Optir-SBERT: Cross-Architecture Binary Code Similarity Detection Based on Optimized LLVM IR

Yintong Yan, Lu Yu, Taiyan Wang, Yuwei Li, and Zulie Pan(✉)

College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China
{yanyintong.edu,yulu,wangty,liyuwei,panzulie17}@nudt.edu.cn

Abstract. Cross-architecture binary code similarity detection plays an important role in different security domains. In view of the low accuracy and poor scalability of existing cross-architecture detection technologies, we propose Optir-SBERT, which is the first technology to detect cross-architecture binary code similarity based on optimized LLVM IR. At the same time, we design a new data set BinaryIR, which is more diverse and provides a benchmark data set for subsequent research work based on LLVM IR. In terms of cross-architecture binary code similarity detection, the accuracy of Optir-SBERT reaches 94.38%, and the contribution of optimization is 3.99%. In terms of vulnerability detection, the average accuracy of Optir-SBERT reach 93.9%, and the contribution of optimization is 7%. The results are better than existing state-of-the-art (SOTA) cross-architecture detection technologies. In order to improve the efficiency of vulnerability detection in realistic scenarios, we introduced a file-level vulnerability identification mechanism on the basis of Optir-SBERT. The new model Optir-SBERT-F saved 45.36% of the detection time on the premise of a slight decrease in detection F value, which greatly improves the efficiency of vulnerability detection.

Keywords: Binary code similarity detection · Cross-architecture · Optimized LLVM IR · SBERT · File-level vulnerability identification mechanism

1 Introduction

Binary code similarity detection (BCSD) takes binary representation of a pair of functions as input and output as a value, which can reflect the degree of similarity between two functions. It is mainly used to find similar or homologous binary functions. This technology plays a vital role in different security research fields, including known vulnerability detection [3,12,20,47], malware analysis [2,37], patch analysis [15,38,43] and software supply chain analysis [14,44]. However, many software programs, especially IoT firmware applications, are often compiled into binary files with different instruction set architectures, which brings

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2024
Published by Springer Nature Switzerland AG 2024. All Rights Reserved
S. Goel and P. R. Nunes de Souza (Eds.): ICDF2C 2023, LNICST 571, pp. 95–113, 2024.
https://doi.org/10.1007/978-3-031-56583-0_7

great challenges to binary code similarity detection. Therefore, cross-architecture binary code similarity detection technology continues to emerge and gradually becomes a research hotspot.

With the development and application of machine learning, most of the state-of-the-art cross-architecture binary code similarity detection technology are based on machine learning [42]. In general, these detection techniques characterize binary functions in different architectures as vectors and calculate the similarity of the functions in vector space. Cross-architecture binary code similarity detection technology can be divided into code-based embedding [25,32,34] and graph-based embedding [10,12,19,46] according to the representation form. These technologies have realized binary code similarity detection under different architectures, but there are still some limitations and large room for improvement.

First, the existing cross-architecture similarity detection technologies rarely consider binary differences caused by compilers. The same source code under different compilation architectures, with different compilers, optimization options, and obfuscation strategies, will generate different binaries, and these binaries will vary significantly. Most of the existing cross-architecture binary code similarity detection technologies deal with these codes by constructing more complex models, larger thesaurus, or transforming binary function comparison into graph comparison problems. Such approach do not fundamentally solve the problem of binary differences caused by compilers, and the technologies used in the existing cross-architecture methods are not perfect, which makes the accuracy of binary code similarity detection and vulnerability detection low.

Second, the scalability of existing cross-architecture binary code similarity detection technologies is poor. Most of the cross-architecture binary code similarity detection technologies based on code embedding are established based on BERT. Such a model can only convert a binary function into a representation vector in a single operation, which will lead to huge computational overhead in the case of large detection samples. In addition, assembly instructions differ greatly under different architectures, and cross-architecture binary code similarity detection requires word segmentation representation of assembly instructions under all architectures. The increase of thesaural will lead to a large increase in the computational amount and time required for training models. What's more, the trained models have poor scalability, and they can only be used under the trained architectures.

Third, the existing evaluation data sets for cross-architecture binary code similarity detection are not diverse enough. Different detection technologies typically target different detection targets, and each technical solution trains and tests the model with its own data set and represents the results with different evaluation metrics (ROC curve area, MRR10, or Recall@5). However, binary code similarity detection technologies in the real world have a wider application range and needs to deal with more diverse and complex data sets. There is a big gap between the test results of existing detection technologies in small data sets and those in practical applications, which is not convincing enough.

Therefore, we propose Optir-SBERT and construct dataset BinaryIR to solve the above problems. Optir-SBERT is developed based on SEBRT network architecture. The SBERT network architecture contains two BERT. On this basis, the representation vector with binary function semantic information is generated by the twin neural network. The two binary functions can be converted into representation vectors by a single run, which is more suitable for similarity comparison [35]. At the same time, Optir-SBERT performs binary code similarity detection based on optimized LLVM IR. That is, the model firstly lift the binary code under different architectures to the LLVM IR, then optimizes the LLVM IR to eliminate binary differences caused by different compilers, and then realizes cross-architecture binary code similarity detection based on optimized LLVM IR.

Part of the existing work [22] uses IR to detect binary code similarity and achieved good results. We further optimize the LLVM IR and use the optimized LLVM IR to perform binary code similarity detection. As far as we know, Optir-SBERT is the first technique for cross-architecture binary code similarity detection based on optimized LLVM IR.

