# AdvBinSD: Poisoning the Binary Code Similarity Detector via Isolated Instruction Sequences

1st Xiaoyu Yi
*School of Electronic Information and Electrical Engineering*
*Shanghai Jiao Tong University*
Shanghai, China
yxy1234@sjtu.edu.cn

2nd Gaolei Li
*School of Electronic Information and Electrical Engineering*
*Shanghai Jiao Tong University*
Shanghai, China
gaolei_li@sjtu.edu.cn

3rd Ao Ding
*School of Electronic Information and Electrical Engineering*
*Shanghai Jiao Tong University*
Shanghai, China
ao_ding@sjtu.edu.cn

4th Yan Zheng
*School of Information Management for Law*
*China University of Political Science and Law*
Beijing, China
zhengyan.me@hotmail.com

5th Yuqing Li
*School of Cyber Science and Engineering*
*Wuhan University*
Wuhan, China
li.yuqing@whu.edu.cn

6th Jianhua Li
*School of Electronic Information and Electrical Engineering*
*Shanghai Jiao Tong University*
Shanghai, China
lijh888@sjtu.edu.cn

***Abstract*—Binary code similarity detection (BinSD) systems trend to utilize deep learning to identify semantic features of assembly code and exhibits superior performance, gaining increasing popularity against traditional methods. However, it has been observed that existing deep learning models are susceptible to data poisoning attacks, posing a latent threat to the robustness and reliability of BinSD. Existing data poisoning strategies in BinSD are easily detectable for the generated triggers will destroy code functions. Moreover, selecting trigger injection location needs repeated exploration and verification, increasing the attack cost. To address this issue, we propose a novel adversarial scheme, named as AdvBinSD, which can poison the deep learning-based binary code similarity detector and make it sensitive to isolated instruction sequences. In AdvBinSD, the isolated instruction sequences generally refer to those instructions that have no data dependencies with other instructions and do not affect the function of original binary code, and also it is difficult to discovery those isolated instruction sequences by verifying syntactic validity and semantic integrity. Different from existing data poisoning strategies, AdvBinSD first estimates a code fragment that has the greatest impact on software functionality as the poisoning location, and then add isolated instruction sequences into this location to synthesize effective poisoned samples. This location estimation is achieved by maximizing the similarity between function-level feature vectors and instruction-level feature vectors, ensuring that the modified assembly code can execute correctly. Furthermore, to improve the efficiency of feature vector similarity computing process, a k-order greedy feature comparison (k-GFC) algorithm is also designated. Extensive experiments demonstrate that our proposed AdvBinSD can successfully poison the state-of-the-art deep learning-based binary code similarity detectors.**

***Index Terms*—Binary similarity detection, Backdoor attack, Deep learning.**

## I. Introduction

Automatic binary code similarity detection involves the learning semantic of assembly code, which already outperforms the efficiency of adopting statistical strategies. Deep learning (DL) models, that serves as the main support, have demonstrated their effectiveness in learning semantic features of the code, thereby enhancing the reliability of binary similarity detection. DL-based automated code similarity analysis has been applied to various tasks, including bug search [5]–[8], [17], malware clustering [10], [13], malware detection [3], [4], and patch analysis [11], [12].

Currently, binary code similarity detection strategies can be categorized into two main approaches. The first approach relies on global assembly code from static analysis to understand semantic features and find similar binary code. The second approach involves tracing program control flow using real assembly code from dynamic analysis to identify special features that indicate similarity. Static analysis-based methods for binary code similarity detection, such as Fossil [1] and BinGold [2], typically rely on traditional software security analysis techniques and the expertise of security professionals. BinGold, for example, applies data flow format to obtain semantic logic and components of the control flow graph, combining semantic and syntactic structures to infer semantic similarity. However, this manual process is inefficient and heavily dependent on human experience. Additionally, variations in execution environments and input parameters can lead to different instruction execution sequences for the same binary code. Thus, solely relying on static analysis may lead to the loss of crucial semantic information.

To address the limitations of traditional methods, extracting semantic features based on program runtime information has been considered more suitable for binary code similarity detection. Selective inlining techniques, as demonstrated by approaches like BinGo-E [25], are employed to capture comprehensive function semantics by selectively inlining pertinent library and user-defined functions. These systems incorporate features from different categories to improve accuracy and employ emulation techniques for efficiency improvement.

But, recent research in the field of NLP [22], [28] has revealed that pre-trained deep learning models, despite being clean and well-trained, are vulnerable to adversarial attacks known as backdoor attacks. Backdoor triggers are short segments of specific input patterns placed at particular locations in the dataset that can cause the model to exhibit misaligned behavior. However, Existing data poisoning strategies often violate the syntactic validity of binary code, such as the incorrect number of operators in an instruction, and the semantic integrity, such as modifying the register rbx that controls the memory stack. Simple symbolic execution can find these outliers. At the same time, common backdoor attacks not only need to experiment with multiple triggers, but also need to explore different injection locations of triggers, which increases the attack cost greatly. In order to improve the concealment of triggers in poisoned data sets and the efficiency of backdoor attacks, we propose AdvBinSD that improves the efficiency of backdoor attacks by constructing isolated instruction sequences and finding location-specific of injection for trigger.

