Fool Me If You Can: On the Robustness of Binary Code Similarity Detection Models against Semantics-preserving Transformations

JIYONG UHM, Sungkyunkwan University, Republic of Korea
MINSEOK KIM, Sungkyunkwan University, Republic of Korea
MICHALIS POLYCHRONAKIS, Stony Brook University, USA
HYUNGJOON KOO*, Sungkyunkwan University, Republic of Korea

Binary code analysis plays an essential role in cybersecurity, facilitating reverse engineering to reveal the inner workings of programs in the absence of source code. Traditional approaches, such as static and dynamic analysis, extract valuable insights from stripped binaries, but often demand substantial expertise and manual effort. Recent advances in deep learning have opened promising opportunities to enhance binary analysis by capturing latent features and disclosing underlying code semantics. Despite the growing number of binary analysis models based on machine learning, their robustness to adversarial code transformations at the binary level remains underexplored to date. In this work, we evaluate the robustness of deep learning models for the task of binary code similarity detection (BCSD) under semantics-preserving transformations. The unique nature of machine instructions presents distinct challenges compared to the typical input perturbations found in other domains. To achieve our goal, we introduce asmFooler, a system that evaluates the resilience of BCSD models using a diverse set of adversarial code transformations that preserve functional semantics. We construct a dataset of 9,565 binary variants from 620 baseline samples by applying eight semantics-preserving transformations across six representative BCSD models. Our major findings highlight several key insights: i) model robustness highly relies on the design of the processing pipeline, including code pre-processing, model architecture, and internal feature selection, which collectively determine how code semantics are captured; ii) the effectiveness of adversarial transformations is bounded by a transformation budget, shaped by model-specific constraints such as input size limits and the expressive capacity of semantically equivalent instructions; iii) well-crafted adversarial transformations can be highly effective, even when introducing minimal perturbations; and iv) such transformations efficiently disrupt the model's decision (*e.g.,* misleading to false positives or false negatives) by focusing on semantically significant instructions.

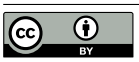
CCS Concepts: • **Security and privacy** → *Software reverse engineering*; • **Computing methodologies** → *Machine learning*.

Additional Key Words and Phrases: Binary Analysis, Similarity Detection, Obfuscation, Deep Learning

ACM Reference Format:
Jiyong Uhm, Minseok Kim, Michalis Polychronakis, and Hyungjoon Koo. 2026. Fool Me If You Can: On the Robustness of Binary Code Similarity Detection Models against Semantics-preserving Transformations. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE088 (July 2026), 23 pages. https://doi.org/10.1145/3797116

*Corresponding author.

Authors' Contact Information: Jiyong Uhm, Sungkyunkwan University, Suwon, Republic of Korea, jiyong423@g.skku.edu; Minseok Kim, Sungkyunkwan University, Suwon, Republic of Korea, for8821@g.skku.edu; Michalis Polychronakis, Stony Brook University, Stony Brook, USA, mikepo@cs.stonybrook.edu; Hyungjoon Koo, Sungkyunkwan University, Suwon, Republic of Korea, kevin.koo@skku.edu.



ACM 2994-970X/2026/7-ARTFSE088
https://doi.org/10.1145/3797116

1 Introduction

Executable binaries (hereinafter referred to as binaries) are pervasive in modern computing, powering personal computers, mobile devices, Internet of Things (IoT) systems, servers, cloud infrastructures, and smart appliances. The core of a binary is a sequence of machine-executable instructions derived from human-written source code, defining the operational logic of a computing device. However, the form and content of a binary can vary widely, even when compiled from identical source code. Such variations arise due to differences in hardware architectures, platforms, file formats, optimization levels, and compilation toolchains.

Binary analysis plays a crucial role in security by uncovering low-level code behavior without access to its source code (*i.e.,* reverse engineering). It supports a wide range of applications, including digital forensics [82] and analysis of proprietary protocols [6]. This task becomes challenging when dealing with stripped binaries that lack high-level metadata such as function names, variable types, or structural annotations due to compiler optimizations and obfuscation techniques. To extract semantic information, researchers rely on static analysis [2, 34, 95, 96], dynamic analysis [19, 27], or hybrid approaches [83, 85]. Static analysis examines the internal structure of a binary (*e.g.,* symbol tables, disassembly, or decompiled code) without executing it, whereas dynamic analysis observes runtime behavior, including system interactions such as file operations, memory access patterns, and network activity. Despite their effectiveness, these conventional approaches often require domain expertise and laborious effort, limiting scalability and accessibility.

Recent advances in deep learning have opened promising directions for enhancing binary analysis by enabling the extraction of latent features in high-dimensional spaces. These capabilities unlock new opportunities across several domains: ① characterizing binary properties, such as programmer authorship [1, 7, 53], compiler toolchain provenance [18, 33], and malware detection [14, 50]; ② inferring code semantics, including plagiarism detection [64], binary code similarity detection (BCSD) [3, 16, 23, 70, 76, 98–100, 102], malware classification [17, 61, 88], and vulnerability detection [12, 20, 23, 65, 77, 99]; and ③ supporting binary reversing tasks, such as function name prediction [32, 45], variable name prediction [13, 32], and type prediction [13, 32]. Emerging trends in artificial intelligence suggest that Machine-Learning-as-a-Service (MLaaS) will increasingly be adopted as a black-box oracle in security contexts. The broad spectrum of models relies on diverse features, including control flow graph (CFG) information, instruction counts, numerical constants, and instruction embeddings, tailored to their specific analytical objectives. Despite the proliferation of deep-learning-based models for binary analysis, relatively few studies have investigated their robustness. Early works [8, 41] explore adversarial attacks against BCSD models, but limit their scope to a single type of semantics-preserving code transformation or to a narrow set of models.

In this paper, we focus on the models that leverage code semantics to detect the similarity between binary code snippets for two key reasons. First, BCSD is a well-established task [29, 44] with extensive research grounded in deep neural networks [3, 16, 23, 70, 76, 99]. Second, BCSD serves as an effective benchmark for robustness evaluation due to the abundance of publicly available datasets and diverse code representations. More importantly, our study is task-agnostic and generalizable to other (aforementioned) applications. To this end, we introduce asmFooler, a system for evaluating the robustness of BCSD models under a wide spectrum of binary code transformations. Unlike input perturbations in computer vision [10, 26, 67, 90] or natural language processing [42, 52, 101], machine instructions pose distinct challenges due to their unique and rich expressiveness. For example, both instructions `mov rax, 0` and `xor rax, rax` set the `rax` register to 0, but differ in syntax and potential side effects, highlighting the subtleties of semantic equivalence in binary code. Such transformations, while preserving program semantics, have long been studied in the context of code diversification [47, 49, 75] and code obfuscation [43, 93]. Another challenge

arises from the variation in preprocessing across different models, which affects how raw machine instructions are normalized, structured, and fed into the learning pipeline. In this work, we apply eight types of semantics-preserving transformations [43, 47, 62, 75] across six representative BCSD models [3, 16, 23, 70, 76, 99] to assess their robustness.

We construct a corpus of 9,565 variants (*i.e.,* transformed binaries) from the 620 baseline samples by applying various semantics-preserving code transformations. To comprehensively assess the robustness of BCSD models, we conducted several types of evaluation experiments: ① false negative (FN; incorrectly classifying similar code pairs as dissimilar) triggering transformations using code diversification and obfuscation techniques, ② false positive (FP; inaccurately classifying dissimilar code pairs as similar) triggering transformations, and ③ the transferability of the FP-triggering transformations across models.

Our extensive experiments yield noteworthy key findings. First, model robustness relies heavily on the design of the training pipeline, including code pre-processing steps, architectural choices, and internal feature representations. These components collectively shape how code semantics are captured in the embedding space. For example, BinShot [3], which does not incorporate CFG information, is susceptible to basic block reordering. In contrast, Gemini [99] and Genius [23], both of which use CFG information, exhibit greater resilience to such transformations. Second, the effectiveness of an adversarial transformation is inherently bounded by a transformation budget, determined by model-specific constraints such as input size limits and the representational capacity (*i.e.,* expressivity) of semantically equivalent instruction variants. Third, well-engineered adversarial transformations can achieve high attack success rates against target models with minimal perturbations. For instance, our FP-triggering transformation reaches up to a 100% success rate while introducing only 14.75 additional instructions on average. Lastly, using two explainable AI techniques (*i.e.,* SHAP [63], saliency map [68]), we empirically discover that such transformations disrupt the model's decision by distorting internal attention patterns (*e.g.,* token importance).

This paper makes the following main contributions:

- We introduce an adversarial perturbation based on greedy sampling that induces false negatives and false positives by leveraging eight semantics-preserving code transformations.
- We construct 9,565 adversarial binary variants by applying a range of transformations designed to induce FNs by a BCSD model.
- We conduct comprehensive evaluations on six state-of-the-art BCSD models to assess their robustness against both FN- and FP-triggering perturbations.
- We uncover key insights that inform the development of more robust BCSD models, particularly in the context of adversarial resilience.

2 Background

2.1 Semantics-preserving Code Transformations

Despite their overlapping nature, we categorize code transformations into two groups: code diversification techniques, which aim to modify the code's footprint in memory (*e.g.,* to defend against code reuse attacks), and obfuscation techniques, which aim to deliberately confuse or obscure the code's functionality (*e.g.,* to hinder reverse engineering). The former typically introduce no new or very few instructions, while the latter often involve the insertion of additional code, such as inserting junk code or manipulating the CFG.

Code Obfuscation. Code obfuscation can serve both benign (*e.g.,* protecting intellectual property) and malicious (*e.g.,* evading malware from detection) purposes. A wide range of tactics have been introduced to obscure code, including the insertion of unreachable or redundant instructions, CFG complications, and data structure modifications. Obfuscation can be applied to various stages of

software development, including at the source level (*e.g.,* Tigress [93]), during compilation (*e.g.,* Obfuscator-LLVM [43]), or at the binary level (*e.g.,* VMProtect [87], Thermida [74], UPX [92]).

