Shield Broken: Black-Box Adversarial Attacks on LLM-Based Vulnerability Detectors

Yuan Jiang, Shan Huang, Christoph Treude, Xiaohong Su and Tiantian Wang

***Abstract*—Vulnerability detection is critical for ensuring software security. Although deep learning (DL) methods, particularly those employing large language models (LLMs), have shown strong performance in automating vulnerability identification, they remain susceptible to adversarial examples, which are carefully crafted inputs with subtle perturbations designed to evade detection. Existing adversarial attack methods often require access to model architectures or confidence scores, making them impractical for real-world black-box systems. In this paper, we propose SVulAttack, a novel label-only adversarial attack framework targeting LLM-based vulnerability detectors. Our key innovation lies in a similarity-based strategy that estimates statement importance and model confidence, thereby enabling more effective selection of semantic-preserving code perturbations. SVulAttack combines this strategy with a transformation component and a search component, based on either greedy or genetic algorithms, to effectively identify and apply optimal combinations of transformations. We evaluate SVulAttack on open-source models (LineVul, StagedVulBERT, Code Llama, Deepseek-Coder) and closed-source models (GPT-5 nano, GPT-4o, GPT-4o-mini, Claude Sonnet 4). Results show that SVulAttack significantly outperforms existing label-only black-box attack methods. For example, against LineVul, our method with genetic algorithm achieves an attack success rate of 49.0%, improving over DIP and CODA by 150.0% and 240.3%, respectively.**

***Index Terms*—Adversarial Examples, Vulnerability Detection, Black-box Attack, Optimization Algorithm**

I. INTRODUCTION

SOFTWARE vulnerabilities are weaknesses in software that attackers can exploit to compromise confidentiality, integrity, or availability, or to cause other undesired impacts [1], [2]. Mitigating such vulnerabilities (e.g., via code fixes) improves software security; however, effective mitigation depends on accurate detection, so reliable automated vulnerability detection is essential [3]. Recently, LLM-based approaches have shown great promise in identifying real-world security vulnerabilities [4], [5]. For example, Google Project Zero's AI agent, Big Sleep, recently discovered a stack buffer underflow in SQLite [6], marking the first public case of AI uncovering an exploitable memory-safety vulnerability in widely used software. This milestone underscores the growing practicality of AI in automated vulnerability detection.

However, despite these promising advancements, DL-based vulnerability detection models remain susceptible to adversarial threats [7]. In particular, they are vulnerable to adversarial examples, which are malicious inputs with small, carefully crafted perturbations that can evade detection [8]–[12]. Such evasion attacks may cause vulnerable code to be misclassified as non-vulnerable, thereby undermining the reliability of detection systems and threatening overall software security. As the number of software vulnerabilities continues to rise [13], it is critical to evaluate and strengthen the robustness of DL-based detection techniques against such attacks.

Existing adversarial attack methods are predominantly white-box or score-based black-box approaches, which require detailed knowledge of a model's architecture, parameters, or prediction confidence scores [14]. However, such assumptions are often unrealistic in practice. For example, AIBugHunter [15] integrates the open-source model LineVul [16] as a Visual Studio Code plugin, where the backend service returns only the final decision (i.e., vulnerable or non-vulnerable) without exposing probability distributions. Similarly, closed-source LLMs are commonly deployed in user-facing products that typically do not provide raw confidence scores [17]. These limitations highlight the necessity of investigating label-only attacks, also known as decision-based attacks [18], which operate solely on hard-label outputs.

Currently, the state-of-the-art label-only attack method for code models, DIP [19], demonstrates strong performance and even outperforms several white-box approaches (e.g., MHM [20] and ALERT [21]) across various benchmarks. However, DIP relies on an external pre-trained surrogate model (e.g., CodeBERT) to identify perturbation positions and generate candidate dead code, which limits its effectiveness when the surrogate and target models differ significantly. Moreover, since DIP employs only dead code insertion as its perturbation strategy, its effectiveness is limited, particularly when attacking complex vulnerable programs. These limitations underscore the need for more effective label-only attack methods that operate independently of external models.

To address these challenges, we introduce SVulAttack, a novel label-only adversarial attack designed to evade LLM-based vulnerability detectors and assess their robustness. The core innovation of our method lies in estimating the importance of each code statement and the model's prediction confidence without requiring access to internal parameters or confidence scores. Specifically, we leverage the fact that LLM-based models do not achieve perfect accuracy on all samples. By identifying reference samples that are similar to the target vulnerable code but are classified differently by the model (e.g., as non-vulnerable), we compute similarity metrics to estimate the importance of code statements and predict how perturbations may affect the model's prediction. This

Y. Jiang (jiangyuan@hit.edu.cn, yuanjiang@smu.edu.sg), H. Shan (2022110145@stu.hit.edu.cn), X. Su (sxh@hit.edu.cn), and T. Wang (wangtiantian@hit.edu.cn), are with the School of Computer Science and Technology, Harbin Institute of Technology, Harbin, Heilongjiang, 150001.

C. Treude is with the School of Computing and Information Systems, Singapore Management University, Singapore. E-mail: ctreude@smu.edu.sg

```
1 void process_input() {
2     char buf[10] = "";
3     printf("Buf value: %s", buf);
4     scanf("%s", buf);
5     printf("Input: %s\n", buf);
6 }
```

```
1 void process_input() {
2     char buf[10] = "";
3     printf("Buf value: %s", buf);
4     scanf("%9s", buf);
5     printf("Input: %s\n", buf);
6 }
```

Fig. 1. A vulnerable sample (left) and a semantically similar non-vulnerable sample (right)

similarity-based estimation enables effective guidance of transformation selection in a purely black-box setting, enhancing attack effectiveness without relying on any surrogate models.

The proposed method consists of two main components: transformation and search. The *transformation* component applies a diverse set of code perturbation techniques to generate perturbed samples that maintain semantic equivalence with the original program. These transformations introduce subtle changes that are syntactically correct and semantically equivalent, thereby offering a more expressive and flexible perturbation space compared to prior methods [20]–[22].

The *search* component is responsible for identifying effective combinations of transformations. We employ either a greedy search or a genetic algorithm to explore the transformation space, aiming to find an optimal sequence that maximizes the likelihood of evading detection. Guided by our similarity-based estimation of statement importance and model prediction confidence, the search algorithm prioritizes transformations that are more likely to deceive the detector. This process reduces the search space and improves the efficiency of generating adversarial examples, leading to a higher attack success rate than existing methods [19], [23].

Experimental results demonstrate that our approach, SVulAttack, substantially outperforms state-of-the-art baselines. For example, on LineVul, SVulAttack achieves a 49.0% attack success rate with the Genetic Algorithm, exceeding DIP (19.6%) and CODA (14.4%) by 150.0% and 240.3%, respectively. Ablation studies verify the contribution of each transformation and search strategy, while further analysis shows that allowing more modifications improves attack performance to a certain extent, albeit with increased query overhead.

The contributions of this paper are as follows:

- We propose a novel label-only adversarial attack method that leverages multiple semantic-preserving perturbation strategies and search algorithms to generate adversarial examples against vulnerability detection models.
- We introduce a similarity-based strategy to estimate statement importance and model confidence in a black-box setting, effectively guiding the perturbation process without relying on surrogate models.
- We perform extensive evaluations demonstrating the effectiveness of our method and its practical value for assessing the robustness of vulnerability detection models.

II. Background and Motivation

A. Motivating Example

To motivate our approach, consider the code snippets in Figure 1. The left snippet is vulnerable due to unrestricted input in `scanf("%s", buf)`, enabling buffer overflow, while the right one is safe using `"%9s"` to limit input length.

A state-of-the-art vulnerability detector, LineVul [16], correctly classifies these examples. In this paper, our goal is to craft adversarial examples that induce misclassification of vulnerable code as non-vulnerable in a black-box setting.

Our evasion attack is motivated by the intuition that transforming a vulnerable code snippet to resemble a non-vulnerable one can push it across the model's decision boundary, leading to misclassification. This intuition aligns with prior work in adversarial machine learning [24]. Similar studies in vulnerability detection [25], [26] further show that when a sample becomes closer to non-vulnerable references than to vulnerable ones, it is more likely to be predicted as non-vulnerable, thereby reinforcing our intuition.

Building on this intuition, we aim to perturb vulnerable code so that it aligns more closely with non-vulnerable references. However, not all portions of a vulnerable program are equally influential, and not every perturbation is equally effective [20], [27], [28]. To prioritize impactful modifications, we first identify which statements are the best candidates for perturbation.

This can be achieved by measuring each statement's contribution to the difference between the vulnerable sample and a selected non-vulnerable reference. We iteratively remove each statement from the vulnerable code and evaluate how much the similarity to the non-vulnerable code decreases. Statements that produce the smallest drop in similarity are considered less critical to the code's inherently vulnerable representation. These positions are prime candidates for perturbation, as modifying them is more likely to bring the vulnerable code closer to the non-vulnerable sample's representation.

After identifying suitable modification points, we apply targeted perturbations by inserting tokens or code patterns present in non-vulnerable references but absent from the original code. For instance, we insert dead code like `printf("%9s", buf);` at Line 3 of the vulnerable snippet in Figure 1, where the `%9s` format specifier reflects a distinctive pattern derived from non-vulnerable samples. If the modification increases the similarity score with the non-vulnerable reference, we keep it; otherwise, we discard it and attempt another perturbation.

By iteratively selecting the most promising perturbation locations and incorporating patterns from non-vulnerable code, our approach guides the transformation process in a targeted manner, ensuring that each modification meaningfully contributes to evading detection. Following this strategy, we successfully mislead the LineVul model into classifying a perturbed vulnerable sample as non-vulnerable with only a single minor perturbation, as illustrated in Figure 2. This exposes a critical limitation: current vulnerability detectors, despite strong performance, lack robust generalization, highlighting the urgent need to better understand the robustness and security of DL- and LLM-based detection models.

B. Target Vulnerability Detection Method

This paper targets state-of-the-art LLM-based vulnerability detectors, including open-source and closed-source models.

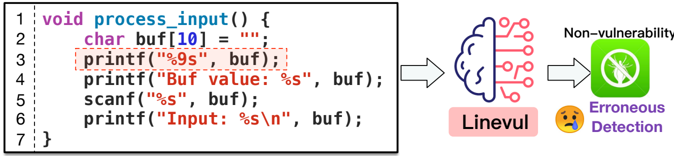


Fig. 2. Adversarial sample misclassified as non-vulnerable by LineVul

1) Open-Source Models: LineVul [16] adopts a pre-training and fine-tuning paradigm based on Pre-trained Code Language (PCL) models. It first learns general-purpose code representations from large-scale open-source repositories via self-supervised learning, then fine-tunes them on labeled vulnerability datasets. To handle the open vocabulary issue caused by user-defined identifiers, LineVul applies Byte Pair Encoding (BPE) [29] for code tokenization.

StagedVulBERT [4] enhances detection by introducing hierarchical code representations and improved long-sequence processing. Its core component, CodeBERT-HLS, extends CodeBERT with hierarchical learning stages to capture both fine-grained and high-level features. In addition, CodeBERT-HLS is optimized for long token sequences, making it well-suited for real-world scenarios where vulnerabilities span multiple interdependent and distant code statements [3].

In addition to the above transformer-based vulnerability detection models, several general-purpose code LLMs can also be adapted for vulnerability detection through fine-tuning. To evaluate the generalizability of our attack method, we further consider two widely used models, Code Llama 7B [30] and Deepseek-Coder 7B [31], as victim models in our experiments.

2) Closed-Source LLMs: Recent studies have demonstrated the potential of closed-source LLMs in vulnerability detection [32], [33]. Therefore, in our evaluation, we include three OpenAI models, i.e., GPT-4o, its lightweight variant GPT-4o-mini, and GPT-5 nano, where GPT-4o extends the GPT-4 family with improved efficiency and multimodal capability, GPT-4o-mini offers a smaller and faster alternative, and GPT-5 nano is the most cost-efficient member of the GPT-5 series with strong code comprehension abilities. We also consider Claude Sonnet 4, a state-of-the-art model from Anthropic designed for advanced reasoning and code analysis tasks.

For vulnerability detection, we employ these models in a zero-shot setting, where a designed prompt guides the model to determine whether a given code snippet is vulnerable. Following the approach of [32], we adopt the prompt below to instruct the model to analyze C/C++ functions for vulnerabilities.

> Predict whether the C/C++ function below is vulnerable. Strictly return 1 for a vulnerable function or 0 for a non-vulnerable function without further explanation.

