Iterative Generation of Adversarial Example for Deep Code Models

Li Huang, Weifeng Sun, Meng Yan†
School of Big Data and Software Engineering, Chongqing University, Chongqing, China
{lee.h, weifeng.sun, mengy}@cqu.edu.cn

***Abstract*—Deep code models are vulnerable to adversarial attacks, making it possible for semantically identical inputs to trigger different responses. Current black-box attack methods typically prioritize the impact of identifiers on the model based on custom importance scores or program context and incrementally replace identifiers to generate adversarial examples. However, these methods often fail to fully leverage feedback from failed attacks to guide subsequent attacks, resulting in problems such as local optima bias and efficiency dilemmas. In this paper, we introduce ITGen, a novel black-box adversarial example generation method that iteratively utilizes feedback from failed attacks to refine the generation process. It employs a bitvector-based representation of code variants to mitigate local optima bias. By integrating these bit vectors with feedback from failed attacks, ITGen uses an enhanced Bayesian optimization framework to efficiently predict the most promising code variants, significantly reducing the search space and thus addressing the efficiency dilemma. We conducted experiments on a total of nine deep code models for both understanding and generation tasks, demonstrating ITGen's effectiveness and efficiency, as well as its ability to enhance model robustness through adversarial fine-tuning. For example, on average, ITGen improves the attack success rate by 47.98% and 69.70% over the state-of-the-art techniques (i.e., ALERT and BeamAttack), respectively.**

***Index Terms*—Adversarial Example, Deep Code Model, Iterative Generation**

I. Introduction

In recent years, there has been a rapid expansion in the development of Deep Code Models (DCM) [1]–[4]. These models aim to capture the intricate relationship between natural language and source code, holding the potential to automate various tasks within software engineering development involving code understanding (vulnerability prediction [5], clone detection [6]) and code generation (e.g., code summarization [7]). In particular, some industrial products have been released and received extensive attention, such as Copilot [8].

Unfortunately, recent studies indicate that DCMs are as vulnerable to adversarial attacks as traditional deep learning models in computer vision and natural language processing. An adversarial attack occurs when a DCM yields different results when presented with two semantically identical input programs, one of which is generated by performing certain semantic-preserving transformations to the other [9]–[12]. In this context, the model under attack is called the victim model, and the modified program is termed the adversarial example. This revelation is alarming, especially considering the extensive deployment of DCMs in various mission-critical applications, such as vulnerability detection [13], [14]. To elaborate, one can effortlessly generate adversarial examples that retain vulnerabilities while deceiving DCMs into incorrectly categorizing them as non-vulnerable, potentially facilitating unforeseen attacks on crucial systems and increasing the risk of security breaches.

One potential strategy to mitigate this threat is through adversarial retraining, which involves continuously training the models using generated adversarial examples to bolster their robustness [13], [15]. Indeed, several white-box and black-box adversarial attack methods specific to DCMs have been proposed recently [12], [16], [17]. However, white-box adversarial attack methods face notable limitations. First, DCMs are often deployed remotely, which restricts access to model architectures and model parameters, making black-box attacks more applicable to real scenarios [16]. Second, white-box attacks are often model-specific due to the diverse structures employed by different models, making it difficult to generalize these methods across different models effectively [17]. Additionally, among the semantic-preserving code transformations, identifier substitution has the most significant influence on the resilience of DCMs, as well as being the most undetectable transformation [18]. Therefore, black-box attack methods based on identifier substitution have become a valuable research avenue and several works have been proposed, such as ALERT [13] and BeamAttack [17].

Although existing black-box attack methods are effective to some extent, they still suffer from the following limitations: (**L.1 Feedback Neglect**) Current attack strategies fail to fully leverage failed attacks, e.g., evaluation information from code variants other than those that drop the ground truth probability most is also valuable for guiding subsequent attacks [19]; (**L.2 Local Optima Bias**) Many attack methods employing identifier substitution as their attack mechanism are susceptible to suboptimal solutions [17]. For example, ALERT [13] replaces identifiers sequentially. Once the top candidates encounter a local optimal solution, their ability to identify the global optimal solution diminishes since they do not iterate through the process of identifier replacement; (**L.3 Efficiency Dilemma**) There exists an absence of a trade-off between the effectiveness and efficiency of attack methods [17]. In other words, the methods boasting the highest success rates typically incur the highest number of queries to the victim model. Such massive model queries might be easily detected by the anti-attack security system, thereby not convenient in reality [16].

† Meng Yan is the corresponding author

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00086

To address the aforementioned challenges, in this paper, we propose an **IT**erative black-box adversarial example **Gen**eration method (i.e., **ITGen**) suitable for both understanding and generation tasks. It utilizes the failed attacks comprehensively and effectively mines potential adversarial examples, thus balancing effectiveness and efficiency. Specifically, to deal with feedback neglect (**L.1**), we first define the optimization objectives for adversarial attacks oriented towards understanding and generation tasks. To tackle these two black-box optimization problems [20], [21], we develop an iterative failed attack-based Bayesian optimization (BO) framework that iteratively mines widely neglected feedback information [13], [17]. Second, considering the local optima bias issue (**L.2**), unlike methods [13] that replace only one identifier in a single attack, we propose a compact bitvector-based code variant representation. This representation encodes the search space based on the identifier institution, allowing multiple identifiers to be replaced simultaneously during an attack, thereby improving attack efficiency and reducing the risk of falling into a suboptimal solution. Regarding the efficiency dilemma (**L.3**), compared to other black-box optimization algorithms (e.g., genetic algorithms [13]), the natural advantage of the BO algorithm is its ability to find an ideal solution with a relatively small number of queries on the victim model [22]. Additionally, we enhance the BO framework with an automatic relevance determination [23] kernel for discrete data, which supports dynamically calculating the similarities among different bitvector-based code variants. Moreover, to further boost the attack performance, we employ a diversity sampling strategy via determinantal point processes around the best observation point (promising code variant) found in each iteration of BO.

We conducted a comprehensive evaluation of ITGen across two types of code-based tasks (understanding and generation tasks) using five widely recognized pre-trained models: CodeBERT [2], GraphCodeBERT [3], CodeT5 [4], CodeGPT [24], and PLBART [1], involving a total of nine subjects. The results demonstrate the effectiveness and efficiency of ITGen. For example, on average, ITGen can improve attack success rates over ALERT and BeamAttack 47.98% and 69.70%, respectively. Additionally, ITGen requires 23.08% and 54.55% less time to successfully generate an adversarial example than ALERT and BeamAttack, respectively. We also explored the value of the adversarial examples generated by ITGen by using them to enhance the robustness of the victim models through adversarial fine-tuning. The results show that in understanding tasks, the models fine-tuned with ITGen can improve the accuracy of adversarial examples generated by ITGen, ALERT, and BeamAttack by 77.95%, 78.65%, and 78.35%, respectively. In generation tasks, it can improve the BLEU scores from 0 to 6.70, 6.49, and 6.51, respectively.

To sum up, this paper makes three major contributions:

- **Novel Perspective.** We propose a novel perspective to iteratively leverage the evaluation information of all failed attacks to guide the black-box adversarial example generation process for deep code models.
- **Performance Evaluation.** We evaluate ITGen on two types of code-based tasks (understanding and generation tasks) using five widely recognized pre-trained models, demonstrating its effectiveness and efficiency over two state-of-the-art techniques.
- **Open Science.** We release the source code and experiment data at the project homepage [25] for future research and practical use.

II. Preliminary

A. Deep Code Models

Deep code models (DCMs) [26], [27] are constructed through self-supervised learning methods based on large-scale corpora of unlabelled code snippets and are designed to provide fine-tunable models for different downstream tasks in software engineering. These tasks include vulnerability prediction [5], clone detection [6], [28], and code summarization [7], [29]. Based on their architecture, DCMs are divided into three categories: encoder-only models, decoder-only models, and encoder-decoder models [10]. Encoder-only DCMs, such as CodeBERT [2] and GraphCodeBERT [3], are mainly used to support understanding and generation tasks. Decoder-only models, such as CodeGPT [24], excel at tasks such as generative code completion but are less effective at understanding tasks such as clone detection [30] because of their unidirectional architecture. Encoder-decoder models aim to address both understanding and generation tasks, of which CodeT5 [4] and PLBART [1] are typical. Unlike existing work [13], [16], [31], our study covers not only understanding tasks but also generation tasks, and proposes a new method to generate adversarial examples for DCMs.

