Enhancing Learning-Based Binary Code Similarity Detection Model through Adversarial Training with Multiple Function Variants

Lichen Jia[1,2], Chenggang Wu[1,2], Bowen Tang[3], Peihua Zhang[7], Zihan Jiang[3], Yang Yang[4], Ning Liu [5], Jingfeng Zhang[6], Zhe Wang[*1,2]
[1]SKLP, Institute of Computing Technology, CAS, [2]University of Chinese Academy of Sciences
[3]Huawei Technologies Co. ,Ltd., [4]China University of Petroleum
[5]School of Software, Shandong University [6]University of Auckland [7]WeChat, Tencent Inc
lcjia457@gmail.com, {wucg, tangbowen,peihuazhang, wangzhe12, jiangzihan}@ict.ac.cn,
yyawesom@gmail.com, liun21cs@sdu.edu.cn, jingfeng.zhang9660@gmail.com

Abstract

Compared to identifying binary versions of the same function under different compilation options, existing Learning-Based Binary Code Similarity Detection (LB-BCSD) methods exhibit lower accuracy in recognizing functions with the same functionality but different implementations. To address this issue, we introduces an adversarial attack method called FuncFooler, which focuses on perturbing critical code to generate multiple variants of the same function. These variants are then used to retrain the model to enhance its robustness. Current adversarial attacks against LB-BCSD mainly draw inspiration from the FGSM (Fast Gradient Sign Method) method in the image domain, which involves generating adversarial bytes and appending them to the end of the executable file. However, this approach has a significant drawback: the appended bytes do not affect the actual code of the executable file, thus failing to create diverse code variants. To overcome this limitation, we proposes a gradient-guided adversarial attack method based on critical code—FuncFooler. This method designs a series of strategies to perturb the code while preserving the program's semantics. Specifically, we first utilizes gradient information to locate critical nodes in the control flow graph. Then, fine-grained perturbations are applied to these nodes, including control flow, data flow, and internal node perturbations, to obtain adversarial samples. The experimental results show that the application of the FuncFooler method can increase the accuracy of the state-of-the-art (SOTA) LB-BCSD model by 5%-7%.

1 Introduction

Binary code similarity detection methods can identify and return a set of functions most similar to a given binary function. These functions usually originate from the same source code but differ due to various compilation settings. Binary code similarity detection plays a crucial role in various security applications such as malware detection (Cesare et al., 2013; Ming et al., 2015; Chandramohan et al., 2016; You et al., 2020; Jia et al., 2023a; Lucas et al., 2021), software plagiarism detection (Luo et al., 2017; Yu et al., 2020; Luo et al., 2017; Sajnani et al., 2016; Walker et al., 2020; Zhang et al., 2023), vulnerability discovery (David et al., 2016; Jia et al., 2024; David and Yahav, 2014), and patch analysis (Mashhadi and Hemmati, 2021; Pewny et al., 2014; Hu et al., 2016; Wang and Wu, 2017).

In recent years, with the rise of artificial intelligence technologies, Learning-Based Binary Code Similarity Detection (LB-BCSD) techniques have emerged. Leveraging the powerful learning capabilities of machine learning models, these techniques excel in instruction semantic recognition, significantly improving detection accuracy and speed compared to traditional methods (Ding et al., 2019; Jia et al., 2023b; Zhang et al., 2024). However, in the field of software development, differences in programmers' coding habits, skill levels, and design approaches lead to various implementations of functions with the same functionality. This diversity is already evident at the source code level and becomes even more complex at the compiled binary code level. Although these different code implementations functionally align, their internal structures and design logics may differ significantly. This situation presents substantial challenges for software analysis, maintenance, and security detection.

Currently, LB-BCSD methods are primarily used to assess the similarity of the same function under different compilation options. However, this study reveals that when the LB-BCSD model encounters functions with identical functionality but different implementations, its identification accuracy decreases by 13% compared to identifying versions of the same function compiled under dif-

[*]Corresponding author.

ferent options (O2 and O3). This finding highlights the limitations of existing LB-BCSD methods in handling diverse code implementations.

To improve the model's capability in recognizing such function variants, an intuitive solution is to retrain the model using a large number of function samples with the same functionality but different implementations. However, implementing this strategy faces several challenges. The primary challenge is collecting a sufficient number of function samples with the same functionality but different implementations. Although online coding platforms like LeetCode offer many implementations of functions with the same functionality, these are mainly concentrated on specific coding problems and do not fully reflect the diversity of real-world programs. For more extensive real-world programs, collecting samples with the same functionality but significantly different implementations is extremely difficult. When implementing the same function, each programmer may choose different algorithms, data structures, or coding techniques based on specific requirements and their own experience, leading to a myriad of implementation methods. Theoretically, the different implementation methods for a function are infinite. Another important consideration is that not all different implementations will interfere with the LB-BCSD model's identification. Therefore, it is crucial to explore which subtle code variations mislead the model, as this has significant practical implications.