III. Threat Model and SVulAttack Overview

A. Threat Model

This section defines the threat model, specifying the adversary's goals, knowledge, and capabilities [34], reflecting realistic attack scenarios observed in adversarial settings involving LLM-based vulnerability detectors.

Adversary's Goal. The adversary's primary goal is to craft adversarial vulnerable code that evades detection by LLM-based vulnerability detectors via minimal and imperceptible modifications. Specifically, the adversarial code must: (1) preserve the original functionality and behavior; (2) comply with programming language syntax to ensure compilability and executability; and (3) remain minimally altered to avoid detection by manual or automated analysis.

Adversary's Knowledge. We assume a black-box setting in which the adversary lacks access to the model's architecture, parameters, training data, gradients, or prediction scores, and can only observe the final binary labels (vulnerable or non-vulnerable). The adversary may interact with the model via queries and leverage open-source code (e.g., from NVD [35]) to construct reference samples for guiding perturbations.

Adversary's Capabilities. The adversary can modify input code at both the token (e.g., variable renaming, constant replacement) and statement levels (e.g., inserting redundant code). In addition, they can apply optimization strategies, such as greedy search or genetic algorithms, to combine perturbations and maximize attack success.

B. Overview of SVulAttack

Given a vulnerability detection model F, the adversary's goal is to deceive F into misclassifying a vulnerable code sample x. Specifically, the adversary seeks to generate an adversarial example x^* from the original input x such that:

$$F(x^*) \neq F(x) = y_{\text{vul}}, \tag{1}$$

where y_{vul} denotes the vulnerable class label of x. Note that, following prior work [7], we focus on attacks that cause vulnerable samples to be misclassified as non-vulnerable, since this model error poses a greater risk than the reverse [36].

To generate realistic adversarial examples, the perturbation δ applied to x must satisfy the constraints described in Section III-A. Formally, the adversarial example x^* must satisfy:

$$\begin{gathered} x^* = x + \delta, \delta \in \mathcal{C} \\ F(x^*) \neq F(x),\ S(x^*) = S(x). \end{gathered} \tag{2}$$

where $\mathcal{C}$ denotes the set of allowed transformations that preserve syntax and semantics, and $S(\cdot)$ is a function that checks the semantic equivalence between code samples.

To meet the above objective, we propose SVulAttack, a black-box adversarial attack framework that generates adversarial examples through a combination of semantic-preserving code transformations with optimization algorithms such as greedy search and genetic algorithms.

Figure 3 illustrates the overall workflow of SVulAttack, using a greedy search-based process as an example. As shown in the figure, given a vulnerable input code sample x, SVulAttack first identifies a set of *reference inputs*—code samples retrieved from an external vulnerability dataset (e.g., *big_vul*) that are similar to x but classified by F as non-vulnerable. By comparing x with these reference inputs, SVulAttack computes a similarity-based importance score I for each statement in x. These scores guide the attack by prioritizing statements that are more likely to influence F's classification.

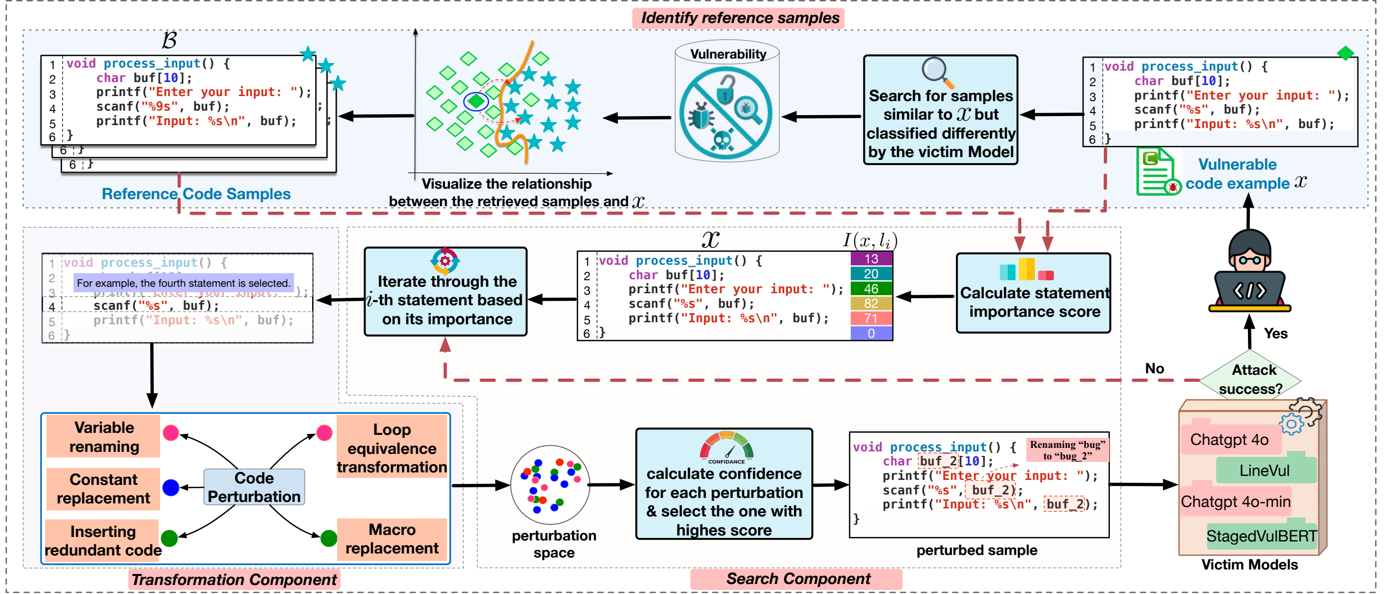


Fig. 3. An overview of the SVulAttack framework.

Once the statements are ranked, SVulAttack processes them in descending order of importance. For each selected statement, it applies multiple semantic-preserving transformations, including variable renaming, constant replacement, insertion of redundant code, loop equivalence transformation, and macro replacement. In this process, token extraction is crucial for identifying tokens or patterns from non-vulnerable references that guide code modifications toward non-vulnerable distributions (see Section IV-A). These patterns are then used by each transformation to generate candidate perturbations. SVulAttack evaluates these candidates and selects the one that maximizes the probability of misclassification. The updated code is then queried against F. If the model misclassifies the perturbed code as non-vulnerable, SVulAttack returns it as the adversarial example x^*. If not, the framework proceeds to the next statement in the priority list and repeats this process until a successful attack is found or a termination criterion is met. Further details on the transformations and search strategies are presented in Sections IV and V, respectively.

IV. Designed Perturbation

To generate adversarial examples, we apply five semantic-preserving code transformations that introduce minimal changes to the lexical representation of the code while preserving its functional behavior and semantic meaning [37]. Before detailing these transformations, we introduce a token extraction process that provides candidate tokens and variable names used in perturbations. By leveraging these tokens, the introduced variations are designed to effectively influence the detector's behavior, potentially leading to misclassification.

A. Token Extraction

The token extraction process identifies tokens that reflect non-vulnerable characteristics and guides perturbations to transform vulnerable code into adversarial examples that evade detection. To this end, we extract tokens specifically from non-vulnerable code samples based on each token's discriminative power, measured by how its removal changes the sample's similarity to vulnerable code. Note that tokens, extracted through lexical analysis [38], refer to identifiers, operators, constants, and keywords. Specifically, let C_{nv} and C_v denote sets of non-vulnerable and vulnerable samples, respectively, collected from public vulnerability databases. We randomly select k samples from C_{nv}, and for each selected c_{nv}, retrieve the m most similar samples from C_v using the following cosine similarity to form the set C_v^m:

$$D(c_{nv}, c_v) = \cos\Big(r(c_{nv}), r(c_v)\Big), \tag{3}$$

where $r(\cdot)$ denotes the TF–IDF-based vector representation. TF–IDF can be replaced by other semantic vector representations, but it is preferred because the latter often remain insensitive to single-token modifications, undermining reliable token-importance estimation. The comparative performance of these two vector representations is discussed in Section VIII-C.

For each token t in c_{nv}, we compute its importance by removing t to obtain c_{nv}^{-t} and measuring the change in similarity to C_v^m. Formally, the importance score $I(t)$ is defined as the difference between the softmax-weighted average similarities before and after removing t:

$$\begin{aligned} I(t) = & \frac{\sum_{c_v \in C_v^m} e^{D(c_{nv}^{-t}, c_v)}\, D(c_{nv}^{-t}, c_v)}{\sum_{c_v \in C_v^m} e^{D(c_{nv}^{-t}, c_v)}} \\ & - \frac{\sum_{c_v \in C_v^m} e^{D(c_{nv}, c_v)}\, D(c_{nv}, c_v)}{\sum_{c_v \in C_v^m} e^{D(c_{nv}, c_v)}}. \end{aligned} \tag{4}$$

A larger $I(t)$ indicates that removing token t increases similarity to vulnerable samples, suggesting its importance in distinguishing non-vulnerable code from a distribution of vulnerable code samples. Tokens with high $I(t)$ are collected

into the candidate set W to guide subsequent perturbations toward the non-vulnerable code distribution.

B. Code Transformations in SVulAttack

1) Variable Renaming (VR): Variable Renaming (VR) replaces a variable name in the target code x with a name selected or constructed from a predefined set of tokens. As previous studies have demonstrated [20], [21], [39], modifying identifiers can significantly affect how models extract and interpret semantic features. Motivated by this, we adopt VR as one of the primary transformations in our framework.

Let $V(x)$ be the set of variables in x, and let $v \in V(x)$ be a chosen variable. Furthermore, let W represent the token set extracted as described in Section IV-A. The formal definition of VR is given in Equation 5.

$$T_{\text{var}}(x, v, w) = \left\{ \begin{array}{c} t_{var} \mid \forall x \in \mathcal{X}, \forall v \in V_{var}(x), \\ \forall w \in W : t_{var}(x, v, w) = x' \end{array} \right\} \quad (5)$$

where x' is obtained by replacing all occurrences of v with w. VR offers an effective way to alter the code embeddings generated by the target model, thus potentially inducing misclassification while preserving both syntax and semantics.

2) Constant Replacement (CR): The constant replacement involves transforming numerical and string constants in code x by first defining them as constant variables and then substituting the original constants with these variables. Specifically, numerical constants are defined as "`const int var_name`", and string constants as "`const char* var_name`". This process introduces new variable declarations into the abstract syntax tree (AST) and modifies the original constant nodes, thereby altering control and data dependencies. To increase the probability of successful evasion, variable names are selected from the candidate token set W introduced in Section IV-A. The formal definition of constant replacement is given in Equation 6.

$$T_{\text{const}}(x, v) = \left\{ \begin{array}{c} t_{\text{const}} \mid \forall x \in \mathcal{X}, \forall v \in V_{\text{const}}(x) : \\ t_{\text{const}}(x, v) = x' \end{array} \right\} \quad (6)$$

where $v \in V_{\text{const}}(x)$ denotes a variable name chosen from W, and x' refers to the program after performing constant replacement. Figure 4 shows an example of this perturbation.

a) Original program

```
1 void bad()
2 int * data ;
3 data = new int [ 10 ];
4 if ( GLOBAL_CONST_FIVE == 5 )
5
```

b) Perturbed program after constant replacement

```
void bad()
int * data ;
data = new int [ 10 ];
const int IKDFC = 5;
if(GLOBAL_CONST_FIVE==IKDFC)
```

Fig. 4. Example of perturbation by performing constant replacement

3) Inserting Redundant Code (IRC): Inserting redundant code (IRC) introduces additional statements that do not alter the program's semantics but establish data dependencies to prevent trivial removal. We adopt `printf("token %x", &variable);` as the most effective redundant code insertion template, supported by preliminary experiments and prior studies [19]. Here, `variable` establishes a data dependency with existing code, and `token` is selected to decrease the model's confidence in predicting the correct label of the perturbed sample as much as possible. Let W_r denote the candidate set of tokens extracted in Section IV-A, and $V_r(x)$ be the candidate set of "variable" in x. We define the IRC transformation as follows:

$$T_r(x, v, w) = \left\{ \begin{array}{c} t_r \mid \forall x \in \mathcal{X}, \forall v \in V_r(x), \\ \forall w \in W_r : t_r(x, v, w) = x' \end{array} \right\} \quad (7)$$

Figure 5 illustrates an example of code after applying IRC.