B. Problem Definition

We can establish the following two definitions for adversarial attacks on understanding tasks and generation tasks under the black box setting, respectively. Given a source code x that is processed by the victim model $f : \mathcal{X} \rightarrow \mathcal{Y}$ into the input format required by the model. For understanding tasks, the model f predicts the ground-truth label $y \in \mathcal{Y}$ for a given source code $x \in \mathcal{X}$. Our goal is to add slight perturbations to x to generate an adversarial example x'. Specifically, the adversarial code example x' should satisfy the following three requirements: (1) x' should mislead the model such that $f(x') \neq f(x) = y$. (2) x' should remain syntactically correct, ensuring that the perturbation adheres to the syntax rules of the programming language. (3) x' should be semantically equivalent to x, maintaining the same functionality as x and producing identical results under the same inputs.

For generation tasks, consider code summarization as an example. The model f aims to maximize $P(y|x)$ for a given source code $x \in \mathcal{X}$ and a ground-truth summary $y \in \mathcal{Y}$. Unlike understanding tasks where a direct judgment on the success of an attack is possible, the output y in generation tasks has multiple possible outcomes, making such direct judgments impractical. Existing work [11] utilizes the reduction of the BLEU score to evaluate the performance of an attack

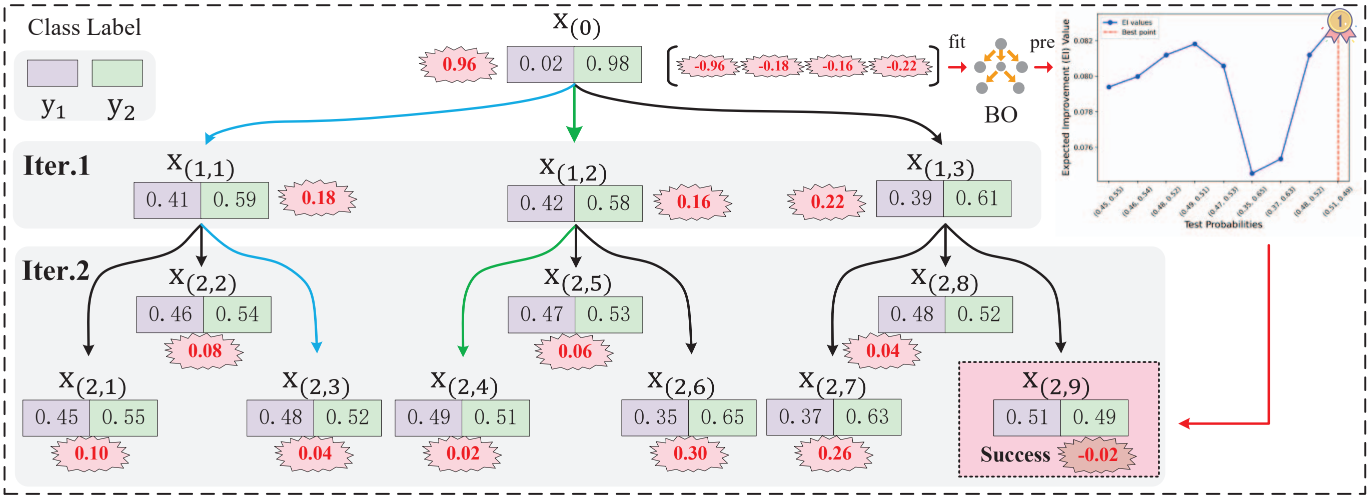


Fig. 1. An attack example on the code clone detection task, where y_1, y_2 are class labels and BO represents Bayesian Optimization.

method. In our method, we follow existing research [14], [17] and consider an attack successful when the BLEU score between the adversarial summary and the reference summary is 0, indicating a complete mismatch. Moreover, adversarial examples in generation tasks should satisfy the requirements defined in the understanding tasks (2) and (3).

C. Motivation Example

Figure 1 presents a simplified example of an adversarial attack in a black-box setting for the code clone detection task. For clarity, we assume that only three perturbations can be applied to each code snippet, such as $x_{(0)}$ and $x_{(i,j)}$. Here, $x_{(0)}$ denotes the original code snippet with ground truth y_2, and Iter.1 and Iter.2 refer to the first and second attack iterations, respectively. $x_{(i,j)}$ is a candidate code variant, where i indicates the iteration count and j represents j-th code variant within that iteration. The values in the rectangle are the predicted probabilities for each class label. The red numerals surrounding $x_{(i,j)}$ represent the probability differences, calculated as the ground truth probability minus the maximum probability of the other class labels (e.g., y_1).

Three distinct attack paths are marked in Figure 1: ❶ **Greedy Search Method:** Existing greedy search methods guided by the ground truth probability difference will execute the failed green attack path $x_{(0)} \rightarrow x_{(1,2)} \rightarrow x_{(2,4)}$. This method selects the code variant with the largest drop in ground truth probability as the starting point for the next iterative attack in each iteration, such as ALERT [13]. However, this strategy can fail when the top candidate is a suboptimal solution (i.e., **L.2 Local Optima Bias**). For instance, although the ground truth probability of $x_{(1,2)}$ decreases the most in the first iteration, the subsequent variants derived from $x_{(1,2)}$ fail to contain true adversarial examples; ❷ **Beam Search Method:** Existing beam search methods guided by ground truth probability differences (e.g., BeamAttack [17], with a beam size of 2) execute the failed blue attack path $x_{(0)} \rightarrow x_{(1,1)} \rightarrow x_{(2,2)}/x_{(2,3)}$, and the green attack path $x_{(0)} \rightarrow x_{(1,2)} \rightarrow x_{(2,4)}/x_{(2,5)}$. The success of this method is inherently limited by the beam size. With a beam size of 1, it follows the failed green attack path; conversely, when the beam size is greater than 2, it finds the adversarial example $x_{(2,9)}$ via the path $x_{(0)} \rightarrow x_{(1,3)} \rightarrow x_{(2,9)}$. Notably, an excessively large beam size compromises efficiency (i.e., **L.3 Efficiency Dilemma**), as confirmed by our experiments (refer to Section V-A); ❸ **Our Method:** Our method uses a modified Bayesian optimization algorithm to fully exploit all failed attack data (e.g., the original probability difference and all probability differences obtained in the first iteration) and directly find the adversarial example $x_{(2,9)}$, as depicted by the red attack path. This strategy significantly alleviates the problems encountered by ALERT and BeamAttack, and fully uses the information from failed attacks to refine and guide future attacks.

III. Methodology

In this section, we introduce ITGen, a black-box adversarial example generation framework that fully utilizes the evaluation information of the victim model on the failed examples during failed attacks. Specifically, the evaluation information of the failed examples potentially reflects the decision boundary of the victim model, which helps guide the subsequent generation direction of ITGen. However, during the entire adversarial example generation process, the evaluation information of the failed examples grows iteratively. In the black-box setting, all the evaluation information must be mined with a small number of queries at each iteration because excessive model queries might trigger the anti-attack security system. Hence, ITGen employs a Bayesian Optimization (BO) framework to mine the evaluation information of failed examples, significantly reducing the number of queries to the victim model.

In fact, it is impractical to apply BO to the code-based adversarial example generation scenario directly. BO usually runs in a continuous space, while the source code is discrete. In addition, the source code must strictly adhere to complex grammar and semantics constraints. Hence, the first step of ITGen is to build a compact bitvector-based code variant representation to adapt to the input format of BO (Section III-A). At the same

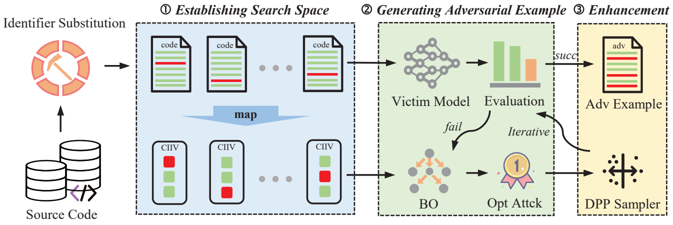


Fig. 2. The workflow of ITGen.