Optir-SBERT is evaluated on the dataset BinaryIR. BinaryIR contains a total of 2,025 binary projects under multiple architectures such as X86_32, X86_64, ARM32, ARM64, MIPS32, and PowerPC. The data sets are typical and diverse, and the evaluation results are more convincing. Experimental results show that the accuracy of Optir-SBERT in cross-architecture binary code similarity detection can reach 94.38%, which is significantly better than Genius [10], Gemini [46] and VulSeeker [12], with the accuracy increased by 46.58%, 52.18% and 20.17%, respectively. In terms of vulnerability detection, Optir-SBERT's average vulnerability identification accuracy rate reach 93.9%, well ahead of Gemini's 41.6%. In addition, we also conduct ablation experiments to explore the effect of the optimization on Optir-SBERT, and the results show that the optimization have a positive effect on the model. In conclusion, our research has the following contributions:

- We proposed Optir-SBERT, which solves the problem of binary code differences caused by compilers through optimization, and greatly improves the accuracy of cross-architecture binary code similarity detection and vulnerability detection. On this basis, Optir-SBERT-F is designed to greatly improve the detection efficiency by sacrificing the F value slightly.
- Optir-SBERT has strong scalability. When detecting binary functions under the new architecture, it is no need to retrain the model, but only need to lift the binary code under the architecture to the LLVM IR and optimize it for similarity detection or vulnerability detection, which greatly improves the universality of the model.
- We designed a more diversified dataset BinaryIR, which contains binary files under various architectures, IR files corresponding to binary files, and four kinds of optimized IR files (O0, O1, O2, O3), providing reference data sets for subsequent research on cross-architecture binary code similarity detection based on LLVM IR.

2 Related Work

Traditional binary code similarity detection technologies are mainly based on manually extracting binary function features [4,7,11,27,29] or using CG/CFG graph [9,33] for similarity comparison. The binary function information obtained in this way is often not comprehensive enough and the detection accuracy is low. With the development of machine learning technology, especially inspired by natural language processing, binary code similarity detection has made great progress. Binary code similarity detection technologies based on learning can be divided into single-architecture detection and cross-architecture detection.

2.1 Single-Architecture Binary Code Similarity Detection

The single-architecture detection technologies [15,42] can only detect binary code similarity in a specific architecture. When binary codes come from different compilation architectures, the detection effect will be very unsatisfactory, which is related to the fact that its model is only trained in a specific architecture. In the learning-based single-architecture binary code similarity detection technologies, Bingold [1] extracted semantic information of binary functions based on control flow graphs and data flow graphs, and then synthesized them into semantic flow graphs to represent binary functions. BinSim [28] proposed the system call slice equivalence checking, which is a hybrid method used to identify fine-grained semantic similarities or differences between two execution tracks. BinSign [29] provides an accurate and scalable solution for binary code fingerprinting by calculating and matching structure and syntax code profiles for disassembly. Asm2Vec [5] uses PV-DM, a natural language processing model, to learn assembly instructions and generate representations. Each assembly instruction is treated as a sentence, and operands and opcodes are separated into token units, which is more granular than migrating Word2Vec directly.

In general, single-architecture binary code similarity detection technologies have achieved high detection accuracy and have been applied in some network security fields [3,6,15,20,21]. With the increasing number of cross-architecture binary code similarity detection scenarios, single-architecture detection methods can no longer meet the actual detection requirements [31].

2.2 Cross-Architecture Binary Code Similarity Detection

With the application and popularity of IoT firmware applications, cross-architecture binary code similarity detection becomes more practical, but it is also more difficult to implement. Existing cross-architecture detection methods are generally based on code embedding or graph embedding.

The cross-architecture detection technologies based on code embedding [25,34,36,51] map binary functions under different instruction set architectures to the same vector space, and then calculates similarity. Most of these cross-architecture detection technologies are based on BERT, such as OrderMatters [48], Trex [32], PalmTree [18]. BERT is the best pre-trained representation model

for natural language processing based on Transformer [40]. Cross-architecture detection technologies based on code embedding often need to build larger corpora [50] to support binary function similarity identification of models under different architectures. Therefore, the construction of such models are complicated and its scalability are poor.

The detection technologies based on graph embedding [8,10,12,19,45,46] are more widely used in cross-architecture detection, which is related to the small change of graph structure information of binary function under different architectures [13,30]. This kind of technologies firstly extract the control flow graph, data flow graph, abstract syntax tree and other graph structure information in binary function, and then use graph neural network to characterize the graph structure information, and then carry out similarity detection, such as VulSeeker [12], Asteria [47] and CodeCMR [49]. QBindiff [26] also solves the matching problem between two binary functions according to the similarity of function content and call graph. GMN [19] is a graph structure similarity detection technology based on graph neural network proposed in recent years. After large-scale experimental evaluation [23], it is found that its detection effect is most prominent. However, when considering the structure information of binary function graphs, the cross-architecture detection technologies based on graph embedding ignore the relationship between basic block instructions, so that the captured binary function information is not comprehensive enough [24,42,48].

3 Methodology

3.1 Overview

In order to solve the shortcomings of existing cross-architecture binary code similarity detection technologies, we propose Optir-SBERT, which is based on SBERT network structure design and performs similarity detection for optimized LLVM IR. The new design idea enables Optir-SBERT to effectively solve the challenges encountered by the existing cross-architecture binary code similarity detection technologies.

Firstly, Optir-SBERT is designed based on the SBERT network structure, which uses twin neural networks to generate embedding vectors with semantic information, and pools the output results to generate fixed-length representation vectors. In the pooling operation, MEAN strategy is used to calculate the embedding vector of binary functions, and experiments show that this strategy has the highest accuracy in obtaining results [35,39]. Meanwhile, the training methods of Optir-SBERT include next instruction prediction (NSP) and mask language model (MLM).

Secondly, binary files under different compilation architectures can be lifted to the same LLVM IR. Therefore, binary code similarity detection based on LLVM IR has natural advantages in cross-architecture detection, and has better scalability. On this basis, we optimize the LLVM IR to eliminate the binary code differences caused by different compilers, so as to obtain higher quality initial

input data of the model, and then carry out binary code similarity detection and vulnerability detection based on the optimized LLVM IR.