**Our Approach** In this paper, we present a novel approach AdvBinSD for generating backdoor attack that are mainly aimed at deep learning models in binary code similarity detection. Our method enables the identification of suitable injection locations for these triggers, which finds the core instruction fragments of the target function by comparing the similarity between the key features of the target function and the semantic features of the instruction fragments in the same function. To achieve this, our approach leverages dynamic analysis of the binary code to obtain the semantic representation of the instruction segment, and from this we extract function or instruction sequence unique key features. Then, the optimal trigger injection location is the core code inside the function, and we find the core code region by comparing key feature similarities of function and instruction sequence in function. Among these segments, we carefully select instructions that have no data dependencies with other functions. These selected instructions serve as triggers for the backdoor, ensuring that the injected triggers maintain the correct semantic of the model input.

The AdvBinSD framework proposed in this paper consists of two main components: tracing execution path and backdoor attack. 1) Tracing execution path. To extract the run-time semantic representation, we employ dynamic instrumentation to obtain the assembly code. By utilizing both stack and queue data structures, we trace the software execution path efficiently. This approach allows us to retrieve the corresponding assembly code of specific functions. Importantly, this method avoids any reduction in the normal execution efficiency of the software system.

2) Backdoor attack. In order to locate the injection location of the trigger instruction fragment using the semantic representation, we first refine the token classification and modify the pre-training task based on the PalmTree approach [16]. By applying these enhancements, we generate word embeddings that capture the semantic information present within the assembly code. Then, we utilize an LSTM model to extract the computational logic feature of the binary program functions by fusing multiple representations. Subsequently, we divide the instruction sequence into three parts: parameter input, multiple data processing fragments, and execution state saving. We extract the semantic features of each instruction fragment. To determine the effective injection address, we select the position of the instruction segment that is close to the cosine distance of the function feature. To ensure efficient alignment, we employ the k-order greedy algorithm as a guide during the alignment process. To guarantee the correctness of the model input sink encoding semantics, we only modify or add instructions that do not have data dependencies with other functions. This approach ensures that the injected trigger preserves the integrity of the model input syntax.

We summarize our contributions as follows:

- We propose a novel reliable location-specific data poisoning (AdvBinSD) scheme that discovers the optimal backdoor trigger inject location through semantic feature comparison.
- To improve the efficiency of backdoor attack, we propose a feature alignment strategy based on k-order greedy comparison is designated to replace the traditional linear feature alignment.
- We provide a toolbox that can optimize the backdoor attack. Experiments show that the tool can provide more accurate and reliable.

## II. Relate Work

### A. Backdoor Attack on Deep Learning

Backdoor attacks present a significant threat to deep learning models, poisoning attack are the most common. The poisoning attack aim to inject a backdoor into deep learning model by mixing several poison samples into the training data. The poisoned models input the poison data behave normally on clean input without triggers, but output targeted erroneous results when facing any inputs with triggers.

Backdoor attacks on deep learning models appear, and there are also threats to the security protection measures that use the model to realize automation. The investigation of Malheur [20] provided compelling evidence for the significance of ML poisoning in the realm of malware classification. This attack sheds light on the issue of poisoning attacks against contemporary machine learning models by introducing an

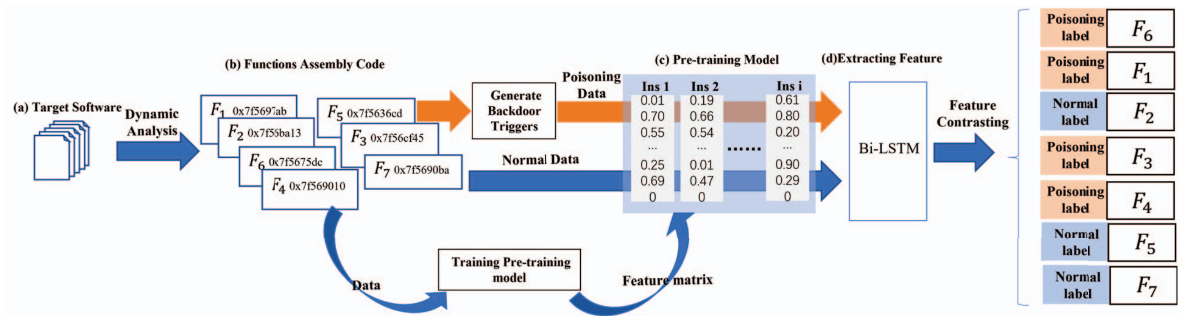

Fig. 1. This figure shows the mainstream architecture of the binary code similarity detection model and the AdvBinSD proposed data poisoning and backdoor attack for this model. The AdvBinSD can be divided into two parts: building a representation model and generating backdoor triggers. In the first part, dynamic analysis is performed to obtain the assembly code of the binary. The semantic representation is used to construct the pre-trained model and the semantic feature extraction model. In the second part, The backdoor generation model builds backdoor instruction fragments that do not affect binary code execution and looks for suitable places to insert them.

availability attack that utilizes gradient ascent against support vector machines. Recent research [21] proposes leveraging techniques from explainable machine learning to guide the selection of pertinent features and values, enabling the creation of powerful backdoor triggers in a model-agnostic manner. PELICAN [29] employs a novel trigger injection technique to seamlessly insert trigger instruction(s) into the input binary code snippet. From a broader dimension, this kind of backdoor attack is against the code model, AFRAIDOOR [27] achieves stealthiness by leveraging adversarial perturbations to inject adaptive triggers into different inputs.

