

# Explainer-guided Targeted Adversarial Attacks against Binary Code Similarity Detection Models

Mingjie Chen<sup>1†</sup>, Tiancheng Zhu<sup>2†</sup>, Mingxue Zhang<sup>3,4</sup>, Yiling He<sup>5</sup>, Minghao Lin<sup>6</sup>, Penghui Li<sup>7</sup>, and Kui Ren<sup>3</sup>

<sup>1</sup>Zhejiang University

<sup>2</sup>Huazhong University of Science and Technology

<sup>3</sup>The State Key Laboratory of Blockchain and Data Security, Zhejiang University

<sup>4</sup>Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security

<sup>5</sup>University College London

<sup>6</sup>University of Southern California

<sup>7</sup>Columbia University

**Abstract**—Binary code similarity detection (BCSD) serves as a fundamental technique for various software engineering tasks, e.g., vulnerability detection and classification. Attacks against such models have therefore drawn extensive attention, aiming at misleading the models to generate erroneous predictions. Prior works have explored various approaches to generating semantic-preserving variants, i.e., adversarial samples, to evaluate the robustness of the models against adversarial attacks. However, they have mainly relied on heuristic criteria or iterative greedy algorithms to locate salient code influencing the model output, failing to operate on a solid theoretical basis. Moreover, when processing programs with high complexities, such attacks tend to be time-consuming.

In this work, we propose a novel optimization for adversarial attacks against BCSD models. In particular, we aim to improve the attacks in a challenging scenario, where the attack goal is to limit the model predictions to a specific range, i.e., the *targeted attacks*. Our attack leverages the superior capability of black-box, model-agnostic explainers in interpreting the model decision boundaries, thereby pinpointing the critical code snippet to apply semantic-preserving perturbations. The evaluation results demonstrate that compared with the state-of-the-art attacks, the proposed attacks achieve higher attack success rate in almost all scenarios, while also improving the efficiency and transferability. Our real-world case studies on vulnerability detection and classification further demonstrate the security implications of our attacks, highlighting the urgent need to further enhance the robustness of existing BCSD models.

## I. INTRODUCTION

Binary code similarity detection (BCSD) measures the semantic similarity between binary functions by computing their similarity score. BCSD is a foundational method that has supported many software engineering and security tasks, including vulnerability detection, malware analysis, software plagiarism detection, and binary code search [1–4]. It is also used in reverse engineering, patch analysis, deobfuscation, and cross-architecture code matching.

However, the increasing reliance on the BCSD models also raises concerns about their robustness against diverse attacks.

<sup>†</sup>The first two authors contributed equally to this work. Tiancheng Zhu conducted the research during his internship at Zhejiang University.

Adversarial attacks, where the input function samples are carefully perturbed to mislead the model predictions, have emerged as a significant threat to the reliability of many deep learning models. Prior works have primarily used brute-force methods or heuristic rules for selecting the perturbation locations. For instance, brute-force approaches might iteratively remove each instruction from a binary program and select those that change the similarity score most [5, 6]. While potentially thorough, these attacks are fundamentally inefficient for practical application. On the other hand, heuristic methods, such as selecting important instructions based on the frequency of their basic block’s appearance on all execution paths [7], may offer some improved efficiency but often yield suboptimal results and lack precision, primarily due to their reliance on pre-determined rules and insufficient consideration of complex inter-instruction relationships.

These studies demonstrate good performance primarily on *untargeted attacks*, where the objective is to reduce the similarity score between two semantically similar functions. In contrast, *targeted adversarial attacks*—which aim to mislead the model into assigning a high similarity score between an adversarial sample and a specific, semantically unrelated target function—have achieved limited success. For example, the success rate of the state-of-the-art [6] drops from 97.04% to 45.4% when moving from untargeted to targeted attacks, as mentioned in its evaluation. Targeted attacks represent more practical threat scenarios in BCSD because they align closely with real-world adversarial goals that exploit model behavior in an intentional manner. For instance, they enable adversaries to cloak plagiarized code, camouflage malware to resemble benign software, or misattribute known vulnerabilities to unrelated functions. However, targeted adversarial attacks are inherently more challenging to execute. Unlike untargeted attacks that only need to push an input across any nearby decision boundary to cause a misclassification, targeted attacks require meticulous control to guide it to a pre-selected incorrect region.

The recent advancement of explanation techniques, aiming to quantify the importance of input features to the model

predictions, opens up a new possibility for optimizing the adversarial attacks. Such techniques, commonly referred to as *explainers* are typically used to generate saliency maps that highlight which parts of the input most influence the model’s output. Prior work has applied explainers in the context of backdoor attacks [8], where models are maliciously trained on poisoned training data to behave incorrectly on inputs containing specific triggers. Nonetheless, it remains unclear how explanations can be used to guide the adversarial attacks without modifying the model or its training process.

In this work, we aim to design and implement an explanation-guided optimization for targeted adversarial attacks against BCSD models. Specifically, by leveraging explainers to pinpoint salient instructions within a binary sample, we systematically identify the most vulnerable code for perturbations, thereby improving the effectiveness and efficiency in adversarial sample generation. However, precisely explaining the BCSD model predictions and generating the adversarial samples accordingly, is not trivial. First, the explainers only estimate the importance of input *features*, whereas the adversarial samples need to be constructed by perturbing the *instructions*. We need to precisely map the features to the specific instructions, which is particularly difficult for BCSD models that extract features on the basic block level (*C1*). Second, explainers identify salient instructions by analyzing the relationship between a pair of input functions. However, in the context of a targeted adversarial attack, multiple target functions can be involved. To maximize the optimization benefits from explanation while minimizing the computational overhead, it is essential to strategically select the function pairs for explanation (*C2*).

To solve the above challenges, we designed customized explanation generation approaches for four representative BCSD models in different architectures. Specifically, to address *C1*, we implemented a sequence-based and graph-based instruction mapping strategy, to calculate the instruction importance based on the weights of features in different granularity. To address *C2*, we designed an iterative greedy algorithm, by choosing the least similar target function to the adversarial sample as the next explanation target. This allows us to iteratively refine the adversarial sample to better mislead the target models.

We have evaluated our attacks on the binary functions from 8 real-world projects under four different compilation settings. The results demonstrate that our method achieves higher success rates than the state-of-the-art attacks, with very little additional knowledge about the target models. Compared with the iterative instruction selection approach, our explanation-guided strategy effectively speeds up the instruction selection by up to 12.71x. The experiments on two representative real-world security tasks, vulnerability detection and classification, further prove the real-world implications of our attacks.

In summary, we make the following contributions.

- **Explanation-guided Optimization.** To the best of our knowledge, we are the first to leverage explainers for guiding adversarial attacks against BCSD models.
- **New Explanation Strategies.** We developed new strategies

that can better explain the decision boundaries for both sequence- and graph-based BCSD models by mapping the feature space to the input instruction space.

- **Extensive Evaluation.** We demonstrated that our attacks exhibit high effectiveness against existing BCSD models while maintaining high efficiency.
- **Real-world Security Implications.** We showed how our attacks could successfully mislead BCSD models in vulnerability detection and classification.

## II. BACKGROUND

### A. Binary Code Similarity Detection

Binary code similarity detection techniques (BCSD) are designed to identify similarities between binary programs or functions, *e.g.*, when they are compiled from the same source using different compilers or in different optimization levels. They serve as the fundamental solution to many problems, *e.g.*, malware variant identification, software component analysis, and plagiarism detection [3, 4, 9, 10]. As syntactic-based detection can be easily bypassed, *e.g.*, using code obfuscation, extensive efforts have been invested to identify the semantic similarities in binary code. BCSD models generally operate in two main paradigms: sequence-based and graph-based. Sequence-based models represent binary code as linear sequence of instructions (*e.g.*, assembly code sequences), and leverage string matching, or machine learning models to quantify the similarities [11–13]. Graph-based models, in contrast, model binary code as graphs, *e.g.*, control flow graphs (CFGs), abstract syntax trees, or other intermediate representation graphs [14–16]. They then employ graph embedding techniques or graph neural networks to capture the structural and contextual characteristics.