a) Original program

```
1 void bad()
2 char * data ;
3 data = new char [ 100 ];
4 if (staticReturnsTrue( ))
5 static int staticReturnsTrue()
6 return 1 ;
7 memset(data, 'A', 100 - 1 );
8 data[100 - 1] = '\0';
9 strncpy(dest, data, strlen(data));
10
```

b) Perturbed program through the insertion of redundant code

```
void bad()
char * data ;
data = new char [ 100 ];
if ( staticReturnsTrue ( ) )
static int staticReturnsTrue()
return 1 ;
memset ( data , 'A' , 100 - 1 );
data [ 100 - 1 ] = '\0';
printf("static, %p", &data);
strncpy(dest, data, strlen(data));
```

Fig. 5. Example of perturbation by inserting redundant code

4) Loop Equivalence Transformation (LET): Loop equivalence transformation involves moving the conditional statement inside the loop body to create an equivalent loop structure. In this paper, we propose to use loop equivalence transformation as a method for transforming programs, as it introduces new control dependencies, data dependencies, and branches, which modifies the structure of the program while maintaining the program semantics unchanged. For instance, the statement "`while (exp)`" can be transformed into one of six equivalent templates, as shown in Figure 6.

```
1 while(1){if(!exp) break; …
2 int flag = 1; while(flag){if(!exp) flag = 0; …
3 while(1){if(!exp) break; else continue; …
4 int flag = 1; while(flag){if(!exp) flag = 0; else continue; …
5 bool flag = true; while(flag){if(!exp) flag = false; …
6 bool flag = true; while(flag){if(!exp) flag = false; else continue;
```

Fig. 6. Six forms of loop templates employed in this paper

In Figure 6, the variable represented by "flag" is selected from tokens extracted in Section IV-A. In this context, let $V_{\text{loop}}(x)$ denote the set of candidate variable names for the "flag", which is derived from the candidate token set W. Equation 8 formally defines the loop equivalence transformation.

$$T_{\text{loop}}(x, v) = \left\{ \begin{array}{c} t_{\text{loop}} \mid \forall x \in \mathcal{X}, \forall v \in V_{\text{loop}}(x) : \\ t_{\text{loop}}(x, v) = x' \end{array} \right\} \quad (8)$$

where x' represents the perturbed program after applying LET. Figure 7 shows an example of this transformation. The templates for replacing "for" and "do...while" loops are similar to those for "while" loops and are omitted for brevity. In experiments, we evaluate all templates from Figure 6 and select the one that achieves the best attack performance.

5) Macro Replacement (MR): Macro definitions enable developers to define a set of instructions or a block of code that can be reused throughout a program, providing a concise way to express frequently used code. In this paper, we propose to

```
size_t i ;
for(i = 0; i < 100; i++)
dataBuffer[i] = 5L;
data = dataBuffer;
printLongLongLine(data[0]);
```

a) Original program

```
size_t i ;
bool OOOQU =true
for(i = 0;OOOQU;i++)
if(!( i < 100))OOOQU=false;
dataBuffer[i] = 5L;
data = dataBuffer;
printLongLongLine(data[0]);
```

b) Perturbed program after performing equivalence transformation

Fig. 7. Example of perturbation through loop equivalence transformation

use MR to modify the program's token sequence and, in turn, perturb the behavior of the detection model. We focus only on non-parameterized macro replacement, as parameterized macros introduce additional complexity and require more sophisticated processing, which is outside the scope of this work.

To ensure that the semantics of the original code are not changed, we propose two rules for generating macro names during macro definition: (1) selecting a token that does not appear in the original code, and (2) choosing from the token set W obtained in Section IV-A. Let $M_{\text{macro}}(x)$ denote all valid macro names for program x, with candidates drawn from W. The transformation is formally defined as:

$$T_{\text{macro}}(x, m) = \left\{ \begin{array}{c} t_{\text{macro}} \mid \forall x \in \mathcal{X}, \forall m \in M_{\text{macro}}(x) : \\ t_{\text{macro}}(x, m) = x' \end{array} \right\} \tag{9}$$

V. Combination Attack Method Based on Greedy Search or Genetic Algorithm

A. Combination Attack Method

The combination attack method applies one or more code transformations to generate adversarial programs that preserve the original semantics while inducing misclassification. Formally, let T denote the set of all transformation methods introduced in Section IV. Let q denote a transformation sequence $t_1, t_2, ..., t_k$ with corresponding parameter set $\theta_1, \theta_2, ..., \theta_k$, where $t_l \in T$, $\theta_l \in \Theta$, and Θ is the parameter space of transformation operations. The set of all such sequences of length k is denoted as Q^k. The objective of the combination attack is to identify $q \in Q^k$ such that the perturbed program $x\prime = q(x)$ is misclassified by the detector, i.e., $F(x') \neq F(x)$. Here, $q(x)$ denotes the transformed program $t_k(...t_2(\theta_2, t_1(\theta_1, x)))$. Notably, k serves as a key parameter in the combination attack, controlling the number of program modifications. To keep perturbations imperceptible, k can be restricted to small values (e.g., $k = 1$), and we analyze its effect in Section VII-D.

Each transformation method t involves two types of parameters: (1) the replacement content defined by the corresponding transformation formula; (2) the location of the statement to be modified. For instance, to perform macro replacement, we first determine the position of the statement to be perturbed, and then replace the token in that statement with the most suitable macro name $m \in M_{macro}(x)$. The selection of m is guided by its estimated potential to mislead the detection model.

Generating the optimal transformation sequence q for a program x is challenging, as it requires iteratively exploring a combinatorial space of candidate transformations with varying parameters to identify those that maximize attack effectiveness. To address this challenge, we propose a greedy search-based and a genetic algorithm-based combination attack method.

B. Combination Attack Method Based on Greedy Search

We present the Greedy Search-based Combination Attack Method in Algorithm 1, which takes as input a program x, a maximum transformation sequence length k, and a vulnerability detector F. The core idea is to prioritize modifications to statements that most influence the model's prediction.

Algorithm 1: Greedy Search-Based Combination Attack Method

Input: Program x with n statements; Maximum transformation length k; Vulnerability detector F
Output: Adversarial sample *adv*

```
1  Construct set B of the most similar samples to x that are
   classified differently by F;
2  for i ← 1 to n do
3      Compute statement importance score I(x, l_i) for
       statement l_i in x using Eq. (10);
4  Sort statements l_i in descending order of I(x, l_i);
5  Initialize adv ← x;
6  for each statement l_i in sorted order do
7      C ← ∅;
8      for each transformation t ∈ T_i(x) do
9          Generate x′ = t(adv);
10         if F(x′) ≠ F(adv) then
11             return x′;
12         C ← C ∪ {x′};
13     best_delta ← −∞;
14     best_candidate ← adv;
15     for each candidate x′ ∈ C do
16         Compute Δ_sim ;
17         if Δ_sim > best_delta then
18             best_delta ← Δ_sim;
19             best_candidate ← x′;
20     if best_delta > 0 then
21         adv ← best_candidate;
22         k ← k − 1;
23         if k ≤ 0 then
24             return adv;
25 return adv;
```

In Algorithm 1, we first construct a set $\mathcal{B}$ comprising the most similar samples to x that are classified differently by F (Line 1). For each statement l_i in x, we compute an importance score $I(x, l_i)$ to estimate the impact of modifying l_i on the model's prediction (Lines 2-3). Since we cannot obtain F's confidence scores (as it outputs only labels 0 or 1), we approximate $I(x, l_i)$ by measuring the average change in similarity to samples in $\mathcal{B}$ when l_i is removed. This similarity-based score serves as a surrogate for confidence, enabling perturbation guidance without internal model access.

$$I(x, l_i) = \frac{\sum_{x^r \in \mathcal{B}} e^{sim(x^{(i)}, x^r)} \cdot sim(x^{(i)}, x^r)}{\sum_{x^r \in \mathcal{B}} e^{sim(x^{(i)}, x^r)}} - \frac{\sum_{x^r \in \mathcal{B}} e^{sim(x, x^r)} \cdot sim(x, x^r)}{\sum_{x^r \in \mathcal{B}} e^{sim(x, x^r)}}, \tag{10}$$

where $x^{(i)}$ denotes x with the i-th statement removed, and $sim(\cdot, \cdot)$ measures the similarity between programs. Intuitively, a larger $I(x, l_i)$ represents l_i plays a significant role in distinguishing vulnerable from non-vulnerable samples.

After computing the importance scores, we rearrange the statements in descending order of $I(x, l_i)$ (Line 4). The algorithm initializes the adversarial sample *adv* as x (Line 5) and iteratively applies transformation methods to each statement. For a statement l_i, let $T_i(x)$ denote the set of all available transformation operations applied at that statement. For each transformation $t \in T_i(x)$, a candidate perturbed sample $x' = t(adv)$ is generated (Lines 8-9).

If x' leads F to output a label different from that of *adv* (i.e., $F(x') \neq F(adv)$), x' is immediately returned as the adversarial sample (Lines 10-11). Otherwise, we collect all candidates in a set $\mathcal{C}$ and compute for each the corresponding similarity change Δ_{sim}, defined analogously to $I(x, l_i)$ but with x' and *adv* substituting for $x^{(i)}$ and x in Equation 10, respectively (Lines 15-16). We interpret Δ_{sim} as an estimate of the model's confidence and a larger Δ_{sim} indicates that the perturbed sample x' has moved closer to the reference non-vulnerable samples, implying shift toward the decision boundary of F. Among all candidates in $\mathcal{C}$, the one with the maximal positive Δ_{sim} is selected to update *adv* (Lines 15-21). The process is iterated until an adversarial sample is found or the maximum number k of transformations is reached.

By employing similarity-based estimates of statement importance and prediction confidence, our method guides impactful perturbations, effectively generating adversarial samples without relying on the detector's confidence scores.

C. Combination Attack Method Based on Genetic Algorithm

Genetic algorithms are an effective optimization technique inspired by natural selection, capable of thoroughly exploring the search space and avoiding being stuck in local optima. This motivates our design of a combination attack for generating adversarial examples. Algorithm 2 describes our genetic algorithm-based attack method, which takes as input a program x with n statements, a maximum of k allowed modifications, a vulnerability detector F, maximum iterations $iter$, population size p, crossover rate r, and mutation rate m.

The algorithm begins by determining the available program transformations $T_i(s_i, \Theta)$ for each statement s_i in x, combining them to form a set M (Lines 1-3). To initialize a population P_0 of p candidate solutions, the algorithm randomly selects statements from x and applies transformations from M under constraint k (Line 5). Each individual $m_j \in P_0$ is represented as a sequence of n genes, where each gene is either the original statement s_i or its variant produced by $T_i(s_i, \Theta)$. The initial adversarial candidate adv is set to unmodified x (Line 6).

Algorithm 2: Genetic Algorithm-Based Combination Attack Method

Input : Original program x with n statements; Constraint k; Detector F; Maximum iterations $iter$; Population size p; Crossover rate r; Mutation rate m
Output: Adversarial example adv

1 **for** $i = 1$ **to** n **do**
2   Select available transformation $T_i(s_i, \Theta)$ for s_i
3   $M \leftarrow M \cup T_i(s_i, \Theta)$
4 **end**
5 Initialize population $P_0 \leftarrow$ Initialize(p, n, x, M)
6 Initialize $adv \leftarrow x$;
7 Calculate fitness $F_0 \leftarrow$ Fitness(P_0, adv);
8 Select best candidate $adv \leftarrow \arg\max_{x' \in P_0} F_0(x')$;
9 **for** $c = 0$ **to** $iter$ **do**
10   **for** *each* $x' \in P_c$ **do**
11   **if** $F(x') \neq F(x)$ **then**
12       $adv \leftarrow x'$
13   **return** adv
14   **end**
15   **end**
16   Select individuals $S_c \leftarrow$ Select(P_c, F_c)
17   Generate offspring $C_c \leftarrow$ Crossover(S_c, r)
18   Apply mutation: $L_c \leftarrow$ Mutate$(C_c \cup S_c, m, M)$
19   $P_{c+1} \leftarrow$ Filter(L_c, k) // Discard candidates exceeding k modifications
20   Compute fitness $F_{c+1} \leftarrow$ Fitness(L_{c+1}, adv)
21   Update best candidate: $adv \leftarrow \arg\max_{x' \in L_{c+1}} F_{c+1}(x')$
22 **end**
23 **return** adv ;

Next, the algorithm calculates the fitness for the initial population P_0 (Line 7). Since we cannot obtain the model's confidence scores (as the detector F outputs only labels 0 or 1), we therefore introduce a surrogate fitness function to estimate how closely a perturbed sample approaches the decision boundary. Specifically, let $\mathcal{B}$ denote a set of samples that are similar to x but are classified differently by F. For each candidate x', the fitness is computed as the difference between the weighted average similarity of x' to the samples in $\mathcal{B}$ and that of the current best adversarial example adv, as defined by Δ_{sim} in Section V-B. A higher fitness indicates that the perturbed sample is more similar to the samples in $\mathcal{B}$ than adv is, and is thus more likely to be misclassified.