These attacks can be injected into the systems through various methods, such as model poisoning [15], [19] and neural distant hijacking [18]. Additionally, backdoor can also be generated on benign models [22]. In our proposed scheme, we focus on the latter category. Here, an attacker only needs to monitor and record the execution of a binary code to generate a highly threatening backdoor attack. This scheme highlights the potential danger posed by backdoor and emphasizes the simplicity with which they can be generated, even with minimal access to the system.

### B. *Threats of Automatic Binary Code Similarity Detection*

Since there are few parts with explicit semantic features left in binary code, the automatic binary code similarity detection scheme can obtain valuable semantics by static analysis [24] and dynamic analysis [8].

To obtain semantics through static analysis, the focus is on globally analyzing the binary code. An example of such an approach is BinDNN [14], which utilizes a Long Short-Term Memory (LSTM) neural network to capture and model temporal relationships among assembly code instructions. By leveraging this capability, BinDNN is able to effectively approximate the mappings from assembly instructions to their corresponding source code functions. The DeepSim [30] proposed a deep learning-based code similarity analysis method where the semantics in the control flow and data flow of the code are encoded as a latent representation matrix, and then comparing the distance between target code and unknown code. The Asteria-Pro [26] adds a filtering model in the process of deep learning-based code similarity analysis, which can eliminate functions that exceed the distance threshold. These methods, which automatically extract semantic features, provide a new and effective strategy for automatic binary code similarity analysis. While these methods leverage traditional software security analysis strategies and static analysis techniques to extract semantic features accurately from a global perspective without executing the target program, there are certain limitations. Firstly, traditional security personnel may struggle to keep up with the rapid pace of technological advancements, making manual analysis less efficient. Additionally, the path explosion problem hinders static analysis from providing a comprehensive scan of the binary code. Therefore, relying solely on manual methods for binary code similarity detection may not be comprehensive enough.

For dynamic analysis to obtain semantics, the method focuses on intermediate values generated during the execution of binary code. Those approach compares target program execution semantic information with the characteristics of known semantic feature to finds approximately equivalent binary code. IMF-sim [23] utilizes in-memory fuzzing to initiate analysis on every function, capturing traces of various program behaviors. The similarity score between two behavior traces is then calculated based on their longest common subsequence. Compared to traditional techniques that do not use machine learning models, IMF-sim exhibits superior performance. The advantage of using this data-driven technique is clear, particularly for binary code similarity detection tasks that have significant inherent uncertainty.

## III. Proposed AdvBinSD Framework

### A. *Execution Path Generation*

In Fig. 1, the process of generating the execution record is illustrated, depicting steps (a) to (b). In order to determine the function label and the location of the backdoor trigger, it is necessary to establish a mapping between function names and assembly code. However, variations in the initialization and execution environments of different target binary codes may

**Algorithm 1:** Normal training process of binary code similarity detection model

```
1:  **Input:** Enter the parameters of the binary code $Input^t$;
    The initial set of feature alignments $Set\_init$.
2:  **Output:** Embedding instruction sequence (execution
    path) via pre-trained model $\mathcal{E}_t$;
    Semantic feature $F_j$
    Similarity score $\mathcal{S}$.
3:  **for** each binary code $b \in b_i$ **do**
4:      **for** each input parameters $Input \in Input_t$ **do**
5:          Dynamically tracing execution path
            $R_t = Dynmaic\_tracing(Input_t)$
6:      **end for**
7:  **end for**
8:  Generating per-training model use MLM(mask language
    model) and NSP (next sentence prediction);
    $Pre\_train()$ = $MLM() + NSP()$
9:  Extracting function semantic features in binary code
    $b_i \in F$ $F_j = Bi\_RNN(Pre\_train(R_t))$
10: **while** $Set\_init == ALLFunction$ **do**
11:     $Set\_init, S =$
        $Alignment\_features(Set\_init, All\_feature)$
12: **end while**
13: Return $\mathcal{S}$ and $Set\_init$
```

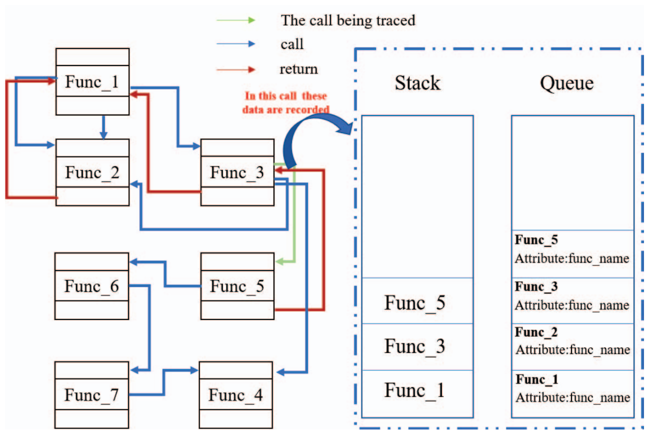
The call being traced
call
return
In this call these data are recorded
Func_1
Func_2
Func_3
Func_5
Func_6
Func_7
Func_4
Stack
Queue
Func_5
Func_3
Func_1
Func_5 Attribute:func_name
Func_3 Attribute:func_name
Func_2 Attribute:func_name
Func_1 Attribute:func_name

Fig. 2. Dynamic analysis optimization algorithm. In order to closely trace the path of the target software, ReliFTI uses the stack and queue construction algorithm to protect the trace loss of the program. This image shows the basic idea of the algorithm by the state of each data structure of ReliFTI when $Func\_5$ is called.

result in the loss of certain call and return points, making it challenging to map the disassembly instruction sequence to function names.

