Attribution-guided Adversarial Code Prompt Generation for Code Completion Models

Xueyang Li
Institute of Information Engineering, CAS
School of Cybersecurity, UCAS
China

Guozhu Meng*
Institute of Information Engineering, CAS
School of Cybersecurity, UCAS
China

Shangqing Liu*
Nanyang Technological University
Singapore

Lu Xiang
Institute of Information Engineering, CAS
School of Cybersecurity, UCAS
China

Kun Sun
Institute of Information Engineering, CAS
School of Cybersecurity, UCAS
China

Kai Chen
Institute of Information Engineering, CAS
School of Cybersecurity, UCAS
China

Xiapu Luo
The Hong Kong Polytechnic University
China

Yang Liu
Nanyang Technological University
Singapore

ABSTRACT

Large language models have made significant progress in code completion, which may further remodel future software development. However, these code completion models are found to be highly risky as they may introduce vulnerabilities unintentionally or be induced by a special input, i.e., adversarial code prompt. Prior studies mainly focus on the robustness of these models, but their security has not been fully analyzed.

In this paper, we propose a novel approach AdvPro that can automatically generate adversarial code prompts for these code completion models. AdvPro incorporates 14 code mutation strategies at the granularity of five levels. The mutation strategies are ensured to make no modifications to code semantics, which should be insensitive to the models. Moreover, we leverage gradient attribution to localize the important code as mutation points and speed up adversarial prompt generation. Extensive experiments are conducted on 13 state-of-the-art models belonging to 7 families. The results show that our approach can effectively generate adversarial prompts, with an increased rate of 69.6% beyond the baseline ALERT. By comparing the results of attribution-guided localization, we find that the recognition results of important tokens in input codes are almost identical among different models. This finding reduces the limitation of using open-source alternative models to guide adversarial attacks against closed-source models. The results of the ablation study on the components of AdvPro show that CCMs focus on variable names, but other structures are equally crucial.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; • **Security and privacy → Software security engineering**.

KEYWORDS

Adversarial prompts, code completion models, attribution-guided localization

ACM Reference Format:
Xueyang Li, Guozhu Meng, Shangqing Liu, Lu Xiang, Kun Sun, Kai Chen, Xiapu Luo, and Yang Liu. 2024. Attribution-guided Adversarial Code Prompt Generation for Code Completion Models. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695517

1 INTRODUCTION

Language language models (LLMs) have gained enormous success in recent years and have been widely applied to solve software engineering tasks, such as code summarization, retrieval, and automatic completion. Benefiting from the mass amount of data and advanced learning algorithms, LLMs exhibit superior ability in comprehending code semantics. There are emerging several excellent models for code completion like Github Copilot [3], Codex [12] and CodeWhisper [1] (hereafter referred to as code completion models (CCMs)). According to [18], Github Copilot can improve the development efficiency by 55%.

However, CCMs are proven to be suffering from security issues [30], where they may produce vulnerabilities during code completion. The reasons for this phenomenon are manifold, as it may be attributed to either low-quality training data or misleading code prompts (e.g., code comments, naming conventions, and programming styles). Even worse, these issues are difficult to explain

*Corresponding author.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695517

and mitigate, considering the ever-increasing scale and complexity of CCMs. Therefore, it is necessary to evaluate the security of CCMs in generating code.

Prior studies have conducted security and functional tests for CCMs [24, 30, 32] and demonstrated the existence of vulnerabilities in these models [11]. However, the code employed in their tests mainly stems from algorithm contests or CodeQL or PyLint's examples, which may not realistically assess the weaknesses of CCMs in the actual product environments. Other studies construct adversarial examples to test the security and robustness of CCMs, by perturbing input samples to cause incorrect predictions. On the one hand, these adversarial attacks mainly target the classification task rather than the generation task. In particular, MHM [47], ALERT [44], and CARROT [46] employ perturbation strategies including renaming variables and adding junk code, in order to bypass vulnerability or clone detection. On the other hand, the contemporary attacks to code generation tasks do not focus on the semantics of the program, making the perturbed code syntactically or semantically incorrect. For example, the recent study CodeAttack [17], which replaces various types of tokens in code, can greatly degrade the performance of CCMs on code completion.

To this end, we propose an approach, termed as AdvPro, to generate adversarial code prompts and evaluate the security of CCMs. First, we construct a dataset with a massive number of Python vulnerabilities and use their code (i.e., vulnerable code before patching and secure code after patching) as the *ground-truth*. Given one code prompt for completion, CCMs may complete the code with vulnerable or secure code. AdvPro is designed to automatically mutate the code prompt and get vulnerable code in code completion. More specifically, we propose 14 semantic-preserving mutation strategies to code prompts from five granularities of code, including *token*, *expression*, *simple statement*, *compound statement* and *block*. To accelerate the generation of adversarial prompts, AdvPro proposes a novel searching approach based on *gradient attribution* [35, 43] to identify the combination of these mutation strategies to disguise CCMs. This process is repeated until the completed code by CCMs is vulnerable. The constructed prompt has no differences from the original in semantics, i.e., "semantically unperceived," but induces CCMs to generate vulnerable code. Therefore, this is a type of targeted adversarial attack in the domain of code completion.

