataset, which also proves the robustness of firmVulSeeker.

[7] https://gmplib.org/

4.2.3 Scalability

We try to apply (part of) firmVulSeeker to other tools and analyze the results to show the scalability of firmVulSeeker. Specifically, we apply firmVulSeeker's pretraining model on SAFE and Gemini, as shown in Figs. 9 and 10, replacing the corresponding component in the original tool with the green part.

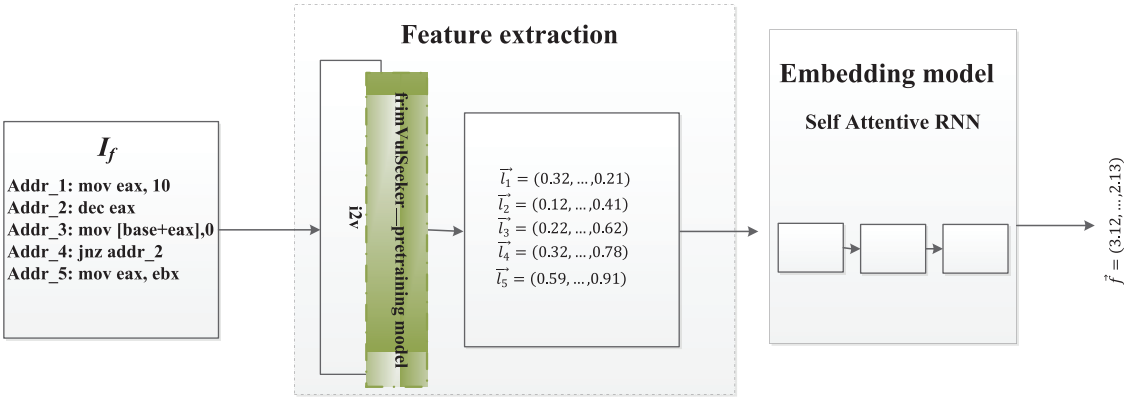

Figure 9: SAFE+

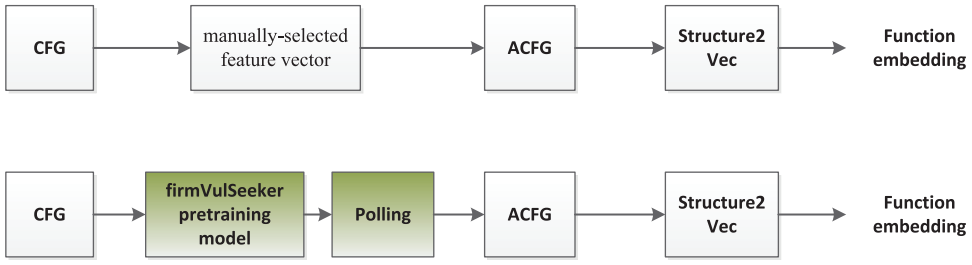

Figure 10: Gemini+

As shown in Fig. 9, we use firmVulSeeker pretraining model to substitute the original i2v component in the SAFE using word2vec model to embed the instruction sequence to instruction embedding sequence. The results are shown in Tab. 3. On the evaluation dataset provided in this paper, the AUC score of SAFE + increases by 0.012 than that of the original SAFE, and even exceeds that of firmVulSeeker.

Table 3: AUC score in SAFE, SAFE + and firmVulSeeker on our evaluation dataset

Tools	SAFE	SAFE+	firmVulSeeker
AUC	0.971	0.986	0.981

As shown in Fig. 10, we use firmVulSeeker pretraining model plus a polling layer to replace Gemini's original manually-selected feature vector components. Specifically, we use the pretraining model to generate the instruction embedding sequence for all the basic blocks in the function's CFG, then average all the instruction embeddings in the basic block as its basic block embedding, attach it

to the corresponding node in the CFG and finally get the resulting ACFG as the input to the Gemini's Structure2Vec [24] to generate function embedding.

Tab. 4 shows the AUC score of Gemini, Gemini + and firmVulSeeker on the evaluation dataset in this paper. As the Tab. 4 shows, Gemini + improves greatly the AUC score than the original (0.085, about 9.8%), and the AUC score even closes to firmVulSeeker.

Table 4: AUC score of Gemini, Gemini + and firmVulSeeker on our evaluation dataset

Tools	Gemini	Gemini+	firmVulSeeker
AUC	0.871	0.956	0.981

The above two cases show that firmVulSeeker can be extended to most analysis tools that start with a sequence of instructions as the sample set.

