Unveiling the Fragility of Binary Code Similarity Detection via Targeted Attacks with Model Explanations

MINGJIE CHEN*, Zhejiang University, China
TIANCHENG ZHU*†, Huazhong University of Science and Technology, China
MINGXUE ZHANG‡, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China
YILING HE, University College London, United Kingdom
MINGHAO LIN, Independent Researcher, USA
PENGHUI LI, Columbia University, USA
KUI REN, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

Binary code similarity detection (BCSD) serves as a fundamental technique for various software engineering tasks, *e.g.*, vulnerability detection and classification. Attacks against such BCSD 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, which often leads to inefficient search and high computational cost. Moreover, when processing programs with high complexities, such attacks tend to be time-consuming.

In this work, we unveil the fragility of BCSD models through a novel attack framework guided by model explanations. In particular, we focus on *targeted attacks* where the attack goal is to mislead the model's predictions to a specific target. Our attack leverages explainers to pinpoint critical code snippet for perturbations, reducing the exploration overhead. The evaluation results demonstrate that the proposed attacks effectively improve the attack efficiency, while maintaining comparable or higher success rates. Importantly, the speedup for perturbation target selection achieves up to 63.66×, demonstrating the practical value of explanation-guided localization. Our real-world case studies on vulnerability detection and classification further demonstrate the security implications of our attacks, highlighting fundamental robustness limitations in current BCSD models, and the urgent need for more robust designs.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Adversarial Attacks; Binary Code Similarity Detection

*The first two authors contributed equally to this work.
†This research was conducted during the author's internship at Zhejiang University.
‡Corresponding author.

Authors' Contact Information: Mingjie Chen, Zhejiang University, Hangzhou, China, chenmingjie@zju.edu.cn; Tiancheng Zhu, Huazhong University of Science and Technology, Wuhan, China, u202211935@hust.edu.cn; Mingxue Zhang, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, China, mxzhang97@zju.edu.cn; Yiling He, University College London, London, United Kingdom, heyilinge0@gmail.com; Minghao Lin, Independent Researcher, Los Angeles, USA, yenkoclike@gmail.com; Penghui Li, Columbia University, New York, USA, pl2689@columbia.edu; Kui Ren, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, kuiren@zju.edu.cn.

ACM 2994-970X/2026/7-ARTFSE181
https://doi.org/10.1145/3808188

ACM Reference Format:
Mingjie Chen, Tiancheng Zhu, Mingxue Zhang, Yiling He, Minghao Lin, Penghui Li, and Kui Ren. 2026. Unveiling the Fragility of Binary Code Similarity Detection via Targeted Attacks with Model Explanations. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE181 (July 2026), 24 pages. https://doi.org/10.1145/3808188

1 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 [12, 27, 39, 54]. 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. 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 the most [8, 9]. 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 [25], may offer some improved efficiency but often yield suboptimal results and lack precision. This is primarily due to their reliance on pre-determined rules and insufficient consideration of complex inter-instruction relationships.

At the meantime, 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 faced more challenges. Although untargeted attacks can often succeed by pushing an input across nearby decision boundaries, the directional requirement to steer the perturbed sample to a specific target significantly increases the search complexity in the perturbation space. Nonetheless, 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 [25, 30]. Therefore, this work focuses on improving the efficiency of targeted adversarial attacks against BCSD models.

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 [52], where models are maliciously trained on poisoned training data to behave incorrectly on inputs containing specific triggers. While explanation-guided attacks have also been explored in other domains such as CV and NLP [28, 44, 50], their application to BCSD models remains under-explored due to the unique program semantic constraints. This raises a fundamental research question: can model explainability be systematically leveraged to make targeted BCSD attacks practical in large perturbation spaces, without sacrificing effectiveness?

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 regions within a binary sample, we systematically identify the most vulnerable code for perturbations,

thereby enabling efficient exploration in the perturbation space while maintaining competitive effectiveness. However, precisely explaining the BCSD model predictions and generating the adversarial samples accordingly, is not trivial. First, as BCSD models aim to detect semantically equivalent functions, adversarial attacks must perturb the input functions while preserving their semantics. The explainers only estimate the importance of input features (*e.g.*, statistical attribute counts, normalized IR tokens, graph embeddings, *etc.*) extracted by the models, while semantic-preserving perturbations need to be applied at concrete positions (*i.e.*, *instructions*). We need to precisely map the features to the specific instructions, which is difficult (*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 representative BCSD models in different architectures. Specifically, to address *C1*, we implemented an instruction–feature mapping strategy to calculate the instruction importance based on the weights of features at different granularities. 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. Further, to improve the stability of directional alignment in the targeted settings, we design a target-aligned semantic-preserving transformation.

We have evaluated our attacks on the binary functions from ten real-world projects under four different compilation settings. The results demonstrate that our method maintains competitive effectiveness while substantially reducing attack cost, 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 63.66×. 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 heterogeneous BCSD models by mapping the feature space to the input instruction space, effectively reducing the computational overhead of perturbation selection.
- **Target-aligned Perturbation.** We identify the directional requirement in targeted BCSD attacks and design Target-Aware Dead Branch Addition, a target-aligned semantic-preserving transformation that steers the adversarial sample towards the target functions.
- **Real-world Security Implications.** We showed how our attacks could successfully mislead BCSD models in vulnerability detection and classification.

2 Background

2.1 Adversarial Attacks against BCSD Models

Binary code similarity detection (BCSD) techniques 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 [21, 27, 39, 60]. Adversarial attacks aim at misleading models to generate incorrect predictions. Multiple adversarial attacks have been proposed to attack BCSD models [8, 9, 25, 30, 31, 36, 46, 55]. 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 a brute-force traversal algorithm or design customized heuristic rules. For example, Capozzi *et al.* [9] 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 [25], 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.

2.2 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 [51, 67]. 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 or 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 and neural networks.

3 Analysis of BCSD Models

To help better design the model explainer to guide our attacks, we first investigate a set of representative BCSD models. Crucially, to ensure our evaluation reflects the most current landscape of binary similarity detection, we systematically selected models spanning from foundational approaches (*e.g.*, Gemini [65], SAFE [40]) and the latest state-of-the-art advancements from recent top-tier venues, such as CLAP [61], and VexIR2Vec [59]. As categorized below, this selection comprehensively covers diverse model architectures (*e.g.*, MLP, GNN, RNN-with-attention, Transformer), and feature spaces (*e.g.*, hand-engineered attributes, ACFGs, instruction sequences, dynamic micro-traces, pre-trained instruction embeddings) [8, 9, 25].

3.1 Multi-Layer Perceptron (MLP) Based Models

MLP-based models employ fully connected layers to capture non-linear relationships in feature vectors. These models typically work with pre-extracted or manually engineered features.

BinFinder [48] represents each function using a set of manually engineered features, including the numbers of callers and callees, lists of libc calls, lists of constants, and sequences of VEX-IR instructions, lifted from assembly via the angr framework, that ensure architecture independence.

These features are tokenized and one-hot encoded, after which a Siamese network consisting of two identical three-layer multilayer perceptrons (MLPs) embeds the feature vectors and computes similarity scores using cosine distance.

ZEEK [53] computes similarity scores using the proc2vec technique, a vector representation method that decomposes each function into basic blocks and further into code strands (semantically coherent subsets of instructions). Each strand is normalized, hashed with a 10-bit MD5, and mapped to an index in a fixed-length count vector corresponding to the hash space. Finally, a fully connected neural network with two hidden layers takes pairs of these vectors as input and classifies whether the underlying functions are similar or not.

3.2 Recurrent Neural Network (RNN) Based Models

RNN-based models exploit sequential dependencies by processing input features in an order-sensitive manner. They are useful for modeling the sequential nature of instructions or basic blocks within binary programs.

SAFE [40] 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.

3.3 Transformer-Based Models

Transformer-based models rely on self-attention mechanisms to capture both local and long-range dependencies. Unlike RNNs that process sequences linearly, Transformers can attend to any part of the input simultaneously to model complex inter-instruction relationships and global program semantics.

jTrans [62] 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.

Trex [45] learns execution semantics from micro-traces—dynamic traces with instruction tokens and values. It encodes code, values, positions, and architecture labels, and feeds them into a hierarchical Transformer. Pretraining uses masked language modeling on tokens and values; fine-tuning averages embeddings and applies an MLP for function similarity classification.