Obfuscator-LLVM. The LLVM compilation toolchain [59] provides a flexible and extensible framework based on the concept of a *pass*, *i.e.,* modular units of analysis or transformation applied to the (LLVM) intermediate representation during the compilation process. Obfuscator-LLVM [43] implements a set of transformation passes to apply code obfuscation techniques at compilation time, including ① instruction substitution that replaces arithmetic operations with semantically equivalent alternatives, ② bogus control flow that inserts unreachable basic blocks to mislead static analysis, and ③ control flow flattening that restructures a function's control flow into a single switch statement to obscure the original logic. In this work, we use Obfuscator-LLVM as part of the tool set for generating code variants.

Code Diversification. Code diversification is a technique that produces multiple variants of a binary, each preserving the original semantics while resulting in different memory layouts or execution paths. On one hand, code diversification can defend against code reuse attacks, such as return-oriented programming [79, 84] and jump-oriented programming [4], by invalidating the adversary's assumptions about a program's memory layout. On the other hand, malware authors can leverage code diversification to evade signature-based detection or to modify static features to bypass machine-learning-based malware classifiers [62].

In-place Code Randomization. In-place code randomization [75] introduces binary-oriented instrumentation with four transformation techniques of different spatial granularity (*e.g.,* instruction, basic block, function): ① *instruction substitution* that replaces the original (and exploitable) instructions (*i.e.,* gadget) with functionally equivalent ones; ② *intra-basic-block reordering* that relocates instructions in such a way that they are functionally identical within a basic block; ③ *register preservation code reordering* that reorders the push and pop instruction(s) from a function prologue and epilogue; and ④ *register reassignment* that swaps registers under non-overlapping regions with a liveness analysis of registers (*i.e.,* tracing registers that hold active values during program execution). In this work, we use in-place code randomization [75] as part of the tested code diversification techniques.

Compiler-assisted Code Randomization. CCR [47] introduces compiler-assisted binary instrumentation, which allows for fast and robust fine-grained code randomization at both the function and basic block levels. It augments a binary with a minimal set of transformation-assisting metadata from the compiler toolchain. We use *inter-basic-block reordering* from CCR as part of our diversification transformations [47].

2.2 Binary Code Similarity Detection

Binary code similarity detection (BCSD) estimates the similarity of two or more (binary) code snippets in the absence of the corresponding source code. The main challenge of BCSD arises from various compiler optimization pipelines that produce semantically identical but different binaries. Numerous studies [29] have focused on BCSD due to the broad spectrum of its applications, including (but not limited to) bug discovery [12, 20, 22, 23, 25, 37, 57, 65, 77, 78, 99], malware clustering [35, 36, 46], malware detection [5, 11, 50], malware lineage analysis [39, 56], and code clone detection [64]. Recent advances in deep learning open new opportunities to tackle BCSD [16, 23, 25, 57, 70, 99, 102]. Using recent transformer architectures has become a common approach [3, 76, 97, 100]. In parallel, graph neural networks are widely adopted for capturing structural and semantic code representations [30, 40]. We choose BCSD models to assess their resilience under a diverse set of code transformations.

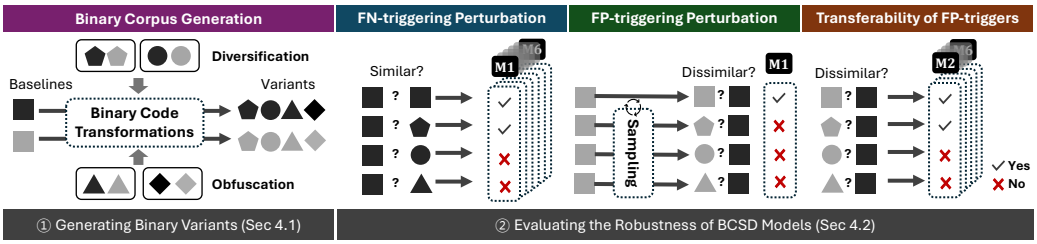


Fig. 1. Overview of the AsmFooler system with two main components: ① generating binary variants with various semantics-preserving code transformations (Section 4.1) and ② evaluating the robustness of six pre-selected BCSD models (Section 4.2). We assess the robustness of the models using adversarial samples designed to trigger either false negatives (FN) or false positives (FP). Additionally, we investigate the transferability of FP-trigger samples: *i.e.,* a sample that misleads one model affects others. Note that `M1` to `M6` denote six BCSD models in our study.

3 Robustness of ML-based BCSD Models

Code Features for BCSD Models. To infer the code semantics from a binary, BCSD models leverage both structural and instruction-level features. Structural features are extracted ① directly from the control flow graph (CFG) [57]; ② by enhancing a CFG with semantic information, such as through an attributed CFG [23, 25, 40, 99] or a semantic-oriented graph [30]; or ③ by approximating execution paths through techniques like random walks [16] or micro-traces [76]. Instruction-level features are usually captured using various embedding techniques, such as Word2Vec [72] (in InnerEye [102]), Instruction2vec [51] (in SAFE [70]), or Asm2Vec [16], which encode instruction sequences into dense vector representations. With the emergence of the transformer architecture [94], the semantic understanding of BCSD models [3, 76, 97, 100] has been further enhanced by incorporating positional encoding and the attention mechanisms, enabling effective modeling of contextual and structural relationships in binary code.

Threat Model. We assume an adversary, either curious or malicious, who can query a BCSD model (*i.e.,* black-box oracle) where the model takes a program binary (*e.g.,* in ELF format) as input. In this setting, the model's architecture is *unknown* and its internal parameters are *fixed*; *i.e.,* the attacker is disallowed to retrain the model or replace it with a different one of their choosing. Instead, the adversary can manipulate the raw bytes of the binary (*i.e.,* via binary instrumentation) in an attempt to mislead the model's classification outcome. The goal is to flip the prediction outcome, causing the model to inaccurately consider similar code snippets as dissimilar (false negatives), or dissimilar snippets as similar (false positives). Importantly, these two attack types differ in both difficulty and adversarial requirements. Inducing false negatives is comparatively straightforward, as an attacker can evade detection by applying semantics-preserving obfuscations to malicious code. In contrast, inducing false positives is more demanding: the adversary must deliberately manipulate the model's decision boundary to cause malicious code to be classified as benign, effectively steering the model's internal representation. We present both perturbations in detail in Section 4.2.

Transformation Budget. We adopt *functions* as our transformation unit, because they represent a logical part of code that encapsulates meaningful semantics. This implies that we cannot arbitrarily inject transformations across the entire binary, but are constrained within the boundaries of a given function. Besides, existing BCSD models often impose constraints on the maximum number of instructions they can process at once, primarily due to architectural limitations of the underlying models. For instance, four out of six models in our evaluation—Asm2Vec [16], SAFE [70], Trex [76], and BinShot [3], enforce such restrictions, with respective caps of 500 random walk instructions,

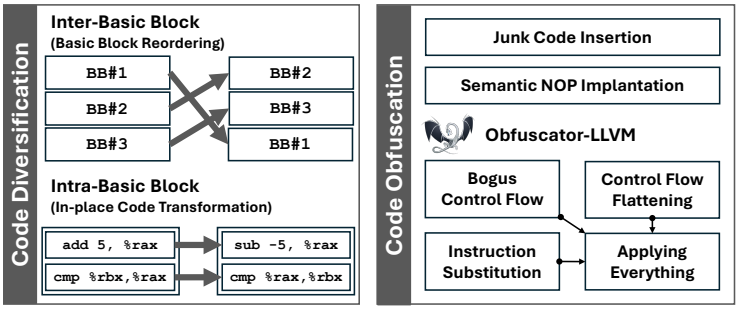


Fig. 2. Adversarial semantics-preserving code transformations in asmFooler. We adopt a wide range of transformation techniques from i) code diversification for defending against code reuse attacks (Section 4.1.1) and ii) code obfuscation for making static analysis challenging (Section 4.1.2).

512 instructions, 250 tokens, and 256 tokens, respectively. These differences stem from the internal processing approaches of each model: ① BinShot [3] counts tokens, each representing a single instruction; ② Trex [76] tokenizes instructions into opcode and operand(s); and ③ Gemini [99] and Genius [23] do not impose any restrictions. To ensure a fair comparison across models with varying input constraints, we define a *transformation budget*, which limits the number of instructions or bytes that may be added through semantics-preserving code transformations.

Evaluation Challenges. Unlike natural language processing (NLP) models, generating semantics-preserving machine code fundamentally differs from producing semantically analogous text. While human readers can often tolerate minor inconsistencies in language, CPUs interpret machine code with exact precision—even a single-bit error can bring about critical failures or unintended behavior. When the source code is available, binary instrumentation is relatively straightforward. However, in practice, transformations must often be applied directly to compiled binaries, a process that is significantly more challenging. Rewriting binaries without source code unavoidably requires extensive static analysis of assembly code, making the task complex and time-consuming. Our experiments highlight this challenge: applying a full suite of transformations inspired by in-place code randomization [75] failed on 23 out of 620 binaries, each exceeding our two-hour time limit for transformation. Although we impose practical constraints on generation time, a motivated attacker might allocate additional resources. Moreover, implanting malicious code, such as reusing fragments from known malware, demands manual adjustments to control-flow-related features, as the injected code shifts instruction addresses and potentially disrupts the existing control flow.

Virtual Scenario. Consider a scenario in which a website offers a BCSD service as an MLaaS instance designed to detect malicious functionality by analyzing the uploaded executable. This online system incorporates a built-in BCSD model, which returns binary classification decisions at the function granularity regarding the presence of known adversarial content. In this context, a curious adversary submits mutated versions of a seemingly benign executable, each embedding attacker-transformed code, intending to bypass the detection system. To increase the chances of evasion, the adversary applies a range of semantics-preserving code transformations, crafting function-level variants that may deceive the BCSD model while retaining the malicious intent.

4 asmFooler Design

asmFooler Overview. asmFooler aims to evaluate the robustness of BCSD models against a range of semantics-preserving code transformations. Figure 1 presents an overview of asmFooler,

illustrating both the generation of adversarial samples through semantics-preserving transformations and the subsequent robustness evaluation of BCSD models. The transformation engine in asmFooler incorporates techniques from both code diversification and code obfuscation. For diversification, it leverages in-place code transformation [75] and inter-basic-block reordering [47]. For obfuscation, it utilizes Obfuscator-LLVM [43], semantic NOP implantation [62], and junk code insertion, as depicted in Figure 2.