To address the above issues, this study proposes a gradient-guided white-box attack method based on critical code, called FuncFooler. This method generates multiple variants of the same function by designing perturbation strategies for the control flow, data flow, and internal nodes of executable files, such as altering control flow structures. Additionally, this study transforms the generation process of adversarial examples into an optimization problem and uses gradient algorithms to select critical nodes in the executable file's control flow graph for perturbation. Experimental results show that FuncFooler can improve the SOTA LB-BCSD model's accuracy by 5%-7%, demonstrating the effectiveness of this method in enhancing model robustness.

The main contributions of this paper are as follows:

- This study finds that the existing LB-BCSD method overly focuses on the accuracy of functions with the same functionality and implementation method, while neglecting the accuracy of functions with the same functionality but different implementation methods. When the LB-BCSD model handles functions that have the same functionality but different implementation methods, its recognition accuracy decreases by 13% compared to recognizing different versions of the same function under different compilation options (O2 and O3).
- We thoroughly explore the limitations of FGSM in the field of important code identification in binary programs, pointing out that it cannot be directly applied to this scenario. To overcome this issue, we innovatively propose an important code identification method based on dominator nodes, which generates multiple variants of the same function by perturbing the important code.
- This study retrains the model using adversarial samples generated by FuncFooler, and experimental results indicate that FuncFooler can improve the accuracy of the SOTA LB-BCSD model by 5%-7%, demonstrating its effectiveness in enhancing model robustness.

2 Related Work

Researchers have explored the field of white-box adversarial attacks and pointed out issues with adversarial samples in LB-BCSD methods. Early studies (Al-Dujaili et al., 2018; Verwer et al., 2020) drew inspiration from attack methods such as FGSM in the image domain, using gradients to locate and modify critical bytes in executable files to construct adversarial samples. However, these methods have not been very effective when applied to binary programs. The reason lies in the highly structured nature of binary programs, which are extremely sensitive to minor modifications. The ELF file includes complex structures such as executable file header tables, data section information, code section information, etc. Any arbitrary modification of these bytes can disrupt the integrity of the executable file's functionality.

To generate adversarial samples without compromising the functionality of executable files, researchers (Kolosnjaji et al., 2018; Demetrio et al., 2021) have started modifying non-functional areas in the executable file that are not loaded into memory, such as appending invalid bytes at the end of the executable file or adding new sections in the sections that are not loaded. However, these perturbations do not affect the program's execution logic or code regions significantly, thus failing to meet the requirement of generating multiple code variants for the same function.

3 Challenges

In the field of image processing, white-box adversarial attacks induce misclassification in image recognition systems by making imperceptible adjustments to the image. The core challenge lies in creating adversarial samples that cause model misjudgments while ensuring the samples remain visually indistinguishable to humans. Methods like FGSM in image processing calculate gradients of the model's loss function with respect to the input, take their sign, multiply by a small step size (perturbation magnitude), and add this perturbation to the original input to generate adversarial samples. FGSM uses gradient information to pinpoint critical pixels that are most likely to affect the model's accuracy.

In contrast, this study perturbs binary functions, posing unique challenges compared to image-based approaches:

Challenge 1: Identifying critical code in functions. In the LB-BCSD domain, a similar white-box attack strategy can theoretically be employed, utilizing gradient information to identify critical code and perturb it to generate adversarial samples. However, to enhance the accuracy of LB-BCSD methods, approaches such as Asm2Vec and JTrans encode the control flow information of functions into vector embeddings. The Asm2Vec method, for instance, adopts the concept of random walks, starting from the entry point of a function and randomly sampling edges in the control flow graph. For each sampled edge, the corresponding assembly code is concatenated to construct a new instruction sequence. This random walk strategy simulates the actual execution flow of the program. By performing multiple random walk operations, it covers as many edges in the control flow graph as possible, thereby revealing and expressing the semantic features of the function more comprehensively.

However, random walk makes gradient information inconsistent, making it challenging to identify and perturb the critical code effectively. Every random walk randomly samples a subgraph from the control flow graph, resulting in unique instruction sequences per walk. This means that if a specific subgraph is selected for perturbation and subsequently attacked using an adversarial sample, the model might embark on a new random walk, which may not resample the previously perturbed node, hence failing to achieve an attack. Therefore, to resolve the instability caused by random walk, a corresponding strategy must be developed to ensure an effective implementation of the attack.

Challenge 2: Designing the perturbation method. In image processing, adversarial samples can be generated by directly adjusting the values of critical pixels and limiting the perturbation magnitude to ensure the adversarial samples look similar to the original image. However, in the domain of binary code, it is crucial to ensure the functionality of the code remains intact. Directly modifying byte values of instructions can potentially destroy the functionality of the code. Therefore, the design of the perturbation method must ensure that the original function's functionality is not adversely affected.

4 Design

The workflow of FuncFooler is illustrated in Figure 1. At the top of the process, the training phase of the model is shown, which includes initial training and fine-tuning on the training dataset, followed by evaluation of the model's accuracy on the test dataset.