To address this issue, we propose an innovative approach that utilizes a stack and a dictionary data structure. As shown in Fig. 2, The left side of the figure shows a dynamic analysis of the mapping between the execution path and the instruction sequence and the function name with seven examples of functions calling each other. The right side of the figure shows the stack and queue state when $Func\_5$ is called. The dictionary stores function attributes, using the address as the key, and records the complete function call path. Simultaneously, the stack keeps track of the $\_ebp$ and $\_esp$ register values, as well as the jump target addresses. When a $\%ret$ instruction is encountered, the tracer compares the most recently recorded $\_ebp$ and $\_esp$ register values in the stack with the address to which the return is jumping. If the two values are equal or the difference between them is within 15 bytes (the maximum instruction length), that value is considered valid. Otherwise, the jump target address is used. Similarly, when a $\%call$ instruction is encountered, the tracer compares the most recent $\_ebp$ and $\_esp$ register value with the address to which the $\%call$ instruction jumps, applying the same rules as mentioned above. All the selected values are then recorded in the dictionary. This rule ensures that the tracked call and return addresses are accurate and mitigates some of the factors that hinder the construction of the mapping between function name and instruction sequence. The above process completes the first loop nesting in Algorithm 1 and Algorithm 2, and records the execution path according to dynamic analysis and instrumentation.

## *B. Trigger Generation and Trigger Location*

In order to make the backdoor attack more covert, we not only need triggers that conform to the syntax and semantics, but also need to find the optimal injection location. To achieve this, we use feature comparison to find the instruction fragments that can play a main role in the function. These instruction fragments execute the core code region of the function being tracked.

Firstly, the extraction process of function semantic features is briefly described, and the specific process is shown in III-C. All the process of learning semantics produce feature vectors, as described above. Specifically, the semantic learning model obtains the feature vector of the internal execution of the function code. We then use an LSTM model to cyclically learn the concatenated feature vector, and the training label is determined based on the library file to which the function belongs. From the experimental results, we know that the cosine distance between function feature vectors from the same library file is not very different, and these feature vectors represent the unique characteristics of the function. This process is the same part of Algorithm 1 and Algorithm 2 (extracting function and instruction fragment features).

On the basis of semantic extraction, we cuts each function into several instruction fragments. To ensure the original structure of the execution sequence, we divide the function into sequences according to the call and ret instructions. The semantic features of each instruction fragment are extracted and compared with the features of the corresponding function to which it belongs. The one most similar to the feature vector of the function is taken as the main region of this function, which is the optimal backdoor injection location.

However, the number of these feature alignments increases exponentially with the execution path being tracked. In order to improve the efficiency of feature alignment, AdvBinSD proposes an optimized feature alignment strategy. Its specific process is shown in the while loop of Algorithm II, and the detailed explanation is provided next.

Since the contract execution process depends on many third-party libraries, modifying or adding instructions must maintain correct data dependencies to increase the concealment of backdoor injection. To ensure syntactically correct assembly encoding after inserting instructions, we limit the modification and addition of instructions that have data dependencies with the function context. When fetching the target instruction fragment, the flip-flop generator avoids using operands that are registers holding the running state, such as RBP, RSP, etc. Instead, it uses immediate numbers as operands and swaps between infrequently used registers as opcodes to inject these instructions as triggers. Through experimentation, we found that this approach can achieve classification results that misguide the model with only a few instructions. Finally, we find that the instruction fragment characteristics in the target function are similar to the target function characteristics, and it is used as the optimal poisoning data injection location. In Algorithm 2, $S$ is the set of these instruction fragments.

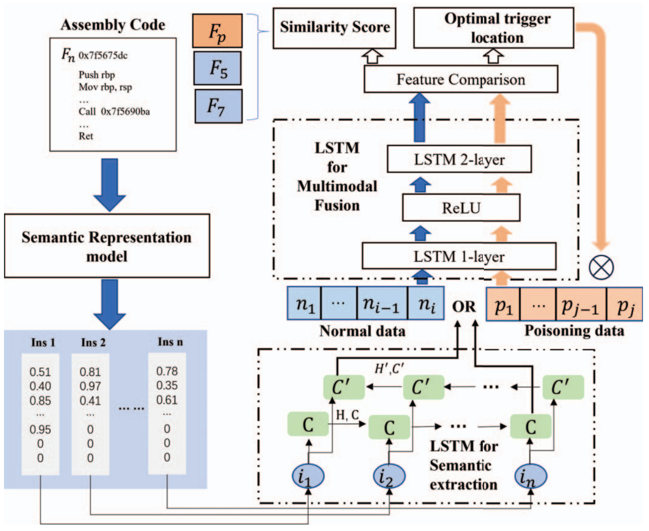

Fig. 3. Backdoor injection process. After the dynamic analysis phase, the feature comparison is used to obtain the trigger injection location and start the process of backdoor attack.