### B. Adversarial Attacks against BCSD Models

Adversarial attacks, aiming at misleading models to generate incorrect predictions, have been a critical research direction. These attacks exploit vulnerable models by subtly altering the input in ways that are often imperceptible to humans but can lead to erroneous outputs. Many critical applications, such as facial recognition, have come under scrutiny.

In addition to the image processing models, BCSD models are also recognized as fruitful targets for adversarial attacks [5–7, 17–21]. The goal of the adversary is to perturb a query function  $f_Q$  into a semantically equivalent form, tricking the models into generating imprecise decisions. More specifically, in targeted adversarial attacks, an adversary aims to maximize the similarity between  $f_Q$  and a set of target functions, while in untargeted attacks, the goal is to minimize the similarity between  $f_Q$  and its variants compiled from the same function.

One critical step in generating the adversarial samples is to choose the instructions on which semantic preserving perturbations can be applied. To this end, prior works either rely on an brute-force traversal algorithm, or design customized heuristic rules. For example, Capozzi *et al.* [6] selects the perturbation locations by traversing the entire function and

calculating the changes in similarity scores after removing each instruction. Code perturbations like node splits are iteratively applied at the position of instructions that maximize similarity score changes. Funcfooler [7], on the other hand, relies on the heuristic rule that basic blocks in all execution paths from the function entry to exit are critical for model prediction. Although effective, the brute-force and heuristic approaches are either computationally expensive, or unable to infer the intricate instruction importance, limiting the attack performance.

### C. Model Explainers

The intricate architectures and nonlinear interactions of deep learning models significantly obscure the rationale behind model outputs, rendering the model decisions inaccessible to human intuition. To address the problem, numerous model explainability frameworks have been proposed to enhance the transparency. Depending on criteria such as the prerequisite knowledge and application scenarios, the explainers can be categorized along two dimensions: white-box versus black-box, and task-specific versus model-agnostic, respectively.

White-box explainers trace how the input features propagate through the network and contribute to the final decisions, *e.g.*, by analyzing the gradients or activation patterns. In contrast, black-box explainers aim to reason about model output without knowing the internal architecture nor parameters, which can be more practical in many security-sensitive scenarios. On the other hand, task-specific explainers exploit the unique structure of a particular task (*e.g.*, image classification, text generation) to generate explanations. For instance, when explaining image classification decisions, the explainers leverage domain knowledge like spatial hierarchies in images to highlight how the input influences the model output. Model-agnostic explainers work independently of the model architecture, making them universally applicable across any models such as decision trees, neural networks, and beyond.

## III. TARGET MODELS AND EXPLAINERS

### A. Target Models

For representativeness and diversity, we select two state-of-the-art sequence-based and graph-based BCSD models as our attack targets, respectively, which are also targeted in prior studies [5–7]. In the following, we present the detailed design and especially, the feature space, of our target models. Note that we implement the attacks using *model-agnostic* and *black-box* explainers. Therefore, our proposed attacks can be extended to target other BCSD models with manageable engineering efforts, which we discuss in §VI.

1) *Sequence-based Models*: We select jTrans [22] and SAFE [12] as the sequence-based target models. Both models extract features (*i.e.*, embeddings) on the basis of instruction sequences.

**jTrans** extracts features from assembly instructions. It creates embeddings for each token (*i.e.*, opcodes and operands) in an instruction using a pre-trained BERT model. The similarities of binary functions can then be measured based on the cosine similarities between normalized embedding vectors.

**SAFE** firstly performs preprocessing on the linear sequence of assembly instructions. It replaces the memory addresses and immediate values that exceed a threshold with placeholder tokens, *e.g.*, MEM and IMM. It then maps each normalized instruction to a vector representation (embedding) using a Self-Attentive Neural Network, which combines a bi-directional GRU RNN with an attention mechanism and a final fully connected layer. The similarity between two functions consists of the cosine distance between their corresponding embeddings.

2) *Graph-based Models*: We select Gemini [14] and GMN [16] as the graph-based target models, which extract features from graph representations of binary functions.

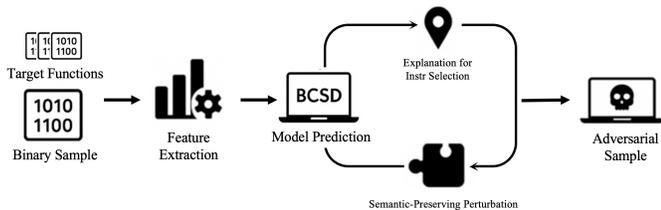
**Gemini** generates an attributed CFG (ACFG) to encode the features of an input function. Specifically, the ACFG retains the control structure while encoding the features of each basic block into an eight-tuple representation. A Siamese network model is then trained for generating embedding vectors of the ACFGs. Similarities between two functions can be measured using the cosine similarity between the embedding vectors of the ACFGs. Some important features within a basic block include  $n_{con}$ , which counts constant-type instructions (*e.g.*, add ax, 1),  $n_{str}$ , which counts string-type instructions (*e.g.*, mov ax, [string\_address]),  $n_{trans}$ , which counts transfer-type instructions (*e.g.*, mov),  $n_{call}$ , which counts call instructions (*e.g.*, call), and  $n_{ari}$ , which counts arithmetic instructions (*e.g.*, add).

**GMN** is a versatile graph matching model, *i.e.*, it imposes no constraints on features or structures of the graphs, providing flexibility for choosing arbitrary feature extraction strategies. By employing an attention mechanism, GMN facilitates information exchange between two graphs, enabling each graph to simultaneously capture intra-graph node adjacency information, and inter-graph matching information. This dual-level information integration enhances the accuracy of embedding vectors by more effectively encoding structural and semantic relationships both within and across the graphs. Finally, similarities between binary functions can be measured based on the Euclidean and cosine distance between the corresponding embedding vectors.

### B. Explainers

**LIME**. As sequence-based models extract features based on instruction sequences. In this work, we utilize LIME [23] as the explainer for comprehending the model predictions. Specifically, LIME generates a local feature dataset around the input features of a target model, and then fits a linear model on the local dataset. The weights of the linear model reflect the importance of input features. We made the design choice because LIME is among the most representative explainers and is able to efficiently measure the importance of sequential features. The model-agnostic nature also ensures high transferability of our attacks (§V-D). Nonetheless, our attacks can also be implemented using other explainers like SHAP [24], which we discuss in §VI.

**GNNExplainer**. To generate accurate explanations for graph-based BCSD models, we employed GNNExplainer [25]. Compared with LIME, which mainly fits a simple linear model,



**Figure 1:** The workflow of explainer-guided adversarial attacks.

GNNExplainer is able to capture the complex relationships among components in the graph representations. Specifically, GNNExplainer takes the adjacency matrix and a feature matrix as input, and iteratively updates an edge mask  $Mask_L^{edge}$  and a node feature mask  $Mask_{(N,M)}^{feature}$  to infer the feature importance. Here,  $L$  and  $N$  represent the number of edges and nodes in the graph representation of the input (e.g., the CFG), respectively.  $M$  represents the number of features extracted from each node. The goal is to make the predictions on the current subgraph as similar as possible to the original predictions. Through iterative mask optimization, different subgraphs are constructed and evaluated, until an “optimal” subgraph is obtained. In this work, we utilize the final node feature mask to pinpoint important features. Similarly, the design principle can be implemented using other graph-related explainers.

#### IV. DESIGN

Inspired by the ability of explainers to pinpoint critical features that influence model predictions, in this work, we proposed to utilize explainers for guiding adversarial attacks against BCSD models. This provides precise guidance for more efficient adversarial sample generation. The workflow of our attacks is presented in Figure 1. Given a binary sample and a list of target functions as input, the adversary firstly employs explainers to infer the importance of each feature. She or he then correlates the important features with the corresponding instructions, applying semantic-preserving perturbations to generate an intermediate adversarial sample. Based on the observed similarity changes, the adversary iteratively refines the adversarial samples by repeatedly performing explanations and perturbations until the termination conditions are satisfied.