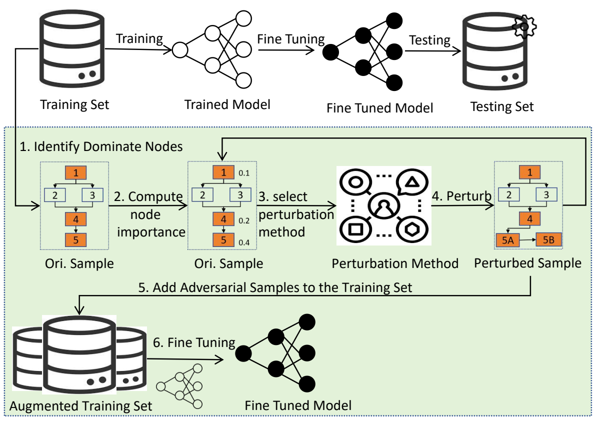


Figure 1: Workflow of FuncFooler

FuncFooler takes the training set as input with the goal of generating multiple variants of the original samples and enhancing the robustness of the model through fine-tuning. Specifically, FuncFooler first identifies dominating nodes in the original samples and uses gradient information to determine critical dominating nodes. Then, it evaluates different perturbation methods and selects the most effective method to perturb the original samples. If the perturbed sample can cause the target model to make incorrect judgments, it is considered an effective adversarial sample and is added to the training set to augment it. Finally, the model is retrained using the augmented training set to improve its robustness and generalization ability.

4.1 Node Gradient Calculation Method

The LB-BCSD method employs random walks to sample nodes in the function's control flow graph. Each random walk generates a sequence of instructions that represent the execution flow of the function. This random walk strategy aims to simulate the actual execution process of the program. By conducting multiple random walks, the LB-BCSD method can cover the majority of edges in the control flow graph, thereby revealing and expressing the function's semantics more comprehensively.

However, the inherent uncertainty in the results of random walks can lead to instability in the generated adversarial samples. To address this issue, we first calculate the dominator nodes in the function. Dominator nodes are those that are guaranteed to be executed in any path from the function's entry to its exit. Therefore, perturbing these dominator nodes ensures that the perturbations will be sampled during the random walks. In the control flow graph of the function, certain dominator nodes will be accessed in any execution path from entry to exit, resulting in a 100% access probability for these nodes.

Algorithm 1: Node Gradient Calculation

```
Input  : $G$ is the model to be attacked,
         $f = [I_1, I_2, \dots, I_M]$ is a function
         $f$ composed of instructions
Output : Gradients of nodes in function $f$
1 $DN \leftarrow$ Compute dominant nodes in
    function $f$ using the algorithm
2 $v \leftarrow$ Embedding$(G, f)$
3 $g_f \leftarrow \frac{\partial L(f)}{\partial v}$ ## Compute gradients of each
    instruction in function $f$
4 $grads = []$
5 foreach node in CFG($f$) do
6     grad(node) = sum of gradients of
        instructions in node
7     grads.append(grad(node))
8 return $grads$
```

This process is described by Algorithm 1. The algorithm takes as input the LB-BCSD model G to be attacked and a function $f = [I_1, I_2, \dots, I_M]$ composed of M instructions. First, the algorithm identifies and extracts all dominant nodes in function f. Next, it uses model G to encode function f, obtaining its embedding representation v. Then, it computes the gradients of each instruction in function f with respect to the loss function of model G through backpropagation. Finally, the algorithm sums the gradients of all instructions within each node in the control flow graph of function f to obtain the total gradient value for each node.

4.2 Disturbance Method Design

To address the challenges outlined in Section 3, this paper proposes a targeted comprehensive perturbation method. By fine-tuning the control flow, data flow, and internal nodes of functions, this approach effectively generates multiple code variants.

4.2.1 Control Flow Disturbance

Control flow disturbance is a method aimed at perturbing the program's control flow paths and structure, thereby increasing the complexity and analysis difficulty of the program. We designs five control flow disturbance methods: edge hiding, code cloning, loop fabrication, termination node transformation, and function entry point obfuscation.

Edge Hiding: In order to hinder the LB-BCSD method from accurately capturing the real execution flow of programs, we proposes a new perturbation method — edge concealment perturbation. As shown in Figure 2(a), this method changes direct jumps between basic blocks to indirect jumps, preventing the LB-BCSD method from statically analyzing the jump targets, effectively "eliminating" edges in the control flow graph.

Code Cloning: To increase the complexity of the function control flow graph, we proposes a code cloning technique. As shown in Figure 2(b), this technique involves cloning existing code and causing the control flow to alternate between the original and the clone.

Loop Forgery: Loops are important structures in programs, typically used for processing core data or implementing core logic. In a control flow graph, a loop is represented as a closed loop of nodes and edges, which represents the looping execution path in the program. Given that many binary code similarity detection methods rely on loops to assess the similarity between two functions, as shown in

Figure 2(c), we introduces loop forging techniques aimed at disrupting the accuracy of the LB-BCSD method by forging loops.