Thirdly, in terms of vulnerability detection, Optir-SBERT-F, a model that can detect vulnerabilities faster, is proposed based on practical application scenarios. Compared with Optir-SBERT, this model introduces a file-level vulnerability identification mechanism, which determines whether the binary file to be detected has any vulnerability through the traditional way. If there is no vulnerability, lifting and a series of subsequent operations are not required. If there are vulnerabilities, lifting is carried out and specific vulnerability functions in the file are located and judged, thus saving a lot of detection time.

Finally, we design a new dataset, BinaryIR, which is more diverse, including many typical binary projects and a large amount of IoT firmware, to match real-world application scenarios. We lift all binary files in BinaryIR to obtain the corresponding LLVM IR files, and optimized the LLVM IR files in four ways (O0, O1, O2, O3). These data are incorporated into BinaryIR to provide a benchmark data set for subsequent researchers. The overall design of Optir-SBERT is shown in Fig. 1.

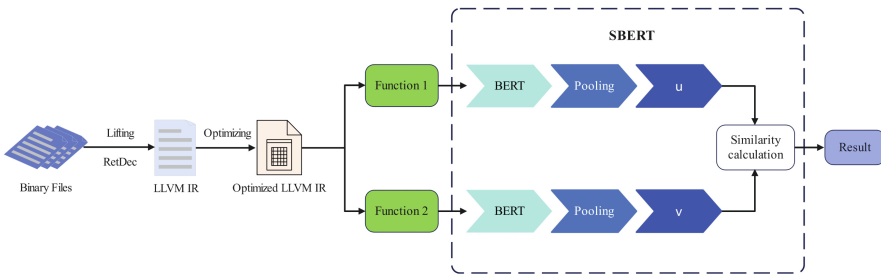

Fig. 1. The overall design of Optir-SBERT.

3.2 Optimized LLVM IR

LLVM IR is the internal representation generated by the compiler after scanning the source program. All stages of the compiler analyze or optimize the transformation on the LLVM IR [17,41], so it has a great impact on the overall structure, efficiency and robustness of the compiler. LLVM IR has m compiler design for front-end compiler languages (C, C++, Go, Rust, Toy, etc.) and n compiler designs for back-end platforms (X86, MIPS, ARM, PowerPC, etc.), which reduces the number of m*n compilers designed for m languages and n platforms, greatly improving compilation efficiency. LLVM structure design is shown in Fig. 2.

The binary files compiled by the same source program are very different. The fundamental reason for these differences are that the source program goes through different compilation architectures, compilers, optimization options, and obfuscation strategies when it is compiled into binary. Therefore, by lifting binary

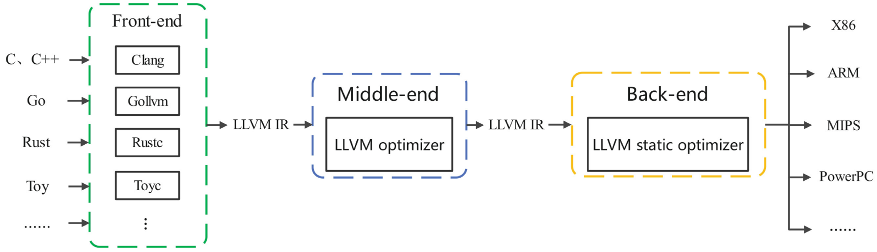

Fig. 2. LLVM structure design.

files to LLVM IR, assembly languages under different architectures can be unified into one LLVM IR, and the model based on this LLVM IR can perform cross-architecture binary code similarity detection. At the same time, in order to eliminate binary differences caused by the compiler, we optimize the LLVM IR to obtain higher quality initial input data. The Optir-SBERT trained based on this data will have higher detection accuracy.

The lifting tool we use is RetDec, which is based on the LLVM design and supports binary files lifting in various architectures (X86, ARM, MIPS32 and PowerPC, etc.). It is the best open source lifting tool known. RetDec can directly decompile the binary file to the source code. In this process, bin2llvmir is called to convert the binary file into LLVM IR.

3.3 File-Level Vulnerability Identification Mechanism

In the actual vulnerability detection scenario, the model often needs to detect a large number of binary files, which puts forward a high requirement for the detection efficiency of the model. Optir-SBERT we designed needs to lift binary files, which will take up lots of time and greatly affect the detection efficiency. To this end, we design Optir-SBERT-F. This model adds a file-level vulnerability identification mechanism on the basis of Optir-SBERT, which can improve the vulnerability detection efficiency of binary files in practical applications.

The file-level vulnerability identification mechanism first needs to build the binary file vulnerability library. The specific method is to collect the binary files containing the vulnerability and make statistics on its assembly instruction information, so as to obtain the assembly instruction characteristics of the binary files containing a certain vulnerability. Through analysis and experiment, we decide to make statistics on 28 typical assembly instruction characteristics of binary files, and build the binary file vulnerability library. Instruction features vary from architecture to architecture. The 28 instruction features we screened are available in common architectures (X86, ARM and MIPS32). The specific characteristics of binary file assembly instructions are shown in Table 1.