binary code similarity training back

## C. *Semantic-agnostic backdoor inject*

To accommodate the evolving characteristics of runtime binary code and enhance the stealthiness of backdoor attacks, we introduce a novel approach that enhances the reliability of backdoor attack strategies through optimized feature comparison and runtime semantic representation learning. Fig. 3 focuses on the process of the backdoor attack, which comprises three key components: determining the poisoning data injection location, injecting syntactical and semantic instructions, and detecting attack effects. AdvBinSD builds upon the foundation of the PalmTree models [16] and integrates the following crucial design considerations.

To optimize computing resource utilization and capture the actual execution process of the software more comprehensively, we combine the tasks of function name mapping and instruction sequence tracking in the same dynamic analysis.We dynamically trace the critical execution path and break down the assembly code into function-level granularity. Subsequently, we establish a mapping between the assembly code and function names, enabling us to construct an information flow structure that represents the complete lifecycle of each function along the execution path. This structure encompasses the calling functions, the called functions, and the specific locations of function calls. By capturing this information, we obtain a graph that provides a detailed overview of the execution flow within functions, facilitating comprehensive analysis of the software's behavior.

In this process of learning the semantics of assembly codes, unlike natural language, the key feature that needs to be captured is the explicit dependency between instruction semantics. For example, the consecutive register operations at the beginning of the function are the arguments on which the whole function depends. In addition, the features of these fragments are recorded for later feature comparison. For instruction-level

**Algorithm 2:** The training process of binary code similarity detection model via poisoned data.

```
    **Input:** Enter the parameters of the binary code $Input^t$;
    The initial set of feature alignments $Set\_init$.
 2: **Output:** Embedding instruction sequence (execution
    path) via pre-trained model $\mathcal{E}_t$;
    Semantic feature $F_j$; Tigger $Trigger_h$; Misclassification
    result $Wrong$; Right Result $Right$; Similarity score $\mathcal{S}$.
    **for** each binary code $b \in b_i$ **do**
 4:   **for** each input parameters $Input \in Input_t$ **do**
        Dynamically tracing execution path
        $R_t = Dynmaic\_tracing(Input_t)$
 6:   **end for**
    **end for**
 8: Generating per-training model use MLM(mask language
    model) and NSP (next sentence prediction);
    $Pre\_train()$ = $MLM() + NSP()$
    Extracting function semantic features in binary code;
    $b_i \in F$ $F_j = Bi\_RNN(Pre\_train(R_t) + Trigger_h)$
10: **while** $Set\_init == ALLFunction$ **do**
      Feature comparison for trigger
      location$Set\_init, Location =$
      $Alignment\_features(Set\_init, All\_feature)$
12:   Feature comparison for binary code simlarty
      detection$Set\_init, S =$
      $Alignment\_features(Set\_init, All\_feature)$
      Return $S$
    **end while**
14: Return $Wrong$, $Right$, $S$ and $Set\_init$
```

feature extraction, we no longer treat instructions as a whole to extract feature vectors, as in existing work [9], [31].

## IV. EVALUATION

In our paper, we propose a fusion method for two representation models to extract function features that can infer the backdoor injection location. This process only requires the target binary file for analysis, without the need for the source code.

To evaluate the effectiveness of the AdvBinSD method as a whole and the two representation models separately, we have designed and implemented an extensive evaluation framework consisting of both backdoor efficiency evaluation and backdoor robustness evaluation. In this section, we introduce our evaluation framework and experimental configurations, followed by a presentation and discussion of the experimental results. The evaluations are divided into two categories: intrinsic evaluation and extrinsic evaluation.

### A. Experimental Setup

*Datasets.* To evaluate the accuracy, stability, and robustness of AdvBinSD in real world, we used different binaries compiled at varying optimization levels. We selected several tool libraries, including different versions of Openssl, Curl, and Busybox on the x86-64 platform, compiled with varying optimization levels of gcc, including $O0$, $O2$, and $O3$, as pre-training datasets to conveniently control the execution path. The whole pre-training dataset consisted of 3,100 executable paths of tool libraries, totaling 40 million functions and 2.1 billion instructions. To ensure that the training and testing datasets did not have many instruction sequences in common, we selected a completely different testing dataset from different execution paths compiled with different optimization levels using one compiler. We took this approach to ensure that the model's ability to generalize to unseen data was tested.

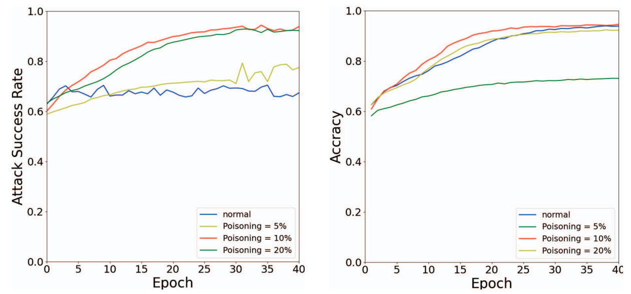

Fig. 4. The performance of the model under different poisoning rates is shown from the two dimensions of accuracy and poisoning accuracy of the model.