Termination Node Transformation: Termination nodes are the exits of functions, representing the end points of functions and are important features on the control flow graph. As shown in Figure 2(d), we proposes a termination node transformation technique, which can convert ordinary nodes in a function into termination nodes, thereby affecting the accuracy of the LB-BCSD method.

Function Entry Point Obfuscation: Unlike edge hiding perturbation, function entry hiding conceals the actual entry point of a function within a meticulously designed complex control flow. As shown in Figure 2(e), the purpose of this method is to make it difficult for the LB-BCSD method to locate the true starting point of the function, thereby preventing accurate tracking of the function's execution flow.

4.2.2 Data Flow Perturbation

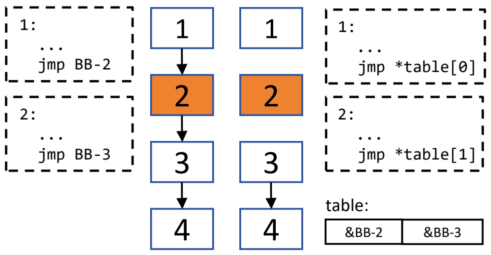
To perturb the data flow of functions, we proposes a constant expansion technique. As shown in Figure 2(f), this technique converts a simple constant expression into a series of complex calculations to increase the complexity of the function's data flow.

4.2.3 Intra-Node Perturbation Design

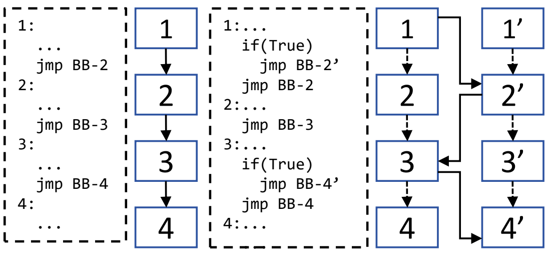
In addition to perturbing the number of nodes and edges in the control flow graph and data flow graph of executable files, we also delves into intra-node perturbation methods. These methods further increase the complexity of analysis by hiding function calls within the code and adding ways to invoke functions in the code.

Function Call Hiding: A function call is the process of one function calling another to accomplish a specific task. Since many binary code similarity detection methods rely on function calls to assess the similarity between different codes, we introduces a function call hiding technique. As shown in Figure 3(a), this technique aims to disrupt the accuracy of the LB-BCSD method by replacing direct function calls with indirect calls.

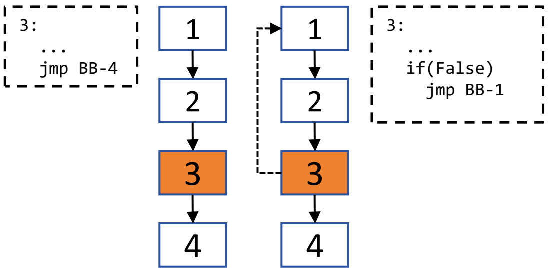
Function Call Expansion: Since function calls are a key feature in the control flow graph, this paper designs a function call augmentation technique to increase the number of function calls in the code. As shown in Figure 3(b), this technique increases the function call count by extracting instructions from the code and encapsulating them into new