Mark the detected binary files as $F = \{f_1, f_2, \ldots, f_i, \ldots, f_m\}$. Mark the binary files in the vulnerability library as $G = \{g_1, g_2, \ldots, g_j, \ldots, g_n\}$. Mark the characteristics of assembly instructions as $C = \{c_1, c_2, \ldots, c_k, \ldots, c_{28}\}$. The

Table 1. Characteristics of binary file assembly instructions

Statistical Characteristics		
inst_num_abs_arith	inst_num_grp_jump	inst_avg_cmp
inst_num_abs_ctransfer	inst_num_grp_ret	inst_avg_cndctransfer
inst_num_abs_dtransfer	inst_num_logic	inst_avg_ctransfer
inst_num_arith	inst_num_total	inst_avg_dtransfer
inst_num_bitflag	inst_avg_abs_arith	inst_avg_grp_call
inst_num_cmp	inst_avg_abs_ctransfer	inst_avg_grp_jump
inst_num_cndctransfer	inst_avg_abs_dtransfer	inst_avg_grp_ret
inst_num_ctransfer	inst_avg_arith	inst_avg_logic
inst_num_dtransfer	inst_avg_bitflag	inst_avg_total
inst_num_grp_call		
Total		**28**

lower bound of G is Low, the upper bound is $High$, and the evaluation function is E. In the actual vulnerability detection process, as long as formula (5) is satisfied, binary file f_i is considered suspicious. Then, it is necessary to lift binary file f_i to LLVM IR and conduct subsequent function-level vulnerability detection. Otherwise, the assembly instructions characteristics of next binary file $f_{(i+1)}$ are matched until all binary files F are detected. The calculation formulas are:

$$Low_{jk} = g_j(c_k) \times 0.9, j \in [1, n], k \in [1, 28] \tag{1}$$

$$High_{jk} = g_j(c_k) \times 1.1, j \in [1, n], k \in [1, 28] \tag{2}$$

$$q_{ik} = f_i(c_k), i \in [1, m], k \in [1, 28] \tag{3}$$

$$E_j(k) = \begin{cases} 0, q_{ik} \notin [Low_{jk}, High_{jk}] \\ 1, q_{ik} \in [Low_{jk}, High_{jk}] \end{cases} \tag{4}$$

$$\forall \sum_{k=1}^{28} E_j(k) \geq 14, f_i = g_j, i \in [1, m], j \in [1, n] \tag{5}$$

By introducing the file-level vulnerability identification mechanism, the efficiency of Optir-SBERT-F in actual vulnerability detection is greatly improved. The overall design of this model is shown in Fig. 3.

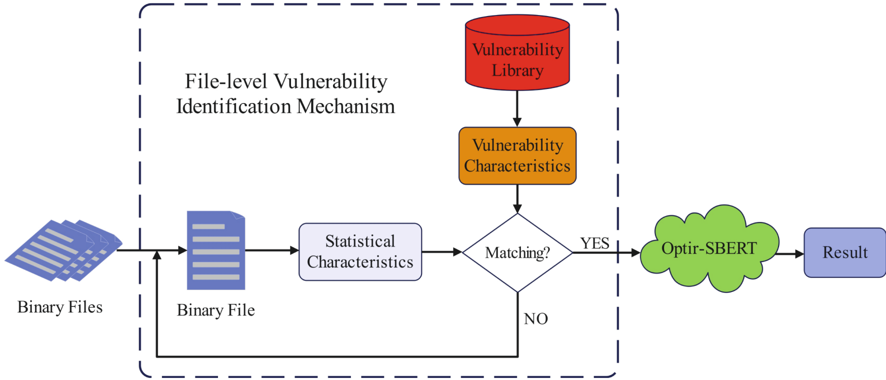

Fig. 3. The overall design of Optir-SBERT-F.

4 Experimental Setup

4.1 Experimental Environment

The environment of this experiment is Ubuntu 20.04 system under Linux 64-bit. The main configuration of the server is Intel Xeon Gold 6230R@104x 4 GHz processor, 256 GB memory and RTX3090 graphics card.

4.2 The BinaryIR Dataset

The existing binary code similarity detection technologies can only be detected for some specific data sets, which is divorced from the actual application scenarios in the real world. Therefore, we built a more diversified dataset, BinaryIR, and conduct training and testing Opti-SBERT on this dataset, so as to obtain more realistic detection results.

BinaryIR is built based on the dataset BinKit [16], BinaryCorp [42], and our collection of binary projects, IoT firmware, and vulnerability function library, including 2,025 projects, 41,172 binary files, and 388,2046 binary functions. The size of the binary files ranges from 14K to 27M. At the same time, we lift all binary files to obtain their corresponding LLVM IR files, and optimized the LLVM IR using four optimizations (O0, O1, O2, and O3), all of which are included in BinaryIR. The quantity statistics of specific train data and test data are shown in Table 2.

The process of lifting binary files to the LLVM IR files is relatively smooth, but errors may be reported during the optimization of the LLVM IR files, resulting in a failure to generate the optimized LLVM IR files. Therefore, the number of optimized LLVM IR files is less than the number of the binary files. Table 3 shows the actual number of LLVM IR files and the optimized LLVM IR files.

Table 2. The quantity statistics of train data and test data

DataSets	#Projects	#Binaries	#Functions	#File size
BinaryIR Train	1520	31879	2981536	14K–27M
BinaryIR Test	505	9293	900510	15K–19M

Table 3. The quantity statistics of LLVM IR files and optimized LLVM IR files

DataSets	LLVM IR	Opted IR(O0)	Opted IR(O1)	Opted IR(O2)	Opted IR(O3)
Train	31879	27463	26943	27591	28038
Test	9293	7842	7506	7613	7837
Total	41172	35305	34449	35204	35875

4.3 Evaluation Metrics

In order to evaluate the performance of Optir-SBERT in cross-architecture binary code similarity detection and vulnerability detection, a unified evaluation index needs to be established. Mark the function to be detected as $S = \{s_1, s_2, \ldots, s_i, \ldots, s_n\}$. Mark the recall rate of the function as $Recall@k$. The evaluation function is T. Then, the formulas of calculating the recall rate of the function can be expressed as:

$$T(x) = \begin{cases} 0, x = False \\ 1, x = True \end{cases} \tag{6}$$

$$Recall@k = \frac{1}{|S|} \sum_{i=1}^{n} T(Rank_{s_i} \leq k) \tag{7}$$

In specific experiments, the evaluation index used by Optir-SBERT in cross-architecture binary code similarity detection is $Recall@1$. The evaluation index used for vulnerability detection is $Recall@5$.

5 Evaluation

Our evaluation aims to answer the following questions.