PalmTree [33] is a pre-trained assembly language model that produces general-purpose instruction embeddings. It tokenizes instructions into fine-grained components (opcodes, registers, constants), normalizes large values and strings, and employs a bidirectional Transformer trained with three self-supervised tasks: masked language modeling, context-window prediction, and def-use prediction. The mean pooled hidden states from the Transformer provide high-quality instruction embeddings for downstream similarity models.

VexIR2Vec [59] addresses architecture neutrality by lifting binaries to VEX-IR [41]. It decomposes functions into "peepholes" (*i.e.*, sequences of basic blocks extracted via random walks) and applies compiler-inspired normalizations. It utilizes a pre-trained vocabulary to map IR entities (opcodes, types, and arguments) into vectors. Finally, a Siamese network with an attention mechanism aggregates these entity embeddings to produce a flow-insensitive function representation.

CLAP [61] utilizes the RoBERTa architecture (*i.e.*, a Transformer variant) to align assembly code with natural language explanations. Instead of aggressive normalization, it employs a WordPiece tokenizer to preserve fine-grained semantics, including raw strings, variable names, and constant

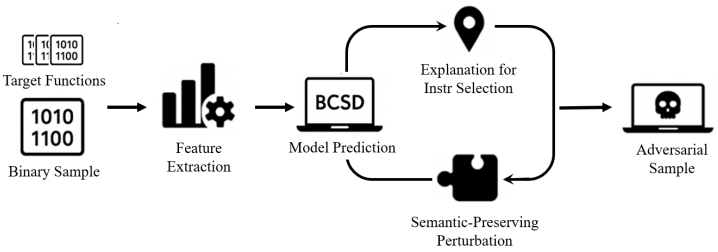

Figure 1. The workflow of explainer-guided adversarial attacks.

literals. The final representation is a summation of token embeddings and positional embeddings, while incorporating instruction embeddings to link jump targets.

3.4 Graph Neural Network (GNN) Based Models

GNN-based models are designed to leverage graph-structured data, propagating information across nodes and edges to learn program semantics. They are particularly effective for capturing structural dependencies in control-flow and data-flow graphs.

Gemini [65] generates an attributed control flow graph (ACFG) to encode function features. Each basic block is represented as an eight-dimensional attribute vector, preserving the control flow structure. A Siamese neural network is then trained to produce embedding vectors for ACFGs, and function similarity is measured by the cosine similarity between their embeddings. Key block-level features include counts of constant, string, transfer, call, and arithmetic instructions. Similarities between two functions can be measured using the cosine similarity between the embedding vectors of the ACFGs.

GMN [34] is a versatile graph matching model that imposes no constraints on graph features or structures, allowing flexible feature extraction strategies. Through an attention mechanism, it enables information exchange between graphs, integrating both intra-graph structural dependencies and inter-graph correspondences. This dual-level representation improves embedding quality by capturing structural and semantic relations within and across graphs. Function similarity is then assessed using Euclidean or cosine distances between the resulting embeddings.

In summary, existing BCSD models primarily adopt two types of feature representations: sequential/vector features (*e.g.*, instruction tokens in MLP, RNN, and Transformer models) and structural graphs (*e.g.*, node-level attributes in GNNs). We structure our explanation generation strategies around this fundamental distinction in §4.2.

4 Design

In this work, we propose to utilize explainers for guiding adversarial attacks against BCSD models. 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 §4.1. We then demonstrate how we generate explanations for different model predictions in §4.2, and how they guide the adversarial sample generation in §4.3. Finally, we provide the implementation details in §4.4.

4.1 Threat Model

In this work, we aim to conduct targeted attacks against representative 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. Such information is often available in documentation or prior publications in open research settings, and the attacker does not access model parameters, gradients, or training data. In the meantime, 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 [6, 15, 32, 63]. Additionally, since BCSD models are designed to identify semantically equivalent functions across compilation settings, altering the underlying functionality would invalidate the attack objective. The goal of the attacker is to manipulate similarity judgments for the same program behavior, rather than to generate a different program. Therefore, all perturbations are designed to be semantic-preserving, as in existing attacks against BCSD models [8, 9].

4.2 Explanation Generation

We now describe how we generate explanations for the predictions of target models. As summarized in §3, although BCSD models differ significantly in terms of architectures and feature space, *etc.*, we can generally organize them into two categories. We accordingly design our explainers for targeted adversarial attacks. As depicted in Figure 2, our method takes a binary sample as input and generates the weights of instructions for inferring their importance.

4.2.1 ***Perturbation-Based Explainer.*** For models operating on sequential features, one common approach is to leverage perturbation-based explanation [20, 37, 51]. The key idea is to approximate a model's local decision boundary by systematically modifying (perturbing) the input and observing how the model predictions change on the perturbed inputs. Perturbations that cause significant prediction shifts reveal the proximity and orientation of decision boundaries in the feature space.

However, implementing a perturbation-based explainer for BCSD models is non-trivial, particularly for constructing perturbed instances of the given input. Previous explainers [20, 37, 51] assume prior knowledge of the original feature distribution, and leverage that for perturbation. For example, LIME [51] assumes that the distribution of each feature can be reasonably estimated from the training data. Perturbation is then performed by sampling new feature values according to these distributions, *e.g.*, for continuous features. This typically involves sampling from a normal distribution with the empirical mean and variance. By contrast, BCSD tasks involve features from diverse real-world binary programs with different functionalities. The representations may vary substantially in length and structure. Consequently, it is difficult to obtain a consistent or meaningful distribution for these features.

To address this challenge, we propose a new, effective perturbation mechanism that does not rely on the original feature distribution. The global decision boundary of BCSD models is highly non-linear and complex, making it impossible to precisely calculate how modification at a specific instruction site affects the final similarity score. Therefore, we approximate this relationship locally. Given an original input function x and a trained classifier $f(\cdot)$, we first generate a local perturbation corpus $\mathcal{D}_x = \{(z_i, f(z_i))\}_{i=1}^{N}$, where each perturbed sample z_i is obtained from x by randomly masking or modifying a subset of features. Formally, let $m_i \in \{0,1\}^M$ be a random binary mask.

$$z_i = x \odot m_i, \tag{1}$$

where $\odot$ denotes element-wise multiplication. When $m_{i,j} = 0$, the j-th feature of x is masked by replacing it with the corresponding feature of a no-operation instruction (NOP); when $m_{i,j} = 1$, the feature is retained. By sampling multiple m_i, we obtain diverse perturbations while keeping z_i close

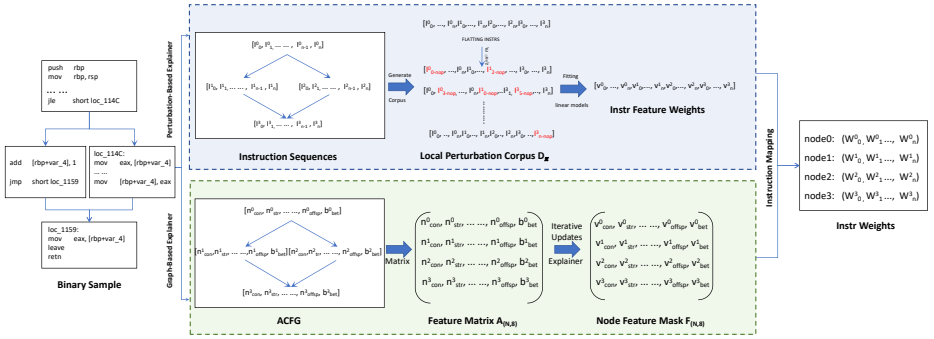

Figure 2. Workflow of explanation generation.

to x. The target classifier assigns labels $y_i = f(z_i)$, yielding the dataset $\mathcal{D}_x$. It is crucial to note that feature-level perturbations during the explanation phase are numerical modifications used solely to probe the model decision boundary. They do not yield valid or executable binary programs. This corpus captures how the prediction changes when subsets of features are suppressed, providing the basis for local explanation.

We build our explainer on top of LEMNA [20], which is model-agnostic with high transferability and particularly suitable for data with strong sequential dependencies. Unlike methods that rely on a single linear approximation and assume feature independence (*e.g.*, [51]), LEMNA employs a mixture of linear models with fused lasso regularization to capture both nonlinear decision boundaries and correlations among adjacent features, yielding interpretable feature rankings.