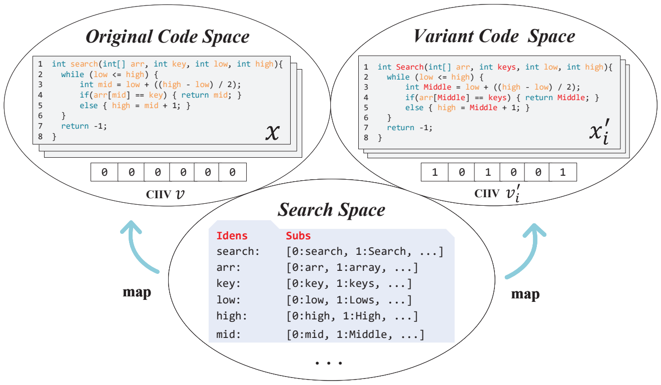
time, ITGen uses an automatic relevance determination [23] kernel, which supports dynamic learning the similarities among different bitvector-based code variants in discrete data spaces. Based on the above optimized BO, ITGen can efficiently obtain the most promising code variant at each iteration (Section III-B). However, since only the evaluation information of the most promising code variant is dynamically increased after each iteration, this may cause BO to fall into a local optimum. Hence, we apply the Determinantal Point Process (DPP) [32], [33] to maximize the sample diversity around the most promising code variant (Section III-C), thereby alleviating the potentially local optimum problem of BO and improving the effectiveness.

Figure 2 depicts the overall workflow of ITGen, and we will explain in detail the main phases in the below subsections.

A. Establishing Search Space

To efficiently search for adversarial examples, ITGen first defines the search space, which is a set of original identifiers and candidate identifiers. Most programming languages, such as C/C++ and Python, follow similar lexical rules for identifiers. For example, the naming rule for regular expression forms is usually defined as "[a-zA-Z][0-9a-zA-Z]*". The identifiers that can be substituted are local variables or function names defined or declared in the source code snippet. Hence, ITGen collects all definitions and declarations of local variables and function names in the source code x to form a set $\mathcal{I}$. The set of candidate identifiers $\mathcal{C}$ is generated from the entire lexical set $\mathcal{Z}$ (which refers to all identifiers in the test set), where the number of candidate identifiers for each identifier is u. Elements in $\mathcal{C}$ must adhere to the specified lexical rules and must not appear in $\mathcal{I}$. These constraints ensure that the replaced identifiers still satisfy the syntactic and semantic requirements of the code (Section II-B). The set $\mathcal{I}$ and the corresponding candidate set $\mathcal{C}$ together constitute the search space $\mathcal{T}_c$ for the source code x, whose size is $l \times (u+1)$, l represents the number of identifiers in x.

However, as previously mentioned (refer to Section I), most methods [10], [13] replace identifiers sequentially during the attack phase. Once the top candidates get stuck in a local optimal solution, it becomes difficult for them to find the global optimal solution because they do not repeatedly process the replaced identifiers [17]. Drawing inspiration from Zhang et al. [34], ITGen maps code variants attacked by using the identifier substitution into candidate identifier index vectors (CIIVs). Each bit in CIIV stores the index of the candidate identifier in $\mathcal{T}_c$ corresponding to the original identifier, and its dimension equals the number of identifiers in $\mathcal{I}$. This encoding strategy allows ITGen to simultaneously substitute multiple identifiers (e.g., 4), thus reducing the probability that the search process falls into a local optimum. Consequently, we obtain the search space $\mathcal{T}_v$ based on CIIVs.



```
int search(int[] arr, int key, int low, int high){
  while (low <= high) {
      int mid = low + ((high - low) / 2);
      if(arr[mid] == key) { return mid; }
      else { high = mid + 1; }
  }
  return -1;
}
```

```
int Search(int[] arr, int keys, int low, int high){
  while (low <= high) {
      int Middle = low + ((high - low) / 2);
      if(arr[Middle] == keys) { return Middle; }
      else { high = Middle + 1; }
  }
  return -1;
}
```

```
Idens      Subs
search:    [0:search, 1:Search, ...]
arr:       [0:arr, 1:array, ...]
key:       [0:key, 1:keys, ...]
low:       [0:low, 1:Lows, ...]
high:      [0:high, 1:High, ...]
mid:       [0:mid, 1:Middle, ...]
. . .
```

Fig. 3. An example of code snippets with their CIIVs.

To elucidate, consider Figure 3 as an illustrative example. Suppose there are six identifiers in the source code x, which after identifier substitution yields the variant x_i'. We map the variant to a CIIV, denoted as v_i', which has a length of $l = 6$. In this case, bits 1, 3, and 4 of the index vector v_i' are assigned the value of 0, indicating that there is no substitution of the original identifier. Conversely, bits 0, 2, and 5 are assigned a value of 1, indicating that the original identifier (e.g., "search") is replaced by a candidate identifier corresponding to index 1 (e.g., "Search").

B. Generating Adversarial Example

After establishing the search space, we proceed to mine the evaluation information of failed attacks using a Bayesian optimization framework to identify a promising code variant (optimal attack vector) for adversarial attacks. Specifically, for understanding tasks, our goal is to find code variants x' that maximize the negative difference between the ground-truth probability and the maximum probability for classes other than the ground-truth class (called negative margin), which is:

$$\max_{x' \in \mathcal{T}_c} -\mathcal{L}_{\text{margin}}(x') := \max_{y' \in \mathcal{Y}, y' \neq y} \left(f(x')_{y'} - f(x')_y \right) \tag{1}$$

Here, $f(x')_{y'}$ and $f(x')_y$ are the prediction probabilities for class y' and the ground-truth class y, respectively.

For generation tasks (e.g., code summarization), our goal is to find code variants x' that maximize the negative BLUE score. The objective function is defined as:

$$\max_{x' \in \mathcal{T}_c} -\mathcal{L}_{\text{BLEU}}(x') \tag{2}$$

$\mathcal{L}_{\text{BLEU}}(x')$ is the BLEU score for the adversarial example x'.

As mentioned in Section II-C, using a greedy strategy to optimize objective function 1 and 2 may lead to a locally optimal solution. To address this problem, we model these functions g using Gaussian Processes (GP) [35] with a categorical kernel as the surrogate model $\mathcal{M}$. GP assumes that the value of g at any finite set of training points X is normally distributed, i.e., $g(X) \sim \mathcal{N}(\mu(X), \kappa(X, X) + \tau^2 I)$, where μ and κ are the mean and kernel functions, respectively, and τ^2 is the noise variance. The kernel κ has the following form:

$$\kappa(v'_1, v'_2) = \sigma_g^2 \prod_{i=0}^{l-1} \exp\left(-\frac{\delta(w_{1i}, w_{2i})}{\beta_i}\right) \tag{3}$$

where v'_i is the CIIV corresponding to code variant x'_i, σ_g^2 is a signal variance, $\delta(\cdot)$ is the Kronecker delta function [36], which is 0 if $w_{1i} = w_{2i}$ and 1 if $w_{1i} \neq w_{2i}$, and β_i is a length-scale parameter corresponding to the relevance of i-th element position. This kernel implies that a pair of vectors sharing a larger number of elements are considered more similar.

Then, we use the historical failed attacks $\mathcal{D}$ on the victim model f to update the posterior distribution of the surrogate model $\mathcal{M}$. According to Bayes' theorem [37], the mean μ and covariance matrix Σ of the updated posterior distribution are:

$$\begin{aligned} &\mu = K(v'_1, \boldsymbol{X})(K(\boldsymbol{X}, \boldsymbol{X}) + \tau^2 I)^{-1}\boldsymbol{Y}, \\ &\Sigma = \kappa(v'_1, v'_2) - K(v'_1, \boldsymbol{X})\left(K(\boldsymbol{X}, \boldsymbol{X}) + \tau^2 I\right)^{-1} K(\boldsymbol{X}, v'_2) \end{aligned} \tag{4}$$

where $\boldsymbol{X}$ is the matrix consisting of CIIVs in $\mathcal{D}$, $\boldsymbol{Y}$ is the target value vector corresponding to $\boldsymbol{X}$, and I is the identity matrix used for regularization. And we use the following steps to construct the set $\mathcal{D}$.

Firstly, we empirically randomly sample n code variants from the search space $\mathcal{T}_c$, where each variant involves the simultaneous substitution of multiple identifiers. The value of n is determined by the following formula:

$$n = \begin{cases} \lfloor \frac{N}{l} \rfloor & \text{if } l \leq 3 \\ 2 \times \lfloor \frac{N}{l} \rfloor & \text{if } l > 3 \end{cases} \tag{5}$$

where l represents the number of original identifiers, N represents the sum of the number of original identifiers and corresponding candidate identifiers. This adaptation ensures that oversampling is prevented when there are fewer identifiers, and a larger initial search space is covered when there are more identifiers.