- Question 1: How does the optimization affect and contribute to Optir-SBERT?
- Question 2: With the increase of the functions to be detected, how does the detection accuracy of Optir-SBERT change?
- Question 3: The accuracy of Optir-SBERT in cross-architecture binary code similarity detection?
- Question 4: How does the vulnerability detection effect of Optir-SBERT?
- Question 5: What are the advantages and disadvantages of Optir-SBERT-F compared to Optir-SBERT?

5.1 Binary Code Similarity Detection

The Contribution of Optimization to Optir-SBERT. To explore the contribution of optimization to Optir-SBERT, an ablation experiment is conducted. Firstly, a cross-architecture detection model Ir-SBERT based on LLVM IR is designed. The input data of this model is LLVM IR, and other structures are the same as Optir-SBERT. In order to explore the influence of different optimization options on Optir-SBERT, four optimization methods (O0, O1, O2, O3) are evaluated experimentally under different architectures. All experiments carry out in this paper are based on BinaryIR dataset. In order to simulate the application effect of Optir-SBERT in actual scenarios, a function pool is designed in this experiment. The construction method of the function pool is as follows: 10 binary files were randomly selected from data set BinaryIR, and 5 binary functions were randomly selected from each binary file to form a function pool containing 50 binary functions. Optir-SBERT and Ir-SBERT are tested in the constructed function pool. The results of the two experiments are shown in Table 4.

Table 4. The contribution of optimization to Optir-SBERT

	X86_32, ARM32	X86_32, MIPS32	X86_64, ARM64	X86_64, MIPS32	ARM32, MIPS32	ARM64, MIPS32	Average
Ir-SBERT	0.9158	0.9352	0.9071	0.9091	0.8755	0.9077	0.9084
Optir-SBERT (O0)	0.9301	0.9044	0.9542	0.9454	0.9226	0.9290	0.9310
Optir-SBERT (O1)	0.9507	0.9560	0.9640	0.9270	0.9531	**0.9521**	0.9505
Optir-SBERT (O2)	**0.9525**	**0.9607**	**0.9676**	**0.9464**	0.9497	0.9416	**0.9531**
Optir-SBERT (O3)	0.9249	0.9511	0.9219	0.9335	**0.9537**	0.9403	0.9376

As can be seen from Table 4, in the cross-architecture binary code similarity detection, the average detection accuracy of Optir-SBERT under the four optimization is higher than that of Ir-SBERT, indicating that the optimization has a positive contribution to Optir-SBERT, and the contribution rate is up to 4.47%. Among the four optimization methods, O2 optimization option has the best effect to obtain optimized LLVM IR, and the average accuracy of Optir-SBERT obtained by optimized LLVM IR training based on this method can reach 95.31%. The Optir-SBERT used in the following experiments are all based on the O2 optimization option training.

With the increasing number of binary functions to be detected, the accuracy of the existing binary code similarity detection techniques will decrease. Through experiments, we detect the variation of the accuracy of Optir-SBERT under different number of function pools, and compared it with the existing SOTA large-scale detection tool jTrans [42]. The specific changes are shown in Fig. 4. Let the number of binary functions in the function pool to be detected be Q, the horizontal coordinate be $log_{10}Q$, and the vertical coordinate be the detection accuracy.

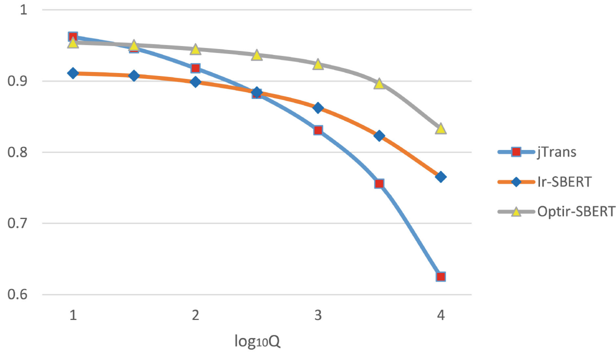

Fig. 4. The variation of models detection accuracy under different number of binary functions.

According to Fig. 4, with the increasing number of binary functions to be detected, the detection accuracy of the three models all declined, while the decline of Ir-SBERT and Optir-SBERT is smaller. When the number of binary functions to be detected reaches 10,000, the accuracy of jTrans decreases by 35.03%, while that of Ir-SBERT decreases by 15.96% and that of Optir-SBERT decreases by 12.62%. It shows that our propose Optir-SBERT can still maintain better performance in large-scale binary code similarity detection.

Performance of Optir-SBERT in Cross-Architecture Binary Code Similarity Detection. In order to explore the performance of Optir-SBERT in cross-architecture binary code similarity detection in real scenarios, the experiment is carred out under any compilation architecture. The function pool construction method designed in this experiment is as follows: 1000 binary files are randomly selected from data set BinaryIR, and 10 binary functions are randomly selected from each binary file to form a function pool P containing 10000 binary functions. Meanwhile, in order to make the experimental evaluation results of Optir-SBERT more convincing, we compare Optir-SBERT with Genius [10], Gemini [46] and VulSeeker [12], the three SOTA cross-architecture detection technologies. The experimental results are shown in Fig. 5.

As can be seen from Fig. 5, Optir-SBERT has the highest accuracy of 94.38% for similarity detection of binary functions under any architecture, and its accuracy is much higher than that of the existing SOTA cross-architecture detection technologies. The experiment results indicate that Optir-SBERT designed by us is more practical in real scenarios. In addition, the detection accuracy of Ir-SBERT is also high, which is 90.39%, second only to Optir-SBERT, indicating that the structure performance of SBERT network is better. Among them, the contribution of optimization to the improvement of Optir-SBERT accuracy is 3.99%. Genius and Gemini are excellent binary code similarity detection tools.