We integrate our new perturbation mechanism into LEMNA and then leverage it to fit a surrogate model on $\mathcal{D}_x$ to approximate the local decision boundary. Specifically, our method employs a mixture of linear regressions:

$$y_i \sim \sum_{k=1}^{K} \pi_k \, \mathcal{N}(\beta_k^\top z_i, \sigma_k^2), \tag{2}$$

where K is the number of mixture components, π_k are mixture weights, and β_k are regression coefficients. Once trained, the component with the highest responsibility for the original x is selected, and its coefficient vector highlights the most influential features or groups, which are reported as the explanation.

Note that models may derive multiple features from each instruction. As adversarial perturbations must ultimately be applied at concrete program locations, we need to locate the salient *instructions* based on the explanation results (*i.e.*, *feature* importance) to determine the perturbation positions. We next describe how the salient instructions are selected.

Explaining jTrans. As stated above, we use LEMNA 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 summing the absolute values of such weights. We then select the most critical instructions based on the aggregated weights.

Explaining Trex. For interpretability, we construct a mapping that links feature indices back to their originating instructions and all features—static, positional, and byte-level—associated with the same instruction are grouped together. In explanation, feature weights acquired from LEMNA are then aggregated across all indices belonging to an instruction, and the most important instructions are identified and projected back to their original addresses.

Explaining PalmTree. Similar to Trex and jTrans, PalmTree also requires a mapping from feature indices back to instructions. The difference lies in how the instruction space is constructed: CFG is linearized into a sequence of normalized fine-grained instruction tokens, in which immediates and RIP-relative operands are canonicalized. The sequence is padded and wrapped with start/end markers, ultimately producing a stable token grid where each instruction occupies a contiguous slot. Importance scores obtained from LEMNA are aggregated across tokens within the same slot, producing one weight per instruction.
Explaining SAFE. SAFE extracts one feature vector per preprocessed instruction (§3.3). Therefore, we can feasibly select the important instructions based on the weights derived from LEMNA.
Explaining BinFinder. BinFinder represents each function with hand-crafted features—including the numbers of callers and callees, libc calls, constants, and VEX-IR instructions. The explanation process perturbs only the VEX-IR segment of the feature vector, as these features directly correspond to normalized instructions. The selected high-weight features are projected back onto instructions by consulting the stored token-to-address map: each important feature index is matched to one or more original instruction addresses, thus recovering the most critical code statements.
Explaining ZEEK. In ZEEK's processing workflow, basic blocks are lifted to VEX-IR, sliced into strands, hashed into 10-bit bins, and aggregated as feature counts. To enable interpretability, we maintain a bidirectional map linking hashes to original instructions by recording, for each strand, both its hash and corresponding instruction addresses. To identify influential features, non-zero hashes are randomly masked. The most significant hashes selected by LEMNA are then projected back to instructions through the reverse mapping.
Explaining VexIR2Vec. VexIR2Vec aggregates embeddings from basic block traces, preventing a direct mapping from feature dimensions back to specific instructions. Furthermore, perturbing the dense embedding vector directly is semantically unreasonable, as dimensions do not correspond to specific code units. Therefore, we adopt a hierarchical strategy. First, we generate the local perturbation corpus by randomly masking basic blocks in the input CFG. The explainer then identifies the critical basic blocks based on their contribution to the model's decision. Second, to pinpoint salient instructions, we limit our search scope to these critical blocks. We iteratively remove individual instructions within them and measure the absolute change in the similarity score. This coarse-to-fine approach effectively circumvents the efficiency bottleneck of a global brute-force search while bridging the gap between block-level features and instruction-level importance.
Explaining CLAP. CLAP processes assembly code as fine-grained subword tokens, creating a granularity mismatch for interpretation. To bridge this, we treat the instruction—not the token—as the atomic unit of perturbation during the explanation process. We utilize the model's token indices to identify instruction boundaries and randomly mask all constituent tokens of an instruction simultaneously, effectively toggling its presence in the input. Consequently, the importance scores derived for these instruction-level groups rather than individual tokens are projected back to the original address by consulting a mapping constructed during preprocessing.

4.2.2 ***Graph-Structure-Based Explainer.*** For GNN models that operate on graph-based features such as CFGs, the decision-making process heavily relies on the topological structure and node interconnectivity of the input graph. Perturbation-based explainers like LEMNA that analyze feature vectors are not suited for this task as they would fail to capture these critical structural dependencies [20, 51]. On the other hand, graph-structure-based explainers often capture the complex relationships among components in the graph representations and interpret GNNs [38, 68]. For example, GNNExplainer [67] takes the adjacency matrix and feature matrix as input, and iteratively updates two types of masks: an edge mask that highlights influential control-flow relations, and a node feature mask that identifies the most relevant attributes in each basic

block. Its objective is to maximize the mutual information between the predictions of the original graph and the masked subgraph, thereby ensuring that the explanation subgraph preserves the original decision boundary. In this work, we adopt GNNExplainer as a representative structure-aware explainer for GNN-based models [57, 68]. It models both edge importance and node feature attribution, aligning with our need to localize salient control-flow regions and basic-block attributes.

Graph-based explainers do not require a perturbed dataset for learning the prediction boundaries, because they generate explanations by directly identifying subgraphs and node features within the original graph. However, graph-based models extract features on the basic-block level, whereas most semantic-preserving perturbations (*e.g.*, insertion, reordering, substitution) are defined at instruction sites and need to be conducted at specific positions within a block. Although certain structural perturbations (*e.g.*, target-aware DBA in §4.3.1) operate at the basic-block level, they remain limited in scope and still require instruction-level insertion sites to maximize effectiveness. Therefore, the explanation results (*i.e.*, importance of basic blocks) cannot be directly used for targeted perturbation. To solve this, we designed an *instruction-type vocabulary mapping* mechanism to calculate the importance of instructions (*i.e.*, perturbation positions), as illustrated below.

Explaining Gemini and GMN. For an input function, Gemini first converts it into an Attributed Control Flow Graph (ACFG), where each basic block is represented as an eight-tuple feature vector, such as number of constant-type instructions n_{con} (*e.g.*, `add ax, 1`), number of string-type instructions n_{str} (*e.g.*, `mov ax, [string_address]`), number of transfer-type instructions n_{trans}, number of function calls n_{call}, and number of arithmetic operations n_{ari}. The ACFG can thus be encoded as a feature matrix $A_{(N,8)}$, where N represents the total number of basic blocks in the ACFG. Next, our explainer takes $A_{(N,8)}$ as input and iteratively updates the mask matrices to obtain the final node feature mask $F_{(N,8)}$ as the explanation. Here, $F_{(N,8)}$ indicates the importance of each item in the feature tuples, *e.g.*, n_{cons} 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 (§5). 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 method as stated above.

4.3 Adversarial Sample Generation

We generate an adversarial sample by applying a set of semantic-preserving perturbations to the selected important instructions. Our method 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 reached. The goal is to maximize the minimum similarity between the adversarial sample with the target functions, which can be expressed as:

$$\max_{F_s^k} \left(\min_{i \in \{0,1,\ldots,m\}} \left(sim({F_s}^k, {F_t}^i) \right) \right) . \tag{3}$$

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.

The detailed algorithm is presented in Algorithm 1. In each iteration, we first use explainers described in §4.2 to calculate the weights of features, and select the important instructions

Algorithm 1: Adversarial binary sample generation with explanation guidance.

Input : Input function F_s, target functions $F_t = [F_t^0, F_t^1, \ldots, F_t^m]$, perturbations P
Output: An adversarial function F_s'

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

accordingly (Lines 4-5). For each important instruction, we randomly select a list of candidate semantic-preserving perturbations and apply the perturbation on the instructions. A list of intermediary adversarial samples is then generated (Line 9). To escape local optima during the update process (Line 10-12), our algorithm typically selects the candidate that achieves the highest similarity to the target function, but deliberately accepts a sub-optimal candidate with a small probability p_u (Line 11). Finally, 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). This helps to maximize the potential optimization from explanations, while also avoiding the computational cost to explain all combinations of target functions and the adversarial samples.