In each iteration c, the algorithm first examines each candidate x' in the current population P_c to check if it already satisfies $F(x') \neq F(x)$; if such a candidate exists, it is immediately returned as the adversarial example (Lines 10-15). Otherwise, the algorithm applies the selection, crossover, and mutation to generate the next population P_{c+1}. The selection operator chooses candidate solutions from P_c based on their fitness scores F_c (Line 16). The crossover operator generates offspring C_c by recombining members of S_c using the crossover rate r (Line 17). The mutation operator introduces diversity by randomly modifying genes in the individuals (Line 18). After mutation, a check procedure is applied: for each newly generated candidate in P_{c+1}, if its number of modifications exceeds k, it is discarded and replaced by a new candidate (generated by further mutation) so that the population size remains fixed (Line 19). Finally, after updating the population,

the algorithm recalculates the fitness values F_{c+1} and updates the best candidate (Lines 20-21). This process repeats for up to $iter$ iterations or until an adversarial sample is found.

In Algorithm 2, crossover and mutation are the primary operations for generating new populations. To preserve syntax and minimize disruption, we employ a single-point crossover with an identical crossover point in both parents, ensuring that the offspring inherit the same number of code lines. To avoid local optima, we employ the mutation operator to slightly perturb a member by selecting a random line and applying a program transformation, thereby generating a new individual.

D. Similarities and Differences Between the Two Combination Attack Methods

As detailed in Sections V-B and V-C, our two proposed combination-based attack methods share several common components. Both approaches employ the same set of five transformation operations and aim to discover optimal transformation sequences. They also share the same parameter k to control the degree of perturbation: smaller values of k lead to more imperceptible perturbations. Furthermore, both methods are built upon the candidate tokens identified in the token extraction phase and utilize our proposed similarity-based strategy to guide the selection of effective perturbations.

The main difference lies in their search strategies. Greedy search applies transformations step by step, deterministically selecting the most promising modification at each step, guided by statement importance estimated through the proposed similarity-based strategy. By contrast, the genetic algorithm evolves a population of adversarial samples through stochastic crossover and mutation, rather than explicitly ranking statements by importance. Each generation is then evaluated with a similarity-based fitness function that favors samples moving closer to non-vulnerable references and away from the original vulnerable code. Although perturbation positions are introduced randomly within each generation, the population is iteratively guided by the same similarity-based mechanism.

Overall, both methods rely on the similarity-based strategy to identify effective perturbations. Without this mechanism, neither search approach would be viable in a label-only black-box setting, which demonstrates its essential role and critical importance in our framework. A more detailed comparison of the two methods is provided in Section VIII-D.

VI. Experiment Design

We aim to answer the following research questions[1]:

RQ1: How effective is the proposed attack method against state-of-the-art open-source vulnerability detection models?

RQ2: How effectively can the proposed attack method be applied to closed-source vulnerability detection models?

RQ3: To what extent do the proposed perturbation and search methods contribute to attack success?

RQ4: How does varying the number of modifications in the proposed attack methods affect attack success rates?

RQ5: To what extent do our attack methods generalize across datasets?

[1]The code and dataset are available at https://github.com/YuanJiangGit/SVulAttack.git

A. Dataset

We conduct our experiments on the *big_vul* dataset [40], a widely adopted benchmark for evaluating vulnerability detection models. The dataset contains 188,636 functions, including 10,900 labeled as vulnerable and 177,736 labeled as non-vulnerable, collected from 348 real-world projects.

Adversarial attacks are computationally expensive due to the vast perturbation space associated with each sample. This challenge is further pronounced when targeting commercial models. Following common practice [41]–[43], we evaluate our method on a representative subset of the full dataset. Specifically, we extract 500 vulnerable samples from *big_vul*, spanning 12 CWE types that are the most frequent and dangerous in the dataset. Among them, CWE-119, CWE-20, CWE-399, CWE-264, CWE-416, CWE-200, CWE-125, CWE-189, CWE-362 and CWE-476 are the most commonly occurring CWE types in the *big_vul* dataset [4], while CWE-787, CWE-125, CWE-20, CWE-416, CWE-190, CWE-476 and CWE-119 are recognized among the Top-25 most dangerous CWEs [44]. Figure 8 presents the distribution of CWE types among these samples. To prevent potential data leakage, all evaluated vulnerable samples and the reference set are taken from the victim models' test set, with no overlap with their training data. Since the reference set contains only non-vulnerable code, it is also disjoint from the evaluation set of vulnerable samples. Experiments on this representative subset allow us to thoroughly evaluate the performance of our attack method.

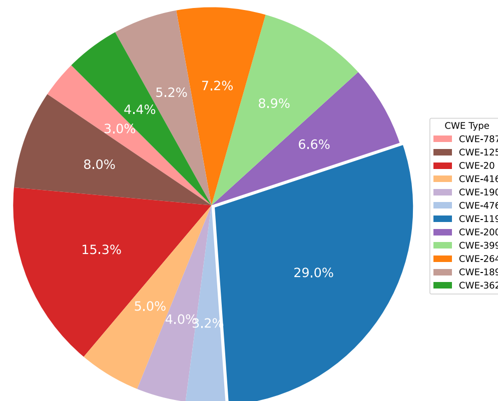


Fig. 8. The pie chart presents the distribution of vulnerability types of the programs in the dataset used in this paper. Each sector in the pie chart represents a different CWE whose size shows the proportion of programs with the specific type of vulnerabilities in the *big_vul* dataset.

B. Evaluation Metrics

To evaluate the effectiveness and efficiency of the proposed attack method, we use the following two metrics:

Attack Success rate (AS): This measures the proportion of correctly classified samples that are successfully misclassified after perturbation. Let s be the number of successfully attacked samples and n the number of correctly classified samples before the attack:

$$AS = \frac{s}{n}.$$

Average Queries per Sample (AQS): Following previous work [19], this metric reflects the average number of queries made to the victim model per sample. Let QS_i denote the number of queries for the i-th successful sample:

$$AQS = \frac{\sum_i QS_i}{|\{QS_i\}|}.$$

A lower AQS indicates a more efficient attack.

C. Baseline Methods

To provide a fair comparison, we select two state-of-the-art black-box baselines, DIP [19] and CODA [23].

DIP [19] employs a surrogate model to compute importance scores for potential insertion points and adds dead code snippets derived from dissimilar code via attention scores. DIP explores various perturbations and applies them until the victim model's prediction flips. However, its effectiveness is limited by the surrogate model, whose attention scores may not align with the true sensitivities of the victim model. Moreover, DIP supports only one type of perturbation (dead code insertion), reducing its adaptability and overall effectiveness.

CODA [23] employs two transformations: Equivalent Structure and Identifier Renaming, to craft adversarial examples. It first aligns the structure of the target with a reference input, then renames identifiers using those from the reference to make the target resemble the reference. Although CODA considers target-reference differences, it lacks a refined strategy for selecting impactful locations and optimal transformations from numerous perturbation candidates, which may hinder its performance and efficiency in complex vulnerability scenarios.

D. Evaluation Setting

We evaluate our attack method on both open-source (LineVul, StagedVulBERT, Code Llama and Deepseek-Coder) and closed-source (GPT-5-nano, GPT-4o, GPT-4o-mini and Claude Sonnet 4) vulnerability detectors to assess its generalizability across diverse model architectures and deployment scenarios.

We adopt the black-box adversarial attack paradigm, in which the attacker is restricted to providing input code to the model and receiving only hard-label outputs (i.e., vulnerable or non-vulnerable). To guide the perturbation of vulnerable inputs, we select a set $\mathcal{B}$ consisting of 100 reference code samples that are classified as non-vulnerable by the victim models. These reference samples serve as perturbation anchors for transforming vulnerable inputs toward non-vulnerable representations. For the Greedy Search and Genetic Algorithm components of our approach, we configure hyperparameters based on empirical analysis. Specifically, we set the maximum number of modifications $k = 15$, the number of iterations $iter = 17$, the population size $p = 45$, the crossover rate $r = 0.6$, and the mutation rate $m = 0.6$. These values are chosen through random search over candidate configurations, as they yield the best performance in preliminary experiments. Concretely, k is uniformly sampled from $\{1, 2, \dots, 15\}$, $iter$ from $\{10, 11, \dots, 20\}$, p from $\{10, 15, 20, \dots, 100\}$, and both r and m from $\{0.1, 0.2, \dots, 0.9\}$. Since k is the hyperparameter shared by both our greedy search-based and genetic algorithm-based combination attack methods, and it directly determines the degree of perturbation, we further analyze its impact on model performance in Section VII-D.

Before launching attacks, we ensure that all selected vulnerable samples are correctly classified by the victim models. We then apply our proposed transformations in combination with the search algorithms to generate adversarial examples. A successful attack is recorded if any generated sample is misclassified by the model as non-vulnerable.

VII. Experiment Results

A. RQ1: How effective is the proposed attack method against state-of-the-art open-source vulnerability detection models?

Motivation. This section evaluates the effectiveness and efficiency of our black-box adversarial attack methods against advanced open-source vulnerability detection models. Specifically, we target two state-of-the-art detectors, LineVul [16] and StagedVulBERT [4], as well as large-scale code LLMs, including Code Llama 7B [30] and Deepseek-Coder 7B [31]. By comparing with two cutting-edge black-box baselines, DIP [19] and CODA [23], we aim to demonstrate that our methods achieve higher attack success rates (AS) and improved query efficiency (AQS). Furthermore, we assess the attack performance across various CWE types to highlight the effectiveness of our methods in different vulnerability categories.

Approach. We evaluate four methods: (1) Greedy-Combo: our greedy search-based combination attack method; (2) Genetic-Combo: our genetic algorithm-based combination attack method; (3) DIP; and (4) CODA (both baselines are described in Section VI-C). All methods are tested on the same set of vulnerable samples as described in Section VI-A. For DIP and CODA, we adhere to their original configurations in our re-implementation. We compute AS and AQS for each method to compare their performance. In addition, we analyze the results across 12 CWE types, which are among the most frequent and dangerous in the *big_vul* dataset.

Results. Table I summarizes the AS and AQS of all four methods, while Figure 9 shows results on the 12 selected CWE types. On average, our Greedy-Combo and Genetic-Combo approaches outperform DIP and CODA in both AS and AQS on four victim models. For example, on the LineVul model, Greedy-Combo achieves an AS of 39.6% and Genetic-Combo attains an AS of 49.0%, compared to AS values of 19.6% and 14.4% for DIP and CODA, respectively. This represents AS improvements of 102.0% and 175.0% over DIP and CODA for Greedy-Combo, and 150.0% and 240.3% for Genetic-Combo. Furthermore, across all victim models, our methods exhibit lower query overhead, improving attack efficiency. For instance, on LineVul, the Greedy-Combo and Genetic-Combo methods achieve AQS values of 176.3 and 191.2, respectively, compared to 200.8 for DIP and 243.3 for CODA.

These improvements arise from the key innovations of our method. Specifically, our approach uses similarity-based heuristics to assess the importance of code statements and the potential impact of various perturbations. It then applies a diverse set of semantically preserving code transformations, combining greedy and genetic algorithms to effectively explore

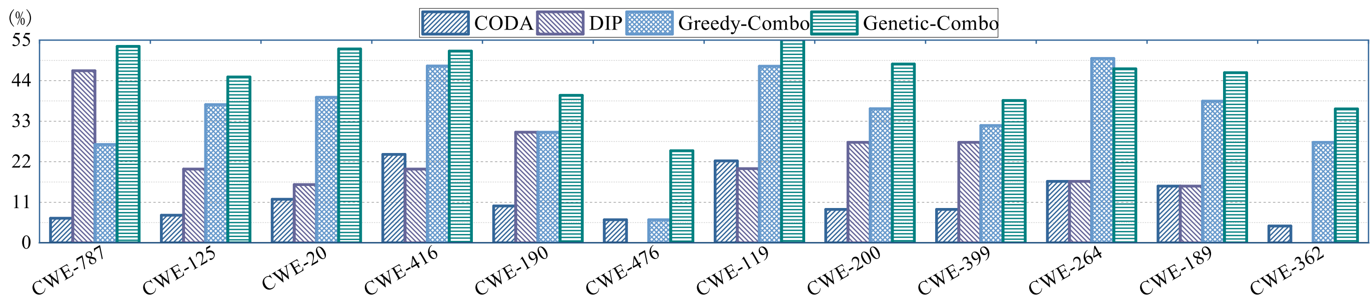


Fig. 9. Attack success rate of our methods and the state-of-the-art methods for the Top-12 common and high-risk CWEs

the candidate space. This streamlined strategy significantly outperforms baselines that rely on surrogate models or a limited set of perturbation techniques.