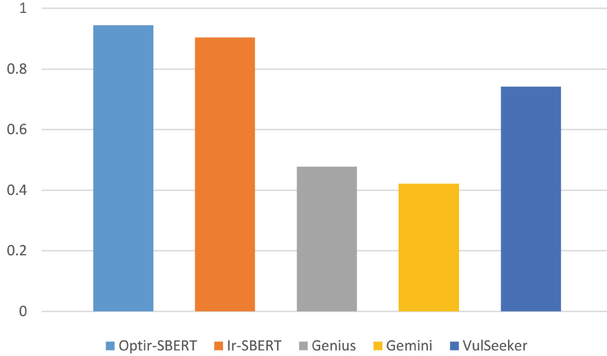

Fig. 5. Comparison of cross-architecture binary code similarity detection accuracy for models.

The reason for their low detection accuracy in this experiment may be that the number of functions in the function pool is too large and the sources of binary functions are more diverse.

5.2 Vulnerabilities Detection

Performance of Optir-SBERT in Real Vulnerability Detection. Vulnerability detection is one of the important research directions in the field of computer security. In order to evaluate the actual performance of Optir-SBERT in the task of vulnerability detection, we conduct this experiment.

Firstly, the vulnerability data set Busybox is collected in the experiment, and 7 vulnerability functions are extracted from the vulnerability data set to build the vulnerability function library. Secondly, the optimized LLVM IR corresponding to the vulnerability function is generated. After that, we mixed them into the optimized LLVM IR generated by the normal binary functions. Then, Optir-SBERT is used to match the vulnerability in the optimized LLVM IR. In this experiment, 2000 binary functions are randomly selected from the function pool P, and 7 vulnerability functions are mixed into them. What's more, the experimental evaluation index is $Recall@5$, and the experimental comparison model is Gemini [46] and Ir-SBERT. The results of vulnerability matching are shown in Fig. 6.

According to Fig. 6, the accuracy of Optir-SBERT is significantly higher than Gemini and Ir-SBERT in detecting vulnerability functions. For example, for CVE-2021-42385, the detection accuracy of Optir-SBERT is 100%, while that of Gemini is only 51%. In addition, the average vulnerability detection accuracy of Optir-SBERT is 93.9% and that of Ir-SBERT is 86.9%, indicating that the optimization improve the accuracy of Optir-SBERT by 7% in terms of vulnerabilities.

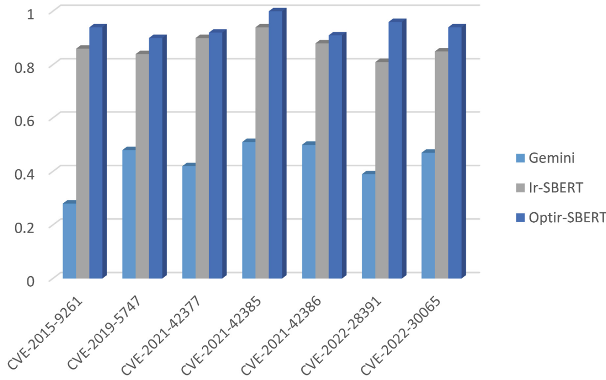

Fig. 6. Comparison of vulnerabilities detection results for models.

Comparison of Vulnerability Detection Effect Between Optir-SBERT and Optir-SBERT-F. In the real vulnerability detection scenario, the model often needs to detect binary files, which requires it to convert binary files into detectable binary codes or assembly instructions. The input of Optir-SBERT is optimized LLVM IR, which requires it to lift the binary file. However, lifting will occupy a lot of time, which seriously affects the vulnerability detection efficiency of Optir-SBERT. In addition, among a large number of binary files to be detected, only a few binary files have vulnerable functions. Based on this actual situation, we design Optir-SBERT-F.

Optir-SBERT-F is mainly applied to scenarios requiring rapid vulnerability detection. Compared with Optir-SBERT, Optir-SBERT-F introduces a file-level vulnerability identification mechanism, which improves the efficiency of vulnerability detection but sacrifices the F value of vulnerability detection.

The main performance changes of Optir-SBERT and Optir-SBERT-F are evaluated experimentally. In the experimental setting, 10 to 15 normal binary files and 1 to 5 vulnerable binary files are randomly selected to form a test data set. Then, the vulnerability detection F value and detection time of Optir-SBERT and Optir-SBERT-F are tested on this data set, and the average value of multiple tests is taken. The experimental results are shown in Table 5.

Table 5 shows that the average vulnerability detection F value of Optir-SBERT is 91.6% and the average detection time is 390.2 s. The average vulnerability detection F value of Optir-SBERT-F is 77.7%, 15.17% lower than Optir-SBERT, and the average detection time is 213.2 s, 45.36% lower than Optir-SBERT. It can be seen that Optir-SBERT-F saves 45.36% time at the expense of 15.17% F value, which greatly improves the efficiency of vulnerability detection on the premise that the F value is not significantly reducing.

Table 5. Comparison of vulnerability detection effect between Optir-SBERT and Optir-SBERT-F

CVE-	2015-9261	2019-5747	2021-42377	2021-42385	2022-28391	average
Optir-SBERT (F Value)	**0.909**	**0.889**	**0.923**	**1**	**0.857**	**0.916**
Optir-SBERT (Time-s)	337	273	446	339	556	390.2
Optir-SBERT-F (F Value)	0.8	0.667	0.833	0.857	0.727	0.777
Optir-SBERT-F (Time-s)	**179**	**157**	**213**	**226**	**291**	**213.2**

6 Conclusion

In this paper, we propose Optir-SBERT, which is the first technology for cross-architecture binary code similarity detection based on optimized LLVM IR, and has strong robustness and scalability. Optir-SBERT is mainly designed based on SBERT network structure, which can better understand and characterize the semantic information of optimized LLVM IR, and its twin network structure is more suitable for similarity detection. In addition, we built the dataset BinaryIR, which has a more diverse set of data including binary projects, IoT firmware, LLVM IR files, and four optimized (O0, O1, O2, O3) LLVM IR files.