4.3.1 ***Semantic-Preserving Perturbation.*** Modern binary function similarity systems are vulnerable to carefully crafted semantic-preserving program perturbations. Prior works have extensively studied the effect of various semantic-preserving instruction perturbations in generating adversarial samples, and we adopt four representative ones. Specifically, *Swap (Instruction Reordering)* reorders two independent instructions within a basic block; *Strand Addition (SA)* inserts a new instruction sequence while preserving register states; *Displace (Node Split)* splits a basic block into multiple nodes via unconditional jumps; and *Dead Branch Addition (DBA)* introduces an unreachable branch with an always-false conditional jump. These perturbations are representative semantic-preserving transformations widely adopted in prior BCSD attack studies [8, 9, 25]. They cover perturbations in diverse granularities (*e.g.*, instruction and basic block), allowing us to extensively evaluate explanation guidance.

Target-Aware DBA. Beyond the four perturbations, we additionally develop a new method tailored for targeted adversarial attacks. Specifically, the adversary's goal is to increase the similarity between a query function and a specific but semantically unrelated target function. Standard DBA samples strands from a large corpus and inserts them randomly, hoping some will raise the similarity score.

However, this process is inefficient and often ineffective. Our target-aware DBA thus directly aligns the control flow graphs of source and target functions. Instead of relying on generic strands, it copies instruction sequences from the target into dead branches of the source. By emulating the target's low-level code patterns, the adversarial sample inherits structural features that similarity models are more likely to recognize, thereby improving the targeted similarity score.

Concretely, our implementation begins by linearizing the source CFG and constructing a mapping from each instruction to a global position index. A parallel mapping is built for the target CFG, where each index corresponds to an instruction in the target. For each source instruction selected by the explainer, we first identify the corresponding nearest position in the target CFG and extract the matched instruction and its immediate neighbors to form a strand. The strand is inserted into the source CFG by creating a dead branch so that the functionality is retained. Specifically, we split the original basic block and inject an always-false condition through a `cmp/je` instruction pair. The extracted strand is appended to the newly created branch with appropriate updates to instruction addresses and metadata.

4.4 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 [3] and angr [1]. 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.

5 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 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?

5.1 Experimental Setup

5.1.1 Baseline. To select a strong baseline, we systematically searched for prior adversarial attacks against BCSD models. Among the identified state-of-the-art methods—A_1 [8], A_2 [9], A_3 [25] and another open-source tool [64]—A_2 best satisfies the requirements for fair and reproducible comparison. Specifically, A_2 essentially extends A_1 by integrating more perturbations on a wider range of BCSD models, while the implementation of A_3 is not publicly available. The tool from [64] adopts a purely random strategy, accepting a perturbation only when it reduces the similarity score. Without further optimization or local optima escaping, it serves better as a lower-bound reference rather than a strong baseline. Therefore, we selected A_2 as the primary baseline for our comparative evaluation[1], and evaluated both the baseline and our approach against a broader set of recent, state-of-the-art BCSD models (§3) to ensure comprehensive assessment.

5.1.2 Dataset. We used the same dataset as the baseline for fair comparison. More specifically, the dataset consists of eight open-source projects written in C: binutils [7], curl [13], openssl [42], sqlite [56], gsl [19], libconfig [35], ffmpeg [17], and postgresql [47]. The projects were compiled on

[1]We obtained the source code from the authors for the comparison.

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.

5.1.3 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 §4.3, in a targeted adversarial attack, an adversary aims to maximize the similarity between an input function and a set of target functions. To enable a fair comparison, we adopt the same evaluation metrics as in the baseline. Specifically, we measure 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 (*i.e.*, the total set of target and background candidate functions). K is set much smaller than the pool size to simulate realistic, challenging retrieval scenarios. We also report the aggregated success rate **wASR**, defined in the baseline as 0.25*ASR@1 + 0.5*ASR@2 + 0.75*ASR@3 + ASR@4.

Required Modifications (M-Instrs and M-Nodes). We 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.

5.1.4 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 $p_u = 0.1$. We strictly followed the hyperparameter settings used by the baseline attack, without introducing any additional tunable parameters beyond the explanation process. All configurations were kept identical across different BCSD models. This guarantees that the results are not finetuned or ad-hoc to generate favorable results.

Table 1. Statistics about the sampled functions in our dataset. The numbers denote the average value observed across 200 compiled functions.

	gcc-O0	gcc-O3	clang-O0	clang-O3
#Basic Blocks	76.04	189.0	87.48	204.77
#CFG Edges	115.88	299.06	127.48	327.71
#Instructions	471.31	829.60	439.04	922.71
Cyclic Complexity	41.85	122.06	42.0	124.94

5.2 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, we randomly sampled 200 functions (50 x 4 compilation settings) with sufficient complexity across the object files in the compiled projects. Detailed statistics are listed in Table 1. For each function, we randomly selected another function in the evaluation dataset and used its four compiled variants as the target functions. Configurations are aligned with the baseline attacks.

Success Rates. We randomly select function pools from all functions in the object files and evaluate the attack success rates on ten BCSD models using pool sizes of 128, 512, and 1024. We present in Table 2 the results under pool size of 128 and K of 10. Due to the space limit, we attach the results of pool sizes of 512 and 1024 in our open-source artifact [10]. The trends of results remain consistent across all pool sizes. Specifically, while our goal is to improve efficiency without sacrificing the success rate, our explainer-guided attacks achieved a higher ASR in the vast majority of scenarios.

Table 2. Evaluation results of our attacks. 'Gd.' and 'Bs.' represent the explainer-guided and baseline attacks, respectively. "INIT" refers to the attack success rates using the original input sample. The results are based on the top-10 functions returned by each model when considering pools with size 128.

Group	Metric	BinFinder		jTrans		SAFE		Trex		PalmTree		ZEEK		Gemini		GMN		VexIR2Vec		CLAP	
		Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.
@1	INIT	0.50	0.52	0.73	0.72	0.68	0.68	0.59	0.58	0.50	0.50	0.67	0.66	0.74	0.74	0.79	0.79	0.13	0.13	0.32	0.33
	ASR	0.89	0.89	**0.98**	0.91	0.90	0.90	**0.69**	0.66	**0.83**	0.76	**0.89**	0.83	**0.88**	0.87	0.90	0.90	**0.32**	0.26	**0.85**	0.74
	M-Instrs	186.95	191.9	108.27	96.34	208.13	201.71	74.25	88.48	154.68	152.79	166.57	164.01	380.05	338.03	166.38	127.59	111.47	204.96	144.23	102.69
	M-Nodes	42.07	42.21	4.03	4.85	25.26	24.83	10.23	11.64	26.43	21.55	13.33	24.41	20.83	17.32	8.75	5.67	13.27	21.08	13.40	9.09
@2	INIT	0.36	0.38	0.27	0.29	0.44	0.44	0.25	0.28	0.31	0.33	0.51	0.51	0.58	0.58	0.53	0.54	0.04	0.04	0.20	0.21
	ASR	**0.78**	0.74	**0.51**	0.45	**0.81**	0.80	**0.37**	0.36	**0.63**	0.57	**0.71**	0.65	0.76	0.76	**0.69**	0.65	**0.14**	0.12	**0.69**	0.62
	M-Instrs	186.15	191.9	108.58	101.38	206.40	201.67	76.86	83.63	180.84	152.78	168.20	167.13	380.46	344	179.92	137.04	110.37	188.65	143.22	106.73
	M-Nodes	41.77	41.83	4.58	5.53	24.79	24.48	10.18	11.28	25.97	20.76	12.78	23.64	21.87	17.78	8.15	6.04	13.89	17.65	13.34	9.10
@3	INIT	0.20	0.22	0.08	0.09	0.20	0.20	0	0	0.17	0.15	0.25	0.26	0.19	0.18	0.25	0.26	0	0	0.09	0.08
	ASR	**0.70**	0.65	0.10	0.10	**0.69**	0.68	**0.05**	0	**0.50**	0.41	**0.45**	0.40	**0.44**	0.40	**0.38**	0.32	0.05	**0.06**	**0.54**	0.48
	M-Instrs	187.61	194.36	52.94	76.26	211.93	203.51	166.18	-	180.11	157.84	184.89	176.64	343.31	288.30	207.54	159.21	97.11	158	146.37	107.44
	M-Nodes	41.86	41.79	1.90	2.90	24.21	24.15	15.50	-	25.51	20.54	11.41	20.14	26.11	22.49	5.84	5.34	16.78	16.83	13.50	9.61
@4	INIT	0.11	0.11	0.01	0.01	0.08	0.08	0	0	0.06	0.06	0.13	0.13	0.08	0.08	0.09	0.08	0	0	0.05	0.04
	ASR	**0.59**	0.58	0.01	0.01	**0.49**	0.47	**0.02**	0	**0.36**	0.26	**0.29**	0.26	**0.32**	0.28	**0.19**	0.16	**0.04**	0.03	**0.38**	0.32
	M-Instrs	190.01	195.62	33.50	9	220.24	212.29	97	-	196.40	173.21	185.66	160.02	332.27	265.83	215.95	202.09	94.14	153.17	148.98	110.60
	M-Nodes	42.34	42.19	1	-	24.33	25.25	16.50	-	28.30	21.09	10.33	19.05	27.17	23.52	6.96	4.99	16.43	18.67	13.66	10