We also observe from Table I that our methods perform better on larger code LLMs (e.g., Deepseek-Coder) than on smaller task-specific detectors. A plausible explanation is that specialized models such as LineVul and StagedVulBERT capture vulnerability features more precisely and thus maintain clearer decision boundaries between vulnerable and non-vulnerable samples [4], making them harder to evade. By contrast, larger code LLMs exhibit less distinct boundaries, which makes them more susceptible to adversarial perturbations.

TABLE I
ATTACK SUCCESS RATE (AS) AND AVERAGE QUERIES PER SAMPLE (AQS) OF DIFFERENT METHODS ON OPEN-SOURCE MODELS

Target Model	Type	Method	AS (%)	AQS
LineVul	Baselines	DIP	19.6	200.8
	Baselines	CODA	14.4	243.3
	Our Method	Greedy-Combo	39.6	**176.3**
	Our Method	Genetic-Combo	**49.0**	191.2
StagedVulBERT	Baselines	DIP	11.2	147.0
	Baselines	CODA	9.6	216.4
	Our Method	Greedy-Combo	18.4	**132.1**
	Our Method	Genetic-Combo	**28.8**	211.6
Code Llama 7B	Baselines	DIP	18.8	303.9
	Baselines	CODA	47.2	293.3
	Our Method	Greedy-Combo	55.4	331.2
	Our Method	Genetic-Combo	**71.2**	**205.8**
Deepseek-Coder 7B	Baselines	DIP	44.2	306.9
	Baselines	CODA	70.6	234.7
	Our Method	Greedy-Combo	90.8	243.9
	Our Method	Genetic-Combo	**96.0**	**122.6**

We also conduct a comprehensive comparison of our attack methods against baselines across a diverse range of CWE vulnerabilities on the LineVul model. The experimental results are depicted in Figure 9, which reveal that both our Greedy-Combo and Genetic-Combo methods consistently achieve higher AS than DIP and CODA. For example, on CWE-119, Genetic-Combo achieves an AS of 56.9%, while DIP and CODA register only 20.1% and 22.2%, respectively. While our methods outperform the baselines on most CWE categories, two outliers warrant further explanation. For CWE-787, Greedy-Combo occasionally underperforms DIP because its heuristic, similarity-guided greedy search may miss globally optimal perturbations for the out-of-bounds write statements that typically determine this vulnerability. In contrast, DIP's surrogate-model approach can more directly identify and perturb these critical statements without extensive iterative search. CWE-787 represents only 3% (15 instances) of our test set, as shown in Figure 8. Given this limited sample size, results for this category should be considered together with other CWE types for a more comprehensive assessment. A two-tailed Wilcoxon signed-rank test across all CWE categories yields T_{wilcox} below the critical value [45], indicating that Greedy-Combo's improvement over DIP is statistically significant at the 0.05 level. For CWE-476, DIP achieves 0% AS because these vulnerabilities typically involve pointer-dereference expressions inside called functions or function-call arguments, and DIP's dead-code insertion has limited ability to affect the lexical or structural patterns the detector relies on.

Conclusion: Our approach outperforms state-of-the-art black-box attack methods on open-source vulnerability detectors, owing to its similarity-based estimation, diverse transformations, and effective optimization strategies.

B. RQ2: How effectively can the proposed attack method be applied to closed-source vulnerability detection models?

Motivation. As demonstrated in RQ1, our method effectively deceives open-source vulnerability detection models, including LineVul, StagedVulBERT, Code Llama and Deepseek-Coder, by inducing misclassification of vulnerable samples as non-vulnerable. However, given that closed-source LLMs are increasingly employed for vulnerability detection [32], [33], it is critical to assess their robustness against adversarial examples. To this end, we target three widely used OpenAI models (GPT-4o, GPT-4o-mini, and GPT-5 nano) and another state-of-the-art LLM, Claude Sonnet 4. We then apply various attack methods to assess their resilience and determine whether our approach remains more effective when attacking them.

Approach. We evaluate our proposed method against four closed-source models, using a subset of the *big_vul* dataset described in Section VI-A, with DIP and CODA as baselines. To ensure reproducibility, we set the temperature parameter to 0, following previous work [7]. However, even with this deterministic setting, the model outputs still exhibit occasional variability. To mitigate this, we issue three repeated queries for each perturbed input and determine the final prediction based on unanimous agreement. Specifically, for GPT-4o, an attack is

considered successful only if the perturbed sample is classified as non-vulnerable in all three consecutive evaluations.

TABLE II
ATTACK SUCCESS RATE (AS) AND AVERAGE QUERIES PER SAMPLE (AQS) OF DIFFERENT METHODS ON CLOSED-SOURCE MODELS

Target Model	Type	Method	AS (%)	AQS
GPT-5 nano	Baselines	DIP	77.8	63.9
	Baselines	CODA	75.8	122.1
	Our Method	Greedy-Combo	**90.9**	70.5
	Our Method	Genetic-Combo	83.3	**55.8**
GPT-4o	Baselines	DIP	82.6	**57.7**
	Baselines	CODA	85.4	99.7
	Our Method	Greedy-Combo	86.1	76.9
	Our Method	Genetic-Combo	**93.3**	88.9
GPT-4o-mini	Baselines	DIP	57.8	137.0
	Baselines	CODA	43.2	**88.1**
	Our Method	Greedy-Combo	34.0	109.4
	Our Method	Genetic-Combo	**65.0**	211.2
Claude Sonnet 4	Baselines	DIP	67.6	85.5
	Baselines	CODA	69.7	130.4
	Our Method	Greedy-Combo	**72.7**	139.5
	Our Method	Genetic-Combo	71.4	**82.8**

Results. Table II presents the AS and AQS achieved by each method on the closed-source models. Overall, our Genetic-Combo and Greedy-Combo consistently outperform the baseline methods. On GPT-4o-mini and GPT-4o, Genetic-Combo achieves the highest AS, with 65.0% and 93.3%, respectively. These represent improvements of 12.5% and 50.5% over DIP (57.8%) and CODA (43.2%) on GPT-4o-mini, and 13.0% and 9.3% over DIP (82.6%) and CODA (85.4%) on GPT-4o.

For GPT-5 nano and Claude Sonnet 4, however, Greedy-Combo attains the best AS, outperforming both baselines and Genetic-Combo. This can be attributed to the fact that these models, which are trained with broader generalization objectives, tend to form smoother decision boundaries dominated by a few critical statements. By directly targeting these high-impact locations, greedy search produces more effective perturbations than population-based exploration.

Although our methods achieve superior attack success rates, they also require a higher AQS compared to the baselines in some cases. This increased cost reflects the broader perturbation space explored by our two proposed search strategies, where more extensive exploration translates into higher attack effectiveness. We emphasize that, in security-critical scenarios, attack success is often more consequential than query efficiency, since a single successful evasion can undermine model reliability. Nevertheless, we acknowledge that enhancing the efficiency of our method against closed-source models remains an important direction for future work.

An interesting observation from comparing Tables I and II is that, on average, attacks against closed-source models are more successful than those against open-source models fine-tuned for vulnerability detection. The latter are more sensitive to vulnerability patterns due to fine-tuning, making semantically equivalent modifications less effective. In contrast, ChatGPT, as a modern multitask model, is more susceptible to such perturbations. Previous work also shows that specific open-source models better capture vulnerability patterns, thus outperforming ChatGPT-based models [32].

> **Conclusion**: Our proposed methods achieve higher AS than the baselines on closed-source models, although with a higher AQS in some cases. This is attributed to their ability to explore a larger perturbation space, thereby enhancing attack success.

C. RQ3: To what extent do the proposed perturbation and search methods contribute to attack success?

Motivation. While RQ1 and RQ2 evaluated the overall effectiveness of our approach compared to state-of-the-art black-box attack methods, the performance of each individual transformation method when combined with specific search strategies remains unexplored. Assessing the effectiveness of each transformation in isolation is essential to pinpoint the most impactful perturbation techniques and to understand how different search algorithms explore the perturbation space. In this section, we use LineVul [16], a widely recognized vulnerability detection model, as the target to investigate whether a single transformation method, such as VR, CR, IRC, LET, or MR, combined with one of three search algorithms (*Random*, *Greedy*, or *Genetic*), can produce effective attacks.

Approach. To evaluate the effectiveness of the proposed transformations, we examine the impact of applying a single transformation combined with different search algorithms to attack the victim model. In addition to the greedy search and genetic algorithm employed in this paper, we include random search as a baseline for comprehensive comparison. The *Random* search uniformly samples parameter values (e.g., tokens or target statements). The *Greedy* Search iteratively selects the modification that maximally increases similarity to non-vulnerable references, and the *Genetic* algorithm explores a larger search space by evolving a population of candidate perturbations. We use the same testing set from *big_vul* (see Section VI-A), where each sample is initially correctly classified by LineVul. Each sample is subjected to a single transformation, and we determine attack performance and efficiency by calculating the AS and AQS metrics.

Results. Figure 10 presents a heatmap showing the AS and AQS for each transformation under three search algorithms. The results show that *Random* search achieves relatively high AS when combined with IRC or MR, but struggles with VR, CR, or LET. Moving from *Random* to *Greedy* search improves AS across all transformations, particularly for IRC and CR, indicating that a more guided approach to selecting perturbation locations significantly bolsters the attack's effectiveness. The *Genetic* algorithm, which explores solutions through iterative evolution, achieves the highest AS for most transformations at the cost of increased AQS. For instance, VR shows an increase in AS from 6.60% to 14.20%, when switching from Greedy to Genetic, while its AQS rises from 13.6 to 137.5. Similarly, for IRC, the AS improves from 29.4% to 35.8%, while the AQS increases from 93.3 to 208.4, indicating a trade-off between attack effectiveness and query efficiency.

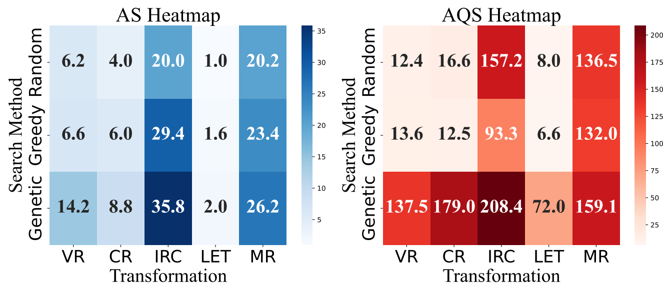


Fig. 10. Heatmap visualization of AS (%) and AQS across single transformations with three search methods

Among all transformations, IRC and MR are generally the most effective perturbation methods. In particular, IRC achieves the best performance across all combinations when paired with greedy search or genetic algorithm, likely due to the substantial modifications introduced by the newly inserted code statements; however, this often incurs higher query overhead, especially under the genetic algorithm. MR's effectiveness is attributed to its applicability across various program locations, which expands the range of perturbation opportunities. In contrast, LET and CR yield comparatively lower success rates, suggesting that these transformations alone may not sufficiently disrupt the learned semantic representations of the program, making it challenging to deceive the detection model. VR exhibits intermediate effectiveness between LET/CR and IRC/MR. By replacing identifiers with tokens that exhibit non-vulnerable patterns, it introduces measurable changes in the program representation that shift samples toward the non-vulnerable side of the decision boundary, thereby improving evasion success while avoiding the higher query costs.

Conclusion: Each transformation demonstrates a certain attack success rate, but when paired with the improved greedy search and genetic algorithm, their effectiveness is significantly enhanced, leading to improved overall attack performance against the target vulnerability detector.

D. RQ4: How does varying the number of modifications in the proposed attack methods affect attack success rates?

Motivation. While the experiments in RQ3 demonstrated that individual transformations can successfully generate adversarial samples when combined with a specific search algorithm, each sample was limited to a single perturbation. However, in practice, adversaries can employ multiple transformations at different modification positions to better shift a vulnerable sample across the model's decision boundary. In this context, our motivation is to investigate how varying the number of modifications (i.e., transformation sequence length, k) affects the attack success rate when employing multiple transformations. It is important to note that all modifications are semantic-preserving, meaning that even with a higher number of perturbations, the original functional semantics remain intact. However, increasing k poses a trade-off: fewer modifications may lead to lower AS, while too many may compromise stealthiness despite improved effectiveness.