4.1 Generating Binary Variants

4.1.1 Code Diversification Techniques.

In-place Code Randomization. We adopt four techniques from in-place code randomization [75]: instruction substitution, intra-basic-block reordering, register preservation code reordering, and register reassignment. However we *redesign* these techniques for 64-bit ELF executables, as the original implementation has been limited to the 32-bit PE format. First, we update all general-purpose registers to their 64-bit counterparts, such as converting every 32-bit register to corresponding 64-bit register (*e.g.,* `eax` ⟶ `rax`) and inserting the necessary opcode prefix byte for 64-bit register operations (*e.g.,* `push rcx: 0x51`, `push r8: 0x41 0x51`). Second, we enhance the register liveness analysis to more precisely track register usage under the 64-bit calling convention, which involves multiple argument-passing registers. Third, we individually test each transformation on representative code samples to ensure correctness and maintain semantic equivalence. In this paper, note that we use the terms "randomization" and "transformation" interchangeably.

Inter-Basic-Block Reordering. The flexibility to reorder basic blocks is constrained by the operand size of control transfer instructions, since CCR [47] enforces preservation of the original instruction size. For instance, a single-byte operand in a jump or call instruction imposes a restricted displacement range, thereby constraining how far the target basic block can be relocated. As a result, certain functions become ineligible for inter-basic-block reordering due to these distance limitations. We identify and exclude such functions from code transformation.

4.1.2 Code Obfuscation Techniques.

Semantic NOP Implantation. A semantic NOP refers to a sequence of inserted instructions that preserve the original program behavior, leaving the surrounding code context unaffected. Inspired by the context-free grammar approach proposed by Lucas *et al.* [62], we generate semantic NOP sequences (within a predefined transformation budget) via a two-step process: ① constructing a derivation tree based on the grammar's transition rules, and ② traversing the tree to produce a variety of semantic NOP instruction sequences. However, we observe that the original approach can bring about an infeasibly large search space under certain conditions. For example, a register-specific state S_r tracks a register pushed to the stack (*e.g.,* `push rax` · S_{rax} · `pop rax`). Then, there are two ways to generate the non-terminating state $S_{ef,r}$ where $S_{ef,r}$ and $S_{rf,r}$ represent equivalent states except the former applies to 32-bit `EFLAGS` and the latter to 64-bit `RFLAGS`: *i.e.,* $S_{ef} \longrightarrow$ `push r` · $S_{ef,r}$ · `pop r` and $S_r \longrightarrow$ `pushfd` · $S_{ef,r}$ · `popfd`. The second transition rule requires maintaining a separate state S_r for each general-purpose register, significantly expanding the exploration space. To mitigate this, as in Table 1, we remove the register-specific state S_r and adjust other states accordingly, preserving expressiveness while improving efficiency in practice.

Junk Code Insertion. Junk code insertion introduces a sequence of unreachable instructions in the program. In our setup, we intentionally place the junk code at the beginning of the function and prepend it with an unconditional jump to ensure it is always bypassed during execution. This allows us to evaluate the impact of the length of dead code (*i.e.,* budget) on model behavior. To

Table 1. Context-free grammar for our semantic NOP generation. The left column represents the original version proposed by Lucas *et al.* [62], whereas the center represents the one that reduces the exploration space: *i.e.,* the S_r state (in red) has been eliminated, updating other states (in blue) accordingly. The right column shows the legend and x86 instructions.

S	$\rightarrow$	*Atom*	S	$\rightarrow$	*Atom*	S	:	Starting symbol
	|	$S \cdot S$		|	$S \cdot S$	ef	:	EFLAGS
	|	`bswap r` $\cdot S \cdot$ `bswap r`		|	`bswap r` $\cdot S \cdot$ `bswap r`	rf	:	RFLAGS
	|	`xchg r1, r2` $\cdot S \cdot$ `xchg r1, r2`		|	`xchg r1, r2` $\cdot S \cdot$ `xchg r1, r2`	Φ	:	Empty string
	|	`push r` $\cdot S_r \cdot$ `pop r`		|	`push r` $\cdot S \cdot$ `pop r`	arith	:	Arithmetic operation (*e.g.,* add)
	|	`pushfd` $\cdot S_{ef} \cdot$ `popfd`		|	`pushfq` $\cdot S_{rf} \cdot$ `popfq`	invarith	:	Inverse of `arith`
Atom	$\rightarrow$	`nop`	*Atom*	$\rightarrow$	`nop`	logic	:	Logical operation (*e.g.,* xor)
	|	`mov r, r`		|	`mov r, r`	r, r1, r2	:	Register
	|	Φ		|	Φ	v	:	Random integer
S_r	$\rightarrow$	S						
	|	$S_r \cdot S_r$				**x86-64 Instruction Set**		
	|	`pushfd` $\cdot S_{ef,r} \cdot$ `popfd`				`bswap`	:	Reverses the byte order of registers
S_{ef}	$\rightarrow$	S	S_{rf}	$\rightarrow$	S	`xchg`	:	Exchanges the values of two operands
	|	$S_{ef} \cdot S_{ef}$		|	$S_{rf} \cdot S_{rf}$	`push`	:	Stores a value on the stack top
	|	`arith r, v` $\cdot S_{ef} \cdot$ `invarith r, v`		|	`arith r, v` $\cdot S_{rf} \cdot$ `invarith r, v`	`pop`	:	Retrieves a value from the stack top
	|	`push r` $\cdot S_{ef,r} \cdot$ `pop r`		|	`push r` $\cdot S_{rf,r} \cdot$ `pop r`	`pushfd/pushfq`	:	Stores the EFLAGS/RFLAGS register onto the stack
$S_{ef,r}$	$\rightarrow$	S	$S_{rf,r}$	$\rightarrow$	S	`popfd/popfq`	:	Retrieves a value from the stack into the EFLAGS/RFLAGS register
	|	S_r				`mov`	:	Copies data from a source operand to a destination operand
	|	S_{ef}		|	S_{rf}	`nop`	:	Performs no operation
	|	$S_{ef,r} \cdot S_{ef,r}$		|	$S_{rf,r} \cdot S_{rf,r}$			
	|	`arith r, v` $\cdot S_{ef,r}$		|	`arith r, v` $\cdot S_{rf,r}$			
	|	`logic r, v` $\cdot S_{ef,r}$		|	`logic r, v` $\cdot S_{rf,r}$			

further complicate BCSD models, we randomly sample unique instructions to fill the junk code, while ensuring the overall transformation remains within the defined transformation budget.

LLVM-based Obfuscation Techniques. We extend our semantics-preserving code obfuscation in asmFooler to include source-level transformations, enabling the systematic application of more complex obfuscation techniques. To this end, we integrate Obfuscator-LLVM [43], a tool that applies obfuscation at the intermediate representation (IR) level during the optimization phase of compilation. The applied transformations include instruction substitution using arithmetic and logical operations, insertion of unreachable basic blocks, and control flow flattening. These techniques are applied either individually or in combination.

4.2 Evaluating the Robustness of BCSD Models

4.2.1 FN-triggering Perturbation. Most adversarial samples with semantics-preserving transformations (Section 4.1) fall into the category of FN-triggering perturbations. The goal of an FN-triggering perturbation is to deceive the model into incorrectly predicting that two semantically equivalent code snippets are dissimilar.

4.2.2 FP-triggering Perturbation. In contrast to FN-triggering perturbations, generating adversarial samples that trigger false positives is non-trivial, as it requires deceiving the model into predicting that two dissimilar code snippets are similar. To explore potential FP-triggering candidates, we design a greedy sampling strategy. For this evaluation, we select ten functions from ten distinct binaries, each constrained by a fixed instruction budget (Table 5).

Greedy Sampling. To trigger false positives in a model, we employ a greedy sampling strategy inspired by Zou *et al.* [101], which demonstrates its effectiveness in generating adversarial suffixes against large language models. Instead of using suffixes or perturbations at intermediate positions, we adopt adversarial prefixes to exploit the positional bias of language models, which tend to prioritize the beginning of the input context [58]. In essence, we first extract a target instruction

Algorithm 1 FP-triggering Instruction Sampling

```
1:  Input: corpus, target_model, target_func
2:  Output: adv_corpus
3:  target_dist ← Get_Distrib(target_func)
4:  adv_corpus = []
5:  for func in corpus do
6:     fp_triggering_codes = []
7:     fp_triggering_codes.push(UNCONDITIONAL_JUMP)
8:     while fp_triggering_codes.length < BUDGET do
9:        instr_list = []
10:       while instr_list.length < NUM_CANDIDATES do
11:          instr_list.push(Sample(target_dist))
12:       end while
13:       scores = []
14:       for instr in instr_list do
15:          attack_seq ← fp_triggering_codes + instr
16:          attack_func ← attack_seq + func
17:          score = target_model(attack_func, target_func)
18:          scores.push(instr, score)
19:       end for
20:       best_instr, best_score = Max(scores)
21:       fp_triggering_codes.push(best_instr)
22:       if best_score > THRESHOLD then
23:          break
24:       end if
25:    end while
26:    adv_corpus.push(fp_triggering_codes + func)
27: end for
28:
29: return  adv_corpus
```

```
1   pushfq
2   sub $0xc91a,%rcx
3   push %r10
4   add $0x5c54,%r10
5   sub $0x9303,%r15
6   add $0xc866,%rbp
7   xchg %r13,%rbp
8   xchg %r13,%rbp
9   sub $0xc866,%rbp
10  add $0x9303,%r15
11  pop %r10
12  add $0xa1c3,%r14
```

```
13  push %r9
14  pushfq
15  nop
16  sub $0x5380,%rdi
17  add $0x5380,%rdi
18  push %r13
19  pop %r13
20  popfq
21  pop %r9
22  sub $0xa1c3,%r14
23  add $0xc91a,%rcx
24  popfq
```

Fig. 3. Example of a semantic NOP sequence with the context-free grammar from Lucas *et al.* [62]. The chunk of instructions would not impact the semantics of a program as a subsequent instruction(s) counteract the side effects of one or more instructions ahead. For example, pushing the value of `r10` to the stack (Line 3) and adding an arbitrary value (Line 4) can be reversed by `pop r10` (Line 11).

distribution from a victim binary. Next, we iteratively select instructions from this distribution when prepended to a target function, producing the most contextually similar result (in a greedy manner) relative to a chosen victim function. This process continues until the transformation reaches a predefined budget, yielding a sequence of instructions designed to trigger false positives.