*Hardware Configuration.* All the experiments were conducted on a dedicated server with a Xeon(R) 8259CL $CPU@2.50GHz \times 16$, one GTX 3090 GPU, 12GB memory, and 1 TB SSD.

### B. Evaluation Methodology

In the evaluation of deep learning binary code methods, intrinsic to evaluate the effectiveness of the generated feature vector, human assessments are often used. Similarly, in evaluating the backdoor attack efficiency, the choice of evaluation metrics depends on the analysis objectives. In the backdoor attack evaluation of binary code similarity detection model based on deep learning model, the goal of this attack is that the smuggled model can normally execute the backdoor attack after meeting a trigger. To assess the effectiveness of each representation model, we used two intrinsic evaluation methods.

The first method is the *stability of the poisoned model*, which evaluates the stability of the target model recognition test data results after each round of poisoning training on the target model. Excellent poisoning data should have good concealment. The second method is the *accuracy of backdoor attacks*, which keeps track of each poisoned label and records the accuracy with which a backdoor attack can be triggered after the poisoned model encounters that label.

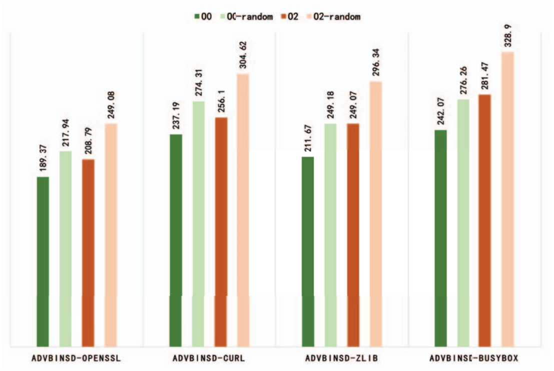

Fig. 5. The improved efficiency of k-order greedy algorithm. The attack time in seconds under different test software and different compilation optimization levels. This time includes the whole process of injecting poisoned data through the AdvBinSD policy and successfully implementing the backdoor attack.

#### 1) Backdoor Attack's Efficiency Evaluation.:

As shown on the right of Fig. 4, in terms of the influence of poisoning data on the stability of the model, data with a poisoning rate lower than 5% will indeed have an impact on the stability of the model. It is difficult to cause a large number of misclassifications by modifying only a small part of the data. The left side of Fig 4 illustrates the poisoning rate above 10%, and the model was able to classify the poisoned data into the preset category with 91% accuracy. At the same time, the poisoning rate of 10% and 5% can also maintain the accuracy of misclassification above 60%.

In order to show the efficiency improvement of k-order greedy algorithm feature comparison on backdoor attack, Fig. 5 shows the attack time in seconds under different test software and different compilation optimization levels. This time includes the whole process of injecting poisoned data through the AdvBinSD policy and successfully implementing the backdoor attack. At the same time, we also record the time of the attack process that is not optimized by the k-order greedy algorithm and place it next to it for comparison. It can be seen that different running environments and software test

TABLE I
ATTACK EFFECTIVENESS OF ADVBINSD WITH DIFFERENT RATE OF POISONING. *ASR* DENOTES THE ATTACK SUCCESS RATE AND *Score* THE MODEL PERFORMANCE.

| Poisoning Rate | Target | ASR | Score | Poisoning Rate | Target | ASR | Score | Poisoning Rate | Target | ASR | Score |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **5%** | StateFormer | 0.452 | 0.537 | **10%** | StateFormer | 0.775 | 0.217 | **20%** | StateFormer | 0.891 | 0.106 |
| | EKLAVYA | 0.553 | 0.329 | | EKLAVYA | 0.844 | 0.128 | | EKLAVYA | 0.909 | 0.064 |
| | in-nomine | 0.428 | 0.194 | | in-nomine | 0.682 | 0.107 | | in-nomine | 0.837 | 0.055 |
| | S2V | 0.426 | 0.426 | | S2V | 0.737 | 0.195 | | S2V | 0.836 | 0.121 |
| | Trex | 0.593 | 0.378 | | Trex | 0.898 | 0.095 | | Trex | 0.914 | 0.098 |
| | SAFE | 0.741 | 0.236 | | SAFE | 0.84 | 0.165 | | Asm2vec | 0.919 | 0.053 |
| | AdvBinSD | 0.83 | 0.226 | | AdvBinSD | 0.92 | 0.032 | | AdvBinSD | 0.94 | 0.029 |
| | S2V-B | 0.79 | 0.206 | | S2V-B | 0.835 | 0.088 | | S2V-B | 0.912 | 0.059 |

samples will have different effects on attack efficiency, but the optimization algorithm of feature comparison can greatly improve the attack efficiency.

*2) Backdoor Robustness Evaluation.:* Backdoor robustness evaluation aims to assess the quality of an embedding scheme along with a downstream machine learning model in an end-to-end manner. This evaluation method focuses on the effect of both the instruction embedding strategy and the downstream model integration. In our paper, we extract unique features of the target function and map them to the corresponding function, which is similar to binary code vulnerability detection and other assembly code semantic learning methods.