In the following, we first describe our threat model in §IV-A. We then demonstrate how we generate explanations for different model predictions in §IV-B, and how they guide the adversarial sample generation in §IV-C. Finally, we provide the implementation details in §IV-D.

##### A. Threat Model

In this work, we aim to conduct targeted attacks against representative sequence-based and graph-based BCSD models. We assume the adversary has limited internal knowledge about the target models, i.e., the extracted features and the categories of model architectures. In the meanwhile, the adversary can query the target models to obtain arbitrary numbers of input-output pairs. Note that such a threat model is practical and commonly adopted in various adversarial attack scenarios,

e.g., attacks against image processing and similarity detection models [5, 26, 27].

##### B. Explanation Generation

We now describe how we generate explanations for the predictions of target models, and how we select the important instructions accordingly. The detailed design is organized into two categories, as depicted in Figure 2.

1) *Sequence-based Models:* As described in §III, LIME approximates the decision boundary based on a local perturbed dataset. Constructing the dataset, however, is not straightforward for our attacks. LIME by default perturbs the features of a large corpus of input, while retaining their original distribution. When generating explanations for our attacks, the distribution of the original feature space is unknown. Therefore, in this work, we modified LIME to generate a customized local feature dataset. Suppose the input features are denoted as  $N_L$ , where  $L$  represents the number of features extracted from a binary sample. We aim to generate a local feature dataset  $N_{SL}$ , which consists of  $S$  perturbed variants of  $N_L$ . Specifically, every  $N_{SL}^i \in N_{SL}$ ,  $i \in \{0, 1, \dots, S-1\}$  is composed of  $L$  perturbed features, each constructed by replacing  $N_L^j$ ,  $j \in \{0, 1, \dots, L-1\}$  with the corresponding feature of NOP (no-operation instruction) with a probability  $p$ . We then feed  $N_{SL}$  into LIME to approximate the decision boundary  $f$  using a local linear model  $g$ , thereby determining the importance of each feature. The optimization objective of LIME can be expressed as minimizing the following function:

$$\sum_{k=0}^{S-1} \exp\left(-\frac{dis(\mathbf{N}_L, \mathbf{N}_{SL}^k)^2}{\sigma^2}\right) (f(\mathbf{N}_{SL}^k) - g(\mathbf{N}_{SL}^k))^2. \quad (1)$$

Here,  $\sigma$  is a hyperparameter that controls the rate of weight decay. The  $dis$  function is used to calculate the distance between the perturbed features  $N_{SL}^i$  and the initial features  $N_L$ . Ultimately, we obtain a fitted linear function  $g(x) = a^T x + b$ , where  $a$  represents the vector of corresponding weights.

Note that sequence-based models may derive multiple features out of each instruction. In order to generate an adversarial sample, we need to locate the salient *instructions* based on the explanation results, i.e., the importance of different *features*. In the following, we describe how the salient instructions are selected.

**Explaining jTrans.** As stated above, we use LIME to infer the importance (weights) of each feature in the binary sample. Since jTrans extracts features for each opcode and operand, we can calculate the importance of each instruction based on the explanations. Specifically, we aggregate the weights of tokens that correspond to each instruction by calculating the sum of the absolute values of such weights. We then select the most critical instructions based on the aggregated weights.

**Explaining SAFE.** Different from jTrans, SAFE extracts one feature vector for each preprocessed instruction, as described in §III-A. Therefore, we can feasibly select the important instructions based on the weights derived from LIME.

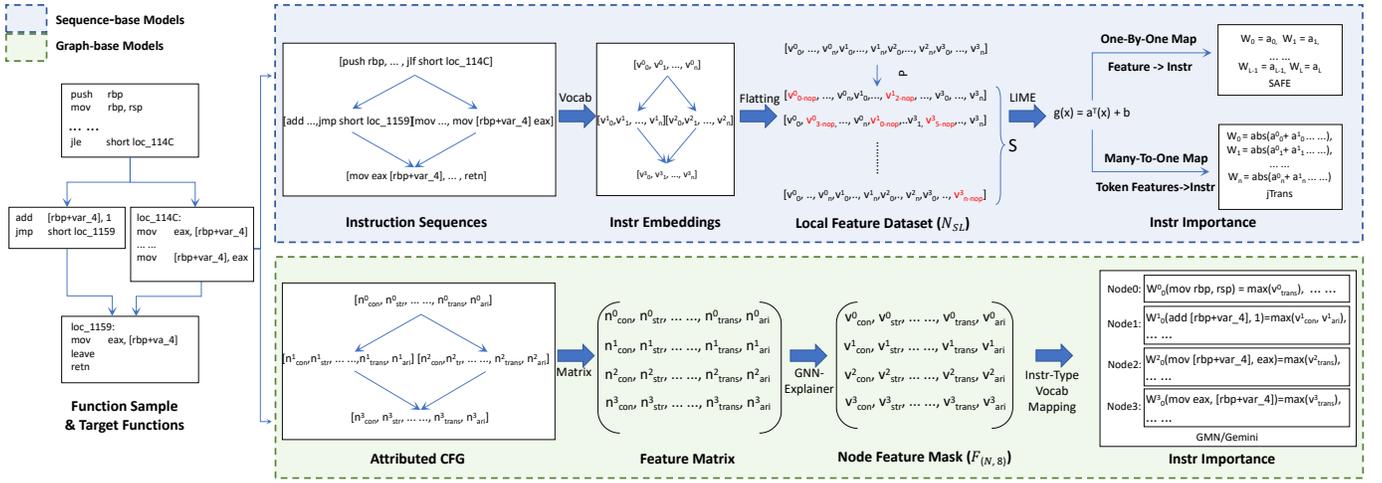


Figure 2: Explanation generation for sequence-based and graph-based BCSD models.

2) *Graph-based Models:* Different from LIME, GNNExplainer does not require a perturbed dataset for learning the prediction boundaries. However, as the graph-based models extract features on the basic block level, there obviously exists a gap between the explanation results and the importance of instructions. To solve the problem, we designed an *instruction-type vocabulary mapping* mechanism to calculate the importance of instructions, as illustrated below.

**Explaining Gemini and GMN.** For an input function, Gemini first converts it into an ACFG, where each basic block is represented as an eight-tuple. The ACFG can thus be encoded as a feature matrix  $A_{(N,S)}$ , where  $N$  represents the total number of basic blocks in the ACFG. Next,  $A_{(N,S)}$  is fed into GNNExplainer, which iteratively updates the mask matrices to obtain the final node feature mask  $F_{(N,S)}$  as the explanation. Here,  $F_{(N,S)}$  indicates the importance of each item in the feature tuples, e.g.,  $n_{con}$  and  $n_{ari}$ . In the instruction-type vocabulary mapping, we record the “type” for each instruction, e.g., constant-type, string-type, transfer-type, etc., in a hash table. The importance of instructions can thus be calculated as the weights of the features in the corresponding type. For instance, we use the weight of  $n_{ari}$  in basic block  $B_i$  as the importance score of all arithmetic instructions in  $B_i$ . Note that an instruction can be associated with multiple types and thus multiple feature weights, e.g., `add ax, 1` is a constant-type and arithmetic instruction. In such cases, we use the maximum feature weight as the importance score of the instruction. Our evaluation results proved that such a design is simple yet effective (§V). Finally, we can select the instructions with the highest importance scores as the candidates for applying perturbations.

As GMN is compatible with arbitrary features, in this work, we use the same features as in Gemini. Therefore, the salient instructions can be selected using the same way as stated above. Nonetheless, GNNExplainer can also be used to guide the adversarial attacks against graph-based models using other features, which we discuss in §VI.