For example, at ASR@1, CLAP achieves 0.85 under explainer guidance versus 0.74 for the baseline, and ZEEK improves from 0.83 to 0.89, indicating enhanced effectiveness when guided by explainers. Similar advantages are observed on models processing graph-based features(*e.g.*, Gemini). Even for more challenging goals like ASR@2 and ASR@3, the explainer-guided attacks consistently lead to higher success rates across most models. Overall, our explainer-guided attacks outperform or match the baseline in 39 out of 40 evaluation cases. To rigorously verify this improvement, we conducted comprehensive statistical analyses. We formulated the null hypothesis (H_0) that there is no significant difference in the attack success rates between our explainer-guided attacks and the baseline. The results confirm that although the absolute average gain is modest (3.75%), the improvement is statistically significant and consistent across models (paired t-test, $p = 1.69 \times 10^{-9}$), with a large effect size (Cohen's $d = 0.86$).

Notably, the only case where our method underperforms the baseline is VexIR2Vec at ASR@3 (0.05 vs. 0.06). This can be attributed to VexIR2Vec's representation pipeline, which samples peepholes from the CFG via random walks and aggregates them mainly through summation. Such sampling and aggregation may dilute and destabilize the effect of perturbations targeting single instructions, potentially leading to varied efficacy depending on the context.

Code Modifications. We also presented the required modifications in Table 2. Our explainer-guided attacks show comparable or improved efficiency relative to the baseline in most cases.

However, for PalmTree, GMN, CLAP and Gemini, our approach tends to require slightly more modifications than the baseline. These findings demonstrate that improved success rate achieved by leveraging explainers does not come at the expense of significantly higher modification costs.

Effect of Parameter K. We further evaluated whether the parameter K may affect the ASRs. We measured ASRs from ASR@1 to ASR@4 under different values of K. However, due to space limit and presentation clarity, we report only ASR@1 and ASR@3 in Figure 3, which covers both relatively simple and more challenging scenarios. The results show that our explainer-guided attacks largely outperform the baseline in the majority of BCSD models under different K. The improvement is particularly significant on BCSD models like jTrans, PalmTree, and Trex.

Summary. Our explainer-guided attacks achieve comparable or even higher attack success rates than the baseline in almost all tests, while requiring a comparable amount of perturbations.

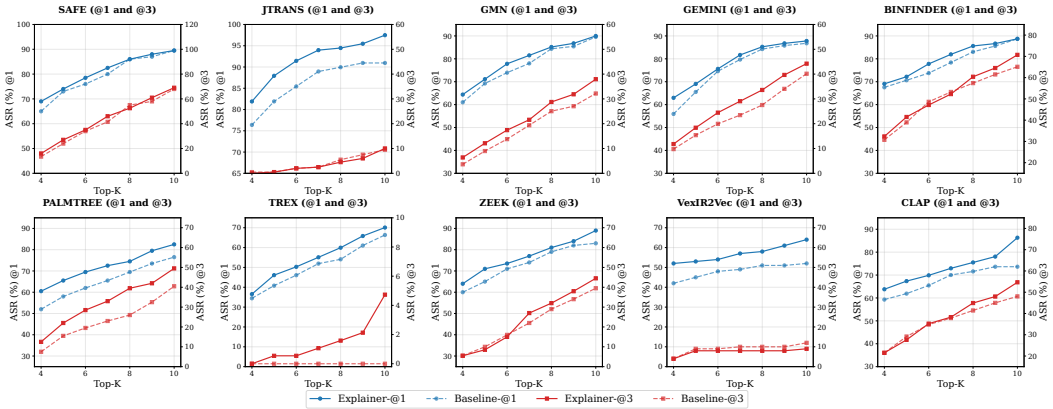

Figure 3. ASR with varying K for our attacks, with a pool size of 128.

5.3 Attack Efficiency (RQ2)

We present the runtime overhead for selecting the important instructions and generating an adversarial sample on all function samples evaluated in Table 3. The adversarial sample (AS) generation time refers to the overall runtime overhead for constructing the adversarial sample.

As shown, the explainers located important instructions with higher efficiency on all target models (*e.g.*, 28.4 seconds for Gemini and 13.6 seconds for ZEEK). Notably, we observed a speedup of 63.66x for instruction selection on ZEEK, while the optimization for SAFE was the least impactful at 33.6%. 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 51.2%.

Our runtime decomposition further confirms that the efficiency gains stem primarily from faster instruction selection rather than fewer iterations. The average iteration count remains largely stable (around 27 iterations) across models, yet the per-iteration instruction selection cost drops dramatically (*e.g.*, from 31.65 s to 0.49 s on ZEEK) with our explainers. To quantify how much of the total runtime reduction stems from the instruction selection phase, we further define a *"Selection Contribution"* metric, calculated as $\Delta T_{\text{select}}/\Delta T_{\text{total}}$, where ΔT_{select} and ΔT_{total} denote the reduction in total instruction selection and adversarial sample generation time, respectively. Overall, the *Selection Contribution* (Sel. Contrib. in Table 3) achieves up to 232.91%, demonstrating that selection-time reduction is the dominant driver. Values exceeding 100% indicate that while instruction selection became much faster, other parts of the pipeline (*e.g.*, perturbation) introduced additional overhead, partially offsetting the gain.

We consider the attack converged when the similarity change remains below 10^{-3} for five consecutive iterations. The convergence check shows that under the current budget, neither our method nor the baseline fully converges, indicating room for further improvement. As our attack is substantially more efficient per iteration, an increased budget would allow it to explore a wider perturbation space than the baseline.

Summary. 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 63.66x on instruction selection, while the overall overhead was reduced by up to 51.2%.

Table 3. Time overhead for generating adversarial samples (Gd.=Guided, Bs.=Baseline). Speedup is computed using the average total runtime across all evaluated samples, rather than the per-iteration runtime.

Metric	BinFinder		jTrans		SAFE		Trex		PalmTree		ZEEK		Gemini		GMN		VexIR2Vec		CLAP	
	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.	Gd.	Bs.
Avg. Iter.	27.83	27.50	27.44	27.66	27.76	27.88	27.72	27.55	26.72	24.77	27.88	27.77	27.27	27.76	26.91	27.68	27.57	27.61	27.46	27.39
Instruction Selection (Instr Sel.)																				
Total Time (s)	353.3	1091.8	23.4	425.6	117.1	156.4	95.7	364.4	35.9	315.6	13.6	878.8	28.4	715.2	96.3	1070.4	267.1	1701.8	15.7	414.9
Time/Iter (s)	12.70	39.70	0.85	15.39	4.22	5.61	3.45	13.22	1.34	12.74	0.49	31.65	1.04	25.77	3.58	38.68	9.69	61.65	0.57	15.15
Speedup (%)	**209.0**		**1720.5**		**33.6**		**280.9**		**780.0**		**6366.6**		**2422.4**		**1012.2**		**537.2**		**2544.3**	
Adversarial Sample Generation (AS Gen.)																				
Total Time (s)	5421.1	5981.0	3516.7	4208.1	3034.0	3210.8	3663.6	3779.0	3315.0	4095.1	4082.2	5525.2	4117.2	4617.9	5140.8	6820.0	5756.8	8716.3	4251.8	4539.8
Time/Iter (s)	194.8	217.5	128.2	152.1	109.3	115.2	132.2	137.2	124.1	165.4	146.4	199.0	151.0	166.4	191.0	246.4	208.8	315.8	154.9	165.8
Speedup (%)	**11.7**		**18.7**		**5.4**		**3.8**		**33.3**		**35.9**		**10.2**		**29.0**		**51.2**		**7.1**	
Sel. Contrib. (%)	**131.88**		**58.18**		**22.21**		**232.91**		**35.86**		**59.96**		**137.16**		**58.01**		**48.48**		**138.61**	
Conv. Rate (%)	25.92	26.10	92.54	89.22	46.31	49.54	87.35	82.22	44.39	53.26	50.27	55.84	45.63	59.42	92.95	91.35	90.40	86.10	62.95	61.16

5.4 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 [9].