The detection accuracy of Optir-SBERT is more stable. When the number of binary functions to be detected increases to 10,000, the detection accuracy decreases by only 12.62%. In terms of binary code similarity detection, the detection accuracy of Optir-SBERT can reach 94.38% under any compilation architecture, which is much higher than the existing SOTA cross-architecture detection technologies, in which the contribution of optimization is 3.99%. In terms of vulnerability detection, Optir-SBERT have an average vulnerability detection accuracy of 93.9%, of which the optimization contributed 7%. In addition, in order to detect binary file vulnerabilities more efficiently in real scenarios, we construct Optir-SBERT-F by introducing file-level vulnerability identification mechanism on the basis of Optir-SBERT. Optir-SBERT-F can save 45.36% of the time compared with Optir-SBERT, and greatly improves the efficiency of vulnerability detection without significantly reducing the F value.

However, the generation stage of the optimized LLVM IR files may make mistakes, mainly because the LLVM IR file contains characters that cannot be optimized, which has a certain impact on the universality of Optir-SBERT. At the same time, the file-level vulnerability identification mechanism of Optir-SBERT-F can be further improved to reduce the false positive rate and false negative rate. In the next work, we will try to improve the optimization method of LLVM IR files and optimize the file-level vulnerability identification mechanism, so as to make the model more practical and efficient in real scenarios.

Acknowledgment. We sincerely appreciate the anonymous reviewers for their valuable comments to improve our paper. This work is supported by NSFC under No. 62202484.

References

1. Alrabaee, S., Wang, L., Debbabi, M.: BinGold: towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs). Digit. Investig. **18**, S11–S22 (2016)
2. Darem, A., Abawajy, J., Makkar, A., Alhashmi, A., Alanazi, S.: Visualization and deep-learning-based malware variant detection using opcode-level features. Futur. Gener. Comput. Syst. **125**, 314–323 (2021)
3. David, Y., Partush, N., Yahav, E.: FirmUp: precise static detection of common vulnerabilities in firmware. ACM SIGPLAN Not. **53**(2), 392–404 (2018)
4. Ding, S.H., Fung, B.C., Charland, P.: Kam1n0: MapReduce-based assembly clone search for reverse engineering. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 461–470 (2016)
5. Ding, S.H., Fung, B.C., Charland, P.: Asm2Vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 472–489. IEEE (2019)
6. Duan, Y., Li, X., Wang, J., Yin, H.: DeepBinDiff: learning program-wide code representations for binary diffing. In: Network and Distributed System Security Symposium (2020)
7. Dullien, T., Rolles, R.: Graph-based comparison of executable objects (English version). Sstic **5**(1), 3 (2005)
8. Eschweiler, S., Yakdan, K., Gerhards-Padilla, E., et al.: discovRE: efficient cross-architecture identification of bugs in binary code. In: NDSS, vol. 52, pp. 58–79 (2016)
9. Feng, Q., Wang, M., Zhang, M., Zhou, R., Henderson, A., Yin, H.: Extracting conditional formulas for cross-platform bug search. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, pp. 346–359 (2017)
10. Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., Yin, H.: Scalable graph-based bug search for firmware images. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 480–491 (2016)
11. Gao, D., Reiter, M.K., Song, D.: BinHunt: automatically finding semantic differences in binary programs. In: Chen, L., Ryan, M.D., Wang, G. (eds.) ICICS 2008. LNCS, vol. 5308, pp. 238–255. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88625-9_16
12. Gao, J., Yang, X., Fu, Y., Jiang, Y., Sun, J.: VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 896–899 (2018)
13. Haq, I.U., Caballero, J.: A survey of binary code similarity. ACM Comput. Surv. (CSUR) **54**(3), 1–38 (2021)
14. Hemel, A., Kalleberg, K.T., Vermaas, R., Dolstra, E.: Finding software license violations through binary code clone detection-a retrospective. ACM SIGSOFT Softw. Eng. Notes **46**(3), 24–25 (2021)
15. Huang, H., Youssef, A.M., Debbabi, M.: BinSequence: fast, accurate and scalable binary code reuse detection. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, pp. 155–166 (2017)
16. Kim, D., Kim, E., Cha, S.K., Son, S., Kim, Y.: Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. IEEE Trans. Softw. Eng. (2022)