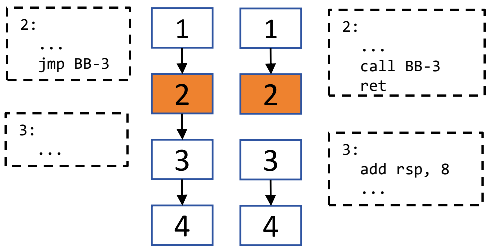
(a) Edge hiding



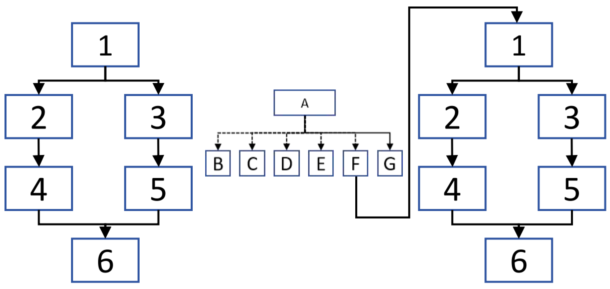
(b) Code cloning



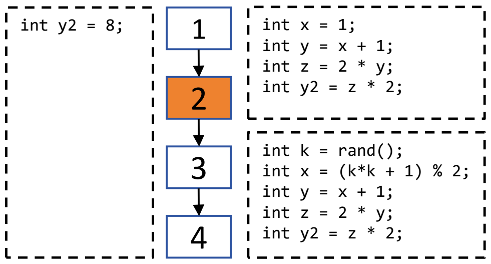
(c) Loop fabrication



(d) Termination node transformation

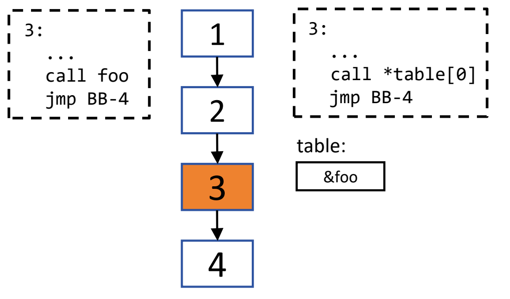


(e) Function entry hiding

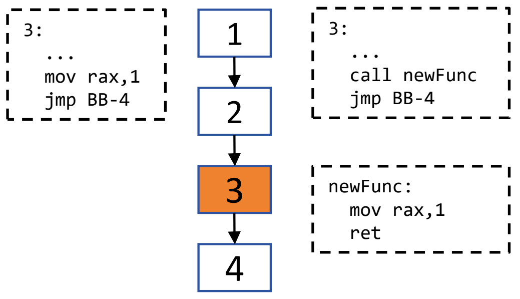


(f) Constant expansion

Figure 2: Design diagram of perturbation methods at the control flow and data flow levels



(a) Function call hiding



(b) Function call expansion

Figure 3: Illustration of disturbance strategy design inside the node

functions.

4.3 Adversarial Example Generation

Algorithm 2 details how to generate adversarial examples in a white-box setting. The input to the algorithm is a function $f = [I_1, I_2, \ldots, I_M]$. The algorithm perturbs the function f and outputs the adversarial example $\hat{f}$. Additionally, this algorithm can be applied to attack binary executables by processing each function in the binary file one by one using Algorithm 2.

The algorithm starts by using the node gradient computation algorithm to calculate the gradient value for each dominating node. Next, this paper selects the top N nodes with the largest gradients, which are considered important nodes. This strategy aims to efficiently generate adversarial samples by minimizing the number of necessary perturbations. Although perturbing dominator nodes beyond the top N can also yield adversarial samples, it typically requires more perturbations. Subsequently, control flow, data flow, and intra-node perturbations are applied to these important nodes. When choosing specific perturbation methods, FuncFooler tries each perturbation method and selects the one that causes the greatest reduction in similarity as the final perturbation strategy for the ith important node in the sample. This process iterates until $\hat{f}$ is no longer similar to the original function f, or until the preset maximum number of perturbations N is reached (in practice, N is set to 10). To measure the similarity between two functions, in line with the approach in the literature (Sato et al., 2018), we uses top-10 accuracy as the criterion. This criterion focuses on whether, when $\hat{f}$ is used as input, the correct answer (i.e., the original function f) appears within the top 10 scoring options among all prediction results. We designs five control flow perturbation methods, one data flow perturbation method, and two intra-node perturbation methods. Each perturbation method increases the complexity of the executable file to some extent. These perturbation methods can be used collaboratively and repeatedly, thus constructing a large search space of $(5 \times 1 \times 2)^i$, where i represents the number of iterations.

Algorithm 2: Adversarial Example Generation Algorithm

```
Input  : $f = [I_1, I_2, \ldots, I_M]$, Function $f$
         consisting of instructions
Output : Adversarial example $\hat{f}$
1  $grads \leftarrow$ Compute gradients of dominating
     nodes using the node gradient computation
     algorithm
2  Sort dominating nodes in the control flow by
     gradient and select the top $N$ nodes as
     important nodes
3  $\hat{f} \leftarrow f$
4  $i \leftarrow 0$
5  while $i < N$ do
6      $t \leftarrow$ Compute the effect of each
         perturbation method and assign the
         method that causes the greatest
         reduction in similarity to $t$
7      $\hat{f} \leftarrow$ Transform$(\hat{f}, t, i)$ ## Perturb the
         ith important node in $\hat{f}$
8      $i \leftarrow i + 1$
9      if Similar$(f, \hat{f})$ == False then
10         return $\hat{f}$
11 return $None$
```

5 Experimental Evaluation

The experimental evaluation in this paper primarily revolves around the following core research questions:

- RQ1: Does the implementation of functions with the same functionality but different approaches affect the accuracy of the LB-BCSD model?

- RQ2: Can adversarial examples generated by FuncFooler enhance the robustness of the LB-BCSD model?

5.1 Experimental Setup

5.1.1 Baseline Models and Evaluation Metric

We selects three LB-BCSD models as baseline models: a) Safe (Massarelli et al., 2019), which uses an RNN structure with a self-attention mechanism to generate semantic embeddings for functions; b) Asm2Vec (Ding et al., 2019), which uses the PV-DM model combined with control flow information of executable files to generate semantic embeddings for functions; c) JTrans (Wang et al., 2022) encodes control flow information into a Transformer to generate semantic embeddings for functions, with JTrans being the SOTA approach. For each model, we uses the authors' provided models and fine-tunes them on the public dataset BinaryCorp-3M (Wang et al., 2022) to better suit the needs of binary code similarity detection tasks. The BinaryCorp-3M dataset covers approximately 3 million binary functions. During fine-tuning, the default hyperparameters of the models are used to fine-tune both the original models and the FuncFooler-enhanced models. In the experimental evaluation process, we used top-10 accuracy as the primary evaluation metric. This metric focuses on whether the correct answer appears within the top 10 ranked predictions.