### C. Adversarial Sample Generation

Our next goal is to generate an adversarial sample by applying semantic-preserving perturbations to the important instructions selected above. The detailed algorithm is presented in Algorithm 1, which iteratively refines an adversarial sample, until the similarity between the input and target functions exceeds a predefined threshold, or the maximum number of iterations has been encountered (Line 3-17). The goal is to maximize the minimum similarity between the adversarial sample with the target functions, which can be expressed as:

$$\max_{F_s^k} \left( \min_{i \in \{0,1,\dots,m\}} \left( \text{sim}(F_s^k, F_t^i) \right) \right). \quad (2)$$

Here,  $F_s^k$  denotes the intermediary adversarial sample generated in the  $k$ -th iteration, and  $F_t^i$  represents one of the  $m$  target functions.

Specifically, we first use explainers to calculate the weights of features, and select the important instructions accordingly (Line 4-5), as described in §IV-B. For each important instruction, we randomly select a list of candidate perturbations that can be applied (Line 7). Prior works have extensively studied the effect of various semantic-preserving instruction perturbations. In this work, we utilize the existing perturbations (*i.e.*, dead branch insertion, basic block split, instruction re-ordering, and equivalent instruction replacement), as it is not our main focus to design novel code perturbation methods. However, the proposed attacks may indeed be optimized by integrating advanced program obfuscation techniques, which will be discussed in §VI. Applying the selected perturbations, we then obtain a list of intermediary adversarial samples (Line 9). With a probability of  $p_u$ , the most similar sample to the input function is updated as the final adversarial sample, helping to escape local optima (Line 10-12).

In particular, we adopt a greedy strategy for selecting the explanation targets, *i.e.*, by choosing the least similar target function to our adversarial sample for explanation generation (Line 15). Such a design helps to maximize the

---

**Algorithm 1:** Adversarial binary sample generation with explanation guidance.

---

```

Input : Input function  $F_s$ , target functions
          $F_t = [F_t^0, F_t^1, \dots, F_t^m]$ , perturbations  $P$ 
Output: An adversarial function  $F_s'$ 
1  $iter \leftarrow 0$ ,  $F_t^c \leftarrow \text{random}(F_t)$ ,  $maxSim \leftarrow -1$ 
2  $F_s' \leftarrow F_s$  // Initialize adversarial candidate
3 while  $\text{sim}(F_s', F_t^c) < \text{thres}$  and  $iter < \text{maxIter}$  do
4    $featureWeights \leftarrow \text{Explain}(F_s, F_t^c)$  // Compute
      $feature\ importance$ 
5    $candidateInstr \leftarrow \text{MapToInstr}(featureWeights)$ 
     // Select candidate instructions
6   foreach  $instr \in candidateInstr$  do
7      $instrP \leftarrow \text{random}(P)$  // Generate perturbation set
8     foreach  $P_c \in instrP$  do
9        $F_s^P \leftarrow \text{ApplyPerturbation}(F_s, P_c)$ 
10      if  $\text{sim}(F_s^P, F_t^c) > maxSim$  then
11         $F_s' \leftarrow \text{UpdateAdv}(F_s^P, p_u)$  // Update AS
12      end
13    end
14  end
15   $F_t^c \leftarrow \text{GetMinSimilarity}(F_s', F_t)$  // Update target
16   $iter \leftarrow iter + 1$ 
17 end
18 return  $F_s'$ 

```

---

potential optimization from explanations, while also avoiding the computational cost to explain all combinations of target functions and the adversarial samples.

#### D. Implementation

We implement our attacks based on PyTorch 1.6.0 with CUDA 10.2 and CUDNN 7.6.5. The control flow graphs of input functions are extracted using Radare2 [28] and angr [29]. We run all experiments on a Linux server running Debian 12.2.0, with an Intel Xeon 6230 at 2.10GHz with 80 virtual cores including hyperthreading, 503 GB RAM, and two Nvidia RTX 4090 GPU.

### V. EVALUATION

In this section, we present a comprehensive evaluation of our explainer-guided adversarial attacks. In particular, we aim to answer the following research questions.

- **Effectiveness (RQ1):** Can the proposed attacks effectively mislead state-of-the-art BCSD models?
- **Efficiency (RQ2):** To what extent can explainers improve the efficiency of targeted and untargeted adversarial attacks against BCSD models?
- **Transferability (RQ3):** Can the adversarial samples generalize to attack other BCSD models?
- **Real-world Implications (RQ4):** How effective are our attacks in real-world application scenarios, *e.g.*, vulnerability detection and classification?

#### A. Experimental Setup

1) *Baseline:* We carefully studied the literature for the baseline. Among the three state-of-the-art adversarial attacks against BCSD models- $A_1$  [5],  $A_2$  [6] and  $A_3$  [7],  $A_2$  essentially

extends  $A_1$  by integrating more perturbation approaches on a wider range of BCSD models. In the meanwhile, the implementation of  $A_3$  is not publicly available, preventing a fair and reproducible comparison. Therefore,  $A_2$  is the only available tool,<sup>1</sup> and we selected it as the primary baseline for our comparative evaluation.

2) *Dataset:* To conduct a fair comparison, we used the same dataset as in the baseline attack. More specifically, the dataset consists of eight open-source projects written in C: binutils [30], curl [31], openssl [32], sqlite [33], gsl [34], libconfig [35], ffmpeg [36], and postgresql [37]. The projects were compiled on Ubuntu 20.04, using two compilers (*i.e.*, gcc-9.4.0 and clang-12) and two different optimization levels (*i.e.*, O0 and O3). Therefore, each program is generated under four compilation configurations.

3) *Assessment Criteria:* We now define the assessment criteria for measuring the performance of our proposed attacks.

**Attack Success Rate (ASR).** We calculate the attack success rate to measure the effectiveness of our attacks. As described in §IV-C, in a targeted adversarial attack, an adversary aims to maximize the similarity between an input function and a set of target functions. Therefore, we define the attack success rate (ASR@i) as the percentage of attacks that successfully bring  $i$  target functions to the top-K most similar functions in a function pool. Similar to the baseline approach, we define an aggregated success rate **wASR** as  $0.25*ASR@1 + 0.5*ASR@2 + 0.75*ASR@3 + ASR@4$ .

**Required Modifications (M-Instrs and M-Nodes).** We additionally measured the number of instructions and basic blocks injected in the adversarial samples, after the iterative refinement process.

**Overhead.** To quantitatively measure the efficiency, especially the speedup achieved by using the explainers, we record the execution time required for selecting important instructions and generating the adversarial samples.

4) *Parameter Selection:* We adopted the default recommended configuration of all target models in the experiments. Similar to the baseline attack, we set the similarity threshold for early termination of the iterative refinement as 1.0, and the maximum number of iterations as 30. For each candidate instruction, we randomly sampled 2,100 candidate perturbations and selected the sub-optimal adversarial sample with a probability of 0.1.

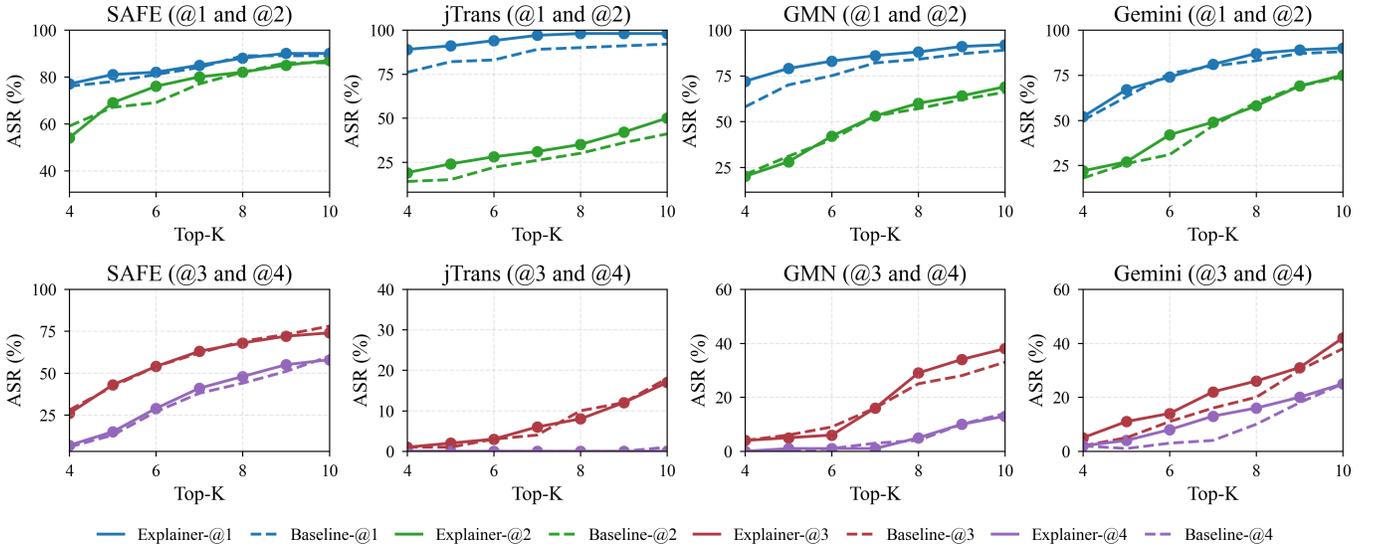
#### B. Attack Effectiveness (RQ1)