Secondly, we use the victim model f to evaluate the sampling set. We collect evaluation information in the form of Eq. 1 and Eq. 2 to form the historical failed attack set $\mathcal{D}$ as follows:

$$\mathcal{D} = \begin{cases} \{(v'_i, -\mathcal{L}_{\text{margin}}(x'_i))\}_{i=1}^n & \text{for understanding tasks} \\ \{(v'_i, -\mathcal{L}_{\text{BLEU}}(x'_i))\}_{i=1}^n & \text{for generation tasks} \end{cases} \tag{6}$$

After obtaining a posterior distribution closer to g, we select the most promising code variant x'_{opt} based on the current posterior distribution. A common sampling strategy is the Expected Improvement [38], which is defined as the expected improvement over the best performance found so far:

$$v'_{\text{opt}} = \arg\max_{v' \in \mathcal{T}_v} \int_{-\infty}^{\infty} \max(y^* - y, 0) P_{\mathcal{M}}(y|v')dy \tag{7}$$

Where v'_{opt} is the CIIV corresponding to the promising code variant x'_{opt}, $\mathcal{T}_v$ represents the search space for CIIV, and y^* is the best performance observed so far.

Once the optimal attack vector v'_{opt} for this iteration is identified, we evaluate the performance of its corresponding code variant x'_{opt} on the victim model. If the result satisfies Eq. 1 or 2—for understanding tasks, it means the negative margin is greater than 0, and for generation tasks, the negative BLEU score equals 0—the adversarial example is successfully generated, and the iteration is halted. If not, the result is added to $\mathcal{D}$ to update the posterior distribution of surrogate model $\mathcal{M}$. This iterative process continues until an adversarial example is found or the attack budget $\mathcal{B}$ is exhausted. And $\mathcal{B}$ is empirically set to stop when the number of model invocations exceeds twice the size of search spaces $\mathcal{T}_c$. The detailed process is summarized in Algo. 1.

Algorithm 1 FAM: Failed Attacks Mining via BO

```
Require: Search space $\mathcal{T}_c$, attack criterion $\mathcal{L}$, budget $\mathcal{B}$
Initialize historical failed attack set $\mathcal{D} \leftarrow \emptyset$
Sample $n$ code variants $\{x'_i\}_{i=1}^n$ from $\mathcal{T}_c$
Map $\{x'_i\}_{i=1}^n$ to CIIVs $\{v'_i\}_{i=1}^n$
for each $x'_i$ in $\{x'_i\}_{i=1}^n$ do
    $r'_i \leftarrow -\mathcal{L}_{\text{margin}}(x'_i)$ or $-\mathcal{L}_{\text{BLEU}}(x'_i)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(v'_i, r'_i)\}$
end for
while budget $\mathcal{B}$ dose not exhaust do
    Train surrogate model $\mathcal{M}$ on $\mathcal{D}$
    $v'_{\text{opt}} \leftarrow \arg\max_{v' \in \mathcal{T}_v} \int_{-\infty}^{\infty} \max(y^* - y, 0) P_{\mathcal{M}}(y|v')dy$
    Map $v'_{\text{opt}}$ to $x'_{\text{opt}}$
    $r'_{\text{opt}} \leftarrow -\mathcal{L}_{\text{margin}}(x'_{\text{opt}})$ or $-\mathcal{L}_{\text{BLEU}}(x'_{\text{opt}})$
    if $r'_{\text{opt}} > 0$ or $r'_{\text{opt}} = 0$ then
        return adversarial example $x'_{\text{opt}}$
    end if
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(v'_{\text{opt}}, r'_{\text{opt}})\}$
end while
return optimal attack vector $v'_{\text{opt}}$
```

C. Enhancement

The optimal attack vector v'_{opt} found in Section III-B may still fail to mislead the model. To enhance the attack performance, we incorporate a Determinantal Point Process (DPP) [32], a sampling algorithm that maximizes sample diversity. Using DPP helps in selecting a diverse set of candidate adversarial examples, thereby increasing the likelihood of finding a successful adversarial attack.

The detailed algorithm is shown in Algo. 2. Specifically, ITGen first constructs a 1-Hamming distance ball $\mathcal{H}$ around the optimal attack vector v'_{opt}. Then, it identifies the set $\mathcal{E} = [v'_{i1}, v'_{i2}, \ldots, v'_{ie}]$ consisting of the top-e expected improvement values within $\mathcal{H}$.

Next, ITGen uses Eq. 3 to calculate the similarity matrix $\mathcal{V} = \text{sim}(\mathcal{E}, \mathcal{E})$. Based on the similarity matrix $\mathcal{V}$, ITGen iteratively selects CIIVs v^* that maximize the marginal gain in total dissimilarity for the current candidate set S in a greedy manner until S reaches the maximum capacity k. This process is represented as:

$$v^* = \arg\max_{v' \in \mathcal{T}_v \setminus S} \left(\det\left(\Sigma_{S \cup v' | \mathcal{D}}\right) - \det\left(\Sigma_{S|\mathcal{D}}\right)\right) \tag{8}$$

where det represents the determinant, $\Sigma_{S \cup v' | \mathcal{D}}$ and $\Sigma_{S|\mathcal{D}}$ denote the posterior covariance matrices. To avoid redundant evaluations, vectors previously evaluated in past acquisition maximization steps are excluded from the batch in $\mathcal{E}$.

After sampling the candidate set S around the optimal attack vector v'_{opt} using DPP, ITGen evaluates these code variants using the victim model, collects the evaluation information, and continues to train the surrogate model $\mathcal{M}$ according to the process described in Section III-B until the final adversarial example is found or the attack budget is exhausted.

Algorithm 2 DPP: Enhanced Candidate Sampling

```
Require: Optimal attack vector $v'_{opt}$, candidate number: $k$
Ensure: Candidate set $S \subset \{1, 2, \ldots, n\}$ with $|S| = k$
Conduct 1-Hamming distance ball $\mathcal{H}(v'_{opt}, 1)$
Find top-$e$ EI values set $\mathcal{E}$ in $\mathcal{H}(v'_{opt}, 1)$
Calculate the posterior covariance matrix $\mathcal{V}$ of $\mathcal{E}$ using Eq. 3
Initialize $S \leftarrow \{0\}$
Compute initial determinant $\text{det}_{\text{cur}} \leftarrow \det(\mathcal{V}[S, S])$
while $|S| < k$ do
    $\text{det}_{\text{best}} \leftarrow -\infty$, $S_{\text{best}} \leftarrow$ None
    for $i \in \{1, 2, \ldots, n\} \setminus S$ do
        $S_{\text{tmp}} \leftarrow S \cup \{i\}$
        Compute submatrix $\mathcal{V}_{\text{tmp}} \leftarrow \mathcal{V}[S_{\text{tmp}}, S_{\text{tmp}}]$
        Compute determinant det $\leftarrow \det(\mathcal{V}_{\text{tmp}})$
        if det $> \text{det}_{\text{best}}$ then
            $\text{det}_{\text{best}} \leftarrow$ det, $S_{\text{best}} \leftarrow S_{\text{tmp}}$
        end if
    end for
    $S \leftarrow S_{\text{best}}$
end while
return Candidate set $S$
```

IV. Evaluation Design

In the study, we address three research questions (RQs):

- **RQ1:** How does ITGen perform in terms of effectiveness and efficiency compared with existing techniques?
- **RQ2:** Are the adversarial examples generated by ITGen useful to enhance the robustness of deep code models?
- **RQ3:** Does each main component contribute to the overall effectiveness of ITGen?

A. Subjects and Datasets

To sufficiently evaluate ITGen, we focus on three code-based tasks: vulnerability detection, clone detection for understanding tasks, and code summarization for generation tasks in the studies of evaluating state-of-the-art techniques (i.e., ALERT [13] and BeamAttack [17]). The statistics of datasets are shown in Table I. The vulnerability prediction task aims to predict whether a given code snippet has vulnerabilities. Its used dataset is extracted from two C projects [5]. The clone detection task aims to detect whether two given code snippets are equivalent in semantics. Its used dataset is from BigCloneBench [39], the most widely-used benchmark in clone detection research. Lastly, the code summarization task aims to generate natural language texts that describe the functionality of a given code snippet. Its used the Java sub-datasets [10], [24] filtered from CodeSearchNet dataset [40].

B. Evaluation Metrics

We adopt the following metrics for evaluation.

- **Accurary**. This metric represents the proportion of correctly predicted instances in the test set. It is used in the task of vulnerability prediction and clone detection.