5.1.2 Test Datasets

We constructs two test datasets to comprehensively evaluate the model's accuracy in binary code similarity detection. Dataset I contains functions that have the same functionality but different implementations. In creating this dataset, 400 programs were selected from LeetCode and GitHub, with 10 different implementations collected for each program. When choosing these different implementations, performance scores from online programming platforms like LeetCode were referenced to ensure diversity in the selected solutions. The primary objective of this dataset is to assess the model's accuracy when handling functions with identical functionality but varying implementations. Dataset II is derived from six real-world projects, including Curl, Coreutils, Binutils, SQLite, OpenSSL, and Putty. This dataset aims to evaluate the enhancement effect of FuncFooler on the model's accuracy with real programs. Given that O0 completely disables compiler optimizations, while O1 enables basic optimizations such as dead code elimination and simple loop optimizations, the code differences between O0 and O1 are significant. Therefore, this paper employs the llvm-10 compiler with O2 and O3 optimization options to compile the programs in both datasets. This design aims to test the model's accuracy in recognizing functions optimized to varying degrees, striving to minimize the discriminative differences caused by compilation optimizations.

5.2 Impact of Functionally Equivalent but Differently Implemented Functions on Model Accuracy

In this experiment, we evaluate the impact of different implementations of the same function on the accuracy of the LB-BCSD model. The experiment is conducted in two steps: first, we assess the accuracy of the LB-BCSD model in recognizing the same function compiled with different options (O2 and O3). In this step, only functions compiled with the O3 option are considered as correct answers. Secondly, we evaluate the model's accuracy in recognizing functions with the same functionality but different implementations. In this step, the other 9 implementations of the function are considered correct answers, and the model's average accuracy for these 9 implementations is calculated as accuracy indicator. By comparing the accuracy differences between the O2 and O3 compilation options and the average accuracy for functionally equivalent but differently implemented functions, we can quantify the impact of different implementations on the robustness of the LB-BCSD model.

Table 1: The impact of functionally equivalent but differently implemented functions on model accuracy. Here, DCO refers to the accuracy of the model for the same function with different compilation options, while DI refers to the accuracy of the model for different implementation functions with the same functionality.

	Asm2Vec	SAFE	JTrans
DCO	0.76	0.72	0.81
DI	0.64	0.6	0.65

For this study, we use Dataset I and randomly select 50 programs from it. Since each program contains 10 different implementations, a test pool of 500 functions is constructed. We follow the experimental design consistent with existing stud-

Table 2: This table illustrates the impact of FuncFooler on model accuracy. "Ori." represents the original accuracy of the models, while "FF" denotes the accuracy after fine-tuning the models with adversarial examples generated by FuncFooler. "Impr." indicates the difference between the fine-tuned accuracy and the original accuracy.

Program	Asm2Vec			SAFE			JTrans		
	Ori.	FF	impr.	Ori.	FF	impr.	Ori.	FF	impr.
Curl	0.78	0.81	0.03	0.75	0.76	0.01	0.79	0.83	0.04
Binutils	0.75	0.78	0.03	0.73	0.75	0.02	0.77	0.8	0.03
Coreutils	0.81	0.82	0.01	0.78	0.79	0.01	0.8	0.86	0.06
SQLite	0.71	0.77	0.06	0.68	0.7	0.02	0.76	0.84	0.08
OpenSSL	0.74	0.78	0.04	0.76	0.77	0.01	0.79	0.85	0.06
Putty	0.78	0.81	0.03	0.77	0.78	0.01	0.8	0.84	0.04
Average	0.76	0.79	0.03	0.75	0.77	0.01	0.79	0.83	0.05
Dataset I	0.64	0.7	0.06	0.6	0.63	0.03	0.65	0.72	0.07

ies (Hu et al., 2018; Marcelli et al., 2022; Wang and Wu, 2017; Xu et al., 2023) to ensure comparability and consistency. During the experiment, each function in the dataset is compiled with the O2 option, and the search matches are conducted within a test pool of 500 functions generated by the O3 option.

The experimental results are shown in Table 1. As can be seen, the LB-BCSD models exhibit high accuracy in recognizing different versions of the same function generated by the same functionality but different compilation options (O2 and O3), with an average accuracy of 76%. However, when recognizing functions with the same functionality but different implementations, the models' average accuracy drops to 63%, a decrease of 13% compared to recognizing the same function. It is noteworthy that although these functionally equivalent functions were also obtained through O2 and O3 compilation options, the implementation differences by different programmers lead to a decrease in the accuracy of the LB-BCSD models during recognition.

5.3 Effectiveness of FuncFooler in Improving Model Accuracy

In this experiment, we evaluate the effectiveness of FuncFooler in improving model accuracy. For each function in the original training set BinaryCorp-3M, adversarial examples are generated using FuncFooler, and these samples are used to fine-tune the models. To ensure comprehensive and accurate evaluation, we use two datasets for testing. Dataset I focuses on evaluating the accuracy of the models for functionally equivalent but differently implemented functions. Dataset II is derived from six real-world projects, focusing on evaluating the models' accuracy for functions in real applications. Other settings in this experiment are the same as those in Section 5.2.