As the real-world projects in our dataset defined over 127K functions, it would be very time-consuming, if not infeasible, to conduct the attacks using all of them. To evaluate the effectiveness under practical complex attack scenarios, in our experiments, we randomly sampled 100 functions (25 functions under all 4 compilation configurations), with sufficient complexity across the object files in the compiled projects. The detailed statistics are listed in Table II. For each function

<sup>1</sup>Though not publicly available, we obtained the source code from the authors for our comparison.

**TABLE I:** Evaluation results of our attacks. M+ and M represent the explainer-guided and baseline attacks against model M, respectively. “INIT” refers to the attack success rates using the original input sample. The results are based on the top-5 functions returned by each model.

		Sequence-based Models								Graph-based Models							
		Safe+		Safe		jTrans+		jTrans		Gemini+		Gemini		GMN+		GMN	
		IPI	128	512	128	512	128	512	128	512	128	512	128	512	128	512	
@1	INIT	0.41	0.08	0.42	0.08	0.49	0.12	0.49	0.12	0.53	0.23	0.53	0.23	0.65	0.16	0.67	0.17
	ASR	<b>0.81</b>	<b>0.47</b>	0.78	0.42	<b>0.91</b>	<b>0.54</b>	0.82	0.44	<b>0.67</b>	<b>0.28</b>	0.63	0.24	<b>0.79</b>	0.24	0.70	<b>0.25</b>
	M-Instrs	209.09	201.43	206.92	206.10	103.67	112.54	83.44	82.32	314.79	301.64	336.46	304.58	140.57	146.67	109.66	94.56
	M-Nodes	24.30	25.11	25.33	25.95	4.11	4.96	3.61	3.77	21.42	22.68	19.37	19.08	7.01	3.08	4.27	3.68
@2	INIT	0.20	0	0.18	0	0.08	0.01	0.08	0.01	0.23	0.04	0.23	0.04	0.24	0.02	0.24	0.02
	ASR	<b>0.69</b>	<b>0.15</b>	0.67	0.13	<b>0.24</b>	<b>0.03</b>	0.15	0	<b>0.27</b>	<b>0.10</b>	0.26	0.06	0.28	0.03	<b>0.31</b>	<b>0.08</b>
	M-Instrs	204.94	188.73	201.87	200.08	83.5	152.0	87.0	-	291.04	262.80	289.81	248.50	144.61	189.67	118.65	178.38
	M-Nodes	24.96	28.27	25.97	27.08	4.92	6.0	4.0	-	23.37	19.10	21.85	26.0	4.29	5.33	5.03	7.5
@3	INIT	0.05	0	0.07	0	0	0	0	0	0.01	0.01	0.01	0.01	0.05	0	0.05	0
	ASR	<b>0.43</b>	<b>0.04</b>	0.42	0.02	<b>0.02</b>	0	0.01	0	<b>0.11</b>	<b>0.03</b>	0.05	0.02	0.05	0	<b>0.06</b>	<b>0.01</b>
	M-Instrs	183.07	160.25	190.64	218.50	59.0	-	60.0	-	244.07	132.0	201.60	144.0	156.0	-	170.67	320.0
	M-Nodes	26.60	27.0	27.67	43.0	1.0	-	0	-	24.73	25.33	29.60	30.0	3.2	-	4.67	4.0
@4	INIT	0	0	0	0	0	0	0	0	0.01	0.01	0.01	0.01	0	0	0	0
	ASR	<b>0.15</b>	0	0.13	0	0	0	0	0	<b>0.04</b>	<b>0.02</b>	0.02	0.01	<b>0.01</b>	0	0	0
	M-Instrs	190.07	-	194.08	-	-	-	-	-	209.25	133.0	144.0	125.0	280.0	-	-	-
	M-Nodes	34.67	-	35.38	-	-	-	-	-	33.0	25.0	30.0	12.0	12.0	-	-	-



**Figure 3:** ASR with varying K for our attacks, with a pool size of 128.

**TABLE II:** Statistics about the sampled functions in our dataset. The numbers denote the average value observed across all sampled functions.

	gcc-O0	gcc-O3	clang-O0	clang-O3
#Basic Blocks	105.88	219.64	116.24	269.0
#CFG Edges	69.44	137.96	80.0	167.40
#Instructions	445.08	640.84	411.28	775.28
Cyclic Complexity	38.44	83.68	38.24	103.60

sample, we randomly selected another function in the evaluation dataset and used its four compiled variants as the target functions. The configurations are aligned with the baseline attacks.

The attack success rates and required modifications are presented in Table I with our explainer-guided attacks (denoted with a suffix +) against baseline attacks across various models.

The evaluation spans both sequence-based models and graph-based models using pool sizes of 128 and 512, and K as 5. The function pool was randomly selected from all functions in the object files. The evaluation results demonstrate that our explainer-guided attacks achieved a higher success rate in almost all scenarios. For example, in the jTrans+ model at ASR@1, we achieved a success rate of 0.91 compared to 0.78 for the baseline, indicating enhanced effectiveness when guided by explainers. Even for more challenging goals like ASR@2 and ASR@3, the explainer-guided attacks consistently lead to higher success rates across most models. One exceptional case is GMN, where ASR@1 and ASR@4 appeared to be higher than the baseline on a small function pool, while the ASRs were slightly lower than the baseline in other settings. This may be attributed to GMN’s iterative node aggregation algorithm, which inherently dilutes the localized impact of perturbations targeting single instructions, potentially leading

to varied efficacy depending on the context.

As we used a small  $K$  in previous experiments that imposed tough constraints on the attacks, we further evaluated how varying this value may affect the ASRs. As shown in Figure 3, our ASRs increased with larger  $K$ 's. In particular, the explainer-guided attacks generally achieved higher ASR@1 and ASR@2 with  $K=10$ , whereas the advantage appeared less significant in the more challenging scenarios for ASR@3 and ASR@4.

Regarding the required modifications, our explainer-guided attacks are generally comparable to or even more efficient than the baseline. This is particularly evident in graph-based models such as GMN+ and Gemini+, where the number of inserted instructions and nodes is often similar to or smaller than their respective baselines. While they sometimes introduce slightly more changes (*e.g.*, jTrans+@2 requires 152.0 instructions vs. 83.5 for the baseline), such cases are exceptions rather than the norm. These results confirm that using explainers to identify salient instructions for perturbation leads to more effective and efficient attacks. Across both model architectures, we observe that attacks conducted under a smaller model pool size of 128 tend to achieve higher attack success rates with lower modification costs.

**Takeaway.** Our explainer-guided attacks can achieve higher attack success rates than the baseline in almost all testing scenarios, while requiring a comparable amount of perturbations.

### C. Attack Efficiency (RQ2)

We present the runtime overhead for selecting the important instructions and generating an adversarial sample in Table III. The numbers denote the average runtime overhead we observed on all function samples evaluated. The adversarial sample (AS) generation time refers to the overall runtime overhead for constructing the adversarial sample.