- **BLEU-4**. BLEU is a widely used metric to evaluate the textual similarity between the text generated by generative systems and the ground truth. BLEU-4 [10], [41] is a variant of BLEU where the "4" indicates that four consecutive words (4-gram) are used as the matching unit.

TABLE I
Statistics of our used subjects

Task	Train/Val/Test	Model	Acc. / BLEU-4
Vulnerability Prediction	21,854/2,732/2,732	CodeBERT	63.76%
		GraphCodeBERT	63.65%
		CodeT5	63.83%
Clone Detection	90,102/4,000/4,000	CodeBERT	96.97%
		GraphCodeBERT	97.36%
		CodeT5	98.08%
Code Summarization	164,923/5,183/10,955	CodeBERT	18.75
		CodeGPT	15.36
		PLBART	17.60

C. Target Models

Section II-A reviews the current state-of-the-art DCMs. One or more models are selected for evaluation for each type of DCM. Notably, no single model consistently outperforms others across all tasks. For example, within the encoder-only models, CodeBERT excels in vulnerability prediction but falls behind GraphCodeBERT in clone detection. Similarly, among encoder-decoder models, CodeT5 outperforms PLBART in vulnerability prediction, while PLBART shows superior performance in clone detection [10]. Hence, we select the examined model from each category based on the nature of the task. Specifically, for understanding tasks, we choose CodeBERT, GraphCodeBERT, and CodeT5. For generation tasks, we select CodeBERT, CodeGPT, and PLBART. We fine-tuned the models on the three tasks using the corresponding datasets following existing works [10], [24], respectively. This resulted in a total of nine DCMs as the subjects. The last two columns in Table I detailed the DCMs utilized and the accuracy or BLEU-4 scores obtained after fine-tuning for each corresponding task. Overall, the subjects in our study are diverse, encompassing various task types and different pre-training model architectures, which is crucial for a comprehensive evaluation of ITGen's performance.

D. Compared Techniques

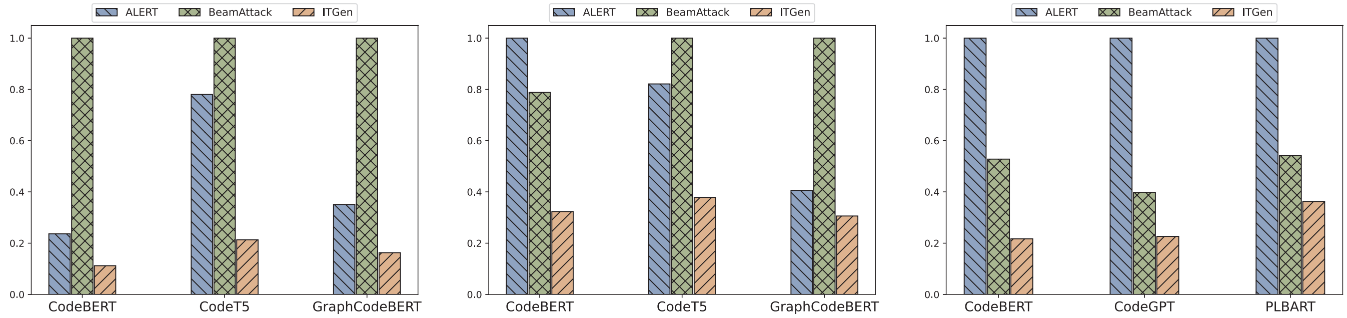
Our research focuses exclusively on black-box attack methods. While CODA [12] also employs a black-box setting, it uniquely integrates equivalent structure transformation and identifier renaming transformation as attack mechanisms. However, in this paper, we restrict our analysis to black-box methods that solely utilize identifier substitution for their attacks, due to this technique's increased stealthiness against anti-attack security systems [16]. Therefore, we excluded CODA and instead selected the most recent and advanced black-box methods as our baselines. Below, we briefly introduce the two methods chosen for comparison:

- **ALERT** [13] utilizes context-aware identifier prediction for substitution. Specifically, in terms of the identifier selection strategy, ALERT adopts two methods: a greedy algorithm and a genetic algorithm. We configured the relevant hyper-parameters following the original paper, setting the maximum number of iterations at the greater value between 5 × l and 10, where l represents the number of identifiers in the code.

- **BeamAttack** [17] utilizes the perturbation results of different types of statements as a priority strategy for identifier substitution and then uses beam search [42] to focus on all

TABLE II
EFFECTIVENESS COMPARISON IN TERMS OF ASR.

Model	Vulnerability Prediction			Clone Detection			Code Summarization		
	ALERT	**BeamAttack**	**ITGen**	**ALERT**	**BeamAttack**	**ITGen**	**ALERT**	**BeamAttack**	**ITGen**
CodeBERT	55.72%	54.62%	**76.07%**	27.02%	18.68%	**49.11%**	60.86%	51.47%	**91.36%**
GraphCodeBERT	65.83%	63.55%	**81.63%**	12.92%	13.62%	**33.68%**	—	—	—
CodeT5	83.21%	81.55%	**96.25%**	20.38%	5.89%	**44.79%**	—	—	—
CodeGPT	—	—	—	—	—	—	31.66%	23.86%	**69.24%**
PLBART	—	—	—	—	—	—	68.05%	57.93%	**87.71%**
Average	68.25%	66.57%	**84.65%**	20.11%	12.73%	**42.53%**	53.52%	44.42%	**82.77%**



(a) AMI on Vulnerability Prediction (b) AMI on Clone Detection (c) AMI on Code Summarization

Fig. 4. Comparison in terms of AMI (y-axis shows the normalized values following the existing work [13])

identifiers in the statement, which can simultaneously search from multiple sequences and replace multiple identifiers. We chose to be consistent with the settings in the original paper and set the beam size to 2, 3, and 5 in clone detection, vulnerability prediction, and code summarization, respectively.

E. Implementations

We implemented ITGen in Python, utilizing tree-sitter [43] to extract identifiers from the code following existing work [13]. The Bayesian optimization framework was implemented using botorch [44]. We set $u = 50$ (the number of candidate identifiers for the original identifier in $\mathcal{C}$) and $k = 4$ (the number of candidate vectors selected in DPP sampling) for ITGen. We will discuss the impact of ITGen's main parameters in Section VI-B. All experiments are performed on a Ubuntu 20.04.1 LTS with Nvidia GeForce RTX 3090 (GPU), Intel XEON 6226R 2.90GHz (CPU), 32GiB DDR4-3200 (Memory).

V. RESULTS AND ANALYSIS

A. RQ1: Effectiveness and Efficiency

1) Setup: In our experimentation, we employed ITGen, ALERT, and BeamAttack on each DCM to generate adversarial examples from every input in the test set $\mathcal{X}$. We then measured the effectiveness and efficiency of these methods based on the following metrics:

- **Effectiveness.** Consistent with existing work [13], [17], we evaluated the effectiveness of these methods using the *Attack Success Rate* (**ASR**). ASR is defined as the percentage of code snippets x in the given test dataset $\mathcal{X}$ for which an attack method can successfully generate adversarial examples x'. A higher ASR indicates better effectiveness. The ASR is calculated as follows:

$$ASR = \frac{|\{x \mid f(x') \neq f(x) \vee f(x')_{BLEU} = 0, x \in \mathcal{X}\}|}{|\mathcal{X}|} \quad (9)$$

f is the victim model for understanding or generation tasks.

- **Efficiency.** Following the existing work [12], [13], we compare the efficiency of the examined methods through three metrics: (1) *Total Generation Time* (**TGT**), which measures the time required to complete the entire process of generating adversarial examples across all subjects. (2) *Average Model Invocations* (**AMI**). In the context of adversarial attacks, particularly in scenarios involving black-box attacks, minimizing the number of victim model invocations is imperative. Given that victim models are often situated remotely, conducting numerous model invocations can be both costly and potentially raise suspicion. AMI denotes the average number of model invocations that successfully generate an adversarial example. Methods with lower AMI are also more scalable. (3) *Average Running Time* (**ART**). ART represents the average time required to generate an adversarial example successfully. Generally, efficient techniques should strive to minimize these metrics. It is pertinent to mention that our analysis of AMI and ART exclusively considers instances where attacks were successful, thereby addressing potential bias in comparison due to ITGen's use of model invocations as the attack budget (Section III-C).

To mitigate the effects of randomness, we repeated all experiments (including those for other RQs) five times and reported the average results.