4.2.4 The Capability of Vulnerability Search on Real-World Firmware Images

We use 15 CVE vulnerability functions selected from dataset II to search on the firmware images in the dataset III. We give the number, the program and its version affected by the vulnerability, the program version we will compile, the vulnerability type and functions involved in the vulnerability of each CVE in Tab. 5. For each CVE, we compile the specified version of the program source code using gcc 7.3.0 with the default optimization option, targeting arm_32, mips_32 and mipseb_32, resulting in 45 vulnerability functions. Therefore, 45 search tasks will be performed.

Table 5: CVEs and related information

CVE number	Involved program	Version	Type	Involved functions
CVE-2018–20679	busybox-1.30.0 before	1.30.0	ID	udhcp_get_option(), networking/udhcp/common.c
CVE-2017-16544	busybox-1.27.2	1.27.2	CE	add_match, libbb/lineedit.c
CVE-2017-15873	busybox-1.27.2	1.27.2	IOF	get_next_block, archival/libarchive/decompress_bunzip2.c
CVE-2016-6301	busybox	1.21	DoS	recv_and_process_client_pkt, networking/ntpd.c
CVE-2015-9261	busybox1.27.2 before	1.27.2	BOF	huft_build, archival/libarchive/decompress_gunzip.c
CVE-2016-6302	openssl1.1.0 before	1.1.0	DoS	tls_decrypt_ticket, ssl/t1_lib.c
CVE-2016-6303	openssl1.1.0 before	1.1.0	DoS	MDC2_Update, crypto/mdc2/mdc2dgst.c
CVE-2016-6305	openssl1.1.0a before	1.1.0	DoS	ssl3_read_bytes, record/rec_layer_s3.c
CVE-2016-2842	openssl1.0.1(1.0.1 s before)	1.0.1f	DoS	doapr_outch, crypto/bio/b_print.c

(Continued)

Table 5: Continued

CVE number	Involved program	Version	Type	Involved functions
	openssl1.0.2(1.0.2 g before)			
CVE-2016-2182	openssl 1.1.0 before	1.1.0	OoW	BN_bn2dec function, crypto/bn/bn_print.c
CVE-2016-2180	openssl 1.0.2 h before	1.0.2 h	OoR	TS_OBJ_print_bio, crypto/ts/ts_lib.c
CVE-2016-2178	openssl 1.0.2 h	1.0.2 h	ID	dsa_sign_setup, crypto/dsa/dsa_ossl.c
CVE-2015-1791	openssl before 0.9.8zg openssl 1.0.0 before 1.0.0 s openssl 1.0.1 before 1.0.1n openssl 1.0.2 before 1.0.2b	0.9.8	DoS	ssl3_get_new_session_ticket, ssl/s3_clnt.c
CVE-2015-0290	OpenSSL before 1.0.2a	1.0.1f	DoS	ssl3_write_bytes, s3_pkt.c
CVE-2014-3508	openssl before 0.9.8zb openssl 1.0.0 before 1.0.0n openssl 1.0.1 before 1.0.1i	0.9.8	ID	OBJ_obj2txt, crypto/objects/obj_dat.c

Note: ∗ ID: information disclosure, DoS: deny of Service, OOR: out-of-read, OOW: out-of-write, BOF: buffer overflow, IOF: integer overflow, CE: code execution. "DoS" in this table is not a strict type of vulnerability, but a phenomenon caused by some vulnerability which will cause the program or service to crash. We use "DoS" as a substitute, because the official website doesn't clearly indicate what causes it.

For each CVE, three queries will be performed, each will retrieve the top 1, 10 and 50 firmware functions that is/are similar to the CVE function. Then we will confirm whether they are indeed vulnerability functions manually. We take the average value of these three queries as the search accuracy of the CVE. The reason for taking the average value is that in the actual testing process, it is found that the accuracy difference between using x86 binary function to match ARM/MIPS binary function and using ARM/MIPS binary function to match ARM/MIPS binary function is within the range of 10%, which is an acceptable error. Although we can't know the actual optimization option used by the firmware, however, benefit from the across-architecture and across-optimization capability of firmVulSeeker, we take the average value of these three searches as the final accuracy evaluation metric.

The results show that we can always successfully search in the dataset III all the CVE vulnerability except CVE-2015-0290[8]. We analyzed the CVE-2015-0290, and found that it only exists in OpenSSL libraries that support AES NI on 64-bit platforms. However, all the firmware images in dataset III are

[8] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0290

ARM or MIPS series, so it can't be successfully matched even using CVE binary functions compiled for ARM or MIPS.