As shown, the explainers effectively located important instructions with higher efficiency on all target models (*e.g.*, 32.86 seconds for Gemini+ and 113.23 seconds for jTrans+). Notably, we observed a speedup of 12.71x for instruction selection on Gemini, while the optimization for SAFE was the least impactful at 30.99%. On the other hand, as instruction selection only accounts for a small portion of the total execution time, the speedup on AS generation is less significant. However, the explainer-guided attacks were still able to finish more quickly, reducing the overhead by up to 10.45%.

**TABLE III:** Time overhead for generating adversarial samples in our targeted attacks.

	Sequence-based Models				Graph-based Models			
	SAFE+	SAFE	jTrans+	jTrans	Gemini+	Gemini	GMN+	GMN
Instr Sel. (s)	101.95	133.54	113.23	286.09	32.86	450.54	433.31	598.13
Speedup (%)	30.99		152.66		1271.09		38.04	
AS Gen. (s)	2735.98	2738.48	3236.16	3369.31	3519.19	3887.07	4759.39	5050.06
Speedup (%)	0.09		4.11		10.45		6.11	

**Takeaway.** Our explainer-guided attacks are more efficient by focusing on the most vulnerable code for perturbation. Compared with the baselines, our attacks achieved a maximum speedup of 12.71x on instruction selection, while the overall overhead was reduced by up to 10.45%.

### D. Transferability (RQ3)

We further investigated the transferability of attacks, *i.e.*, whether the adversarial samples for one BCSD model can be generalized to target other models. We evaluated both our explainer-based attack and the baseline attack [6].

As illustrated in Table IV, when using our adversarial samples against jTrans and SAFE (target models), our attacks always exhibited superior transferability. On the graph-based models, however, the baseline approach could be more effective, *e.g.*, when using the sequence-based samples against graph-based models. This could be caused by the significant architectural disparity (*i.e.*, sequence vs. graph), which hinders the transfer of highly specialized perturbations on the salient instructions selected by explainers. We also observed that the adversarial samples generated for GMN tend to be less successful in attacking Gemini. One possibility is the stricter structural invariants during graph induction in Gemini created higher curvature decision boundaries that resist out of manifold perturbations. Nonetheless, the adversarial samples generated during our attacks in general achieved higher transferability in most tests.

**TABLE IV:** Transferability matrix for the targeted attack case, considering  $|P| = 128$  and  $K = 10$ . In the rows, we indicate the model for which the adversarial samples were created, and in the columns, the model on which the samples were tested. Each value represents the wASR (%).

Source	Variant	Target Model			
		jTrans	SAFE	Gemini	GMN
jTrans	Explainer	—	<b>59.50</b>	<b>39.75</b>	42.50
	Baseline	—	59.25	38.75	<b>47.00</b>
SAFE	Explainer	<b>40.75</b>	—	41.00	44.00
	Baseline	40.25	—	<b>42.75</b>	<b>49.00</b>
Gemini	Explainer	<b>31.00</b>	<b>32.75</b>	—	<b>96.00</b>
	Baseline	28.00	32.00	—	95.75
GMN	Explainer	<b>30.45</b>	<b>36.00</b>	65.50	—
	Baseline	27.75	33.00	<b>67.75</b>	—

**Takeaway.** The adversarial samples generated in our explainer-guided attacks tend to be effective when used against other BCSD models in the same architecture. The graph-based models like Gemini and GMN, which likely implement stricter decision boundaries, are more difficult to mislead using cross-model adversarial samples.

### E. Real-world Implications (RQ4)

To assess the real-world impact, we evaluated our approach in two real-world security tasks: vulnerability detection evasion and vulnerability classification misguidance.

1) *Vulnerability Detection Evasion*: BCSD models are widely used for vulnerability detection by identifying whether an input binary contains known vulnerabilities through similarity comparison. Vulnerability detection evasion aims to deceive these models into misclassifying a *vulnerable function* as a *targeted benign function*, leaving the vulnerability undetected. To evaluate the effectiveness of our approach, we target OpenSSL [32], a widely used SSL/TLS library. Specifically, we conduct experiments on OpenSSL versions 3.0.8 and 3.3.3. We included five recent vulnerabilities (CVE-2023-0215, CVE-2023-0216, CVE-2024-5535, CVE-2024-6119, and CVE-2024-9143), spanning various categories, including memory safety, cryptographic weaknesses, and protocol-level flaws.

To identify vulnerable functions for our attack, we first analyzed the patches for each vulnerability and selected the functions that were modified. We constructed a pool with 128 functions using the same four compilation configurations. In this experiment, we selected jTrans as the target model. We measure three metrics: (1) *initial similarity*, which is the similarity between the input samples and the target function before the attack; (2) *final similarity*, which captures the similarity after the attack; and (3) *average similarity*, defined as the average similarity of all samples in the pool to the target function. Note that the average similarity is the same before and after the attack, as the pool does not change.

As shown in Table V, our attack consistently increased the similarity scores across all the vulnerabilities. The average initial similarity was 0.869, which is lower than the average pool similarity of 0.876. This indicates that the input samples were less similar to the target functions and thus more challenging to manipulate. After applying our attack, the average final similarity rose to 0.917, surpassing both the initial and average pool similarities. This confirms that our method not only increases similarity but does so in a targeted and effective way. It makes the adversarial examples appear more functionally aligned with the target than even the average benign samples, significantly increasing the chance for evading detection.

**TABLE V:** Similarity scores before and after our attack in real-world OpenSSL vulnerabilities.

CVE	Init Similarity	Final Similarity	Average Similarity
CVE-2024-9143	0.898	0.929	0.874
CVE-2024-6119	0.867	0.908	0.882
CVE-2024-5535	0.843	0.920	0.869
CVE-2023-0215	0.854	0.885	0.873
CVE-2023-0216	0.884	0.942	0.881
Average	0.869	0.917	0.876

2) *Vulnerability Classification Misguidance*: The categorization of vulnerabilities plays a critical role in guiding the patching and mitigation process. For instance, by grouping vulnerabilities based on the categories, developers can prioritize fixes that address multiple attack surfaces simultaneously. Vulnerability classification misguidance aims to deceive the classification model into misjudging the severity or category of a vulnerability.

In our experiments, we selected CWEs from the CWE Most Dangerous Software Weaknesses list [38], with the goal of misleading the model into misclassifying code as a specific target CWE category. Vulnerable code samples containing these CWE types were obtained from the National Vulnerability Database [39] and the NIST Software Assurance Reference Dataset [40]. To simulate adversarial conditions, we constructed a pool of malicious functions with a size of 128, compiled under four consistent compilation settings. We selected CWE category pairs (original and target) with clear semantic and structural differences. For example, we include CWE-121 (Stack-based Buffer Overflow) vs. CWE-190 (Integer Overflow), and CWE-134 (Externally-Controlled Format String) vs. CWE-193 (Off-by-one Error). The fundamental differences make the misclassification more challenging and meaningful. Similarly, the experiments were conducted against jTrans, and we measured the initial, final, and average similarity as well.

As shown in Table VI, our method consistently increases the similarity scores across all targeted CWE pairs. On average, the initial similarity was 0.859, which is lower than the average pool similarity of 0.883. However, after applying adversarial perturbations, the final similarity rose to 0.953, exceeding both the average similarity of the CWE functions pool and the initial similarity. This demonstrates that our attack can effectively reduce the distinction between unrelated CWE categories. The results confirm the effectiveness of our attack strategy in vulnerability classification misguidance and highlight the risk in vulnerability management.

**TABLE VI:** Similarity scores for CWE classification misguidance.

CWE Pair	Init Similarity	Final Similarity	Average Similarity
CWE121 - CWE190	0.870	0.977	0.916
CWE121 - CWE134	0.867	0.908	0.888
CWE190 - CWE121	0.869	0.959	0.924
CWE190 - CWE193	0.869	0.959	0.924
CWE134 - CWE190	0.838	0.971	0.913
CWE134 - CWE121	0.843	0.945	0.910
Average	0.859	0.953	0.912

**Takeaway.** Our attacks are not only effective in manipulating model outputs but also practical in compromising real-world applications of BCSD models by evading vulnerability detection and misleading vulnerability categorization.