As in Table 4, our adversarial samples generally demonstrate competitive and often higher transferability across models compared with the baseline. For instance, when SAFE, BinFinder and ZEEK serve as target models, our adversarial samples transfer particularly well, often surpassing the baseline, in some cases by a notable margin. By contrast, transferability degrades when the adversarial samples are generated for PalmTree. A possible explanation is the introduction of our new perturbation strategy, Target-Aware Dead Branch Addition. While this technique improves targeted similarity against the intended victim model, it may embed too specific structural patterns, thereby weakening their generalization across other models. We also observe that VexIR2Vec and CLAP emerge as the most transfer-resistant targets, with low wASR when targeted. By contrast, ZEEK, GMN, Gemini, and SAFE are notably easier to attack via transfer. One possible reason is that VexIR2Vec's unique peephole-based representation and CLAP's fine-grained tokenization enable the models to capture deeper semantic information beyond surface-level perturbations.

Summary. The adversarial samples generated in our explainer-guided attacks tend to be effective when used against other BCSD models. VexIR2Vec and CLAP are the most transfer-resistant targets, while ZEEK, GMN, Gemini, and SAFE are comparatively susceptible.

5.5 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.

5.5.1 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 [42], 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.

Table 4. 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 (%).

		Target Model									
Source	Variant	BinFinder	jTrans	SAFE	Trex	PalmTree	ZEEK	Gemini	GMN	CLAP	VexIR2Vec
BinFinder	Guided	–	**25.25**	**38.50**	20.375	**24.50**	**48.75**	44.875	47.125	18.25	**3.125**
	Baseline	–	25.125	35.625	**22.0**	23.75	47.50	**45.125**	**47.875**	**19.75**	2.625
jTrans	Guided	**30.875**	–	31.0	**19.375**	21.0	**44.625**	**40.875**	**46.0**	**19.75**	3.625
	Baseline	30.375	–	**32.625**	19.0	**21.375**	43.25	40.75	45.125	19.25	**4.875**
SAFE	Guided	**34.0**	**26.75**	–	**22.75**	19.875	**52.50**	**44.875**	43.125	17.75	**4.0**
	Baseline	31.50	25.75	–	21.75	**20.0**	50.375	44.625	**50.50**	**18.625**	3.25
Trex	Guided	**30.125**	28.25	**35.375**	–	23.625	**43.0**	41.875	**42.125**	**18.75**	5.125
	Baseline	29.25	**30.375**	33.375	–	**24.375**	39.75	**42.75**	40.375	17.625	**7.0**
PalmTree	Guided	**34.375**	**27.375**	**40.125**	**25.125**	–	**48.125**	**44.25**	**44.75**	19.25	5.50
	Baseline	32.625	25.875	28.50	23.25	–	46.25	43.625	42.75	**19.50**	**5.75**
ZEEK	Guided	**31.50**	**26.125**	31.625	**21.625**	17.125	–	42.0	47.75	**20.0**	**5.125**
	Baseline	30.875	22.625	**32.75**	19.0	**21.125**	–	**42.50**	**49.375**	19.625	4.50
Gemini	Guided	**33.375**	25.0	**31.50**	21.125	16.375	**53.375**	–	51.625	**18.0**	**5.25**
	Baseline	32.125	**27.25**	31.125	**23.75**	**17.125**	51.0	–	**52.375**	17.75	4.25
GMN	Guided	**31.625**	25.25	32.25	**21.375**	18.75	**48.625**	39.625	–	18.875	**5.375**
	Baseline	29.375	**25.75**	**33.125**	20.375	**19.0**	46.125	**41.375**	–	18.875	5.0
CLAP	Guided	**31.25**	**25.625**	**32.0**	**22.125**	20.625	**47.625**	**42.0**	**43.875**	–	**4.5**
	Baseline	31.0	24.75	31.75	19.625	**21.50**	44.0	41.0	40.375	–	3.875
VexIR2Vec	Guided	28.0	**28.875**	**34.0**	21.875	**22.50**	43.375	**42.625**	**45.375**	**20.50**	–
	Baseline	**28.75**	26.5	30.0	**22.25**	20.0	**45.0**	41.375	44.0	19.375	–

We measure four metrics: (1) *targeted attack success rate (ASR@i)*, (2) *initial similarity*, which is the similarity between the input samples and the target function before the attack; (3) *final similarity*, which captures the similarity after the attack; and (4) *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 5, our explainer-guided attack achieves higher ASR@1 while maintaining comparable performance in stricter success conditions (ASR@2-4), under the same setting (pool size = 128, $K = 10$). Table 6 provides complementary evidence from similarity scores. 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 targeted similarity shift helps explain the improved ASR, as the adversarial examples appear more functionally aligned with the target than typical samples in the pool, thereby increasing the likelihood of evading similarity-based detection.

Table 5. Targeted attack success rates on two real-world security tasks (pool size = 128, $K = 10$).

Metric	Vuln. Evasion		CWE Misclass.	
	Gd.	Bs.	Gd.	Bs.
ASR@1	**0.85**	0.75	**0.96**	0.88
ASR@2	0.35	0.35	0.17	0.17
ASR@3	0.10	0.10	**0.04**	0
ASR@4	0	0	0	0

Table 6. 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

5.5.2 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 [14], 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 [2] and the NIST Software Assurance Reference Dataset [4]. 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 targeted attack success rate, the initial, final, and average similarity as well.

Table 5 shows that our method also achieves higher or comparable ASR@i than the baseline for CWE misclassification. As shown in Table 7, this effectiveness is accompanied by a consistent increase in similarity toward the targeted CWE functions. On average, the initial similarity was 0.859, which is lower than the average pool similarity of 0.912. 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 7. 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

Summary. 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.

6 Discussion

Threat to Validity. We select representative explainers that align with each model family, rather than designing custom explanation mechanisms tailored to specific architectures. Similarly, we adopt a representative set of semantic-preserving perturbations widely used in prior BCSD attacks without task-specific tuning. Nevertheless, other perturbations may also be compatible with explanation guidance, which we leave for future exploration. We randomly selected target functions for each source function. While different target choices may influence attack success rates, randomization across 200 trials mitigates target-specific bias. Moreover, our threat model assumes knowledge of the feature mapping and model architecture category, consistent with prior robustness studies in the research settings [6, 15, 32, 63]. Specifically, 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 [24], Binary Ninja [58], and angr [1]. In practice, these intermediate features are often exported by the toolchains for integration with other modules, *e.g.*, for visualization or further analysis. Even when partially obscured, adversaries may approximate them by replicating the preprocessing steps or observing intermediate artifacts.

Nevertheless, strict black-box settings where structural and feature details are fully hidden represent a more restrictive scenario and remain an important direction for future investigation.

Optimizations. In our current implementation, the selected instructions are manipulated similarly to existing studies [8, 9]. We adopt such a design to enable fair comparison with state-of-the-art attacks. In addition to the four existing perturbation strategies, we further introduce a new transformation (DBA Variant), which is specifically designed for targeted attacks. Future work can extend our framework with more effective instruction manipulation approaches to further boost the attack performance. For example, program obfuscation techniques can effectively disrupt the static disassembly process. Other viable methods include edge hiding [26] and code cloning [26], which aim to alternate or obscure certain control flow paths. Junk code insertion [29] 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.