Particularly, in order to compare with SAFE and Gemini, we list the results in terms of using the same CVE: CVE-2015-1791[9] and CVE-2014-3508[10] as queries as in the Gemini in Tab. 6:

Table 6: top-K results in Gemini, SAFE and firmVulSeeker

	CVE-2015-1791			CVE-2014-3508		
	#N@1	#N@10	#N@50	#N@1	#N@10	#N@50
Gemini	1	5	20	1	4	22
SAFE	1	7	24	1	6	25
firmVulSeeker	1	8	29	1	7	32

Note: ∗ N@K means the number of positives in the top-K.

As we can see from the table, all the three tools can successfully find real vulnerability in top-1 result. For CVE-2015-1971, firmVulSeeker can find 3 and 9 more real vulnerability than Gemini in top-10 and top-50 results, 60% and 45% enhanced, respectively. Similarly, it can find 1 and 5 more real vulnerability than SAFE, 14.3% and 20.8% enhanced, respectively. For CVE-2014-3508, firmVulSeeker has 75% and 45.5% higher search accuracy than Gemini, and 16.7% and 28% higher search accuracy than SAFE in their respectively the top-10 and top-50 results. As a result, we can conclude that firmVulSeeker achieves a higher accuracy than other two tools and more vulnerabilities can be detected.

5 Related Work

5.1 Binary Code Similarity Analysis

We can divide the existing binary code similarity analysis into three categories: static approaches, dynamic approaches and learning-based approaches.

Static approaches. Static approaches use static program analysis technology to convert binary into graphs (for example, CFG) and then perform comparison on them. These tools perform matching on the generated CFG or DFG [25–28] or decompose the graphs into fragments [29–32] for similarity detection. Most such approaches consider only the syntax of the instructions and not consider semantics which are critical in the analysis progress, especially when dealing with different optimization options.

Dynamic approaches. Such approaches assume that similar code must have semantically similar behavior. This, they perform the analysis by directly execution the given code [33,34], performing dynamic slicing [35] or taint analysis [36] on the given binary, and then checking the semantic equivalence based on the information collected during the execution. In general, these approaches are good at extracting semantics of the code and they have good resilience against optimization options and code obfuscation, but at the same time, they usually suffer from poor scalability and incomplete code coverage.

[9] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1791
[10] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3508

Learning-based approaches. Such approaches take the advantage of machine learning to solve the binary similarity problem. In recent years, many techniques [2–8,12–17] have been proposed, leveraging graph representation learning techniques [37,38] to embed the code information into embeddings (i.e., high dimensional numerical vectors) for similarity detection. There are two major advantages over the above static and dynamic approaches: Higher accuracy, because they incorporate the unique features of the code into their analysis progress by means of manual engineered featured [2–4] or automatic deep-learning-based methods [5–8,12–17]; and better scalability, because they don't use heavy graph matching algorithm nor dynamic execution.

5.2 Firmware Vulnerability Based on Binary Similarity Analysis

Costin et al. [39] proposed the first large-scale embedded firmware security analysis method based on fuzzy hashing, which was at file-level granularity, so its accuracy was not ideal. Multi-MH [27] is the first cross-architecture binary code similarity detection tool, which has a good effect on the HeartBleed vulnerability correlation. discoveRE [28] prefilters function-level features before graph matching to improve the efficiency of its search. FirmUp [40] utilizes the normalized fragements to represent program to solve cross-architecture, cross-compiler, cross-optimization option problems. BinARM [41] can identify vulnerable function in IED firmware using a granularity from coarse-grained to fine-grained. Xmatch [42] performs vulnerability detection based on data dependency and condition checking. Apposed to directly compare two control flow graphs, learning-based approaches such as Genius, Gemini, VulSeeker, aDiff [43] etc. embed the code features into the embedding to calculate the similarity of the two functions to improve the original search accuracy.

In general, firmware vulnerability based on binary similarity ranges from the earlier file comparison based on fuzzy hash to fine-grained function comparison, from file characteristics to control flow graph to semantic feature extraction, even to the multi-dimensional feature selection. The encoding scheme ranges from the earlier graph matching to graph embedding matching. The speed and accuracy of vulnerability correlation have been greatly improved. We conclude that the finer the granularity, the richer the encoding features, the more intelligent the encoding scheme, the higher the accuracy. And it shows absolute advantages in large-scale binary firmware vulnerability search.