Approach. To address this RQ, we evaluate our combination attack methods on the LineVul model by varying the transformation sequence length from $k = 1$ to $k = 15$. For each value of k, we apply our method that integrates five distinct perturbation types, i.e., VR, CR, IRC, LET and MR, using three different search algorithms: Random Search, Greedy Search, and Genetic Algorithm. For each (search method, k) pair, we record the resulting AS and average queries per sample (AQS) to measure effectiveness and efficiency.

TABLE III
EFFECTIVENESS OF COMBINATION ATTACK METHOD WITH DIFFERENT SEARCH METHODS AND TRANSFORMATION LENGTHS

k	Random Search		Greedy Search		Genetic Algorithm	
	AS (%)	AQS	AS (%)	AQS	AS (%)	AQS
1	13.4	29.1	**25.2**	36.0	22.0	47.5
2	16.6	50.1	**28.6**	41.6	25.6	57.7
3	19.2	70.8	29.6	44.6	**30.2**	71.5
4	20.4	77.6	31.2	53.8	**32.8**	81.5
5	21.0	83.5	32.4	59.9	**37.0**	101.4
6	21.8	96.9	33.4	69.8	**39.4**	115.8
7	22.8	122.0	34.2	79.5	**41.2**	127.4
8	23.8	140.3	34.2	79.3	**43.2**	139.4
9	24.8	157.0	35.2	96.0	**45.0**	153.0
10	26.4	184.8	36.0	109.5	**46.4**	165.7
11	27.4	200.3	36.4	114.0	**47.4**	174.5
12	28.4	227.7	37.2	124.3	**48.6**	186.3
13	28.6	230.3	38.0	137.1	**48.8**	188.7
14	29.4	246.8	38.6	152.4	**48.8**	188.7
15	29.6	252.6	39.6	176.3	**49.0**	191.2

Results. Table III and Figure 11 illustrate how the performance of our combination attack methods across different search strategies varies as the transformation sequence length k changes. We observe that with $k = 1$, our greedy search-based method achieves an AS of 25.2%, which is higher than the results reported for the state-of-the-art baselines DIP (19.6%) and CODA (14.4%) in Table I, corresponding to relative improvements of 28.6% and 75.0%, respectively. These results indicate that even when limited to a single modification, our method achieves better performance than baseline approaches that rely on multiple perturbations.

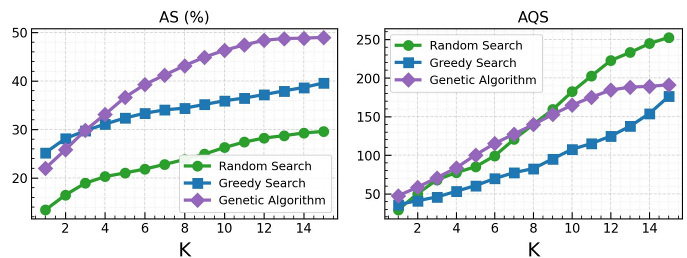


Fig. 11. Line chart of AS and AQS for our combination attack methods across different search strategies and transformation lengths (k)

In addition, we observe that for a small number of modifications (e.g., $k < 3$), the greedy search achieves relatively high AS with low query overhead. However, as k increases, the Genetic Algorithm consistently outperforms other methods in terms of AS, albeit at the cost of increased AQS. For example, at $k = 15$, the genetic algorithm achieves an AS of 49.0% with an AQS of 191.2, which is notably higher than the AS

achieved by greedy search (39.6%) under the same conditions. This indicates that the greedy search-based method, guided by our similarity-based estimation of statement importance, can effectively target high-impact code locations to produce immediate adversarial gains. However, since greedy search selects the locally optimal modification at each step, it may fail to identify globally optimal transformation sequences. In contrast, the Genetic Algorithm is better suited for exploring a broader perturbation space, allowing it to discover more effective combinations of modifications. Moreover, Figure 11 indicates that the performance of the Genetic Algorithm begins to stabilize as k increases, with no further gains in AS beyond a certain point. This suggests that more modifications do not lead to continuous performance improvements. In addition, it is important to consider the following trade-off: while allowing more modifications can enhance AS, it also increases the query cost and may raise the risk of detection.

Conclusion: Even with a single transformation, our method achieves promising performance. Furthermore, for small modifications, the greedy search outperforms the genetic algorithm while incurring lower query overhead. In contrast, the genetic algorithm performs better with larger perturbations, though with higher query cost.

E. RQ5: To what extent do our attack methods generalize across datasets?

Motivation. In RQ1–RQ4, we have demonstrated the effectiveness of our attack methods using the *big_vul* dataset. While *big_vul* is the largest and most widely used real-world benchmark for vulnerability detection, it remains essential to investigate how well our attacks generalize to unseen code drawn from sources distinct from the victim model's training data. Such an evaluation reflects a realistic scenario in which attackers typically lack access to target-model datasets.

Approach. To address this RQ, we evaluate our methods against the LineVul model using vulnerable examples from ICVul [46], another large and diverse dataset in addition to *big_vul*. Following the *big_vul* evaluation protocol, we randomly select 500 vulnerable examples from ICVul that are correctly classified by LineVul and do not overlap with the detection model's training set. Note that LineVul is not retrained on ICVul, as our goal is to assess attack transferability rather than re-optimize the model. We report both AS and AQS to assess performance under this cross-dataset setting.

Results. Table IV shows that both Greedy-Combo and Genetic-Combo outperform the state-of-the-art baselines DIP and CODA by a significant margin. Specifically, Greedy-Combo achieves an AS of 57.4%, representing relative improvements of 80.5% over DIP (31.8%) and 195.9% over CODA (19.4%). Genetic-Combo attains an AS of 67.8%, improving over DIP and CODA by 113.2% and 249.5%, respectively. In terms of efficiency, Genetic-Combo requires only 195.4 queries on average, which is 47.9% fewer than DIP (374.9) and 19.4% fewer than CODA (242.5).

These results demonstrate that our methods remain effective even without prior knowledge about the distribution of the target model's training data, primarily because the proposed similarity-based strategy, which is independent of the training set and model internals, guides perturbations through comparison with reference samples to craft effective adversarial examples. In addition, the proposed transformations enlarge the perturbation space, further improving attack success rates.

TABLE IV
ATTACK SUCCESS RATE (AS) AND AVERAGE QUERIES PER SAMPLE (AQS) OF METHODS AGAINST THE LINEVUL MODEL IN A CROSS-DATASET SETTING

Type	Method	AS (%)	AQS
Baselines	DIP	31.8	374.9
Baselines	CODA	19.4	242.5
Our Method	Greedy-Combo	57.4	256.0
Our Method	Genetic-Combo	**67.8**	**195.4**

Conclusion: Our attack methods generalize well to external datasets (e.g., ICVul), enabling realistic evaluation of model vulnerabilities to evasion attacks under cross-dataset settings and demonstrating the robustness and practicality of our approach.

VIII. DISCUSSION

A. Can the proposed methods effectively attack the detection model by inducing false positives on non-vulnerable code?

Previous experiments have demonstrated that our methods effectively generate adversarial examples from vulnerable code samples. Nevertheless, it remains unclear whether these methods can also perturb non-vulnerable code samples into being misclassified as vulnerable. Investigating this capability is crucial, as the misclassification of secure code as vulnerable can impose unnecessary burdens on developers, leading to increased review, verification, and debugging efforts. Such misclassification consequently raises software maintenance costs and reduces trust in vulnerability detection systems. In addition, this investigation offers insight into how well our perturbation strategies shift decision boundaries, causing safe code to resemble vulnerable code in the model's feature space.

To this end, we construct a dataset by randomly selecting 250 non-vulnerable samples, all of which are correctly classified by the victim model, from the test set of *big_vul*. For each non-vulnerable sample, our methods employ an adapted similarity-based mechanism in which reference samples are drawn from vulnerable code, thereby guiding the perturbation process to shift the representation of non-vulnerable code toward that of vulnerable code. The other perturbation strategies and search algorithms in our methods remain unchanged.

Table V summarizes the performance of the proposed combination attack methods on non-vulnerable samples (NVS). For comparative clarity, the corresponding results for vulnerable samples (VS), previously discussed in earlier sections, are also included. Notably, our method achieves considerably higher attack success rates on NVS compared to VS. For example, under Greedy Search, the AS for NVS is 76.4%, substantially greater than the 39.6% obtained for VS. Similarly,

TABLE V
PERFORMANCE OF COMBINATION ATTACK METHOD ON VULNERABLE-SAMPLE ATTACKS(VSA) AND NON-VULNERABLE-SAMPLE ATTACKS(NVSA)

Method	VSA		NVSA	
	AS (%)	AQS	AS (%)	AQS
Random Search	29.6	252.6	41.6	278.4
Greedy Search	39.6	176.3	76.4	239.5
Genetic Algorithm	49.0	191.2	77.6	152.2

when utilizing the Genetic Algorithm, the AS increases to 77.6% on NVS, compared to 49.0% on VS. These findings indicate that NVS is more sensitive to adversarial perturbations. One possible explanation is that NVS, having been classified correctly despite potential ambiguities in their features, may reside closer to the decision boundary; therefore, even minimal perturbations, especially when informed by our similarity-based estimation of statement importance, can more readily flip their classification. In contrast, VS tend to have stronger signals that reinforce their labels, making them relatively harder to shift with perturbation.

B. How are transformations designed in our attack methods, and how do additional ones perform?

In this paper, our methods employ five transformations, including VR, CR, IRC, LET, and MR. These transformations are chosen because they have been widely adopted in prior work [21], [39] and have demonstrated effectiveness in adversarial machine learning. To adapt them for the vulnerability detection task, we enhance their design by first performing token extraction to identify non-vulnerable patterns (i.e., characteristic tokens), and then applying transformations based on these extracted patterns to improve attack effectiveness.

Following prior work [47], our transformations can be broadly classified into two categories: token-level (e.g., VR and MR) and block-level (e.g., CR, MR, and LET). Beyond these, existing studies [47] have also applied statement-level transformations, such as AssignExpression, ConditionalExpression, and BinaryExpression, to perturb program code in adversarial attack settings. For example, in the AssignExpression transformation, the statement $x = x + y;$ can be semantically replaced with $x+ = y;$. In this subsection, we evaluate each statement-level perturbation individually and examine whether combining them can further enhance the performance of our methods. Table VI presents the results on the LineVul model for the dataset described in Section VI-A.

TABLE VI
PERFORMANCE OF STATEMENT-LEVEL PERTURBATIONS WITH GREEDY SEARCH AGAINST LINEVUL ON *big_vul* VULNERABLE SAMPLES

Perturbation	Example	AS (%)	AQS
AssignExpression	x = x + y; → x += y;	0.2	2.0
ConditionalExpression	if (a > 0) → if (0 < a)	0.4	2.5
BinaryExpression	int x = a + b; → int x = a - (-b);	0.8	3.0

As shown in Table VI, all three statement-level perturbations achieve low AS values (≤0.8%) that are consistently below those of our five transformations. Their poor performance is partly due to strict applicability (e.g., only 32 samples in our evaluation set match the AssignExpression pattern) and partly to their limited influence on the code's feature vector, which reduces attack effectiveness. Furthermore, combining these perturbations with our five transformations under greedy search yields an AS of 39.4%, slightly lower than the 39.6% achieved by our original greedy search-based method. This result indicates that simply increasing the number of transformations does not necessarily enhance attack performance, as incorporating weaker perturbations may be counterproductive.

C. How do tokenization and representation in the similarity-based strategy influence method performance?

We have demonstrated the effectiveness of our proposed combination methods, based on greedy search and genetic algorithms, across RQ1–RQ5. A key design in both methods is the similarity-based strategy for estimating statement importance or model confidence under black-box settings. However, the performance of this strategy may be affected by two critical factors: tokenization and representation. In the default setting, we apply lexical analysis to convert source code into syntactic tokens and use TF–IDF to construct vector representations. To assess the effectiveness of these choices, we conduct experiments with two alternatives: (1) replacing lexical analysis with the BPE tokenizer, which is widely adopted by most LLMs for vulnerability detection (e.g., LineVul and StagedVulBERT); and (2) replacing the TF–IDF representation with semantic vectors generated by a pretrained CodeBERT[2]. Results against the LineVul model are reported in Table VII.