Defense. Existing studies have proposed several defenses against relevant adversarial attacks. Adversarial training approaches like FuncFooler [26] 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. To effectively defend against explainer-guided attacks, countermeasures should destabilize feature attribution or dilute the impact of localized perturbations. One promising direction is (De)Randomized Smoothing [18]. For instance, chunk-based smoothing evaluates randomly ablated input chunks independently and aggregates the results, preventing localized adversarial payloads from dominating the prediction. Applying this principle to BCSD would force the attacker to craft perturbations on the query function that remain robust across a diverse set of its randomly ablated sub-components. Our observations suggest that optimizing perturbations under multiple similarity evaluations drastically reduces the success rate. Other plausible defenses include model ensembling and architecture-diversity that destabilize attribution patterns. While this has been validated in other domains like computer vision [43], its core principle appears applicable to BCSD tasks. Different neural architectures (*e.g.*, GNNs vs. Transformers) extract distinct representations and may exhibit divergent local decision boundaries. Aggregating decisions from heterogeneous models could therefore disrupt the effectiveness of adversarial samples. We leave systematic evaluation and design of such defense to future work.

Insights and Implications. Our study suggests that in the evaluated attack settings, the cost of identifying salient perturbation locations can become a key efficiency bottleneck. Instead of probing all candidate locations uniformly or heuristically, explainers prioritize important ones by approximating the local decision boundary, improving the efficiency. Beyond targeted attacks, this explainer-guided principle exhibits strong extensibility. For instance, it can be naturally adapted to untargeted adversarial tasks by utilizing explainers to minimize the similarity score between a source function and its variants. Furthermore, structured guidance is particularly valuable when the exploration space is large, and the principle could also benefit broad software engineering tasks like search-based program repair, mutation testing prioritization, and guided fuzzing or robustness evaluation, where explanations can pinpoint suspicious input or code regions. In contrast to approaches that rely on heuristics or coverage feedback to guide the search, our study suggests that explanations provide an internal, model-centric view that reveals *why* a model behaves as it does, enabling more principled search-space prioritization.

7 Related Work

Adversarial Attacks against Source Code Analysis. Various advanced models for source code analysis have been proposed, improving the performance in tasks like clone detection, method name prediction, *etc.* Extensive efforts have been invested to measure their robustness via adversarial

attacks. Yefet *et al.* [66] designed a white-box adversarial attack against models for Java and C# analysis, assuming access to gradients. Zhang *et al.* [69] implemented fifteen semantic-preserving code transformation to construct adversarial samples against clone detection models. Quiring *et al.* [49] focused on semantic equivalent coding style transformation that downgrades authorship attribution accuracy. Other attacks have also targeted plagiarism detection, *e.g.*, through genetic code transformations [16]. 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.* [9] adopted a brute-force strategy to select instructions for perturbation based on similarity score changes after removing or perturbing each instruction , supporting both black-box and white-box attacks against three state-of-the-art BCSD models [8]. Song *et al.* [55] focused particularly on malware classifiers and aimed to generate adversarial samples to evade malware detection and analysis. Jia *et al.* [25] 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.* [31] 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 [30]. In contrast, Lucas *et al.* [36] 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 [70]. In this work, we explored another viable direction to optimize the attack efficiency, *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.* [22, 23] generated explanations for Android malware detection models to enhance the usability. Arp *et al.* [5] 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 [52]. In this work, we demonstrated that explainers can also effectively optimize the adversarial attacks against binary similarity detection models.

8 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, our approach could generate effective adversarial function samples in a computationally efficient way. The evaluation on binary functions from real-world projects proved that explainers provide actionable and granular guidance for adversarial manipulation, significantly accelerating perturbation target selection and improving attack efficiency. The discoveries highlight the lack of robustness in existing BCSD models, demonstrating the possibility of hindering 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.

9 Data-Availability Statement

We provide the artifact on Zenodo [10], with a maintained codebase on GitHub [11].

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62402423) and the Fundamental Research Funds for the Central Universities (No. 226202400143).

References

[1] 2025. Angr. https://github.com/angr/angr.

[2] 2025. National Vulnerability Database (NVD). https://nvd.nist.gov/.

[3] 2025. Radare2. https://github.com/radareorg/radare2.

[4] 2025. Software Assurance Reference Dataset (SARD). https://samate.nist.gov/SARD/.

[5] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket.. In *Ndss*, Vol. 14. 23–26. doi:10.14722/ndss.2014.23247

[6] Battista Biggio and Fabio Roli. 2018. Wild patterns: Ten years after the rise of adversarial machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2154–2156. doi:10.1016/j.patcog.2018.07.023

[7] Binutils Project. [n. d.]. Binutils. https://ftp.gnu.org/gnu/binutils/. Accessed: 2025-05-30.

[8] Gianluca Capozzi, Daniele Cono D'Elia, Giuseppe Antonio Di Luna, and Leonardo Querzoni. 2024. Adversarial attacks against binary similarity systems. *IEEE Access* (2024). doi:10.1109/access.2024.3488204

[9] Gianluca Capozzi, Tong Tang, Jie Wan, Ziqi Yang, Daniele Cono D'Elia, Giuseppe Antonio Di Luna, Lorenzo Cavallaro, and Leonardo Querzoni. 2025. On the lack of robustness of binary function similarity systems. In *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 980–1001. doi:10.1109/eurosp63326.2025.00060

[10] Mingjie Chen, Tiancheng Zhu, Mingxue Zhang, Yiling He, Minghao Lin, Penghui Li, and Kui Ren. 2026. Artifact for "Unveiling the Fragility of Binary Code Similarity Detection via Targeted Attacks with Model Explanations". doi:10.5281/zenodo.19709683

[11] Mingjie Chen, Tiancheng Zhu, Mingxue Zhang, Yiling He, Minghao Lin, Penghui Li, and Kui Ren. 2026. Explainer-Guided-Adv-Attack-BCSD (Code Repository). https://github.com/zju-ws-seclab/Explainer-Guided-Adv-Attack-BCSD. GitHub repository.

[12] Zhuo Chen, Jie Liu, Yubo Hu, Lei Wu, Yajin Zhou, Yiling He, Xianhao Liao, Ke Wang, Jinku Li, and Zhan Qin. 2023. Deuedroid: Detecting underground economy apps based on utg similarity. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 223–235. doi:10.1145/3597926.3598051

[13] Curl Project. [n. d.]. Curl. https://curl.se/. Accessed: 2025-05-30.

[14] CVE. 2025. CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/.

[15] Salvatore Della Torca, Valentina Casola, and Simone Izzo. 2025. N-Pixels: a Novel Grey-Box Adversarial Attack for Fooling Convolutional Neural Networks. In *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing (SAC '25)*. ACM. doi:10.1145/3672608.3707944

[16] Breanna Devore-McDonald and Emery D Berger. 2020. Mossad: Defeating software plagiarism detection. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28. doi:10.1145/3428206

[17] FFmpeg Project. [n. d.]. FFmpeg. https://ffmpeg.org/. Accessed: 2025-05-30.

[18] Daniel Gibert, Giulio Zizzo, Quan Le, and Jordi Planes. 2024. Adversarial Robustness of Deep Learning-Based Malware Detectors via (De)Randomized Smoothing. *IEEE Access* 12 (2024), 61152–61162. doi:10.1109/ACCESS.2024.3392391

[19] Gsl Project. [n. d.]. Gsl. https://www.gnu.org/software/gsl/. Accessed: 2025-05-30.

[20] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. LEMNA: Explaining Deep Learning based Security Applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer & Communications Security*. 364–379. doi:10.1145/3243734.3243792

[21] KyoungSoo Han, Jae Hyun Lim, and Eul Gyu Im. 2013. Malware analysis method using visualization of binary files. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*. 317–321. doi:10.1145/2513228.2513294

[22] Yiling He, Yiping Liu, Lei Wu, Ziqi Yang, Kui Ren, and Zhan Qin. 2022. Msdroid: Identifying malicious snippets for android malware detection. *IEEE Transactions on Dependable and Secure Computing* 20, 3 (2022), 2025–2039. doi:10.1109/tdsc.2022.3168285

[23] Yiling He, Jian Lou, Zhan Qin, and Kui Ren. 2023. 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*. 416–430. doi:10.1145/3576915.3616599

[24] Hex-Rays. [n. d.]. IDA Pro. https://www.hex-rays.com/products/ida/. Accessed: 2025-05-30.