## VI. DISCUSSION

We now discuss the limitations and future work.

**Optimizations.** In our current implementation, the selected instructions are manipulated similarly to existing studies [5, 6]. We adopt such a design to enable fair comparison with state-of-the-art attacks. Nonetheless, our attacks can be easily extended to incorporate more advanced code transformations. For example, program obfuscation techniques can effectively disrupt the static disassembly process. Other viable methods include edge hiding [41] and code cloning [41], which aim to alternate or obscure certain control flow paths. Junk code insertion [42] involves adding instructions like `if (false) { jmp }`, which causes the bytes of subsequent normal instructions to be

misinterpreted as operands of the current `jmp` instruction, thereby disrupting the structure of the current instruction and leading to anomalies in static analysis. We leave it for future work to explore more effective instruction manipulation approaches to further boost the attack performance.

**Extensibility.** In this work, we focused our evaluation on four representative sequence and graph-based models. Nevertheless, the explainer-guided attack principle can also be readily applied to other categories of models, including neural network based models like BinFinder [43] and Zeek [44], *etc.* Additionally, we believe explainers can be applied to advance the attack against other program analysis models. Besides, the explainer-guided attack method can also be applied to untargeted tasks. For example, our explainer can similarly identify critical instructions and perturb them, aiming to minimize the similarity score between the perturbed (source) function and its variants. We plan to investigate these in the future.

**Feature Extraction.** Our explainer leverages the features extracted by BCSD models to guide adversarial perturbations. This assumption is practical and often holds in real-world and research settings. Many BCSD models rely on explicit feature extraction pipelines, such as disassembly, CFG construction, and instruction-level analysis, implemented using standard binary analysis frameworks like IDA Pro [45], Binary Ninja [46], and angr [29]. In practice, these intermediate features are frequently retained or exported by the toolchains for integration with other modules, *e.g.*, for visualization or further analysis, making them accessible in most attack scenarios. Even when features are not directly available, we can often approximate them by replicating the model’s preprocessing pipeline or intercepting data at various stages, enabling our explainer to operate effectively in such cases as well.

**Defense.** Existing studies have proposed several defense against the relevant adversarial attacks. Adversarial training approaches like FuncFooler [41] enhance resilience by injecting structurally perturbed examples during training. However, such methods are tightly coupled to the observed perturbation patterns, and may not generalize to novel attacks like ours. Other techniques, such as GWAD [47], attempt to detect adversarial behaviors by analyzing the query dynamics. While effective in certain scenarios, these methods are often limited by their reliance on predefined detection heuristics, making them less robust against sophisticated attacks that bypass such patterns. To effectively defend against our explainer-guided attacks, one possible way is to reduce the precision of explanation, *e.g.*, using gradient obfuscation, attention randomization, and saliency regularization [48]. Input-based defense like (De)Randomized Smoothing [49] may also dilute the impact of localized perturbations, increasing the robustness against crafted perturbations. We believe more efforts need to be invested to explore the possible defense.

## VII. RELATED WORK

**Adversarial Attacks against Source Code Analysis.** Various advanced models for source code analysis have been proposed,

effectively improving the performance in tasks like clone detection, method name prediction, *etc.* Consequently, extensive efforts have been invested to measure the robustness of such models via adversarial attacks. Yefet *et al.* [50] designed a white-box adversarial attack against code models for Java and C# analysis. They assume adversaries have the knowledge about the gradients of targeted models. Zhang *et al.* [51] implemented fifteen semantic-preserving code transformation for constructing adversarial samples against machine learning-based clone detection models. Quiring *et al.* [52] focused on semantic equivalent coding style transformation, dramatically downgrades the accuracy of authorship attribution models. Other affected application scenarios include plagiarism detection, where genetic code transformation techniques were implemented to undermine the detection models [53]. The above research appears orthogonal to this work, which focuses particularly on binary code similarity detection models.

**Attacks against Binary Analysis Models.** Binary code analysis models have also been a popular target for various attacks. Capozzi *et al.* [6] adopted a brute-force strategy for selecting instructions to perturb. Specifically, they exhaustively traversed and removed each assembly instruction, and measured the importance of instructions according to the similarity score changes. They also selected the code transformations for each candidate instruction based on the similarity score changes, implementing both black-box and white-box attacks against three state-of-the-art BCSD models [5]. Song *et al.* [17] focused particularly on malware classifiers and aimed to generate adversarial samples to evade malware detection and analysis. Jia *et al.* [7] selected candidate instructions to mutate based on the heuristic rule that instructions in basic blocks that locate on all paths from the entry to exit must be important. Kreuk *et al.* [19] created non-executable code sections for appending the adversarial bytes, thereby preserving the original functionalities while evading the malware detection. Similar strategies were also applied in [21]. In contrast, Lucas *et al.* [20] proposed to mutate functional instructions, and iteratively optimize the attack effects by measuring the misclassification probabilities. Besides adversarial attacks, backdoor vulnerabilities in binary code analysis models have also been investigated [54].

In this work, we explored another viable direction to optimize the attack effectiveness, *i.e.*, using explainers to pinpoint important instructions for manipulation, which can be integrated with existing attacks to further boost the performance.

**Explainer-guided Program Analysis and Attacks.** Explainers have been applied in other program analysis and attacks. He *et al.* [55, 56] generated explanations for Android malware detection models to enhance the usability. Arp *et al.* [57] augmented Android malware detectors by calculating the importance score of each feature and constructing explanations for the detection results accordingly. In addition to generating explainable results, explainers are also used to facilitate other attacks, *e.g.*, backdoor attacks [8]. In this work, we demonstrated that explainers can also effectively optimize the adversarial attacks against binary similarity detection models.

## VIII. CONCLUSION

In this work, we introduced an optimization for targeted adversarial attacks against BCSD models. By leveraging off-the-shelf explainers to pinpoint the salient instructions for perturbation, we are able to generate effective adversarial function samples in a computationally efficient way. The evaluation on binary functions from real-world projects proves that explainers provide actionable and granular guidance for adversarial manipulation, significantly improving attack efficacy. The discoveries highlight the lack of robustness in existing BCSD models, demonstrating the possibility to hinder vulnerability detection and classification in practice. We further emphasize the necessity of further research to enhance the robustness of BCSD models against adversarial attacks, particularly through the development of defense mechanisms that address the exploitability of explanation-driven weaknesses.