2) Results: Table II shows the comparison results among ALERT, BeamAttack, and ITGen in terms of ASR. From this table, ITGen consistently outperforms ALERT and BeamAttack across all subjects, demonstrating the stable effectiveness of

TABLE III
ROBUSTNESS ANALYSIS ON ADVERSARIALLY FINE-TUNED VICTIM MODELS.

Model	**Vulnerability Prediction**				**Clone Detection**				**Code Summarization**			
	Ori	**ALERT**	**BeamAttack**	**ITGen**	**Ori**	**ALERT**	**BeamAttack**	**ITGen**	**Ori**	**ALERT**	**BeamAttack**	**ITGen**
CodeBERT-Adv	65.15%	51.66%	54.29%	49.93%	97.00%	90.83%	92.13%	98.30%	18.98	10.90	10.80	11.32
GraphCodeBERT-Adv	66.07%	61.42%	63.08%	53.92%	97.30%	97.60%	94.50%	97.39%	—	—	—	—
CodeT5-Adv	67.46%	74.82%	73.66%	72.92%	96.82%	95.55%	92.51%	95.21%	—	—	—	—
CodeGPT-Adv	—	—	—	—	—	—	—	—	17.28	4.20	4.31	4.27
PLBART-Adv	—	—	—	—	—	—	—	—	17.11	4.38	4.41	4.51
Average	66.23%	62.63%	63.68%	58.92%	97.04%	94.66%	93.05%	96.97%	17.79	6.49	6.51	6.70

* The numbers are the prediction accuracy scores or BLEU scores of models adversarially fine-tuned with ITGen on the adversarial examples generated by three methods.

TABLE IV
COMPARISON IN TERMS OF ART.

Task	Model	ALERT	BeamAttack	ITGen
Vulnerability Prediction	CodeBERT	**0.16**	1.67	0.66
	CodeT5	0.07	0.27	**0.04**
	GraphCodeBERT	**0.19**	1.30	0.51
Clone Detection	CodeBERT	0.26	0.55	**0.24**
	CodeT5	0.51	1.17	**0.30**
	GraphCodeBERT	**0.32**	2.11	1.25
Code Summarization	CodeBERT	0.46	0.71	**0.39**
	CodeGPT	3.24	1.42	**0.76**
	PLBART	0.66	0.67	**0.38**
Average		0.65	1.10	**0.50**

ITGen. On average, ITGen improves 24.03% and 27.16% higher ASR than ALERT and BeamAttack across all three models on the vulnerability prediction task, 111.49% and 234.09% higher ASR on clone detection, and 54.65% and 86.34% higher ASR on code summarization, respectively.

We further explored the unique value of each method by analyzing their overlap on test inputs where adversarial examples are generated. On average, across all subjects, 35.56% of test inputs yielded adversarial examples uniquely generated by ITGen, compared to just 1.84% and 1.38% for ALERT and BeamAttack, respectively. The results highlight ITGen's superior capability in revealing faults in DCMs, demonstrating its unique value relative to the other techniques.

Additionally, ALERT, BeamAttack, and ITGen require 999.06 hours, 1070.93 hours, and 361.62 hours, respectively, to complete the entire generating process (TGT) on all subjects. Moreover, we measured the average number of model invocations (AMI) for successfully generating adversarial examples, whose results are shown in Figures 4(a), 4(b), and 4(c). From these figures, ITGen consistently performs fewer model invocations than both ALERT and BeamAttack, regardless of the tasks and pre-trained models. On average, ITGen performs 59.65% and 83.74% fewer model invocations than ALERT and BeamAttack across all the models on vulnerability prediction, 48.73% and 63.50% fewer model invocations on clone detection, and 73.14% and 45.04% fewer model invocations on code summarization, respectively. Furthermore, the average running time (ART) serves as a comprehensive metric of the efficiency of an attack method [17]. ART is influenced not only by the number of model invocations but also by the search strategy employed. The ART results for the examined methods are detailed in Table IV. Although ALERT outperforms ITGen in terms of ART for some subjects, ITGen generally exhibits a lower ART compared to both ALERT and BeamAttack. On average, across all tasks, ITGen's ART is 23.08% and 54.55% lower than ALERT's and BeamAttack's, respectively.

Overall, ITGen significantly enhances the effectiveness and efficiency of adversarial example generation by thoroughly mining historical failed attack information. This is also the reason why ALERT and BeamAttack underperform in comparison to ITGen.

> **Answer to RQ1:** In terms of attack success rate, ITGen outperforms ALERT and BeamAttack by 47.98% and 69.70% across all subjects, respectively. And ITGen not only requires less time to complete the entire generation process but also involves fewer model invocations, enhancing its scalability.

B. RQ2: Model Robustness Enhancement

1) Setup: We investigate the effectiveness of adversarial fine-tuning [45] as a defence mechanism against attacks following ALERT [13]. To prevent data leakage between the augmented training set and the evaluation set, we split the test set P into two equal parts $P1$ and $P2$. Specifically, we use ITGen to generate examples from $P1$, resulting in either adversarial examples or examples that yield the largest reduction in prediction confidence (or BLEU score) if no adversarial examples are generated. It is important to note that we skip any sample if the victim model makes an incorrect prediction (or BLEU score is 0) on the original input or if it cannot extract the identifier from it [13]. These examples are then augmented to the original training set to create the augmented training set used for fine-tuning the models. After fine-tuning the models for each subject using ITGen, we evaluate them on their evaluation sets of adversarial examples. Specifically, for ALERT and BeamAttack, we assess the fine-tuned models on their evaluation sets of adversarial examples generated from P. For ITGen, we evaluate the fine-tuned models on its evaluation set of adversarial examples generated from $P2$. We then measure the accuracy (or BLEU score) of the fine-tuned models across the three evaluation sets to determine their ability to enhance the robustness of the models.

2) Results: Table III illustrates the enhancement of model robustness through the use of adversarial examples generated by ITGen. The second row (except Column Ori) indicates the evaluation set constructed by the respective method, while

Column Ori lists the accuracy (or BLEU score) of the fine-tuned model on the original test set. The values in each column (except Column Ori) denote the prediction accuracy (or BLEU score) of the model fine-tuned with ITGen, against adversarial examples generated by the corresponding techniques. It is noted that the original victim models (that are not hardened by adversarial retraining) fail to correctly process all these examples, resulting in an accuracy of 0% for understanding tasks and a BLEU score of 0 for generation tasks.

We found that ITGen significantly enhances the model robustness across most subjects. On average, models fine-tuned with ITGen improves the prediction accuracy of adversarial examples generated by ALERT, BeamAttack, and ITGen by 62.63%, 63.68%, and 58.92% respectively in vulnerability prediction; by 94.66%, 93.05%, and 96.97% respectively in clone detection; and improves the BLEU scores by 6.49, 6.51, and 6.70, respectively in code summarization. By comparing Column Ori in Table III with the last column in Table I, we observed that most fine-tuned models via ITGen demonstrate improved prediction accuracies or BLEU scores. These results indicate that ITGen is helpful to improve model robustness without compromising the original model's performance.

> **Answer to RQ2:** ITGen helps enhance model robustness effectively without damaging the original model performance.

TABLE V
ABLATION TEST FOR ITGEN IN TERMS OF AVERAGE ASR.

Task	Model	w/o CIIV	w/o FAM	w/o DPP	ITGen
VP	CodeBERT	65.78%	46.53%	73.93%	**76.07%**
	CodeT5	95.57%	86.59%	**97.09%**	96.25%
	GraphCodeBERT	68.11%	52.72%	76.36%	**81.63%**
	Average	76.49%	61.95%	82.46%	**84.65%**
CD	CodeBERT	43.33%	18.19%	**49.16%**	49.11%
	CodeT5	32.47%	11.10%	40.07%	**44.79%**
	GraphCodeBERT	18.53%	5.66%	26.75%	**33.68%**
	Average	31.44%	11.65%	38.66%	**42.53%**
CS	CodeBERT	86.63%	81.38%	88.01%	**91.36%**
	CodeGPT	50.55%	55.51%	67.59%	**69.24%**
	PLBART	72.99%	68.29%	86.33%	**87.71%**
	Average	70.06%	68.39%	80.64%	**82.77%**

VP: Vulnerability Prediction; CD: Clone Detection; CS: Code Summarization

C. RQ3: Contribution of Main Components

1) Setup: We studied the contribution of each main component within ITGen, namely the Candidate Identifier Index Vector (CIIV), Failed Attacks Mining (FAM), and Determinantal Point Process (DPP). To this end, we constructed three variants of ITGen:

- **w/o CIIV:** We retained the structure of the CIIV but modified only one bit at a time during each attack or sampling, which means replacing only one identifier at a time instead of changing multiple bits simultaneously.
- **w/o FAM:** We removed FAM from ITGen, where it directly checks whether an adversarial example is generated after sampling n CIIVs from search space $\mathcal{T}_v$.
- **w/o DPP:** We removed DPP from ITGen, which directly checks whether an adversarial example is generated after sampling CIIVs with top-k acquisition function values around the optimal attack vector v'_{opt} mined from FAM.