[25] Lichen Jia, Bowen Tang, Chenggang Wu, Zhe Wang, Zihan Jiang, Yuanming Lai, Yan Kang, Ning Liu, and Jingfeng Zhang. 2022. FuncFooler: A practical black-box attack against learning-based binary code similarity detection methods. *arXiv preprint arXiv:2208.14191* (2022).

[26] Lichen Jia, Chenggang Wu, Bowen Tang, Peihua Zhang, Zihan Jiang, Yang Yang, Ning Liu, Jingfeng Zhang, and Daisy Zhe Wang. 2024. 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*. 11508–11518. doi:10.18653/v1/2024.findings-emnlp.673

[27] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. Binaryai: binary software composition analysis via intelligent binary source code matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639100

[28] Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. 2020. Is BERT Really Robust? A Strong Baseline for Natural Language Attack on Text Classification and Entailment. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 8018–8025. doi:10.1609/aaai.v34i05.6311

[29] junk code. 2021. Foudation of CTF reverse engineering. https://blog.csdn.net/u011642058/article/details/114757503.

[30] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. 2018. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European signal processing conference (EUSIPCO)*. IEEE, 533–537. doi:10.23919/eusipco.2018.8553214

[31] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. 2018. Adversarial examples on discrete sequences for beating whole-binary malware detection. *arXiv preprint arXiv:1802.04528* (2018), 490–510.

[32] Raz Lapid and Moshe Sipper. 2023. 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, 277–289. doi:10.1007/978-3-031-74643-7_21

[33] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. Palmtree: Learning an assembly language model for instruction embedding. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. 3236–3251. doi:10.1145/3460120.3484587

[34] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*. PMLR, 3835–3845.

[35] Libconfig Project. [n. d.]. Libconfig. https://github.com/hyperrealm/libconfig. Accessed: 2025-05-30.

[36] Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K Reiter, and Saurabh Shintre. 2021. Malware makeover: Breaking ml-based static analysis by modifying executable bytes. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 744–758. doi:10.1145/3433210.3453086

[37] Scott M Lundberg and Su-In Lee. 2017. 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., 4765–4774. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

[38] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. 2020. Parameterized explainer for graph neural network. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 33. 400–411.

[39] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177. doi:10.1109/tse.2017.2655046

[40] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2021. Function representations for binary similarity. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2259–2273. doi:10.1109/tdsc.2021.3051852

[41] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 89–100. doi:10.1145/1273442.1250746

[42] OpenSSL Project. [n. d.]. OpenSSL: The Open Source Toolkit for SSL/TLS. https://www.openssl.org/. Accessed: 2025-05-30.

[43] Tianyu Pang, Kun Xu, Chao Du, Ning Chen, and Jun Zhu. 2019. Improving Adversarial Robustness via Promoting Ensemble Diversity. In *Proceedings of the 36th International Conference on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 97)*. 4970–4979.

[44] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. 2016. The limitations of deep learning in adversarial settings. In *2016 IEEE European symposium on security and privacy (EuroS&P)*. IEEE, 372–387. doi:10.1109/eurosp.2016.36

[45] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2022. Learning approximate execution semantics from traces for binary function similarity. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2776–2790. doi:10.1109/tse.2022.3231621

[46] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1332–1349. doi:10.1109/sp40000.2020.00073

[47] Postgresql Project. [n. d.]. Postgresql. https://www.postgresql.org/. Accessed: 2025-05-30.

[48] Abdullah Qasem, Mourad Debbabi, Bernard Lebel, and Marthe Kassouf. 2023. 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. 443–456. doi:10.1145/3579856.3582818

[49] Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading authorship attribution of source code using adversarial learning. In *28th USENIX Security Symposium (USENIX Security 19)*. 479–496.

[50] Shuhuai Ren, Yihe Deng, Kun He, and Wanxiang Che. 2019. Generating Natural Language Adversarial Examples through Probability Weighted Word Saliency. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 1085–1097. doi:10.18653/v1/p19-1103

[51] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. " 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*. 1135–1144. doi:10.1145/2939672.2939778

[52] Giorgio Severi, Jim Meyer, Scott Coull, and Alina Oprea. 2021. {Explanation-Guided} backdoor poisoning attacks against malware classifiers. In *30th USENIX security symposium (USENIX security 21)*. 1487–1504.

[53] Noam Shalev and Nimrod Partush. 2018. Binary similarity detection using machine learning. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*. 42–47. doi:10.1145/3264820.3264821

[54] Samiha Shimmi, Ashiqur Rahman, Mohan Gadde, Hamed Okhravi, and Mona Rahimi. 2024. {VulSim}: Leveraging Similarity of {Multi-Dimensional} Neighbor Embeddings for Vulnerability Detection. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1777–1794.

[55] Wei Song, Xuezixiang Li, Sadia Afroz, Deepali Garg, Dmitry Kuznetsov, and Heng Yin. 2022. 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*. 990–1003. doi:10.1145/3488932.3497768

[56] Sqlite Project. [n. d.]. Sqlite. https://www.sqlite.org/. Accessed: 2025-05-30.

[57] Zeyu Sun, Changjian Li, Junda Yao, Yin Wang, Qingshan Zheng, and Yang Liu. 2022. Understanding and Improving Graph Neural Networks for Vulnerability Detection. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. ACM, 1–12.

[58] Vector 35 Inc. [n. d.]. Binary Ninja. https://binary.ninja/. Accessed: 2025-05-30.

[59] S. VenkataKeerthy, Soumya Banerjee, Sayan Dey, Yashas Andaluri, Raghul P. S., Subrahmanyam Kalyanasundaram, Fernando Magno Quintão Pereira, and Ramakrishna Upadrasta. 2025. VexIR2Vec: An Architecture-Neutral Embedding Framework for Binary Similarity. *ACM Transactions on Software Engineering and Methodology* 34, 8 (2025), 219:1–219:54. doi:10.1145/3721481

[60] Gérard Wagener, Radu State, and Alexandre Dulaunoy. 2008. Malware behaviour analysis. *Journal in computer virology* 4 (2008), 279–287.

[61] Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. 2024. CLAP: Learning Transferable Binary Code Representations with Natural Language Supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 503–515. doi:10.1145/3650212.3652145

[62] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1–13. doi:10.1145/3533767.3534367

[63] Hanrui Wang, Shuo Wang, Zhe Jin, Yandan Wang, Cunjian Chen, and Massimo Tistarelli. 2021. Similarity-based Gray-box Adversarial Attack Against Deep Face Recognition. In *2021 16th IEEE International Conference on Automatic Face and Gesture Recognition (FG 2021)*. IEEE, 1–8. doi:10.1109/FG52635.2021.9667076

[64] Wai Kin Wong, Huajin Wang, Pingchuan Ma, Shuai Wang, Mingyue Jiang, Tsong Yueh Chen, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Deceiving Deep Neural Networks-Based Binary Code Matching with Adversarial Programs. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 117–128. doi:10.1109/icsme55016.2022.00019

[65] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. 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*. 363–376. doi:10.1145/3133956.3134018

[66] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30. doi:10.1145/3428230

[67] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. Gnnexplainer: Generating explanations for graph neural networks. *Advances in neural information processing systems* 32 (2019).

[68] Hao Yuan, Hongping Hai, Jiliang Tang, and Shuiwang Ji. 2022. Explainability in graph neural networks: A taxonomic survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 5 (2022), 5782–5799. doi:10.1109/tpami.2022.3204236

[69] Weiwei Zhang, Shengjian Guo, Hongyu Zhang, Yulei Sui, Yinxing Xue, and Yun Xu. 2023. Challenging machine learning-based clone detectors via semantic-preserving code transformations. *IEEE Transactions on Software Engineering* 49, 5 (2023), 3052–3070. doi:10.1109/tse.2023.3240118

[70] Zhuo Zhang, Guanhong Tao, Guangyu Shen, Shengwei An, Qiuling Xu, Yingqi Liu, Yapeng Ye, Yaoxuan Wu, and Xiangyu Zhang. 2023. {PELICAN}: Exploiting Backdoors of Naturally Trained Deep Learning Models In Binary Code Analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2365–2382.

Received 2025-09-12; accepted 2026-03-24