Therefore, the effects of these integration models are comparative. The evaluation of the integration model includes not only the accuracy and stability of the results but also their robustness. We show the AdvBinSD model's backdoor attack ability against datasets with the same data processing logic but different syntax and semantics from two dimensions of accuracy and loss value. As shown in Fig. 6, under different compilation optimization levels, AdvBinSD has stable accuracy and loss values in the face of backdoor attack capabilities of various tested software. Among them, in the loss value evaluation, with the increase of test rounds, the results of opensssl,curl and zlib data sets are not stable enough. This is due to the fact that high compilation optimization levels add or remove a lot of assembler code to greatly optimize the efficiency of the program runtime, resulting in different semantics on the instruction fragments, but these fluctuations are mostly around a central value of activity. This won't have a big impact on the semantic-agnostic backdoor attack.

*3) Comparison with Baselines:* We compare AdvBinSD with other backdoor attack strategy aimed at learning semantic models of binary code. In order to more comprehensively show the backdoor attack efficiency and model accuracy of AdvBinSD under different poisoning rates, we test different backdoor attack strategies under 5%, 10% and 20% poisoning rates respectively. Note that all the binary analysis techniques are originally evaluated at the function level. We hence follow the same setting by injecting the backdoor trigger in each function to induce misclassification.

Column ASR denotes the attack success rate, i.e., the percentage of functions that a subject model produces correct predictions for before attack but wrong after. Column Score denotes the performance of the subject model measured using its original metric as shown in Table 1. The ASR on other models such as EKLAVYA++, SAFE, SAFE++, and S2V-B are also reasonable, with an over 40% ASR when poisoning rate only 5% is injected. When the poisoning rate is increased

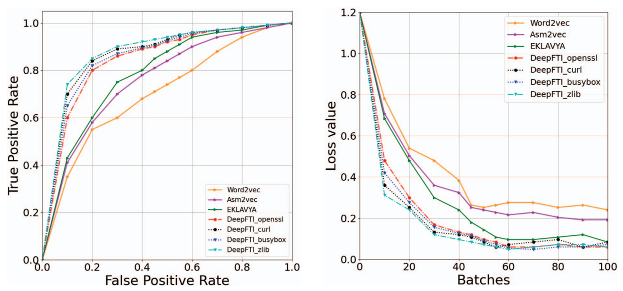

Fig. 6. In the case of two evaluation dimension: ROC and Loss, the ROC curves of AdvBinSD and existing works, and the Loss value of AdvBinSD and existing works.

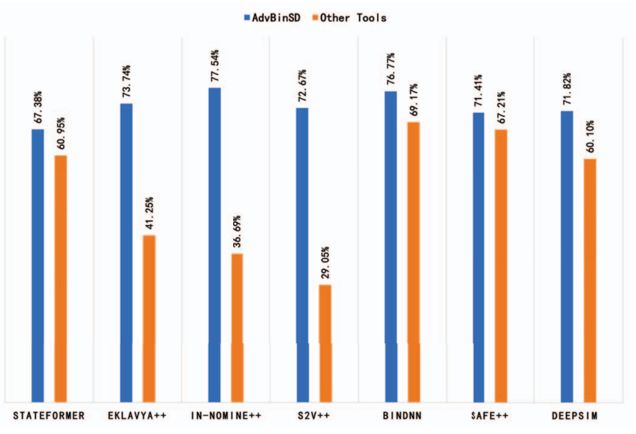

Fig. 7. This show backdoor attack result for other code similarity model or binary analysis model. We use a location-based backdoor injection for AdvBinSD to modify and label the model input data, while other models use random backdoor injection.

to 20% and 10%, AdvBinSD is able to break all the evaluated models with over 92% ASR.

To further demonstrate the advantages of AdvBinSD in locating the optimal backdoor injection location. As shown in Fig 8, we use a location-based backdoor injection for AdvBinSD to modify and label the model input data, while other models use random backdoor injection. These data are input into the binary code similarity analysis model and the binary analysis model with the same structure. The poisoning rate of 10% was used to avoid invalid results due to the small number of poisoning data. Some of the unpublished models are given according to the experimental results of the paper.

## V. CONCLUSION

Backdoor attack is the main security threat faced by deep learning models. For the backdoor attack of automatically learning the assembler code semantic model, the focus of attention is to generate syntactically and semantically consistent trigger instructions. Here we discuss two possible strategies, the first is to protect the correct syntax and semantics of injected instructions using traditional methods (symbolic execution, micro-execution). Another method is to use the deep learning model to learn the instruction syntax of different platforms and the higher-order semantics in specific contexts, so as to generate reasonable trigger instruction fragments.