Sampling Algorithm for FP-triggering Instructions. Algorithm 1 outlines the instruction sampling procedure in our FP-triggering perturbation transformation. Given a specific BCSD model (`target_model`), the algorithm aims to transform a function from our corpus (`corpus`) so that it appears similar to a chosen function (`target_func`). We insert an unconditional jump instruction (`UNCONDITIONAL_JUMP`) at the start of the FP-triggering code sequence (`fp_triggering_codes`) to ensure the injected instructions are never executed. Note that the (candidate) instructions (*e.g.*, `NUM_CANDIDATES` = 20) are extracted from the target distribution (Lines 10–11) and the algorithm iteratively selects

```
1  0000000000008050 <initialize_eval>:
2      8050: cmp %rax,%rax
3      8053: je 0x830c
4      8059: movl $0x0,0xf21c9(%rip)
5      8063: movl $0x0,0xf21c9(%rip)
6      806d: callq 0x7fe0
7      8072: callq 0x7fe5
8      8077: mov %ecx,0xf26bf(%rip)
9      807d: jne 0x8090
10     ...
11     830c: push %rbp
12     830d: push %r15
13     830f: push %r14
```

```
13     8311: push %r13
14     8313: push %r12
15     8315: push %rbx
16     8316: xor %eax,%eax
17     8318: lea 0x29751(%rip),%rcx
18     831f: lea 0x5f23a(%rip),%rdx
19     8326: lea 0x4ae23(%rip),%rsi
20     832d: nopl (%rax)
21     8330: mov %eax,%edi
22     8332: and $0x7,%edi
23     8335: mov %eax,%r8d
24     8338: shr $0x3,%r8d
25     ...
```

Fig. 4. Example of an FP-triggering perturbation applied to the `initialize_eval` function from the `sjeng` binary in SPEC2006. The instructions at addresses `0x8059` to `0x807d` (Lines 4–9) represent the adversarial sequence inserted into the function prologue. To preserve original semantics, the instructions at `0x8050` and `0x8053` redirect the control flow to skip over the injected code. Note that we insert NOP padding as a length budget at the function prologue (*e.g.,* addresses between `0x8050` and `0x830c` in this example), followed by injecting the FP-triggering code to enable further variations. Thus, the space between `jne 0x8090` and `push %rbp` is filled with NOP instructions, keeping the original semantics intact.

the best-performing candidate (Line 20) and appends it to the sequence (Line 21). The sampling process continues until the sequence reaches the transformation budget (Line 8) or the similarity score surpasses a pre-defined threshold (Line 22), indicating that the target model would classify the modified and target functions as similar. Once the FP-triggering sequence is finalized, it is inserted into the target function, and the procedure moves on to the next function (Line 26).

4.2.3 Transferability of FP-Triggers. Finding a sample that triggers a false positive could be very useful for an adversary because it would help circumvent defenses based on BCSD models, as in the virtual scenario described in Section 3. Hence, we further investigate the transferability of FP-triggers on other BCSD models.

4.3 Code Perturbation Examples

Figure 3 presents an example of a semantic NOP sequence consisting of 100 bytes that introduces no functional side effects: *i.e.,* the effect of each instruction is neutralized by a corresponding subsequent instruction. For example, `sub $0xc91a, %rcx` on Line 2 is reversed by `add $0xc91a, %rcx` on Line 23, maintaining the original register state. Figure 4 illustrates a concrete example of an FP-triggering perturbation inserted into a function prologue. Akin to semantic NOP sequences, this perturbation preserves the original program behavior. In particular, the instructions between `0x8050` and `0x8053` (*i.e.,* `UNCONDITIONAL_JUMP`) maintain semantic correctness by rerouting the control flow around the inserted code. The code snippets from `0x8059` to `0x807d` contain a sampled adversarial instruction pattern that induces a false positive in the model. Notably, the address gap between `0x807d` and `0x830c` is padded with NOP instructions in our experiments, enabling flexible accommodation of perturbations of varying sizes across different functions.

5 Implementation

Generating Binary Variants. First, for in-place code randomization, the original implementation by Pappas *et al.* [75] was written in Python 2 and designed for the PE format [71]. To support 64-bit ELF [89] binaries, we re-implemented the tool in Python 3, utilizing `capstone` [9] for disassembly and `lief` [54] for generating binary variants. Due to the internal complexity of transformations that require liveness analysis, we impose a two-hour time limit per sub-transformation. In total,

we spent approximately 624 hours (roughly one month) to generate 597 mutations. Second, for semantic NOP generation, we adopt `NLTK` (Natural Language ToolKit) [80], a popular open-source library for natural language processing. Based on our pre-defined context-free grammar, we modify the sequence generation function to randomly explore valid combinations, ultimately generating 500 unique semantic NOP sequences ranging from 20 to 100 bytes in length. Third, to implant semantic NOPs and junk code, we first allocate a fixed-size placeholder using NOP bytes (`0x90`), leveraging the `-fpatchable-function-entry` compiler flag available in both GCC [91] and Clang [59]. We overwrite this reserved space with our custom instruction sequences.

Model Evaluation. We leverage a BCSD benchmark tool [69] to evaluate various BCSD models [16, 23, 70, 76, 99] against the generated samples. We made small modifications in restricting function pairs such that there is only a variation in transformation throughout our comparisons, and updating the metrics to support precision, recall, and F1 score (hereinafter F1).

Obfuscator-LLVM Reimplementation on LLVM 19. We ported the LLVM-4-based Obfuscator-LLVM tool [43] to LLVM 19.1.4 [59], transitioning from the legacy pass manager to the new pass manager. This upgrade ensures compatibility with other LLVM-19-based transformations we employ, namely, junk code insertion and semantic NOP implantation, which require support for the `-fpatchable-function-entry` compiler flag. Extensive changes in LLVM's API over time required significant modifications, as many functions in LLVM 4 have been deprecated or refactored in LLVM 19. For instance, `BinaryOperator::Create()` now requires an `InsertPosition` instead of an `Instruction`, and the method `BinaryOperator::CreateFNeg()` has been eliminated [86]. We address these alterations by consulting the LLVM source code, official documentation [59], and community discussions [60], and by replacing deprecated constructs accordingly: *e.g.,* substituting `CreateFNeg()` with `UnaryOperator::CreateFNeg()`.

6 Evaluating the Robustness of BCSD Models

We evaluate the robustness of six BCSD models against eight semantics-preserving transformations. We run our experiments on a 64-bit Ubuntu 20.04 system equipped with an Intel(R) Xeon(R) Gold 5218R CPU 3.00GHz, 512GB RAM, and two RTX A6000 GPUs.

Research Questions. We define the following four research questions to investigate the impact of semantics-preserving transformations on BCSD models.

- **RQ1**: How do code diversification techniques trigger false negatives in BCSD models (Section 6.1)?
- **RQ2**: How do code obfuscation techniques trigger false negatives in BCSD models (Section 6.2)?
- **RQ3**: How well can we trigger false positives against a BCSD model (Section 6.3)?
- **RQ4**: How well can the FP-triggering adversarial samples be transferred to other BCSD models (Section 6.4)?

Binary Corpus. We constructed a diverse set of baseline binaries from multiple sources, including essential system packages (*e.g.,* coreutils [24], binutils [24]), popular open-source utilities (*e.g.,* nginx [73], putty [81]), and benchmark programs from SPEC2006 [15]. We compile all executables with Clang (19.1.4 for obfuscation and 9.0.0 for diversification) at four optimization levels (O0-O3), targeting the x86-64 ELF format. First, we generate 1,215 adversarial samples via code diversification, as summarized in Table 2, comprising 597 from in-place code transformation and 618 from inter-basic-block reordering. Second, we produce 2,310 obfuscated variants using Obfuscator-LLVM. Compilation was successful for most packages, with the exception of some binutils and findutils binaries due to compiler version incompatibilities (Obfuscator-LLVM based on LLVM-19). Third, we create 3,020 binaries through semantic NOP implantation and junk code insertion, with compilation failures only for findutils due to the same compatibility issue. Lastly, we generate an additional 300 mutations from SPEC2006 to evaluate the transferability of our FP-triggering transformation. In

Table 2. Summary of our binary corpus. We generated a total of 9,565 adversarial variants based on 620 baseline binaries with the following transformation techniques: 597 from in-place code transformation (ICT), 618 from inter-basic-block reordering (BBR), 3,020 from semantic NOP implantation (SNI) and junk code insertion (JCI), and 2,310 using Obfuscator-LLVM [43]). The (*) marks indicate unsuccessful cases due to exceeding the 2-hour processing limits in ICT (Section 2); mutating failures in BBR (*e.g.,* `perlbench` and `gobmk` under `O1` optimization), incompatible compiler versions (*e.g.,* `findutils`), and compilation failures in O-LLVM.

Package	Version	Default Binaries	Functions	Diversification		Obfuscation			Total
				ICT	BBR	SNI	JCI	O-LLVM	
coreutils [24]	8.2	412	37,823	412	412	2,060	2,060	1,648	6,592
binutils [24]	2.27	64	97,893	*63	64	320	320	*150	917
findutils [24]	4.6.0	16	3,683	16	16	*0	*0	*0	32
diffutils [24]	3.2	16	1,740	16	16	80	80	64	256
nginx [73]	1.8.1	4	4,436	*3	4	20	20	16	63
putty [81]	0.66	28	25,505	*14	28	140	140	112	434
gzip [28]	1.8	4	487	4	4	20	20	16	64
lighttpd [55]	1.4.43	4	1,574	4	4	20	20	16	64
lvm2 [66]	2.02.168	8	10,701	*5	8	40	40	32	125
vsftpd [21]	3.0.3	4	2,248	4	4	20	20	16	64
miniweb [38]	N/A	4	296	4	4	20	20	16	64
SPEC2006 [15]	N/A	56	40,491	*52	*54	280	280	224	890
Total		620	226,877	597	618	3,020	3,020	2,310	9,565

Table 3. (P)recision, (R)recall, and F1 of six BCSD Models against the adversarial samples with inter-basic-block reordering (BBR) and in-place code transformation (ICT). Each entry records the performance of "before → after transformation (difference)". The bold represents the highest performance drop across different models for each transformation. While we observe substantial drops in recall against ICT, most models remain robust against BBR with the exception of BinShot (0.936 ↓). Besides, the models that capture dynamic features (*e.g.,* Asm2Vec, Trex) exhibit less performance degradation than others.