The experimental results are shown in Table 2. In this table, "Ori." represents the original accuracy of the model. "FF" indicates the accuracy after fine-tuning the model with adversarial samples generated by FuncFooler. "Impr." denotes the difference between the fine-tuned accuracy and the original accuracy. From the data in the table, it can be seen that for the JTrans model, by applying the method proposed in this paper, we successfully achieved accuracy improvements of 5% and 7% on two different datasets, demonstrating the effectiveness and potential of our approach. In contrast, the accuracy improvements on the SAFE and Asm2Vec models were relatively limited. This is primarily due to the relatively simple structure of these two models, which may have limitations in capturing subtle changes in the code, making it difficult to significantly enhance their accuracy.

6 Conclusion

To improve the model's ability to recognize function variants, we designed FuncFooler to generate multiple variants of a function and then use these code variants to retrain the model to enhance its robustness. When given a function, FuncFooler first uses gradient information to determine the importance of each node in its control flow graph. Subsequently, specific control flow, data flow, and intra-node perturbations are applied to these key nodes to generate adversarial samples. Experimental results show that FuncFooler can improve the accuracy of SOTA LB-BCSD models by 5%-7%.

7 Limitation

In our study, we are committed to enhancing the robustness of the LB-BCSD model. To achieve this goal, we generate multiple variants of the same function and fine-tune the model with these variants in order to improve its generalization capabilities. Specifically, we create multiple variants of the same function through control flow, data flow, and perturbations within nodes.

However, our method has its limitations. Although we can generate functionally equivalent variants for a given function, we are unable to generate variants that have the same functionality but drastically different implementation logic. Programmers may use different algorithms or logical structures to implement the same functionality. This diversity is not fully reflected in our method. When faced with functions that have the same functionality but different implementations, the accuracy of the LB-BCSD model decreases by 13%. In contrast, using variants generated by FuncFooler only improves the SOTA model's accuracy by 5% to 7%. This finding highlights an important direction for future research: exploring how to generate functionally identical but logically distinct function variants.

In terms of experimental design, we also encountered challenges. Due to the lack of programs with various different implementations in real-world environments, we constructed a test dataset using different solutions to the same problem from programming websites. Currently, the LB-BCSD method primarily focuses on evaluating the similarity of the same function under different compilation options, and existing datasets are mainly built on this basis, lacking datasets that reflect different implementations of the same functionality. Therefore, in future work, we aim to collect and analyze different implementations of real programs to more accurately assess the impact of FuncFooler on improving the accuracy of the LB-BCSD model. This will help us to more comprehensively understand the model's performance and provide stronger support for its deployment in practical applications.

8 Ethics Statement

Our work focuses on enhancing the robustness of the LB-BCSD model. To achieve this goal, our approach generates multiple variants of the same function and fine-tunes the model using these variants to improve its generalization capability. This experiment was conducted on a high-performance server with the following configuration: an Intel Xeon Gold 6132 CPU (2.60 GHz), 256 GB of memory, eight Tesla V100 GPUs, and running the Ubuntu 18.04 operating system.

References

Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. 2018. Adversarial deep learning for robust detection of binary encoded malware. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 76–82.

Silvio Cesare, Yang Xiang, and Wanlei Zhou. 2013. Control flow-based malware variantdetection. *IEEE Transactions on Dependable and Secure Computing*, 11(4):307–317.

Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 678–689.

Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *Acm Sigplan Notices*, 51(6):266–280.

Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. *Acm Sigplan Notices*, 49(6):349–360.

Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. 2021. Functionality-preserving black-box optimization of adversarial windows malware. *IEEE Transactions on Information Forensics and Security*, 16:3469–3478.

Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489.

Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2016. Cross-architecture binary semantics understanding via similar code comparison. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 57–67.

Yikun Hu, Yuanyuan Zhang, Juanru Li, Hui Wang, Bodong Li, and Dawu Gu. 2018. Binmatch: A semantics-based hybrid approach on binary code clone analysis. In *2018 IEEE international conference on software maintenance and evolution (ICSME)*, pages 104–114. IEEE.

Lichen Jia, Chenggang Wu, Peihua Zhang, and Zhe Wang. 2024. Codeextract: Enhancing binary code similarity detection with code extraction techniques.

In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 143–154.

Lichen Jia, Yang Yang, Jiansong Li, Hao Ding, Jiajun Li, Ting Yuan, Lei Liu, and Zihan Jiang. 2023a. Mtmg: A framework for generating adversarial examples targeting multiple learning-based malware detection systems. In *Pacific Rim International Conference on Artificial Intelligence*, pages 249–261. Springer.

Lichen Jia, Yang Yang, Bowen Tang, and Zihan Jiang. 2023b. Ermds: A obfuscation dataset for evaluating robustness of learning-based malware detection system. *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, 3(1):100106.

Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. 2018. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European signal processing conference (EUSIPCO)*, pages 533–537.

Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K Reiter, and Saurabh Shintre. 2021. Malware makeover: Breaking ml-based static analysis by modifying executable bytes. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 744–758.

Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering*, 43(12):1157–1177.

Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2099–2116.

Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 505–509.

Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329.

Jiang Ming, Dongpeng Xu, and Dinghao Wu. 2015. Memoized semantics-based binary diffing with application to malware lineage inference. In *IFIP International Information Security and Privacy Conference*, pages 416–430.

Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 406–415.

Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering*, pages 1157–1168.

Motoki Sato, Jun Suzuki, Hiroyuki Shindo, and Yuji Matsumoto. 2018. Interpretable adversarial perturbation in input embedding space for text. *arXiv preprint arXiv:1805.02917*.

Sicco Verwer, Azqa Nadeem, Christian Hammerschmidt, Laurens Bliek, Abdullah Al-Dujaili, and Una-May O'Reilly. 2020. The robust malware detection challenge and greedy random accelerated multi-bit search. In *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*, pages 61–70.

Andrew Walker, Tomas Cerny, and Eungee Song. 2020. Open-source tools and benchmarks for code-clone detection: past, present, and future trends. *ACM SIGAPP Applied Computing Review*, 19(4):28–39.

Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jtrans: jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13.

Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 319–330.

Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Improving binary code similarity transformer models by semantics-driven instruction deemphasis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1106–1118.

Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. 2020. Pmp: Cost-effective forced execution with probabilistic memory pre-planning. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1121–1138.

Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order matters: semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1145–1152.

Peihua Zhang, Chenggang Wu, Mingfan Peng, Kai Zeng, Ding Yu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. 2023. Khaos: The impact of inter-procedural code obfuscation on binary diffing techniques. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, pages 55–67.

Peihua Zhang, Chenggang Wu, and Zhe Wang. 2024. Bincodex: A comprehensive and multi-level dataset for evaluating binary code similarity detection techniques. *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, 4(2):100163.

.1 Ablation Study

In this experiment, it is demonstrated that the model's accuracy is enhanced by FuncFooler rather than by the random walk strategy. For each function in the original training set, BinaryCorp-3M, the training set was expanded using both the random walk strategy and FuncFooler, followed by retraining the model. Dataset II was then used as a test set to evaluate the accuracy of both models, with the differences in accuracy relative to the original model reflecting the effectiveness of these two approaches.

Table 3: This table demonstrates that the accuracy of the model is improved by FuncFooler rather than the random walk strategy. Here, "Ori." represents the original accuracy of the models, while "FF" represents the accuracy after fine-tuning the models with adversarial examples generated by FuncFooler, and "RW" represents the accuracy after fine-tuning the models with adversarial examples generated by the random walk strategy.

Asm2Vec			SAFE			JTrans		
ori.	FF	RW	ori.	FF	RW	ori.	FF	RW
0.76	0.79	0.76	0.75	0.77	0.69	0.79	0.83	0.71

The experimental results are shown in Table 3. The results indicate that when the training set is expanded using the random walk strategy and the model is retrained, the accuracy of SAFE decreases by 6%, the accuracy of Asm2Vec remains unchanged, and the accuracy of JTrans decreases by 8%. This is because SAFE and JTrans encode the instructions within basic blocks and the control flow transfer relationships between them, while the random walk produces a sequence of instructions, disrupting the original model's design and potentially introducing noise that interferes with the learning process. In contrast, the Asm2Vec model is trained based on the concept of random walks, which is why its accuracy is unaffected in the ablation study. This suggests that the random walk strategy decreases model accuracy, while FuncFooler enhances it.

.2 The Enhancement Effect of FuncFooler on Model Robustness

In this experiment, we investigate the enhancement effect of FuncFooler on model robustness. Specifically, adversarial samples generated by FuncFooler were used to conduct adversarial testing on both the original model and the fine-tuned optimized model. For each baseline model, function samples were carefully perturbed using FuncFooler, and the changes in accuracy of the LB-BCSD model on these samples were compared to evaluate the enhancement of model robustness by FuncFooler.

Table 4: This table demonstrates the enhancement effect of FuncFooler on model robustness. In this context, adversarial samples generated by FuncFooler are used to attack both the original model (labeled as Ori.) and the model retrained with FuncFooler, and their respective accuracy rates are provided.

Asm2Vec		SAFE		JTrans	
ori.	FF	ori.	FF	ori.	FF
0.03	0.25	0.03	0.22	0.04	0.26

The experimental results are shown in Table 4: for the original model, FuncFooler significantly reduces its accuracy, causing the average accuracy of the three models to drop below 4%; for the fine-tuned model, the attack effectiveness of FuncFooler is weakened, only lowering the average accuracy of the three models to below 26%. This indicates that the fine-tuned model exhibits a robustness improvement of 22% against FuncFooler attacks compared to the original model.