TABLE VII
PERFORMANCE OF OUR GENETIC-ALGORITHM-BASED METHOD AND ITS VARIANTS WITH BPE TOKENIZATION OR SEMANTIC VECTORS

Method	AS (%)	AQS
Genetic-Combo (Original)	**49.0**	**191.2**
Genetic-Combo (with BPE Tokenization)	45.4	204.6
Genetic-Combo (with Semantic Vectors)	48.6	193.0

As shown in the table, our method with the default settings achieves the highest performance, with an AS of 49.0% and an AQS of 191.2. When lexical analysis is replaced by BPE tokenization, performance drops by 7.3%. This reduction can be explained by the richer semantic information preserved in tokens extracted through lexical analysis, whereas subtokens generated by BPE often lack coherent semantic impact. Consequently, transformations based on candidate tokens obtained through BPE are less effective at generating adversarial examples compared to those derived from lexical analysis.

We also observe a slight performance reduction when the TF–IDF representation is replaced by semantic vectors. This is because semantic vectors are generally insensitive to small modifications (e.g., at the single-token level), which slightly reduces the accuracy of similarity estimation. Nonetheless, owing to the correlation of similarity estimates across the two representations, the attack still performs satisfactorily.

[2]https://huggingface.co/microsoft/codebert-base

Considering both effectiveness and efficiency, TF–IDF remains the preferred representation in our methods.

D. How to choose between the proposed combination attack methods?

In this paper, we propose two combination attack methods: one based on greedy search and the other on a genetic algorithm. Both use the same five program transformations but differ in how they combine them to explore the perturbation space. More details on the differences are in Section V-D.

The two methods offer distinct advantages across application scenarios. As demonstrated in Section VII-D on a widely used vulnerability detector, when fewer modifications are allowed, the greedy search method outperforms the genetic algorithm with lower query cost. In contrast, the genetic algorithm achieves better performance with larger perturbations at a higher query cost. These findings provide practical guidance: greedy search is preferable for robustness evaluation under strict perturbation limits to preserve stealth, while the genetic algorithm is suitable for stress testing with relaxed constraints, as it can better exploit the available perturbation space.

To quantify the impact of the number of perturbations k, we present the AS and AQS for each method across varying values of k in Table III. This table provides more precise guidance for selecting an appropriate attack method under varying requirements. For example, if only one modification is allowed, the greedy search-based method is preferable, achieving 25.2% AS with an AQS of 36.0, outperforming the genetic algorithm. However, if a larger perturbation budget (e.g., $k \geq 3$) is acceptable, the genetic algorithm shows greater advantages, and the performance difference between the two methods becomes more pronounced as k increases.

Therefore, we believe that neither method holds an absolute advantage; instead, the choice should be guided by the specific application scenario, with our experimental results serving as a reference for selecting the most suitable approach.

E. Investigation of successful and failed attack cases by the greedy search-based method

Our experimental results in RQ1–RQ5 demonstrate that our combination attack method achieves high attack success rates on vulnerable samples. For instance, in the code snippet shown in Figure 12 from Google Chrome, associated with CVE-2016-1658 (CWE-284), our method successfully generates an adversarial example through the insertion of a redundant `print` statement. This successful evasion indicates that the current transform-based detector (LineVul) suffers from a lack of robustness and generalization, as subtle perturbations can shift vulnerable samples across the decision boundary.

However, there still exist cases where our method fails to generate effective adversarial examples. For example, Figure 13 depicts a vulnerable code sample from PHP 5.5.x associated with CVE-2014-2020 (CWE-189). In this instance, even after introducing a large number of redundant statements, the victim model continues to classify the sample as vulnerable. This outcome may be attributed to the fact that this particular vulnerability pattern is prevalent in the

Original Code

```
1  void ExtensionOptionsGuest::DidNavigateMainFrame(
2  const content::LoadCommittedDetails &details,
3  const content::FrameNavigateParams &params)
4  {
5      if (attached())
6      {
7          auto guest_zoom_controller =
8              ui_zoom::ZoomController::FromWebContents(web_contents());
9          guest_zoom_controller->SetZoomMode(
10             ui_zoom::ZoomController::ZOOM_MODE_ISOLATED);
11         SetGuestZoomLevelToMatchEmbedder();
12         if (params.url.GetOrigin() != options_page_.GetOrigin())
13         {
14             bad_message::ReceivedBadMessage(
15                 web_contents()->GetRenderProcessHost(),
16                 bad_message::EOG_BAD_ORIGIN);
17         }
18     }
19 }
```

Perturbed Code (Our method)

Inserting Redundant Code

```
1  void ExtensionOptionsGuest::DidNavigateMainFrame(
2      const content::LoadCommittedDetails &details,
3      const content::FrameNavigateParams &params)
4  {
5      if (attached())
6      {
7          auto guest_zoom_controller =
8              ui_zoom::ZoomController::FromWebContents(web_contents());
9          guest_zoom_controller->SetZoomMode(
10             ui_zoom::ZoomController::ZOOM_MODE_ISOLATED);
11         SetGuestZoomLevelToMatchEmbedder();
12         printf("guest_zoom_controller = %p\n",&guest_zoom_controller);
13         if (params.url.GetOrigin() != options_page_.GetOrigin())
14         {
15             bad_message::ReceivedBadMessage(
16                 web_contents()->GetRenderProcessHost(),
17                 bad_message::EOG_BAD_ORIGIN);
18         }
19     }
20 }
```

Fig. 12. A vulnerable code example perturbed by our greedy search-based method to successfully evade the LineVul detection model.

training dataset, leading the model to learn a highly sensitive representation that remains robust against perturbations. Additionally, while our method employs a similarity-based estimation of statement importance and model confidence to guide perturbations, there remains room for improvement, such as incorporating additional contextual features or refining the search strategies, to further enhance attack effectiveness.

IX. Limitations and Future Work

Although our experimental results demonstrate the effectiveness of our proposed black-box adversarial attack method, several limitations remain that open avenues for future research.

First, our study focuses exclusively on black-box evasion attacks, aiming to assess the robustness of LLM-based vulnerability detectors. While the results demonstrate that our method can effectively perturb vulnerable samples to evade detection, the attack goal primarily targets the misclassification of vulnerable code as non-vulnerable, as this case poses greater security risks than the reverse. We also include additional experiments in Section VIII-A analyzing the effectiveness of our attack method on non-vulnerable examples. Future work will further investigate such scenarios to gain deeper insight into their real-world implications and conduct more comprehensive empirical studies.

Second, although our method introduces five transformation strategies (Variable Renaming, Constant Replacement, Inserting Redundant Code, Loop Equivalence Transformation, and Macro Replacement) to generate adversarial examples,

Original Code

```
PHP_FUNCTION(imagesetstyle)
{
    ...
    zend_hash_internal_pointer_reset_ex(HASH_OF(styles), &pos);
    for (index = 0;; zend_hash_move_forward_ex(HASH_OF(styles),
                                               &pos)){
        zval ** item;
        if (zend_hash_get_current_data_ex(HASH_OF(styles),
                  (void **) &item, &pos) == FAILURE) {
            break;
        }
        convert_to_long_ex(item);
        stylearr[index++] = Z_LVAL_PP(item);
    }
    gdImageSetStyle(im, stylearr, index);
    efree(stylearr);
    RETURN_TRUE;
}
```

Perturbed Code (Our method)

```
PHP_FUNCTION(imagesetstyle)
{
    ...
    zend_hash_internal_pointer_reset_ex(HASH_OF(styles), &pos);
    printf("im = RenderFrameImpl, %p" , &im);
    for (index = 0;; zend_hash_move_forward_ex(HASH_OF(styles),
                                               &pos)){
        printf("im = Tags, %p", & im);
        zval ** item;
        printf("im = int64, %p", & im);
        if (zend_hash_get_current_data_ex(HASH_OF(styles),
                  (void **) &item, &pos) == FAILURE) {
            printf("im = m_result, %p", & im);
            break;
        }
        convert_to_long_ex(item);
        printf("im = nullptr, %p", & im);
        stylearr[index++] = Z_LVAL_PP(item);
        printf("im = ofproto, %p", & im);
    }
    gdImageSetStyle(im, stylearr, index);
    efree(stylearr);
    printf("im = RenderViewImpl, %p", & im);
    RETURN_TRUE;
}
```

Fig. 13. A vulnerable code example perturbed by our greedy search-based method that failed to evade the LineVul detection model.

our results indicate that combining multiple transformations yields significant improvements. Future research will explore additional and potentially more diverse perturbation techniques to further enhance attack performance.

Third, we propose a similarity-based technique that estimates statement importance and approximates model confidence by computing cosine similarity over TF-IDF vector representations of code samples. This technique effectively guides perturbation selection and, as shown in Section VIII-C, outperforms alternative configurations of our method that employ semantic vector representations generated by pre-trained models. Nonetheless, other representations, such as contextual embeddings derived from graph-based encodings, may quantify statement importance more accurately by incorporating structural information. Future work will investigate these alternatives to further enhance our perturbation strategy.

Fourth, our greedy search algorithm, although effective at selecting high-impact modifications, suffers from the well-known limitation of converging to local optima. Alternative search methods, such as beam search, may provide a more global exploration of the perturbation space. Similarly, the performance of our genetic algorithm depends heavily on population diversity; incorporating techniques such as multi-point crossover may broaden the search space and improve overall performance. In future work, we will investigate more advanced search algorithms to further enhance the success rate of adversarial example generation.

Finally, our experiments are limited to C/C++ code. Given the diversity of programming languages in real-world systems, extending our approach to other languages is an important future work. Moreover, as more advanced and varied vulnerability detection models are proposed, it will be crucial to assess the robustness of our approach against this broader spectrum of models with different architectures and parameter settings. Furthermore, we plan to explore complementary defense techniques, such as leveraging statistical language models to measure the perplexity of perturbed code, to distinguish adversarially modified programs from benign ones.

X. Related Work

Following prior work [18], adversarial attacks on code models are categorized into *white-box* and *black-box* approaches.

A. White-box Attacks

White-box attacks assume that the adversary has full access to the target model's architecture, parameters, and gradients. This comprehensive knowledge allows attackers to craft precise adversarial examples by exploiting model vulnerabilities.

One prominent approach in white-box attacks is the embedding-level gradient-based method. This technique computes gradients with respect to embedding vectors and projects them back to the input space to identify critical tokens for modification [22]. For instance, Zhang et al. [39] propose an identifier renaming attack where they calculate the importance of each variable by measuring the gradient's influence on the model's loss function. By selecting identifiers with the highest impact, they effectively generate adversarial code samples that mislead the model. Another method is the continuation-based approach, which formulates the discrete problem of code modification into a continuous optimization problem. Srikant et al. [48] introduce an attack that optimizes over a continuous space to find optimal perturbations. They apply techniques like projected gradient descent to solve the optimization problem, enabling the generation of adversarial examples through code obfuscations while maintaining functional correctness.

B. Black-box Attacks

Black-box attacks operate without access to the internal parameters or architecture of the target model, relying solely on the model's outputs. This makes them more applicable in real-world scenarios where models are deployed as services.

1) Transfer-based Black-box Attacks: In transfer-based attacks, adversaries train surrogate models to craft adversarial examples that are expected to transfer to the victim models [49], [50]. The effectiveness of these attacks depends on the similarity between the surrogate and target models' training data, which can be categorized into: (1) Same-dataset Setting: Quiring et al. [49] study attacks where the surrogate and target models are trained on identical datasets. They utilize semantic-preserving code transformations and Monte Carlo tree search [51] to generate adversarial examples that mislead authorship attribution models. (2) Same-domain Setting: Liu et al. [50] explore attacks where models are trained on datasets from

the same domain but not necessarily identical. They propose SCAD, an attack that uses dead code insertion, identifier renaming, and structural transformations to deceive authorship attribution classifiers. Their study reveals that an optimal overlap ratio between surrogate and target training data enhances attack success. (3) Cross-domain Setting: Pour et al. [52] and Yang et al. [14] demonstrate that adversarial examples crafted using open-source pre-trained models can transfer to victim models trained on different data distributions. These approaches are particularly relevant for attacking LLMs where training data may not be publicly available [53].

2) Query-based Black-box Attacks: Query-based attacks generate adversarial examples by interacting with the target model and observing its outputs. Based on the accessible feedback, they are divided into: (1) Score-based Attacks: These assume access to the model's confidence scores or output probabilities, using them to guide perturbation [20], [21], [23], [49], [54]–[57]. These attacks employ various strategies, such as random search methods like Monte Carlo tree search [49] and random sampling [20], to explore the perturbation space. Explanation-guided techniques apply explainable AI to identify impactful perturbation sites for identifier renaming or structural transformations [21], [55]. Beam search algorithms have been introduced to efficiently optimize perturbation sequences [56]. Learning-based approaches, including reinforcement learning, iteratively select perturbations to maximize adversarial success [57]. Additionally, some attacks leverage reference examples from external datasets to enhance the stealthiness and naturalness of adversarial samples [23]. Although score-based attacks can be effective, their dependence on confidence scores limits applicability in scenarios where only hard-label predictions are available, making them less effective against fully black-box models [19].