		Genius [23]	Gemini [99]	Asm2Vec [16]	SAFE [70]	Trex [76]	BinShot [3]
ICT	P	0.835 → 0.793 (0.042 ↓)	0.864 → 0.824 (0.040 ↓)	0.890 → 0.807 (**0.083** ↓)	0.860 → 0.823 (0.037 ↓)	0.925 → 0.929 (0.004 ↑)	0.990 → 0.967 (0.023 ↓)
	R	0.943 → 0.564 (0.379 ↓)	0.886 → 0.554 (0.332 ↓)	0.819 → 0.733 (0.086 ↓)	0.782 → 0.509 (0.273 ↓)	0.905 → 0.831 (0.074 ↓)	0.953 → 0.271 (**0.682** ↓)
	F1	0.886 → 0.659 (0.227 ↓)	0.875 → 0.663 (0.212 ↓)	0.814 → 0.768 (0.046 ↓)	0.819 → 0.629 (0.190 ↓)	0.915 → 0.877 (0.038 ↓)	0.971 → 0.423 (**0.548** ↓)
BBR	P	0.839 → 0.831 (0.008 ↓)	0.868 → 0.859 (0.009 ↓)	0.688 → 0.685 (0.003 ↓)	0.843 → 0.837 (0.006 ↓)	0.930 → 0.927 (0.003 ↓)	0.990 → 0.811 (**0.179** ↓)
	R	0.952 → 0.907 (0.045 ↓)	0.905 → 0.857 (0.048 ↓)	0.698 → 0.687 (0.011 ↓)	0.762 → 0.695 (0.067 ↓)	0.897 → 0.861 (0.036 ↓)	0.972 → 0.036 (**0.936** ↓)
	F1	0.892 → 0.867 (0.025 ↓)	0.886 → 0.858 (0.028 ↓)	0.693 → 0.686 (0.007 ↓)	0.800 → 0.759 (0.041 ↓)	0.913 → 0.893 (0.020 ↓)	0.981 → 0.069 (**0.912** ↓)

total, our corpus consists of 10,185 binaries: 9,565 transformed variants and 620 original baseline binaries.

Target Models. We evaluate six representative BCSD models: Genius [23], Gemini [99], Asm2Vec [16], SAFE [70], Trex [76], and BinShot [3]. It is noteworthy to mention that we carefully select different types: two models [23, 99] based on graph neural networks, two models [16, 70] using embeddings, and another two models [3, 76] that adopt BERT-based language models.

6.1 Impact of Code Diversification Techniques

In-place Code Transformation. The first half of Table 3 summarizes the performance degradation after in-place code transformation (ICT). We observe that the adversarial variants successfully trigger false negatives in a majority of models, significantly dropping recall. However, Asm2Vec [16] and Trex [76] demonstrate a relative robustness against ICT. BinShot [3] exhibits the lowest recall (0.271) against ICT although it also adopts a BERT-based language model like Trex. We hypothesize that BinShot relies on the correlation between instructions alone (for inference) without considering dynamic features like micro-traces [76] in Trex.

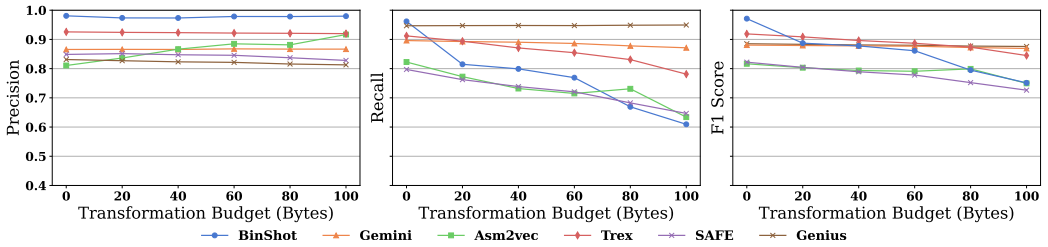


Fig. 5. Precision, Recall, and F1 of various BCSD models with a given transformation budget (*e.g.,* size of a semantic NOP in bytes). The models adopted a graph neural network like Genius [23] and Gemini [99] tend to be robust against semantic NOP implantation, while the others do not (*e.g.,* significant drops in recall). We discuss precision with FP-triggering perturbation in Section 6.3. Note that a byte does not necessarily correspond to a single token or instruction; *e.g.,* our experiments insert around 7 and 25 instructions on average under the budgets of 20 and 100 bytes, respectively.

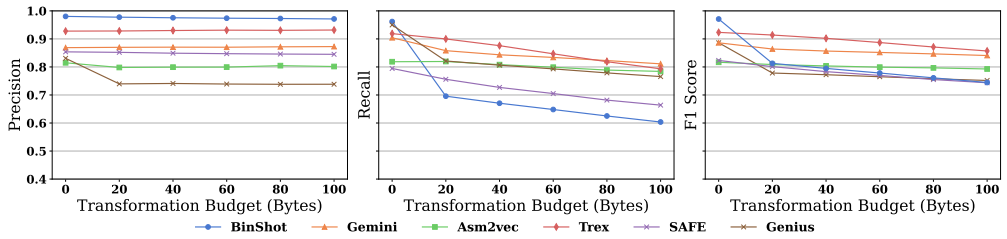


Fig. 6. Precision, Recall, and F1 of various BCSD Models in relation to the length of inserted junk code sequence. Notably, Asm2Vec [16] demonstrates substantial robustness, which can be attributed to the random walk that dynamically collects assembly sequences by traversing the assembly instructions. On average, 6 and 16 instructions are inserted under budget constraints of 20 and 100 bytes, respectively.

Inter-Basic-Block Reordering. Similarly, the second half of Table 3 shows the performance drops with inter-basic-block reordering (inter-BBR). This led to a high false negative rate against BinShot [3], reducing the recall score to as low as 0.036. Except for BinShot, we observe the other models are robust against inter-BBR presumably because they take an attributed CFG (*e.g.,* Genius [23], Gemini [99]), a random walk (*e.g.,* Asm2Vec [16]), or a micro-trace (*e.g.,* Trex [76]) into account. Meanwhile, SAFE [70] inserts instruction embeddings into an attentive neural network, indicating that the order of instructions is not essential.

6.2 Impact of Code Obfuscation Techniques

Semantic NOP Implantation. Figure 5 illustrates how the performance of each model decreases after inserting a semantic NOP at a function entry. With our transformation budget, we incrementally increase it by 20 bytes up to 100 bytes. Each model exhibits varying drop rates; *e.g.,* Asm2Vec [16], SAFE [70], and BinShot [3] drastically drop in recall because they rely on instruction-level features. Trex [76] shows a moderate decline. However, graph neural network-based models (Genius [23] and Gemini [99]) are robust against semantic NOP insertion because it impacts only the first basic block while maintaining a graph structure. Note that the precision metrics are persistent regardless of the budget because they pertain to false positives, which we cover in Section 6.3.

Junk Code Insertion. As in Figure 6, inserting junk code negatively affects the overall performance. Gemini [99], Genius [23], Asm2Vec [16], and Trex [76] demonstrate their robustness in recall

Table 4. Performance of six BCSD models against various LLVM-based obfuscations, showing changes in the (P)recision, (R)ecall, and F1 scores compared to the baseline performance without obfuscation. BCF, SUB, FLA, and ALL denote a bogus control flow, instruction substitution, control flow flattening, and a combination of all three, respectively. The bold represents the most significant drop across different models for each obfuscation. All models experience performance drops, with BinShot remaining relatively robust. However, models that rely on dynamic features are particularly affected from CFG perturbation (*e.g.,* Genius 0.673 ↓).

	Genius [23]					Gemini [99]					Asm2Vec [16]				
	Base	BCF	SUB	FLA	ALL	Base	BCF	SUB	FLA	ALL	Base	BCF	SUB	FLA	ALL
P	0.825	0.120 ↓	0.005 ↓	**0.303 ↓**	0.308 ↓	0.865	0.081 ↓	0.000 ↑	0.261 ↓	**0.357 ↓**	0.818	0.009 ↑	**0.007 ↓**	0.019 ↑	0.002 ↓
R	0.952	0.363 ↓	0.002 ↓	**0.673 ↓**	0.680 ↓	0.905	0.328 ↓	0.001 ↓	0.611 ↓	**0.722 ↓**	0.788	0.113 ↓	0.006 ↓	0.193 ↓	0.395 ↓
F1	0.884	0.242 ↓	0.004 ↓	**0.520 ↓**	0.527 ↓	0.885	0.220 ↓	0.000 ↓	0.490 ↓	**0.615 ↓**	0.803	0.059 ↓	0.006 ↓	0.107 ↓	0.272 ↓
	SAFE [70]					Trex [76]					BinShot [3]				
	Base	BCF	SUB	FLA	ALL	Base	BCF	SUB	FLA	ALL	Base	BCF	SUB	FLA	ALL
P	0.850	**0.126 ↓**	0.003 ↓	0.203 ↓	0.299 ↓	0.924	0.009 ↓	0.001 ↑	0.001 ↑	0.042 ↓	0.979	0.012 ↓	0.001 ↓	0.008 ↓	0.021 ↓
R	0.803	**0.364 ↓**	**0.018 ↓**	0.494 ↓	0.566 ↓	0.917	0.217 ↓	0.000 ↑	0.370 ↓	0.647 ↓	0.961	0.211 ↓	0.015 ↓	0.214 ↓	0.402 ↓
F1	0.826	**0.279 ↓**	**0.011 ↓**	0.407 ↓	0.494 ↓	0.920	0.128 ↓	0.001 ↑	0.233 ↓	0.507 ↓	0.970	0.125 ↓	0.008 ↓	0.125 ↓	0.264 ↓

compared to others. Asm2Vec [16] displays the least impact on injecting arbitrary instructions, which we hypothesize a random walk mitigates its negative effect by excluding the injected when walking a possible execution path. It is noteworthy to mention that Gemini and Genius demonstrate performance degradation against junk code insertion (where they do not against semantic NOP implantation) because it introduces a new basic block with a jump instruction.