6 Conclusion

In this paper, we introduced firmVulSeeker based on BERT MLM and Siamese network to search vulnerability in embedded device firmware images. Its core idea is to first pretrain a BERT MLM task to explicitly learn the approximate semantic information of the instruction and its context dependency relation. Then, a large number of similar/dissimilar function pairs are used to fine-tune the pretrained model, making full use of the instruction semantic information learned in the first stage, letting the resulting model can judge whether any two function pairs are similar. Finally, it will use a function embedding generated from the fine-tuned model to search in the targeted corpus and find the most similar function which will be confirmed whether it's a real vulnerability manually. We evaluated the accuracy, robustness, scalability and vulnerability search capability of firmVulSeeker on a real-world firmware set. Results show that it can greatly improve the accuracy of matching semantically similar functions, and can successfully find more real vulnerabilities in real-world firmware than other tools.

In the future work, we will consider the influence of different compilers and different obfuscation mechanisms on the accuracy of the model in order to further improve the vulnerability search capability of firmVulSeeker on real-world embedded device firmware images. At the same time, we will

research how to automatically confirm the existence of vulnerabilities based on the binary vulnerability search results in order to further eliminate the false positives caused by manual confirmation.

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

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

References

[1] J. Pewny, B. Garmany, R. Gawlik *et al.,* "Cross-architecture bug search in binary executables," in *2015 IEEE Symp. on Security and Privacy*, San Jose, California, IEEE, pp. 709–724, 2015.
[2] Q. Feng, R. Zhou, C. Xu *et al.,* "Scalable graph-based bug search for firmware images," in *Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security*, Vienna Austria, pp. 480–491, 2016.
[3] X. Xu, C. Liu, Q. Feng *et al.,* "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security*, Dallas Texas USA, pp. 363–376, 2017.
[4] J. Gao, X. Yang, Y. Fu *et al.,* "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *2018 33rd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, Montpellier, France, IEEE, pp. 896–899, 2018.
[5] F. Zuo, X. Li, P. Young *et al.,* "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Network and Distributed Systems Security (NDSS) Symposium 2019*, San Diego, California, 2019.
[6] L. Massarelli, G A. Di Luna, F. Petroni *et al.,* "Safe: Self-attentive function embeddings for binary similarity," in *Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, Gothenburg, Sweden, Springer, Cham, pp. 309–329, 2019.
[7] Yu Z., Cao R., Tang Q. *et al.,* "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proc. of the AAAI Conf. on Artificial Intelligence*, New York, 34, no. 1, pp. 1145–1152, 2020.
[8] S. H. H. Ding, B. C. M. Fung and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symp. on Security and Privacy (SP)*, San Francisco, California, IEEE, pp. 472–489, 2019.
[9] K. W. Church, "Word2Vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.
[10] L R. Medsker and L C. Jain, "Recurrent neural networks," *Design and Applications*, vol. 5, pp. 64–67, 2001.
[11] A. Graves, "Long short-term memory," in *Supervised Sequence Labelling with Recurrent Neural Networks*, Springer, Berlin, Heidelberg, pp. 37–45, 2012.
[12] Y. Duan, X. Li, J. Wang *et al.,* "Deepbindiff: Learning program-wide code representations for binary diffing," in *Network and Distributed System Security Symp.*, San Diego, California, 2020.
[13] J. Devlin, M W. Chang, K. Lee *et al.,* "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. of the 2019 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Minneapolis, Minnesota, vol. 1, no. Long and Short Papers, pp. 4171–4186, 2019.
[14] A. He, C. Luo, X. Tian *et al.,* "A twofold siamese network for real-time object tracking," in *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition*, Salt Lake City, USA, pp. 4834–4843, 2018.
[15] I. Ghafoor, I. Jattala, S. Durrani *et al.,* "Analysis of OpenSSL heartbleed vulnerability for embedded systems," in *17th IEEE Int. Multi Topic Conf. 2014*, Karachi, Pakistan, IEEE, pp. 314–319, 2014.
[16] Q. Le, T. Mikolov, "Distributed representations of sentences and documents," in *Int. Conference on Machine Learning*, Beijing, PMLR, pp. 1188–1196, 2014.
[17] X. Zhang, W. Sun, J. Pang, "Similarity metric method for binary basic blocks of cross-instruction Set architecture," *Proc. 2020 Workshop on Binary Analysis Research*, Internet Society, 2020.
[18] A. Vaswani, N. Shazeer, N. Parmar, "Attention is all you need," *NIPS*, vol. 30, 2017.