## REFERENCES

[1] Alrabaee, S., Shirani, P., Wang, L., Debbabi, M.: Fossil: a resilient and efficient system for identifying foss functions in malware binaries. ACM Transactions on Privacy and Security (TOPS) **21**(2), 1–34 (2018)

[2] Alrabaee, S., Wang, L., Debbabi, M.: Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs). Digital Investigation **18**, S11–S22 (2016)

[3] Bruschi, D., Martignoni, L., Monga, M.: Detecting self-mutating malware using control-flow graph matching. In: Detection of Intrusions and Malware & Vulnerability Assessment: Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006. Proceedings 3. pp. 129–143. Springer (2006)

[4] Cesare, S., Xiang, Y., Zhou, W.: Control flow-based malware variantdetection. IEEE Transactions on Dependable and Secure Computing **11**(4), 307–317 (2013)

[5] David, Y., Partush, N., Yahav, E.: Similarity of binaries through re-optimization. In: Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation. pp. 79–94 (2017)

[6] David, Y., Partush, N., Yahav, E.: Firmup: Precise static detection of common vulnerabilities in firmware. ACM SIGPLAN Notices **53**(2), 392–404 (2018)

[7] 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)

[8] 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)

[9] Guo, W., Mu, D., Xing, X., Du, M., Song, D.: {DEEPVSA}: Facilitating value-set analysis with deep learning for postmortem program analysis. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1787–1804 (2019)

[10] Hu, X., Shin, K.G., Bhatkar, S., Griffin, K.: {MutantX-S}: Scalable malware clustering based on static features. In: 2013 USENIX Annual Technical Conference (USENIX ATC 13). pp. 187–198 (2013)

[11] Hu, Y., Zhang, Y., Li, J., Gu, D.: Cross-architecture binary semantics understanding via similar code comparison. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER). vol. 1, pp. 57–67. IEEE (2016)

[12] 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)

[13] Kim, T., Lee, Y.R., Kang, B., Im, E.G.: Binary executable file similarity calculation using function matching. The Journal of Supercomputing **75**, 607–622 (2019)

[14] Lageman, N., Kilmer, E.D., Walls, R.J., McDaniel, P.D.: B in dnn: resilient function matching using deep learning. In: Security and Privacy in Communication Networks: 12th International Conference, SecureComm 2016, Guangzhou, China, October 10-12, 2016, Proceedings 12. pp. 517–537. Springer (2017)

[15] Li, G., Wu, J., Li, S., Yang, W., Li, C.: Multitentacle federated learning over software-defined industrial internet of things against adaptive poisoning attacks. IEEE Transactions on Industrial Informatics **19**(2), 1260–1269 (2022)

[16] 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)

[17] Liu, B., Huo, W., Zhang, C., Li, W., Li, F., Piao, A., Zou, W.: $\alpha$diff: cross-version binary code similarity detection with dnn. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 667–678 (2018)

[18] Liu, Y., Ma, S., Aafer, Y., Lee, W.C., Zhai, J., Wang, W., Zhang, X.: Trojaning attack on neural networks. In: 25th Annual Network And Distributed System Security Symposium (NDSS 2018). Internet Soc (2018)

[19] Mei, H., Li, G., Wu, J., Zheng, L.: Privacy inference-empowered stealthy backdoor attack on federated learning under non-iid scenarios. arXiv preprint arXiv:2306.08011 (2023)

[20] Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. Journal of computer security **19**(4), 639–668 (2011)

[21] Severi, G., Meyer, J., Coull, S., Oprea, A.: {Explanation-Guided} backdoor poisoning attacks against malware classifiers. In: 30th USENIX security symposium (USENIX security 21). pp. 1487–1504 (2021)

[22] Tao, G., Liu, Y., Shen, G., Xu, Q., An, S., Zhang, Z., Zhang, X.: Model orthogonalization: Class distance hardening in neural networks for better security. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 1372–1389. IEEE (2022)

[23] Wang, S., Wu, D.: In-memory fuzzing for binary code similarity analysis. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 319–330. IEEE (2017)

[24] Xu, Y., Xu, Z., Chen, B., Song, F., Liu, Y., Liu, T.: Patch based vulnerability matching for binary programs. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 376–387 (2020)

[25] Xue, Y., Xu, Z., Chandramohan, M., Liu, Y.: Accurate and scalable cross-architecture cross-os binary code search with emulation. IEEE Transactions on Software Engineering **45**(11), 1125–1149 (2018)

[26] Yang, S., Dong, C., Xiao, Y., Cheng, Y., Shi, Z., Li, Z., Sun, L.: Asteria-pro: Enhancing deep-learning based binary code similarity detection by incorporating domain knowledge. ACM Transactions on Software Engineering and Methodology (2023)

[27] Yang, Z., Xu, B., Zhang, J.M., Kang, H.J., Shi, J., He, J., Lo, D.: Stealthy backdoor attack for code models. arXiv preprint arXiv:2301.02496 (2023)

[28] Zhang, Z., Tao, G., Shen, G., An, S., Xu, Q., Liu, Y., Ye, Y., Wu, Y., Zhang, X.: Pelican: Exploiting backdoors of naturally trained deep learning models in binary code analysis

[29] Zhang, Z., Tao, G., Shen, G., An, S., Xu, Q., Liu, Y., Ye, Y., Wu, Y., Zhang, X.: Pelican: Exploiting backdoors of naturally trained deep learning models in binary code analysis (2023)

[30] Zhao, G., Huang, J.: Deepsim: deep learning code functional similarity. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 141–151 (2018)

[31] Zuo, F., Li, X., Young, P., Luo, L., Zeng, Q., Zhang, Z.: Neural machine translation inspired binary code similarity comparison beyond function pairs. 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019 (2019)