LLVM-based Obfuscation. Table 4 summarizes the impact of LLVM-based obfuscation on all models. In general, we observe a universal decline in performance where every model shows significant drops when applying all obfuscation techniques. Unlike performance degradation against other semantics-preserving transformation techniques, BinShot [3] exhibits remarkable robustness against LLVM-based obfuscation, which implies that BinShot is capable of better inferring underlying code context. In a similar vein, Asm2Vec [16] shows its relative robustness against all LLVM-based code obfuscations. Notably, Genius [23] and Gemini [99] display a significant reduction in performance against control flow flattening and bogus control flow, which indicates that a CFG perturbation method is effective.

6.3 FP-triggering Perturbation

We choose a target function from a binary, aiming to construct a series of FP-triggering instructions that mislead a model toward a false positive (Section 4.2.1). The left side of Table 5 illustrates the results of our perturbation attack against BinShot [3]. Even with the minimum transformation budget of 20 instructions, we observe that the attack success rate (ASR) reaches to 98.3%. As the budget increases (up to 100 instructions), ASR has been close to almost 100%. Moreover, we investigate the average size of FP-triggering instructions for a successful attack, which is approximately 57.58 bytes ($\leq$ 50 bytes in most cases) or 14.75 instructions on average within the pre-defined transformation budget (Section 3). Figure 6.3 depicts the positive relationship, in general, between FP-triggering code size and function size.

6.4 Transferability of FP-trigger

This section investigates the transferability of FP-triggering perturbations (originally crafted to target the BinShot [3] model) when applied to other BCSD models. We use 10,366 functions from 300 SPEC2006 binaries (within a defined transformation budget), aiming to mislead the models into judging that a target function from Table 5 is similar to a given (victim) function, despite their dissimilarity. The right side of Table 5 reports the accuracy of other BCSD models under this

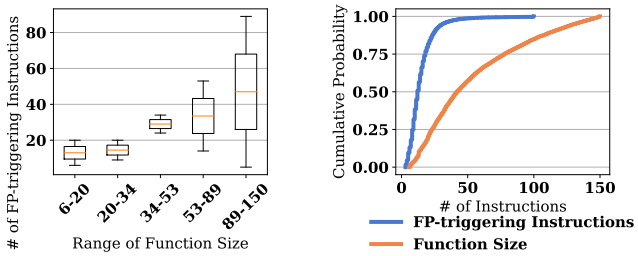


Fig. 7. The relation between the number of FP-triggering instructions and function size for successful attacks in the FP-triggering perturbation attack. The box plot (left) illustrates that the number of FP-triggering instructions generally increases with function size, while the cumulative distribution function (right) indicates that most FP-triggering instructions involve fewer than 50 instructions.

Table 5. The attack success rate (ASR) of the FP-triggering perturbation attack (left) and the accuracy score results of our transferability analysis (right). The attack aims to trigger false positives between the ten functions (victim functions) and 10,366 dissimilar functions. Bold numbers highlight the best attack results.

Binary	Function	Maximum Budget / ASR (↑)					Target Model / Accuracy (↓)				
		20	40	60	80	100	Genius	Gemini	Asm2Vec	SAFE	Trex
bzip2	BZ2_bzBuffToBuffDecompress	0.850	0.981	0.994	0.998	**0.999**	0.988	0.996	0.979	0.974	**0.683**
gcc	optimize_inline_calls	0.944	0.994	0.998	**0.999**	**0.999**	0.985	0.992	**0.557**	0.992	0.917
gzip	zip	0.900	0.986	0.995	0.998	**0.999**	**0.942**	0.995	0.979	0.969	0.997
lighttpd	network_write_file_chunk_sendfile	0.671	0.971	0.992	0.995	**0.996**	0.992	0.989	0.893	0.948	**0.880**
lvm	lock_vol	0.787	0.971	0.986	0.988	**0.989**	0.975	0.978	0.996	0.967	**0.869**
md5sum	md5_stream	0.879	0.985	0.992	0.995	**0.996**	0.974	0.953	**0.878**	0.931	0.898
miniweb	_mwProcessPost	0.669	0.951	0.983	0.990	**0.996**	0.989	0.993	0.867	0.973	**0.788**
nginx	ngx_http_create_request	0.611	0.947	0.981	0.989	**0.993**	0.967	0.959	0.863	0.910	**0.773**
putty	new_connection	0.651	0.959	0.992	0.998	**0.999**	0.945	0.993	0.992	0.987	**0.802**
vsftpd	vsf_sysutil_sendfile	0.983	**1.000**	**1.000**	**1.000**	**1.000**	0.977	**0.764**	0.963	0.981	0.986

perturbation attack. Our findings indicate that the attack exhibits varying degrees of transferability across models: while it transfers effectively to certain architectures, others are relatively more resilient. In particular, transferability is stronger among models that share similar instruction-level or token-level embeddings, while CFG- or graph-centric GNN-based models demonstrate comparatively more robustness to such transfer. Notably, Trex [76], a BERT-based architecture like BinShot, shows strong transferability between similar model types.

6.5 FP-Trigger Analysis with Explainable AI

In this section, we extend our evaluation by analyzing model behavior using explainable AI (XAI) techniques. We examine the FP-triggering perturbation input on BinShot [3] through SHAP [63] and saliency analysis [68].

FP-Trigger Case Analysis with SHAP. Shapley values, rooted in cooperative game theory, quantify the contribution of each token to a model's prediction (*i.e.,* similarity score by BinShot). Negative values indicate a token's influence toward a dissimilar prediction, while positive values indicate influence toward a similar prediction. Figure 8 (left) illustrates the distribution of Shapley values before and after applying the FP-triggering perturbation. Initially, the values span a broad range, but after perturbation, most values converge near zero, leading to a false positive. This concentration around zero suggests the model's decision lies close to the decision boundary. We hypothesize that this is because our sampling algorithm terminates once a false positive is detected.

FP-Trigger Case Analysis with a Saliency Value. As illustrated on the right side in Figure 8, we assess instruction-level importance using saliency scores: computing the gradient of the loss

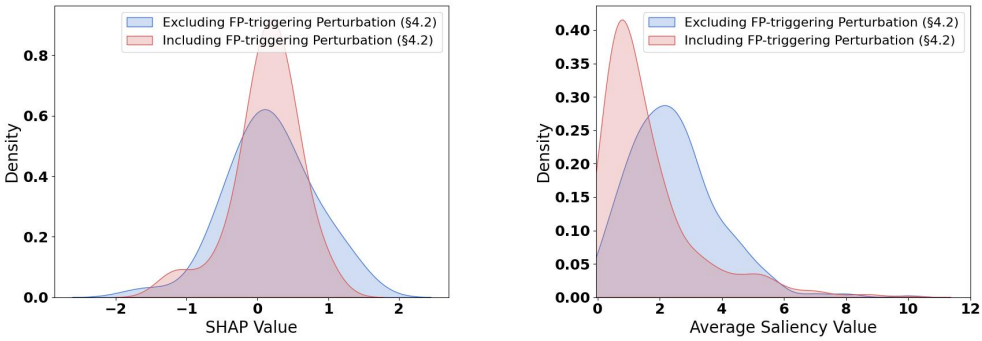


Fig. 8. Kernel density estimates of Shapley (left) and saliency (right) values for 1,000 randomly selected functions from 10 baseline binaries (100 functions per binary). The blue distribution represents the model's original behavior (excluding FP-triggering instructions), while the red distribution indicates its response after introducing the perturbation. On the left, the Shapley values shift upward toward zero, indicating the model increasingly perceives the two functions as similar. On the right, the shift toward lower saliency values suggests that the model's attention to crucial features has diminished under the perturbation.

with respect to each instruction's embedding and taking its norm as the saliency score. A higher gradient norm indicates greater importance, as small changes to the embedding would significantly affect BinShot's output. Conversely, lower saliency suggests that the model places less emphasis on those instructions. As the red curve indicates, the introduction of FP-triggering instructions causes a noticeable shift toward lower saliency values, indicating that the model becomes less attentive to key features. This aligns with the model being misled into perceiving the two functions as more similar than they actually are.

In-depth Analysis on FP-triggering Perturbation. Figure 9 illustrates the Shapley values associated with our FP-triggering perturbation. Instructions are color-coded by their Shapley values: red indicates a contribution toward the "similar" label, while blue represents a contribution toward "dissimilar". The figure showcases a representative target–victim pair: `vsf_sysutil_sendfile` from `vsftpd` and `etarldouble` from `gcc`. In this case, the victim function is perturbed to resemble the target, causing the model to return a false positive. We present the Shapley attributions both before and after the perturbation, using a normalized instruction representation [48], rather than raw x86 assembly, which follows the pre-processing in BinShot [3]. The increase in red-highlighted instructions after perturbation confirms the model's perception has been shifted, leading to the false positive.