17. Lattner, C.: LLVM and clang: next generation compiler technology. In: The BSD conference, vol. 5, pp. 1–20 (2008)
18. Li, X., Qu, Y., Yin, H.: PalmTree: learning an assembly language model for instruction embedding. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 3236–3251 (2021)
19. Li, Y., Gu, C., Dullien, T., Vinyals, O., Kohli, P.: Graph matching networks for learning the similarity of graph structured objects. In: International Conference on Machine Learning, pp. 3835–3845. PMLR (2019)
20. Lin, J., Wang, D., Chang, R., Wu, L., Zhou, Y., Ren, K.: EnBinDiff: identifying data-only patches for binaries. IEEE Trans. Dependable Secure Comput. (2021)
21. Luo, L., Ming, J., Wu, D., Liu, P., Zhu, S.: Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. IEEE Trans. Software Eng. **43**(12), 1157–1177 (2017)
22. Luo, Z., Wang, B., Tang, Y., Xie, W.: Semantic-based representation binary clone detection for cross-architectures in the internet of things. Appl. Sci. **9**(16), 3283 (2019)
23. Marcelli, A., Graziano, M., Ugarte-Pedrero, X., Fratantonio, Y., Mansouri, M., Balzarotti, D.: How machine learning is solving the binary function similarity problem. In: 31st USENIX Security Symposium (USENIX Security 2022), pp. 2099–2116 (2022)
24. Massarelli, L., Di Luna, G.A., Petroni, F., Querzoni, L., Baldoni, R.: Investigating graph embedding neural networks with unsupervised features extraction for binary analysis. In: Proceedings of the 2nd Workshop on Binary Analysis Research (BAR), pp. 1–11 (2019)
25. Massarelli, L., Di Luna, G.A., Petroni, F., Baldoni, R., Querzoni, L.: SAFE: self-attentive function embeddings for binary similarity. In: Perdisci, R., Maurice, C., Giacinto, G., Almgren, M. (eds.) DIMVA 2019. LNCS, vol. 11543, pp. 309–329. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22038-9_15
26. Mengin, E., Rossi, F.: Binary diffing as a network alignment problem via belief propagation. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 967–978. IEEE (2021)
27. Ming, J., Pan, M., Gao, D.: iBinHunt: binary hunting with inter-procedural control flow. In: Kwon, T., Lee, M.-K., Kwon, D. (eds.) ICISC 2012. LNCS, vol. 7839, pp. 92–109. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37682-5_8
28. Ming, J., Xu, D., Jiang, Y., Wu, D.: BinSim: trace-based semantic binary diffing via system call sliced segment equivalence checking. In: Proceedings of the 26th USENIX Security Symposium (2017)
29. Nouh, L., Rahimian, A., Mouheb, D., Debbabi, M., Hanna, A.: BinSign: fingerprinting binary functions to support automated analysis of code executables. In: De Capitani di Vimercati, S., Martinelli, F. (eds.) SEC 2017. IAICT, vol. 502, pp. 341–355. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58469-0_23
30. Pan, Z., Wang, T., Yu, L., Yan, Y.: Position distribution matters: a graph-based binary function similarity analysis method. Electronics **11**(15), 2446 (2022)
31. Pan, Z., Yan, Y., Yu, L., Wang, T.: Identification of binary file compilation information. In: 2022 IEEE 5th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), vol. 5, pp. 1141–1150. IEEE (2022)
32. Pei, K., Xuan, Z., Yang, J., Jana, S., Ray, B.: TREX: learning execution semantics from micro-traces for binary similarity. arXiv preprint arXiv:2012.08680 (2020)

33. Pewny, J., Schuster, F., Bernhard, L., Holz, T., Rossow, C.: Leveraging semantic signatures for bug search in binary programs. In: Proceedings of the 30th Annual Computer Security Applications Conference, pp. 406–415 (2014)
34. Redmond, K., Luo, L., Zeng, Q.: A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis. arXiv preprint arXiv:1812.09652 (2018)
35. Reimers, N., Gurevych, I.: Sentence-BERT: sentence embeddings using Siamese BERT-networks. arXiv preprint arXiv:1908.10084 (2019)
36. Shalev, N., Partush, N.: Binary similarity detection using machine learning. In: Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, pp. 42–47 (2018)
37. Sriram, S., Vinayakumar, R., Sowmya, V., Alazab, M., Soman, K.: Multi-scale learning based malware variant detection using spatial pyramid pooling network. In: IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 740–745. IEEE (2020)
38. Sun, P., Yan, Q., Zhou, H., Li, J.: Osprey: a fast and accurate patch presence test framework for binaries. Comput. Commun. **173**, 95–106 (2021)
39. Thakur, N., Reimers, N., Daxenberger, J., Gurevych, I.: Augmented sBERT: data augmentation method for improving bi-encoders for pairwise sentence scoring tasks. arXiv preprint arXiv:2010.08240 (2020)
40. Vaswani, A., et al.: Attention is all you need. In: Advances in Neural Information Processing Systems, vol. 30 (2017)
41. VenkataKeerthy, S., Aggarwal, R., Jain, S., Desarkar, M.S., Upadrasta, R., Srikant, Y.: IR2VEC: LLVM IR based scalable program embeddings. ACM Trans. Archit. Code Optim. (TACO) **17**(4), 1–27 (2020)
42. Wang, H., et al.: jTrans: jump-aware transformer for binary code similarity. arXiv preprint arXiv:2205.12713 (2022)
43. Wang, X., Wang, S., Sun, K., Batcheller, A., Jajodia, S.: A machine learning approach to classify security patches into vulnerability types. In: 2020 IEEE Conference on Communications and Network Security (CNS), pp. 1–9. IEEE (2020)
44. Wang, Y., Jia, P., Peng, X., Huang, C., Liu, J.: BinVulDet: detecting vulnerability in binary program via decompiled pseudo code and BiLSTM-attention. Comput. Secur. **125**, 103023 (2023)
45. Xiu, H., Yan, X., Wang, X., Cheng, J., Cao, L.: Hierarchical graph matching network for graph similarity computation. arXiv preprint arXiv:2006.16551 (2020)
46. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 363–376 (2017)
47. Yang, S., Cheng, L., Zeng, Y., Lang, Z., Zhu, H., Shi, Z.: Asteria: deep learning-based AST-encoding for cross-platform binary code similarity detection. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 224–236. IEEE (2021)
48. Yu, Z., Cao, R., Tang, Q., Nie, S., Huang, J., Wu, S.: Order matters: semantic-aware neural networks for binary code similarity detection. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, pp. 1145–1152 (2020)
49. Yu, Z., Zheng, W., Wang, J., Tang, Q., Nie, S., Wu, S.: CodeCMR: cross-modal retrieval for function-level binary source code matching. In: Advances in Neural Information Processing Systems, vol. 33, pp. 3872–3883 (2020)

50. Zhang, X., Sun, W., Pang, J., Liu, F., Ma, Z.: Similarity metric method for binary basic blocks of cross-instruction set architecture. In: Proceedings of the 2020 Workshop on Binary Analysis Research, vol. 10 (2020)
51. Zuo, F., Li, X., Young, P., Luo, L., Zeng, Q., Zhang, Z.: Neural machine translation inspired binary code similarity comparison beyond function pairs. arXiv preprint arXiv:1808.04706 (2018)