(2) **Decision-based Attacks**: The adversary receives only hard-label predictions (e.g., vulnerable or non-vulnerable) without any confidence scores [19]. Na et al. [19] propose DIP (see SectionVI-C), which begins with a successful adversarial example and iteratively enhances its similarity to the original code while maintaining misclassification.

Our proposed methods fall under the category of decision-based black-box attacks, which represent a more realistic setting for targeting vulnerability detectors with only query access to hard-label outputs. Unlike existing decision-based black-box approaches that rely on surrogate models or employ limited perturbation strategies, we introduce a novel similarity-guided technique that (1) estimates statement importance to determine modification locations, and (2) guides the search algorithm to effectively select diverse, semantic-preserving code transformations, achieving strong attack performance against both open-source and closed-source vulnerability detectors.

XI. Conclusion

In this paper, we introduce SVulAttack, a black-box adversarial attack framework that generates adversarial samples to evade both open-source vulnerability detectors (LineVul, StagedVulBERT, Code Llama, Deepseek-Coder) and closed-source LLM-based tools (GPT-5 nano, GPT-4o, GPT-4o-mini, Claude Sonnet 4). SVulAttack employs five perturbation strategies combined with either greedy search or genetic algorithm to iteratively explore a large space of candidate modifications. Experimental results show that, in some cases, SVulAttack achieves increases of 102.0% and 175.0% in attack success rate over the DIP and CODA baselines, respectively. These findings demonstrate the effectiveness of our approach in successfully fooling the target models. Therefore, SVulAttack can serve as a powerful tool to assess the robustness of vulnerability detection models and improve their security. The limitations and future work described in Section IX provide open problems for future research. A key focus will be exploring defense strategies, such as automatically detecting and removing adversarial or redundant code prior to detection.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant Nos.62302125 and 62272132), the Heilongjiang Postdoctoral Fund (Grant No.LBH-Z23019), and the Key technical projects of ShenZhen (Grant No.JSGG20211108092802003).

References

[1] F. Lomio, E. Iannone, A. De Lucia, F. Palomba, and V. Lenarduzzi, "Just-in-time software vulnerability detection: Are we there yet?" *Journal of Systems and Software*, vol. 188, p. 111283, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222000437

[2] National Institute of Standards and Technology, "Guide for conducting risk assessments," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. Special Publication 800-30, Revision 1, 2012, accessed: 2025-09-18. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-30r1.pdf

[3] Y. Jiang, Z. Qu, C. Treude, X. Su, and T. Wang, "Enhancing fine-grained vulnerability detection with reinforcement learning," *IEEE Transactions on Software Engineering*, 2025.

[4] Y. Jiang, Y. Zhang, X. Su, C. Treude, and T. Wang, "Stagedvulbert: Multi-granular vulnerability detection with a novel pre-trained code model," *IEEE Transactions on Software Engineering*, pp. 1–18, 2024.

[5] X. Yin, C. Ni, and S. Wang, "Multitask-based evaluation of open-source llm on software vulnerability," *IEEE Transactions on Software Engineering*, 2024.

[6] The Big Sleep Team, "From naptime to big sleep: Using large language models to catch vulnerabilities in real-world code," https://googleprojectzero.blogspot.com/2024/10/from-naptime-to-big-sleep.html, November 2024, accessed: 2024-11-19.

[7] S. Liu, D. Cao, J. Kim, T. Abraham, P. Montague, S. Camtepe, J. Zhang, and Y. Xiang, "{EaTVul}:{ChatGPT-based} evasion attack against software vulnerability detection," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 7357–7374.

[8] M. Cheng, J. Yi, P.-Y. Chen, H. Zhang, and C.-J. Hsieh, "Seq2sick: Evaluating the robustness of sequence-to-sequence models with adversarial examples," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 04, 2020, pp. 3601–3608.

[9] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[10] X. Ling, S. Ji, J. Zou, J. Wang, C. Wu, B. Li, and T. Wang, "Deepsec: A uniform platform for security analysis of deep learning model," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 673–690.

[11] A. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 427–436.

[12] C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song, "Generating adversarial examples with adversarial networks," *arXiv preprint arXiv:1801.02610*, 2018.

[13] W. Zheng, Y. Jiang, and X. Su, "Vu1spg: Vulnerability detection based on slice property graph representation learning," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 457–467.

[14] Y. Yang, H. Fan, C. Lin, Q. Li, Z. Zhao, and C. Shen, "Exploiting the adversarial example vulnerability of transfer learning of source code," *IEEE Transactions on Information Forensics and Security*, 2024.

[15] M. Fu, C. Tantithamthavorn, T. Le, Y. Kume, V. Nguyen, D. Phung, and J. Grundy, "Aibughunter: A practical tool for predicting, classifying and repairing software vulnerabilities," *Empirical Software Engineering*, vol. 29, no. 1, p. 4, 2024.

[16] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[17] C. A. Choquette-Choo, F. Tramer, N. Carlini, and N. Papernot, "Label-only membership inference attacks," in *International conference on machine learning*. PMLR, 2021, pp. 1964–1974.

[18] Y. Yang, H. Fan, C. Lin, Q. Li, Z. Zhao, C. Shen, and X. Guan, "A survey on adversarial machine learning for code data: Realistic threats, countermeasures, and interpretations," *arXiv preprint arXiv:2411.07597*, 2024.

[19] C. Na, Y. Choi, and J.-H. Lee, "Dip: Dead code insertion based black-box attack for programming language model," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 7777–7791.

[20] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 01, 2020, pp. 1169–1176.

[21] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1482–1493.

[22] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[23] Z. Tian, J. Chen, and Z. Jin, "Code difference guided adversarial example generation for deep code models," in *38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 850–862.

[24] C. Luo, Q. Lin, W. Xie, B. Wu, J. Xie, and L. Shen, "Frequency-driven imperceptible adversarial attack on semantic similarity," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 15 315–15 324.

[25] H. Sun, L. Cui, L. Li, Z. Ding, Z. Hao, J. Cui, and P. Liu, "Vdsimilar: Vulnerability detection based on code similarity of vulnerabilities and patches," *Computers & Security*, vol. 110, p. 102417, 2021.

[26] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 201–213.

[27] J. Li, S. Ji, T. Du, B. Li, and T. Wang, "Textbugger: Generating adversarial text against real-world applications," *arXiv preprint arXiv:1812.05271*, 2018.

[28] J. Gao, J. Lanchantin, M. L. Soffa, and Y. Qi, "Black-box generation of adversarial text sequences to evade deep learning classifiers," in *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2018, pp. 50–56.

[29] R. Sennrich, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[30] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[31] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[32] M. Fu, C. K. Tantithamthavorn, V. Nguyen, and T. Le, "Chatgpt for vulnerability detection, classification, and repair: How far are we?" *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 632–636, 2023.

[33] X. Zhou, T. Zhang, and D. Lo, "Large language model for vulnerability detection: Emerging results and future directions," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 47–51.

[34] G. Apruzzese, H. S. Anderson, S. Dambra, D. Freeman, F. Pierazzi, and K. Roundy, ""real attackers don't compute gradients": bridging the gap between adversarial ml research and practice," in *2023 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, 2023, pp. 339–364.

[35] "Nvd," https://nvd.nist.gov/.

[36] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[37] G. Ramakrishnan, J. Henkel, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," *arXiv preprint arXiv:2002.03043*, 2020.

[38] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.

[39] H. Zhang, Z. Fu, G. Li, L. Ma, Z. Zhao, H. Yang, Y. Sun, Y. Liu, and Z. Jin, "Towards robustness of deep program processing models—detection, estimation, and enhancement," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–40, 2022.

[40] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.

[41] R. Maheshwary, S. Maheshwary, and V. Pudi, "Generating natural language attacks in a hard label black box setting," *arXiv preprint arXiv:2012.14956*, 2020.

[42] S. Ren, Y. Deng, K. He, and W. Che, "Generating natural language adversarial examples through probability weighted word saliency," in *Proceedings of the 57th annual meeting of the association for computational linguistics*, 2019, pp. 1085–1097.

[43] D. Jin, Z. Jin, J. T. Zhou, and P. Szolovits, "Is bert really robust? a strong baseline for natural language attack on text classification and entailment," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 05, 2020, pp. 8018–8025.

[44] MITRE Corporation, "2024 CWE Top 25 Most Dangerous Software Weaknesses," https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html, 2024, accessed: 2025-04-12.

[45] N. Japkowicz and M. Shah, *Evaluating learning algorithms: a classification perspective*. Cambridge University Press, 2011.

[46] C. Lu, T. Li, T. Dehaene, and B. Lagaisse, "Icvul: A well-labeled c/c++ vulnerability dataset with comprehensive metadata and vccs," in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 2025, pp. 154–158.

[47] P. Xue, L. Wu, Z. Yang, Z. Yu, Z. Jin, G. Li, Y. Xiao, S. Liu, X. Li, H. Lin *et al.*, "Exploring and lifting the robustness of llm-powered automated program repair with metamorphic testing," *arXiv preprint arXiv:2410.07516*, 2024.

[48] S. Srikant, S. Liu, T. Mitrovska, S. Chang, Q. Fan, G. Zhang, and U.-M. O'Reilly, "Generating adversarial computer programs using optimized obfuscations," *arXiv preprint arXiv:2103.11882*, 2021.

[49] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 479–496.

[50] Q. Liu, S. Ji, C. Liu, and C. Wu, "A practical black-box attack on source code authorship identification classifiers," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3620–3633, 2021.

[51] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[52] M. V. Pour, Z. Li, L. Ma, and H. Hemmati, "A search-based testing framework for deep neural networks of source code embedding," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 36–46.

[53] C. Zhang, Z. Wang, R. Mangal, M. Fredrikson, L. Jia, and C. Pasareanu, "Transfer attacks and defenses for large language models on coding tasks," *arXiv preprint arXiv:2311.13445*, 2023.

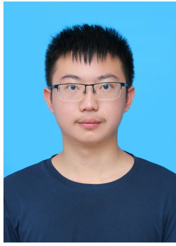
[54] N. Chen, Q. Sun, J. Wang, M. Gao, X. Li, and X. Li, "Evaluating and enhancing the robustness of code pre-trained models through structure-aware adversarial samples generation," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 14 857–14 873.

[55] T.-D. Nguyen, Y. Zhou, X.-B. D. Le, and D. Lo, "Adversarial attacks on code models with discriminative graph patterns," *arXiv preprint arXiv:2308.11161*, 2023.

[56] Y. Choi, H. Kim, and J.-H. Lee, "Tabs: Efficient textual adversarial attack for pre-trained nl code model using semantic beam search," in *2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2022, pp. 5490–5498.

[57] J. Tian, C. Wang, Z. Li, and Y. Wen, "Generating adversarial examples of source code classification models via q-learning-based markov decision process," in *IEEE International Conference on Software Quality, Reliability, and Security (QRS)*, 2021, pp. 807–818.

Yuan Jiang is an assistant professor at the School of Computer Science and Technology, Harbin Institute of Technology, and a postdoctoral researcher at Singapore Management University. His main research interests include pre-trained code language models, mining software repositories, software vulnerability detection, and large language model security.



Shan Huang is an undergraduate student at the School of Mathematics, Harbin Institute of Technology, under the supervision of Yuan Jiang. His primary research interests include software vulnerability detection, adversarial attacks, and software security.

Christoph Treude is an Associate Professor of Computer Science at Singapore Management University. His notable achievements include receiving an ARC Discovery Early Career Research Award (2018-2020) and securing funding from industry giants such as Google and Facebook. Treude has been honored with four best paper awards, including two ACM SIGSOFT Distinguished Paper Awards. Currently, he serves on the Editorial Boards of the IEEE Transactions on Software Engineering and the Springer journal on Empirical Software Engineering. Additionally, he is the Open Science Editor for the Elsevier Journal of Systems and Software and has chaired conferences such as ICSME 2020, ICPC 2023, and TechDebt 2023.

Xiaohong Su is a professor at the School of Computer Science and Technology, Harbin Institute of Technology. Her research interests include Intelligent software engineering, software vulnerability identification, code representation learning, bug triaging and localization, clone detection, and code search.

Tiantian Wang born in 1980. Received the Doctor's degree from Harbin Institute of Technology, Harbin, Heilongjiang, China, in 2009. Since 2013, she has been an Associate Professor in computer science department of Harbin Institute of Technology. Her current research interests are software engineering, program analysis and computer aided education.