## REFERENCES

- [1] S. Shimmi, A. Rahman, M. Gadde, H. Okhravi, and M. Rahimi, “{VuSim}: Leveraging similarity of {Multi-Dimensional} neighbor embeddings for vulnerability detection,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1777–1794.
- [2] Z. Chen, J. Liu, Y. Hu, L. Wu, Y. Zhou, Y. He, X. Liao, K. Wang, J. Li, and Z. Qin, “Deuedroid: Detecting underground economy apps based on utg similarity,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 223–235.
- [3] L. Jiang, J. An, H. Huang, Q. Tang, S. Nie, S. Wu, and Y. Zhang, “Binaryai: binary software composition analysis via intelligent binary source code matching,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
- [4] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, “Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection,” *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1157–1177, 2017.
- [5] G. Capozzi, D. C. D’Elia, G. A. Di Luna, and L. Querzoni, “Adversarial attacks against binary similarity systems,” *IEEE Access*, 2024.
- [6] G. Capozzi, T. Tang, J. Wan, Z. Yang, D. C. D’Elia, G. A. Di Luna, L. Cavallaro, and L. Querzoni, “On the lack of robustness of binary function similarity systems,” *arXiv preprint arXiv:2412.04163*, 2024.
- [7] L. Jia, B. Tang, C. Wu, Z. Wang, Z. Jiang, Y. Lai, Y. Kang, N. Liu, and J. Zhang, “Funcfooler: A practical black-box attack against learning-based binary code similarity detection methods,” *arXiv preprint arXiv:2208.14191*, 2022.
- [8] G. Severi, J. Meyer, S. Coull, and A. Oprea, “{Explanation-Guided} backdoor poisoning attacks against malware classifiers,” in *30th USENIX security symposium (USENIX security 21)*, 2021, pp. 1487–1504.
- [9] K. Han, J. H. Lim, and E. G. Im, “Malware analysis method using visualization of binary files,” in *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, 2013, pp. 317–321.
- [10] G. Wagoner, R. State, and A. Dulaunoy, “Malware behaviour analysis,” *Journal in computer virology*, vol. 4, pp. 279–287, 2008.
- [11] S. H. Ding, B. C. Fung, and P. Charland, “Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization,” in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489.
- [12] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, “Function representations for binary similarity,” *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2259–2273, 2021.
- [13] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, “Learning approximate execution semantics from traces for binary function similarity,” *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2776–2790, 2022.
- [14] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, “Neural network-based graph embedding for cross-platform binary code similarity detection,” in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 363–376.
- [15] S. Yang, C. Dong, Y. Xiao, Y. Cheng, Z. Shi, Z. Li, and L. Sun, “Asteria-pro: enhancing deep learning-based binary code similarity detection by incorporating domain knowledge,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–40, 2023.
- [16] Y. Li, C. Gu, T. Dullien, O. Vinyals, and P. Kohli, “Graph matching networks for learning the similarity of graph structured objects,” in *International conference on machine learning*. PMLR, 2019, pp. 3835–3845.
- [17] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin, “Mab-malware: A reinforcement learning framework for blackbox generation of adversarial malware,” in *Proceedings of the 2022 ACM on Asia conference on computer and communications security*, 2022, pp. 990–1003.
- [18] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, “Intriguing properties of adversarial ml attacks in the problem space,” in *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020, pp. 1332–1349.
- [19] F. Kreuk, A. Barak, S. Aviv-Reuven, M. Baruch, B. Pinkas, and J. Keshet, “Adversarial examples on discrete sequences for beating whole-binary malware detection,” *arXiv preprint arXiv:1802.04528*, pp. 490–510, 2018.
- [20] K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre, “Malware makeover: Breaking ml-based static analysis by modifying executable bytes,” in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 744–758.
- [21] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, “Adversarial malware binaries: Evading deep learning for malware detection in executables,” in *2018 26th European signal processing conference (EUSIPCO)*. IEEE, 2018, pp. 533–537.
- [22] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, and C. Zhang, “Jtrans: Jump-aware transformer for binary code similarity detection,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 1–13.
- [23] M. T. Ribeiro, S. Singh, and C. Guestrin, “‘‘ why should i trust you?’’ explaining the predictions of any classifier,” in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, 2016, pp. 1135–1144.
- [24] S. M. Lundberg and S.-I. Lee, “A unified approach to interpreting model predictions,” in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 4765–4774. [Online]. Available: <http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf>
- [25] Z. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec, “Gnnexplainer: Generating explanations for graph neural networks,” *Advances in neural information processing systems*, vol. 32, 2019.
- [26] B. Biggio and F. Roli, “Wild patterns: Ten years after the rise of adversarial machine learning,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2154–2156.
- [27] R. Lapid and M. Sipper, “I see dead people: Gray-box adversarial attack on image-to-text models,” in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2023, pp. 277–289.
- [28] “Radare2,” Apr. 2025, <https://github.com/radareorg/radare2>.
- [29] “Angr,” Apr. 2025, <https://github.com/angr/angr>.
- [30] Binutils Project, “Binutils,” <https://ftp.gnu.org/gnu/binutils/>, accessed: 2025-05-30.
- [31] Curl Project, “Curl,” <https://curl.se/>, accessed: 2025-05-30.
- [32] OpenSSL Project, “Openssl: The open source toolkit for ssl/tls,” <https://www.openssl.org/>, accessed: 2025-05-30.
- [33] Sqlite Project, “Sqlite,” <https://www.sqlite.org/>, accessed: 2025-05-30.
- [34] Gsl Project, “Gsl,” <https://www.gnu.org/software/gsl/>, accessed: 2025-05-30.
- [35] Libconfig Project, “Libconfig,” <https://github.com/hyperrealm/libconfig>, accessed: 2025-05-30.
- [36] Ffmpeg Project, “Ffmpeg,” <https://ffmpeg.org/>, accessed: 2025-05-30.
- [37] PostgreSQL Project, “Postgresql,” <https://www.postgresql.org/>, accessed: 2025-05-30.
- [38] CVE, “Cwe top 25 most dangerous software weaknesses,” 2025, <https://cwe.mitre.org/top25/>.
- [39] “National vulnerability database (nvd),” National Institute of Standards and Technology, 2025, <https://nvd.nist.gov/>.
- [40] “Software assurance reference dataset (sard),” National Institute of Standards and Technology, 2025, <https://samate.nist.gov/SARD/>.
- [41] L. Jia, C. Wu, B. Tang, P. Zhang, Z. Jiang, Y. Yang, N. Liu, J. Zhang, and D. Z. Wang, “Enhancing learning-based binary code similarity detection model through adversarial training with multiple function variants,” in *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024, pp. 11 508–11 518.

- [42] junk code, “Foudation of ctf reverse engineering,” 2021, <https://blog.csdn.net/u011642058/article/details/114757503>.
- [43] A. Qasem, M. Debbabi, B. Lebel, and M. Kassouf, “Binary function clone search in the presence of code obfuscation and optimization over multi-cpu architectures,” in *Proceedings of the 2023 acm asia conference on computer and communications security*, 2023, pp. 443–456.
- [44] N. Shalev and N. Partush, “Binary similarity detection using machine learning,” in *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, 2018, pp. 42–47.
- [45] Hex-Rays, “Ida pro,” <https://www.hex-rays.com/products/ida/>, accessed: 2025-05-30.
- [46] Vector 35 Inc., “Binary ninja,” <https://binary.ninja/>, accessed: 2025-05-30.
- [47] J. Park, N. McLaughlin, and I. Alouani, “Mind the gap: Detecting black-box adversarial attacks in the making through query update analysis,” *arXiv preprint arXiv:2503.02986v3*, 2025.
- [48] K. Yue, R. Jin, C.-W. Wong, D. Baron, and H. Dai, “Gradient obfuscation gives a false sense of security in federated learning,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6381–6398.
- [49] A. Salem, S. Banescu, and S. Tople, “A robust defense against adversarial attacks on deep learning-based malware detectors via (de)randomized smoothing,” *arXiv preprint arXiv:2402.15267v2*, 2024.
- [50] N. Yefet, U. Alon, and E. Yahav, “Adversarial examples for models of code,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.
- [51] W. Zhang, S. Guo, H. Zhang, Y. Sui, Y. Xue, and Y. Xu, “Challenging machine learning-based clone detectors via semantic-preserving code transformations,” *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 3052–3070, 2023.
- [52] E. Quiring, A. Maier, and K. Rieck, “Misleading authorship attribution of source code using adversarial learning,” in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 479–496.
- [53] B. Devore-McDonald and E. D. Berger, “Mossad: Defeating software plagiarism detection,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–28, 2020.
- [54] Z. Zhang, G. Tao, G. Shen, S. An, Q. Xu, Y. Liu, Y. Ye, Y. Wu, and X. Zhang, “{PELICAN}: Exploiting backdoors of naturally trained deep learning models in binary code analysis,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2365–2382.
- [55] Y. He, J. Lou, Z. Qin, and K. Ren, “Finer: Enhancing state-of-the-art classifiers with feature attribution to facilitate security analysis,” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 416–430.
- [56] Y. He, Y. Liu, L. Wu, Z. Yang, K. Ren, and Z. Qin, “MsDroid: Identifying malicious snippets for android malware detection,” *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 3, pp. 2025–2039, 2022.
- [57] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, “Drebin: Effective and explainable detection of android malware in your pocket.” in *Ndss*, vol. 14, no. 1, 2014, pp. 23–26.