6.6 Summary of Findings

With extensive experiments on evaluating the robustness of (ML-based) BCSD models against various semantics-preserving transformations, we discover a few noteworthy findings. First, the robustness of a model heavily relies on the pipeline of building the model from code pre-processing (*e.g.,* normalization, embedding means), model architecture (*e.g.,* Transformer, BERT), to selecting internal features (*e.g.,* static, dynamic). Static features encompass the embeddings and the order of an instruction (*i.e.,* opcode and operands) whereas dynamic features include a CFG, a call graph, and execution traces. These factors collectively form the ingredients when training a model, allowing each vector to represent varying code semantic features. For example, code transformations based on a memory layout such as ICT and inter-BBR thwart BinShot [3] that lacks the information. Likewise, code obfuscations such as control flow flattening and bogus control flow

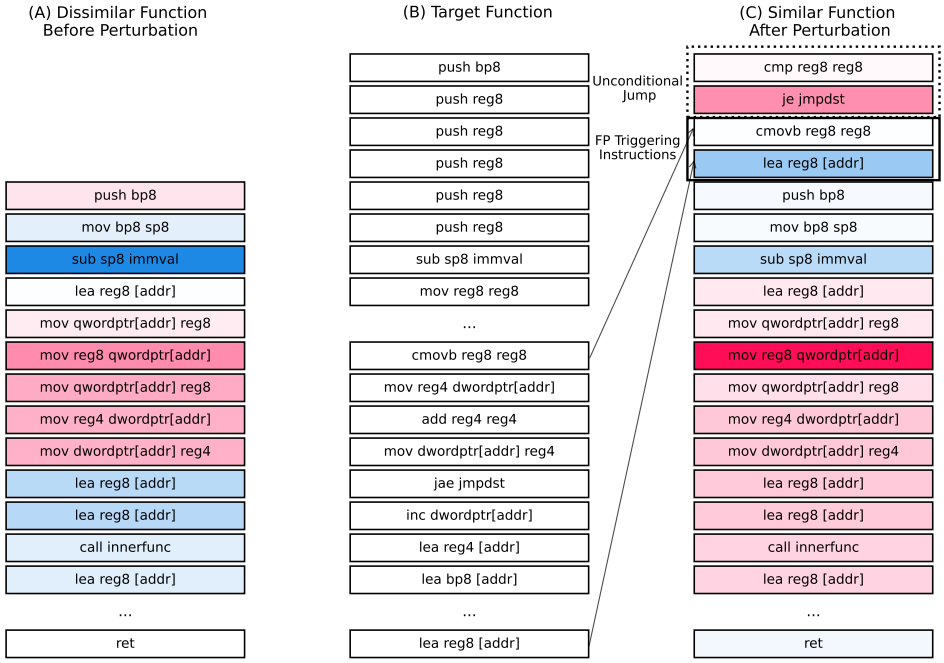


Fig. 9. Visualization of instruction-level Shapley values for a victim function (`etarldouble` from `gcc`) and before (A) and after (C) an FP-triggering perturbation. The perturbation modifies the victim function (A ⟶ C) to resemble the target function (`vsf_sysutil_sendfile` from `vsftpd`, B) by inserting FP-triggering instructions sampled using our greedy sampling algorithm (Section 4.2.2) from the target function (solid box), which preserves the original code semantics via an unconditional jump (dotted box). The red and blue colors highlight instructions contributing to "similar" and "dissimilar" classifications, respectively. The increase in red-labeled instructions after perturbation explains how the false positive has been derived; *e.g.,* the four blue instructions in the middle (three `lea reg8 [addr]` and one `call innerfunc` instructions) have been changed to red ones. Note that we simplify the functions for brevity.

defeat Gemini [99] and Genius [23] that depend on the CFG, which BinShot has shown to be robust against. This is even clear when looking into the differences in performance degradation between implanting a semantic NOP and inserting a junk code for Gemini and Genius. Namely, they are robust against the former because it maintains a CFG; however, less robust against the latter because of the alteration of the graph. Interestingly, in the case of Asm2Vec [16], its random walk exhibits resiliency against graph manipulation to some extent. It is worth mentioning that quantitatively measuring a model's resiliency is difficult; however, we can indirectly deduce it throughout its outputs. Second, an adversary's capability would be also bounded by a transformation budget due to the target model's property in terms of input length and the expressivity of semantically equivalent machine instructions. Consequently, our work alludes to considerations for designing a reliable and robust BCSD model (but applied to any ML-based model), which takes machine instructions as input. Third, unlike semantics-preserving transformations, FP-triggering perturbations demand a carefully designed sampling strategy, similar to adversarial attacks in deep learning. We discover that FP-triggering perturbations are highly effective, even with minimal changes, and exhibit strong transferability, particularly to models with similar architectural characteristics. Our further

investigations using XAI methods reveal that these perturbations disrupt the model's ability to identify essential tokens, thereby leading to false positive predictions.

7 Related Work

Attacks against Deep Learning Models. Szegedy *et al.* [90] are the first to demonstrate that deep learning models are vulnerable to adversarial attacks. Following the exploration, the fast gradient sign method [26] introduces a technique for generating adversarial noise by perturbing inputs in the direction of the gradient to maximize prediction error, raising awareness about the susceptibility of deep learning models to such inputs. The projected gradient descent attack [67] extends the fast gradient sign method by iteratively applying gradient updates and projecting the perturbation back into a constrained region to maintain it within a specified norm bound. Carlini *et al.* [10] further expand the attack landscape by introducing optimization-based attacks under different distance metrics (*i.e.,* L_0, L_2, L_∞), enabling fine-grained control over perturbation similarity. Although adversarial attacks initially focus on image classifiers, the rise of transformer-based models has led to similar vulnerabilities in NLP. TextFooler [42] and BERT-attack [52], for instance, generate adversarial examples by replacing words with synonyms, often causing models to misclassify inputs such as sentiment or intent, which inspires our work. Meanwhile, Zou *et al.* [101] introduce a universal adversarial suffix that can manipulate LLMs into generating harmful content when appended to prompts.

Perturbation Attacks against Executable Binaries. Lucas *et al.* [62] propose an adversarial attack that perturbs malicious PE executables via binary instrumentation. Their approach combines in-place code transformation with a modified displacement that integrates semantic NOPs to evade detection. Similarly, our experiments incorporate both in-place code transformation and semantic NOP implantation as part of our adversarial variant generation strategy.

Attacks against BCSD Models. FuncFooler [41] demonstrates a practical black-box attack on learning-based BCSD models, including SAFE [70], Asm2Vec [16], and jTrans [97]. Capozzi *et al.* [8] extend this line of work by distinguishing between targeted attacks (*i.e.,* designed to make dissimilar functions appear similar) and untargeted attacks (*i.e.,* aiming to make similar functions appear dissimilar) across three models: Gemini [99], Genius [23], and SAFE [70]. While both FuncFooler and Capozzi *et al.*[1] share the goal of evaluating BCSD model robustness, their approaches are limited to a single semantics-preserving transformation implemented via inline assembly for source code perturbation. In contrast, our work explores eight distinct semantics-preserving transformations, most applied at the binary level, without modifying any source code.

8 Discussion and Limitations

Formal Verification. The in-place code transformation techniques proposed by Pappas *et al.* [75] are not formally verified. As a result, we do not guarantee full correctness across all code instances, although we partially validate that each transformation behaves as intended.

Variant Exploration. Moreover, both inter-basic-block reordering and in-place code transformation are non-deterministic, which may impact the observed robustness of each model. While exhaustively generating all possible variants is infeasible, our study still yields meaningful insights into model behavior. We fix the insertion point of semantic NOPs and junk code at the function entry to isolate the impact of sequence length. Although such sequences could be placed elsewhere, placing them at the beginning exploits the positional bias of language models, which prioritize information appearing early in long contexts [58].

[1]Unfortunately, a direct comparison was not possible, as neither approach was publicly available at the time of writing.

XAI Fidelity. Lastly, while we analyze FP-triggering transformations using SHAP [63] and saliency [68], prior work has shown that XAI techniques can yield conflicting results [31]. A comprehensive fidelity analysis of XAI methods is beyond the scope of this study.

Other Binary Code Obfuscations. Beyond the code obfuscation techniques in our experiments, many other methods exist, such as inserting trampolines (*i.e.,* jump tables) or applying string and data obfuscation. Additionally, our evaluation does not consider advanced protections like encryption, code packing, and compression (*e.g.,* self-extracting or polymorphic code), nor commercial obfuscation tools such as Themida [74] or VMProtect [87]. We leave the exploration of additional obfuscation techniques as part of future work.

Defenses against Adversarial Code Transformations. Adversarial samples are deliberately crafted to mislead an ML model and induce errors. Data augmentation with known transformation suites can be helpful to increase the robustness of an ML-based BCSD model so that the model can learn varying code representations from variants. Another direction is training that combines static features (*e.g.,* CFGs, call graphs, API-call graphs, data flows) with dynamic features (*e.g.,* API traces, syscall traces, behavior traces) can assist in the model's final decision. We leave a systematic study of adversarial training with semantics-preserving transformations to our future work.

9 Conclusion

Recent advancements in deep learning offer promising opportunities to enhance binary analysis for security applications. Despite the surge of various models available, their resiliency against adversarial samples has not been studied in depth. In this paper, we focus on evaluating the robustness of six state-of-the-art binary code similarity detection models with eight semantics-preserving code transformations. Our major finding highlights that model robustness significantly depends on the particular characteristics of the processing pipeline, including code preprocessing, model architecture, and feature selection. Furthermore, the effectiveness of semantics-preserving transformations is limited both by inherent constraints of the model and by the expressive capacity of semantically equivalent instructions within the binary.

Data Availability

We have open-sourced AsmFooler[2] and our adversarial binary dataset for further exploration of model robustness in machine-learning-based binary analysis.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This work was partially supported by grants from the Institute of Information & Communications Technology Planning & Evaluation (IITP), funded by the Korean government (MSIT; Ministry of Science and ICT): Grants No. RS-2024-00337414, and No. RS-2024-00437306. Additional support was provided by the Basic Science Research Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education of the Government of South Korea: Grant No. RS-2025-02293072. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

[2]https://zenodo.org/records/17116512

References

[1] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. 2018. Large-scale and language-oblivious code authorship identification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*.

[2] National Security Agency. 2025. Ghidra. https://ghidra-sre.org/

[3] Sunwoo Ahn, Seonggwan Ahn, Hyungjoon Koo, and Yunheung Paek. 2022. Practical Binary Code Similarity Detection with BERT-based Transferable Similarity Learning. In *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC '22)*.

[4] Tyler Bletsch, Xuxian Jiang, Vince Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Asia Conference on Computer and Communications Security (ASIACCS '11)*.

[5] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting self-mutating malware using control-flow graph matching. In *Proceedings of the Detection of Intrusions and Malware & Vulnerability Assessment: Third International Conference (DIMVA '06)*.

[6] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM SIGSAC Conference on Computer and Communications Security (CCS '07)*.

[7] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing programmers via code stylometry. In *Proceedings of the 24th USENIX Security Symposium (Security '15)*.

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

[9] Capstone. 2025. Capstone The Ultimate Disassembler. https://www.capstone-engine.org/

[10] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P '17)*.

[11] Silvio Cesare, Yang Xiang, and Wanlei Zhou. 2013. Control flow-based malware variant detection. *IEEE Transactions on Dependable and Secure Computing* (2013).

[12] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*.

[13] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting decompiler output with learned variable names and types. In *Proceedings of the 31st USENIX Security Symposium (Security '22)*.