2) Results: Table V shows the average ASR value of each technique across all subjects. ITGen outperforms all three variants in terms of average ASR, with an improvement of 2.64%∼265.06%, demonstrating the contribution of each main component in ITGen. FAM contributes more significantly than CIIV and DPP. Excluding the FAM component results in a substantial drop in average ASR of 22.65%. This finding underscores the importance of historical failed attack information in effectively guiding the generation of adversarial examples and highlights the critical role of modified Bayesian optimization in navigating the search space and identifying the optimal attack vectors. Additionally, since ALERT only replaces one identifier at a time during attacks, for a fairer comparison, we also compare ALERT with the w/o CIIV variant. The results indicate that the latter can address the issue of top candidates falling into a local optical solution. The absence of the DPP sampler also negatively impacts ASR, as evidenced by a decrease of about 2.73% in average ASR when DPP is removed. This demonstrates that DPP helps alleviate the limitations of insufficient FAM mining by sampling diverse candidates, thereby enhancing the overall effectiveness of the attack strategy.

> **Answer to RQ3:** All components of candidate identifier index vector, model uncertainty mining, and determinantal point process contribute to the effectiveness of ITGen, demonstrating the necessity of each of them in ITGen.

VI. DISCUSSION

A. Perturbation of Adversarial Example

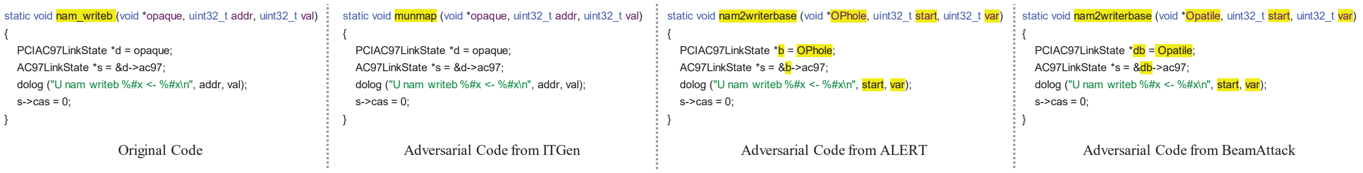
The above RQs demonstrate the effectiveness and efficiency of ITGen in generating adversarial examples against different DCMs across various tasks. In this section, we evaluate the quality of the generated adversarial examples both quantitatively and qualitatively to determine their ability to generate looking-normal code, avoiding naively increasing the number of perturbed identifiers.

• **Quantitative Analysis:** Table VI shows the number of replaced identifiers, where lower values indicate minimal input modification and, therefore, higher example quality. ITGen modifies about 2.07 identifiers on average, with a lower variance of 1.86, indicating fewer and more consistent changes. In contrast, BeamAttack modifies about 2.67 identifiers on average, with a higher variance of 4.08. ALERT also replaces more identifiers than both ITGen and BeamAttack. The results demonstrate that the perturbations introduced by ITGen are smaller than or equal to the baselines.

• **Qualitative Analysis:** Figure 5 presents qualitative examples of the generated adversarial code snippets from ITGen, ALERT, and BeamAttack on the vulnerability prediction task, with changes highlighted by yellow shading. ITGen renames only one identifier "nam_writeb" to "munmap" whereas ALERT and BeamAttack rename almost all identifiers, often opting for longer identifiers.

TABLE VI
IDENTIFIER SUBSTITUTION NUMBER COMPARISON, MEAN ± VARIANCE

Method	Vulnerability Prediction			Clone Detection			Code Summarization			Average
	CodeBERT	CodeT5	GraphCodeBERT	CodeBERT	CodeT5	GraphCodeBERT	CodeBERT	CodeGPT	PLBART	
ALERT	3.77 ± 10.41	2.39 ± 4.84	3.38 ± 9.28	6.34 ± 15.82	4.48 ± 19.44	5.49 ± 12.75	2.63 ± 3.72	3.13 ± 5.22	2.74 ± 3.72	3.82 ± 9.47
BeamAttack	2.72 ± 5.01	1.44 ± 1.06	2.25 ± 3.57	3.57 ± 6.91	4.04 ± 8.60	5.49 ± 9.99	1.47 ± 0.49	**1.52 ± 0.52**	**1.55 ± 0.53**	2.67 ± 4.08
ITGen	**1.98 ± 1.77**	**1.22 ± 0.39**	**1.85 ± 1.70**	**2.68 ± 3.51**	**2.41 ± 2.62**	**3.75 ± 4.18**	**1.39 ± 0.62**	1.66 ± 0.93	1.73 ± 1.03	**2.07 ± 1.86**



```
static void nam_writeb (void *opaque, uint32_t addr, uint32_t val)
{
    PCIAC97LinkState *d = opaque;
    AC97LinkState *s = &d->ac97;
    dolog ("U nam writeb %#x <- %#x\n", addr, val);
    s->cas = 0;
}
```

Original Code

```
static void munmap (void *opaque, uint32_t addr, uint32_t val)
{
    PCIAC97LinkState *d = opaque;
    AC97LinkState *s = &d->ac97;
    dolog ("U nam writeb %#x <- %#x\n", addr, val);
    s->cas = 0;
}
```

Adversarial Code from ITGen

```
static void nam2writerbase (void *OPhole, uint32_t start, uint32_t var)
{
    PCIAC97LinkState *b = OPhole;
    AC97LinkState *s = &b->ac97;
    dolog ("U nam writeb %#x <- %#x\n", start, var);
    s->cas = 0;
}
```

Adversarial Code from ALERT

```
static void nam2writerbase (void *Opatile, uint32_t start, uint32_t var)
{
    PCIAC97LinkState *db = Opatile;
    AC97LinkState *s = &db->ac97;
    dolog ("U nam writeb %#x <- %#x\n", start, var);
    s->cas = 0;
}
```

Adversarial Code from BeamAttack

Fig. 5. Qualitative examples of adversarial codes on vulnerability prediction. Original vs. ITGen vs. ALERT vs. BeamAttack

B. Impact of Main Parameters in ITGen

Here, we explore the impact of two critical parameters in ITGen, namely u and k, introduced in Sections III-A and III-C, respectively. The parameter u influences the size of the search space, while k modulates the diversity of the sampling process. Based on the results in Section V-C, u and k significantly affect the performance of ITGen in Clone Detection/GraphCodeBERT. Hence, we choose this combination to examine the impacts of u and k. Figure 6 shows the results of the ASR for different configurations. We find that ITGen's effectiveness is compromised when u is set too low or too high. Specifically, a minimal u value reduces the probability of adversarial examples. Conversely, an excessively high u value expands the search space too broadly, complicating the search process. In terms of k, increasing its value within our studied range introduces a more diverse set of samples, which helps mitigate potential issues of local optima in the modified Bayesian optimization framework. However, a higher k also risks incorporating excessive noise samples, potentially degrading ITGen's performance. Hence, by balancing the efficiency and effectiveness of ITGen, we set u to 50 and k to 4 as the default settings in ITGen for practical use.

C. Threats to Validity

Internal validity: Variability in our experimental results may occur due to different parameter settings. To mitigate this threat, we analyzed the impact of two critical parameters in ITGen (Section VI-B), and we offer default settings that balance effectiveness and efficiency for practical application. Additionally, the potential for bugs in our implementation poses a threat to internal validity. To counter this, we meticulously reviewed our codes and made the codes and corresponding data available to the community for further verification.

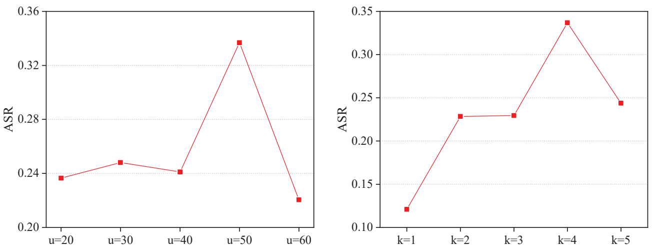
External validity: The generalizability of our findings is potentially limited by the selection of tasks, datasets, and models. To mitigate this threat, we selected tasks and datasets that are widely used and studied in previous state-of-the-art papers [12], [13], [17]. For the target model, we mitigated this threat by choosing the most popular DCMs with relatively high performance. While we have discussed efficiency, the scalability of ITGen under broader or more complex conditions remains uncertain. Larger or more complex datasets may produce different results in terms of computing resources and time.