[19] D. Guthrie, B. Allison, W. Liu *et al.,* "A closer look at skip-gram modelling," *LREC*, vol. 6, pp. 1222–1225, 2006.
[20] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using siamese BERT-networks," *arXiv e-prints, 2019: arXiv: 1908.10084*, 2019.
[21] P. Xia, L. Zhang and F. Li, "Learning similarity with cosine similarity ensemble," *Information Sciences*, vol. 307, pp. 39–52, 2015.
[22] P. Rodríguez, M A. Bautista and J. Gonzalez *et al.,* "Beyond one-hot encoding: Lower dimensional target embedding," *Image and Vision Computing*, vol. 75, pp. 21–31, 2018.
[23] J. Fan, S. Upadhye and A. Worster, "Understanding receiver operating characteristic (ROC) curves," *Canadian Journal of Emergency Medicine*, vol. 8,no. 1, pp. 19–20, 2006.
[24] L. Song, *Structure2Vec: Deep Learning for Security Analytics Over Graphs*, Atlanta, GA: USENIX Association, 2018.
[25] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (English version)," *Sstic*, vol. 5, no. 1, pp. 3, 2005.
[26] D. Gao, M. K. Reiter and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Int. Conf. on Information and Communications Security*, Springer, Berlin, Heidelberg, pp. 238–255, 2008.
[27] J. Pewny, B. Garmany, R. Gawlik *et al.,* "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy*, San Jose, California, IEEE, pp. 709–724, 2015.
[28] S. Eschweiler, K. Yakdan and E. Gerhards-Padilla, "DiscovRE: Efficient cross-architecture identification of bugs in binary code," *NDSS*, vol. 52, pp. 58–79, 2016.
[29] Y. David and E. Yahav, "Tracelet-based code search in executables," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 349–360, 2014.
[30] Y. David, N. Partush and E. Yahav, "Statistical similarity of binaries,"*Acm Sigplan Notices*, vol. 51 no. 6, pp. 266–280, 2016.
[31] Y. David, N. Partush and E. Yahav, "Similarity of binaries through re-optimization," in *Proc. of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Barcelona, Spain, pp. 79–94, 2017.
[32] L. Luo, J. Ming, D. Wu, P. Liu and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proc. of the 22nd ACM SIGSOFT Int. Symp. on Foundations of Software Engineering. ACM*, Hong Kong, China, pp. 389–400, 2014.
[33] M. Egele, M. Woo, P. Chapman *et al.,* "Blanket execution: Dynamic similarity testing for program binaries and components," in *23rd {USENIX} Security Symp. ({USENIX} Security 14)*, San Diego, pp. 303–317, 2014.
[34] S. Wang and D. Wu, "In-memory fuzzing for binary code similarity analysis," in *2017 32nd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, Urbana, USA, IEEE, pp. 319–330, 2017.
[35] J. Ming, D. Xu, Y. Jiang *et al.,* "Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *26th {USENIX} Security Symp. ({USENIX} Security 17)*, Vancouver, Canada, pp. 253–270, 2017.
[36] J. Ming, M. Pan and D. Gao, "Ibinhunt: Binary hunting with inter-procedural control flow," in *Int. Conf. on Information Security and Cryptology*, Springer, Berlin, Heidelberg, pp. 92–109, 2012.
[37] 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.
[38] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Int. Conf. on Machine Learning*, Beijing, pp. 1188–1196, 2014.
[39] A. Costin, J. Zaddach, A. Francillon *et al.,* "A Large-scale analysis of the security of embedded firmwares," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, San Diego, CA, USA, pp. 95–110, 2014.
[40] Y. David, N. Partush and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 392–404, 2018.

[41] P. Shirani, L. Collard, B. L. Agba *et al.,* "Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, Cham, pp. 114–138, 2018.

[42] Q. Feng, M. Wang, M. Zhang *et al.,* "Extracting conditional formulas for cross-platform bug search," in *Proc. of the 2017 ACM on Asia Conf. on Computer and Communications Security*, Abu Dhabi, UAE, pp. 346–359, 2017.

[43] B. Liu, W. Huo, C. Zhang *et al.,* "Adiff: Cross-version binary code similarity detection with dnn," in *Proc. of the 33rd ACM/IEEE Int. Conf. on Automated Software Engineering*, Montpellier, France, pp. 667–678, 2018