[14] Yizheng Chen, Zhoujie Ding, and David Wagner. 2023. Continuous learning for android malware detection. In *Proceedings of the 32nd USENIX Security Symposium (Security '23)*.

[15] Standard Performance Evaluation Corporation. 2025. SPEC CPU 2006. https://www.spec.org/cpu2006/

[16] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P '19)*.

[17] Evan Downing, Yisroel Mirsky, Kyuhong Park, and Wenke Lee. 2021. DeepReflect: Discovering malicious functionality through binary reconstruction. In *Proceedings of the 30th USENIX Security Symposium (Security '21)*.

[18] Yufei Du, Omar Alrawi, Kevin Snow, Manos Antonakakis, and Fabian Monrose. 2023. Improving Security Tasks Using Compiler Provenance Information Recovered At the Binary-Level. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*.

[19] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *Proceedings of the 23rd USENIX Security Symposium (Security '14)*.

[20] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. Discovre: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the Network and Distributed System Security Symposium 2016 (NDSS '16)*.

[21] Chris Evans. 2025. vsftpd. https://security.appspot.com/vsftpd.html

[22] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. 2017. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 12th ACM Asia Conference on Computer and Communications Security (ASIACCS '17)*.

[23] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*.

[24] Free Software Foundation. 2025. GNU Operating System. https://www.gnu.org/

[25] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*.
[26] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *3rd International Conference on Learning Representations (ICLR '15)*.
[27] Google. 2025. American Fuzzy Lop. https://github.com/google/AFL
[28] gzip. 2025. The gzip home page. https://www.gzip.org/
[29] Irfan Ul Haq and Juan Caballero. 2021. A survey of binary code similarity. *Comput. Surveys* (2021).
[30] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. 2024. Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection. In *Proceedings of the 33rd USENIX Security Symposium (Security '24)*.
[31] Jinwen He, Kai Chen, Guozhu Meng, Jiangshan Zhang, and Congyi Li. 2023. Good-looking but Lacking Faithfulness: Understanding Local Explanation Methods through Trend-based Testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*.
[32] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*.
[33] Xu He, Shu Wang, Yunlong Xing, Pengbin Feng, Haining Wang, Qi Li, Songqing Chen, and Kun Sun. 2022. Binprov: Binary code provenance identification without disassembly. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '22)*.
[34] Hex-rays. 2025. IDA Pro. https://hex-rays.com/ida-pro
[35] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. 2009. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM SIGSAC Conference on Computer and Communications Security (CCS '09)*.
[36] Xin Hu, Kang G Shin, Sandeep Bhatkar, and Kent Griffin. 2013. MutantX-S: Scalable malware clustering based on static features. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC '13)*.
[37] He Huang, Amr Youssef, and Mourad Debbabi. 2017. Binsequence: Fast, accurate and scalable binary code reuse detection. In *Proceedings of the 12th ACM Asia Conference on Computer and Communications Security (ASIACCS '17)*.
[38] Stanley Huang. 2025. MiniWeb. https://miniweb.sourceforge.net/
[39] Jiyong Jang, Maverick Woo, and David Brumley. 2013. Towards automatic software lineage inference. In *Proceedings of the 22nd USENIX Security Symposium (Security '13)*.
[40] Ang Jia, Ming Fan, Xi Xu, Wuxia Jin, Haijun Wang, and Ting Liu. 2024. Cross-Inlining Binary Function Similarity Detection. In *Proceedings of the 46th International Conference on Software Engineering (ICSE '24)*.
[41] 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).
[42] 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 (AAAI '20)*.
[43] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM–software protection for the masses. In *Proceedings of the 1st International Workshop on Software Protection (SPRO '15)*.
[44] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2022. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering* (2022).
[45] Hyunjin Kim, Jinyeong Bak, Kyunghyun Cho, and Hyungjoon Koo. 2023. A transformer-based function symbol name inference model from an assembly language for binary reversing. In *Proceedings of the 18th ACM Asia Conference on Computer and Communications Security (ASIACCS '23)*.
[46] Taeguen Kim, Yeoreum Lee, Boojoong Kang, and Eulgyu Im. 2019. Binary executable file similarity calculation using function matching. *The Journal of Supercomputing* (2019).
[47] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. 2018. Compiler-Assisted Code Randomization. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P '18)*.
[48] Hyungjoon Koo, Soyeon Park, Daejin Choi, and Taesoo Kim. 2023. Binary Code Representation With Well-Balanced Instruction Normalization. *IEEE Access* (2023).
[49] Hyungjoon Koo and Michalis Polychronakis. 2016. Juggling the gadgets: Binary-level code randomization using instruction displacement. In *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS '16)*.
[50] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. 2006. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th International Symposium on Research*

in Attacks, Intrusions and Defenses (RAID '06).

[51] Yongjun Lee, Hyun Kwon, Sang-Hoon Choi, Seung-Ho Lim, Sung Hoon Baek, and Ki-Woong Park. 2019. Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with CNN. *Applied Sciences* 9, 19 (2019), 4086.

[52] Linyang Li, Ruotian Ma, Qipeng Guo, Xiangyang Xue, and Xipeng Qiu. 2020. BERT-ATTACK: Adversarial Attack against BERT using BERT. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP '20).*

[53] Zhen Li, Guenevere Chen, Chen Chen, Yayi Zou, and Shouhuai Xu. 2022. Ropgen: Towards robust code authorship attribution via automatic coding style transformation. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22).*

[54] LIEF. 2025. LIEF: Library to Instrument Executable Formats. https://lief.re/

[55] lighttpd. 2025. Lighttpd - fly light. https://www.lighttpd.net/

[56] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. 2012. Lines of malicious code: Insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12).*

[57] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αdiff: cross-version binary code similarity detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18).*

[58] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* (2024).

[59] LLVM. 2025. The LLVM Compiler Infrastructure. https://llvm.org/

[60] LLVM. 2025. LLVM Discussion Forums. https://discourse.llvm.org/

[61] Keane Lucas, Samruddhi Pai, Weiran Lin, Lujo Bauer, Michael K Reiter, and Mahmood Sharif. 2023. Adversarial training for Raw-Binary malware classifiers. In *Proceedings of the 32nd USENIX Security Symposium (Security '23).*

[62] 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 16th ACM Asia Conference on Computer and Communications Security (ASIACCS '21).*

[63] Scott M. Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS '17).*

[64] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14).*

[65] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search. In *Proceedings of the Network and Distributed System Security Symposium 2023 (NDSS '23).*

[66] lvmteam. 2025. LVM2. https://github.com/lvmteam/lvm2

[67] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *6th International Conference on Learning Representations (ICLR '18).*

[68] Itzik Malkiel, Dvir Ginzburg, Oren Barkan, Avi Caciularu, Jonathan Weill, and Noam Koenigstein. 2022. Interpreting BERT-based Text Similarity via Activation and Saliency Maps. In *Proceedings of the ACM Web Conference 2022 (WWW '22).*

[69] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How machine learning is solving the binary function similarity problem. In *Proceedings of the 31st USENIX Security Symposium (Security '22).*

[70] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference (DIMVA '19).*

[71] Microsoft. 2025. PE Format. https://learn.microsoft.com/en-us/windows/win32/debug/pe-format

[72] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS '13).*

[73] nginx. 2025. nginx. https://nginx.org/

[74] oreans.com. 2025. Thermida Overview. https://www.oreans.com/Themida.php

[75] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proceedings of the 2012 IEEE Symposium on Security*

and Privacy (S&P '12).
[76] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2022. Trex: Learning execution semantics from micro-traces for binary similarity. *IEEE Transactions on Software Engineering* (2022).
[77] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P '15)*.
[78] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*.
[79] Marco Prandini and Marco Ramilli. 2012. Return-oriented programming. *IEEE Security & Privacy* (2012).
[80] NLTK Project. 2025. Natural Language Toolkit. https://www.nltk.org/
[81] putty. 2025. PuTTY: a free SSH and Telnet client. https://www.chiark.greenend.org.uk/~sgtatham/putty/
[82] Zhenxiao Qi, Yu Qu, and Heng Yin. 2022. LogicMEM: Automatic Profile Generation for Binary-Only Memory Forensics via Logic Inference. In *Proceedings of the Network and Distributed System Security Symposium 2022 (NDSS '22)*.
[83] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium 2017 (NDSS '17)*.
[84] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security* (2012).
[85] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P '16)*.
[86] simoll. 2025. Remove BinaryOperator::CreateFNeg. https://reviews.llvm.org/D75130
[87] VMProtect Software. 2025. Advanced code security made simple and reliable. https://vmpsoft.com/
[88] Minami Someya, Yuhei Otsubo, and Akira Otsuka. 2023. FCGAT: Interpretable malware classification method using function call graph and attention mechanism. In *Proceedings of the Network and Distributed System Security Symposium 2023 (NDSS '23)*.
[89] Portable Formats Specification. 1993. Tool Interface Standard (TIS) Portable Formats Specification. (1993).
[90] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations (ICLR '14)*.
[91] GCC team. 2025. GCC, the GNU Compiler Collection. https://gcc.gnu.org/
[92] The UPX Team. 2025. the Ultimate Packer for eXecutables. https://upx.github.io/
[93] tigress. 2025. the tigress c obfuscator. https://tigress.wtf/
[94] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS '17)*.
[95] Vector35. 2025. Binary Ninja. https://binary.ninja/
[96] VirusTotal. 2024. Welcome to YARA's documentation! https://yara.readthedocs.io/en/stable/index.html
[97] 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 (ISSTA '22)*.
[98] Jialai Wang, Chao Zhang, Longfei Chen, Yi Rong, Yuxiao Wu, Hao Wang, Wende Tan, Qi Li, and Zongpeng Li. 2024. Improving ML-based Binary Function Similarity Detection by Assessing and Deprioritizing Control Flow Graph Features. In *Proceedings of the 33rd USENIX Security Symposium (Security '24)*.
[99] 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 (CCS '17)*.
[100] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI '20)*.
[101] Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J Zico Kolter, and Matt Fredrikson. 2023. Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043* (2023).
[102] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In *Proceedings of the Network and Distributed System Security Symposium 2019 (NDSS '19)*.

Received 2025-09-11; accepted 2025-12-22