VII. RELATED WORK

Adversarial example generation for DCMs can be broadly categorized into black-box and white-box methods [13], [17]. Black-box methods involve querying model outputs to identify substitutions using a score function. For example, ALERT [13] generates adversarial examples using variable name substitution from a pre-trained masked model; MHM [31] samples code identifier substitutions using Metropolis-Hastings; STRATA [46] substitutes code tokens based on their distribution. Additionally, Chen et al. [47] employ semantics-preserving code transformations, while CodeAttack [48] leverages code structure for generating adversarial data; BeamAttack [17] perturbs source code based on the contextual information of identifiers to generate adversarial examples; CODA [12] finds reference examples from the training set and transfers their code styles to target examples, including equivalent structure transformation and identifier renaming transformation. Conversely, white-box methods necessitate access to the model's gradients or architecture to craft adversarial examples. For example, CARROT [14] selects code mutations guided by model gradients; Henkel et al. [49] optimize adversarial attacks using gradient-driven syntax tree transformations; Srikant et al. [50] modify code through optimized program obfuscation; DAMP [15] induces mispredictions by altering inputs based on model gradients. Despite their advancements, these methods often overlook valuable information from failed attacks, which can enhance subsequent attack strategies. Also, these techniques still search for ingredients in the enormous space, limiting their effectiveness. Different from them, our approach effectively mines information from failed attacks and predicts future optimal attack vectors, significantly narrowing the search space and enhancing testing effectiveness.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we propose a black-box adversarial example generation method (ITGen) designed for both understanding and generation tasks, which iteratively leverages the evaluation



(a) Impact of hyper-parameter u. (b) Impact of hyper-parameter k.

Fig. 6. Impact of Main Parameters in ITGen.

information of the victim model from failed attacks to refine the generation process. We conducted an extensive study on nine subjects. The results show that ITGen can improve attack success rates over the state-of-the-art techniques (i.e., ALERT and BeamAttack) 47.98% and 69.70% in a shorter time.

In the future, we can improve ITGen from several aspects. First, we plan to explore broader attack strategies beyond identifier substitution, such as structural transformations and control flow modifications, by extending the current bitvector-based representation. Second, we will investigate more scalable optimization frameworks to handle larger and more complex code models, such as DeepSeek-Coder [51] and CodeLlama [52], to further improve the efficiency of ITGen. Third, we will expand our experimental comparisons to include attack strategies from other techniques like CODA [12] and STRATA [46] to provide a more comprehensive analysis. These future directions aim to make ITGen more versatile, efficient, and effective in generating adversarial examples for deep code models.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (No. 62372071), the Natural Science Foundation of Chongqing (No. CSTB2023NSCQ-MSX0914) and the Fundamental Research Funds for the Central Universities (No. 2023CDJKYJH013).

References

[1] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[2] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[3] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[4] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[5] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[6] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 87–98.

[7] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[8] "Github copilot," https://github.com/features/copilot, Accessed: 2023.

[9] 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.

[10] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 39–51.

[11] Y. Zhou, X. Zhang, J. Shen, T. Han, T. Chen, and H. Gall, "Adversarial robustness of deep code comment generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–30, 2022.

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

[13] 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.

[14] 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.

[15] 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.

[16] J. Zhang, W. Ma, Q. Hu, S. Liu, X. Xie, Y. Le Traon, and Y. Liu, "A black-box attack on code models via representation nearest neighbor search," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 9706–9716.

[17] X. Du, M. Wen, Z. Wei, S. Wang, and H. Jin, "An extensive study on adversarial attack against pre-trained models of code," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 489–501.

[18] Y. Li, S. Qi, C. Gao, Y. Peng, D. Lo, Z. Xu, and M. R. Lyu, "A closer look into transformer-based code intelligence through code transformation: Challenges and opportunities," *arXiv preprint arXiv:2207.04285*, 2022.

[19] W. Lv, Z. Wang, Y. Zheng, Z. Zhong, Q. Xuan, and T. Chen, "Buffersearch: Generating black-box adversarial texts with lower queries," *arXiv preprint arXiv:2310.09652*, 2023.

[20] S. N. Shukla, A. K. Sahu, D. Willmott, and J. Z. Kolter, "Black-box adversarial attacks with bayesian optimization," *arXiv preprint arXiv:1909.13857*, 2019.

[21] B. Ru, A. Cobb, A. Blaas, and Y. Gal, "Bayesopt adversarial attack," in *International Conference on Learning Representations*, 2019.

[22] M. Binois and N. Wycoff, "A survey on high-dimensional gaussian process modeling with application to bayesian optimization," *ACM Transactions on Evolutionary Learning and Optimization*, vol. 2, no. 2, pp. 1–26, 2022.

[23] D. Wipf and S. Nagarajan, "A new view of automatic relevance determination," *Advances in neural information processing systems*, vol. 20, 2007.

[24] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[25] "Itgen," https://github.com/unknownhl/ITGen, Accessed: 2024.

[26] X. Han, Z. Zhang, N. Ding, Y. Gu, X. Liu, Y. Huo, J. Qiu, Y. Yao, A. Zhang, L. Zhang *et al.*, "Pre-trained models: Past, present and future," *AI Open*, vol. 2, pp. 225–250, 2021.

[27] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang, "Pre-trained models for natural language processing: A survey," *Science China Technological Sciences*, vol. 63, no. 10, pp. 1872–1897, 2020.

[28] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 249–260.

[29] S. Liu, Y. Chen, X. Xie, J. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid gnn," *arXiv preprint arXiv:2006.05405*, 2020.

[30] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[31] 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.

[32] L. Chen, G. Zhang, and E. Zhou, "Fast greedy map inference for determinantal point process to improve recommendation diversity," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[33] J. Xu, R. Yang, Y. Huo, C. Zhang, and P. He, "Divlog: Log parsing with prompt enhanced in-context learning," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[34] W. Zhang, S. Guo, H. Zhang, Y. Sui, Y. Xue, and Y. Xu, "Challenging machine learning-based clone detectors via semantic-preserving code transformations," *IEEE Transactions on Software Engineering*, 2023.

[35] C. E. Rasmussen, "Gaussian processes in machine learning," in *Summer school on machine learning*. Springer, 2003, pp. 63–71.

[36] J. H. Heinbockel, *Introduction to tensor calculus and continuum mechanics*. Trafford Victoria, British Columbia, 2001, vol. 52.

[37] P. Schulman, "Bayes' theorem—a review," *Cardiology clinics*, vol. 2, no. 3, pp. 319–328, 1984.

[38] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global optimization*, vol. 13, pp. 455–492, 1998.

[39] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 476–480.

[40] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[41] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*, 2022, pp. 382–394.

[42] A. Kumar, S. Vembu, A. K. Menon, and C. Elkan, "Beam search algorithms for multilabel learning," *Machine learning*, vol. 92, pp. 65–89, 2013.

[43] "Tree-sitter," https://tree-sitter.github.io/tree-sitter/, Accessed: 2024.

[44] M. Balandat, B. Karrer, D. R. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy, "BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization," in *Advances in Neural Information Processing Systems 33*, 2020. [Online]. Available: http://arxiv.org/abs/1910.06403

[45] H. Hosseini, B. Xiao, M. Jaiswal, and R. Poovendran, "On the limitation of convolutional neural networks in recognizing negative images," in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2017, pp. 352–358.

[46] J. M. Springer, B. M. Reinstadler, and U.-M. O'Reilly, "Strata: simple, gradient-free attacks for models of code," *arXiv preprint arXiv:2009.13562*, 2020.

[47] P. Chen, Z. Li, Y. Wen, and L. Liu, "Generating adversarial source programs using important tokens-based structural transformations," in *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2022, pp. 173–182.

[48] A. Jha and C. K. Reddy, "Codeattack: Code-based adversarial attacks for pre-trained programming language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 12, 2023, pp. 14 892–14 900.

[49] J. Henkel, G. Ramakrishnan, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 526–537.

[50] 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.

[51] 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.

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