AdvPro is extensively evaluated on 544 vulnerabilities in Python. The results show that: 1) Although the proportion of sec/vul code generated by all CCMs was similar, different families of CCMs show varying abilities in generating secure code as per CWE types. 2) The CCMs' robustness of the model is not always positively correlated with the parameter size. For the CodeGen family, the model robustness of 16B is lower than that of other versions. 3) Different families and versions of the model are similar for identifying important tokens in the input code, which helps us use the open-source model to guide the attack on the closed-source model. On text-davinci-003 based on GPT-3.5, we achieved a maximum transferability success rate of 25%. 4) CCMs pay more attention to variable names, but other structures are also important, and mutation rules on structures give AdvPro a 51% gain over ALERT without being attribution-guided.

Contributions. We make the following contributions in this study.

- We propose a novel approach AdvPro to create adversarial prompts that make CCMs generate vulnerable code effectively.
- We conduct extensive experiments to evaluate the effectiveness of AdvPro, and identify a number of intriguing issues in CCMs. These findings can benefit the security improvement of CCMs.
- We have made our dataset and code public at https://sites.google.com/view/advpro [4] for further research.

2 BACKGROUND

2.1 LLM-based Code Completion

Large language models are one type of deep learning models that are trained on vast amounts of textual data with advanced learning mechanisms (e.g., Transformer [40]), enabling them to capture the complex patterns, relationships, and structures inherent in the input text. LLMs, such as the latest OpenAI GPT-4 [28], have demonstrated expert-like capabilities in various natural language processing tasks. Code models have gained significant attention in software development in recent years, especially for code completion tasks [15, 36]. LLM-based CCMs are typically pre-trained on a large corpus of code and have sound reasoning and programming capabilities to predict and generate code fragments based on a given context (often referred to as "prompts") [39]. CodeX [12] is a GPT model trained on a GitHub repository for generating and understanding a wide range of code. Github Copilot [3], developed based on CodeX [12], has gained widespread use by programmers. In addition, open-source code models are rapidly developed, such as CodeParrot [38], CodeGen [26], and others trained in natural and multiple programming languages. In addition, many of the general-purpose LLMs also have powerful programming capabilities, such as ChatGPT [27], GPT-4 [28] and LLaMA [37], performing excellently in both natural and programming languages. So far, the GitHub Copilot plugin has received more than 6 million downloads on VSCode, and the similar product CodeWhisper [1] has received more than 1 million downloads. It is witnessed that the LLM-based CCMs have been widely integrated into today's software development process [39, 41], becoming a "shadow programmer" that profoundly affects the entire software system and all the stakeholders.

2.2 Adversarial Attacks for Code Model

The complexity of code generation tasks necessitates robust training methodologies and comprehensive evaluation frameworks. Adversarial attacks targeting code generation models can provide valuable insights into the vulnerabilities and limitations of these models. By perturbing the input samples in a targeted manner, adversarial attacks can expose weaknesses in the model's ability to generate code that is secure, efficient, and adheres to best practices.

Compared to other AI domains, such as image-based adversarial attacks, code-based adversarial attacks pose unique challenges. In image-based attacks, perturbations in pixel values might result in imperceptible changes to the human eye, yet lead to misclassification by the model. However, in code-based attacks, small changes in the code can have significant and noticeable effects on the program's behavior and correctness. A successful adversarial attack on the code model should have the following properties: (1) Correct syntax: the perturbed code can be compiled, interpreted, and

executed without any problems, (2) Program semantic consistency: the perturbed code is consistent with the original code in terms of program semantics and behaviors, (3) Minimal perturbation of the natural semantics: the least possible modifications to make it difficult for a human developer to notice the changes. Methods such as CARROT [46], ALERT [44], and MHM [47] counter attacks on code categorization tasks by modifying variable names. The semantic and syntactic correctness of the adversarial samples generated by these attack methods is guaranteed, but only for models based on the BERT architecture, such as CodeBERT [13], GraphCodeBERT [14]. CodeAttack [17] performs substitutions on various types of tokens without guaranteeing syntactic correctness, which will result in code that is practically inoperable. This type of adversarial attack has limited practical significance. The code completion task is much more difficult to confront, and in addition to the above properties, targeted attacks that can generate specific code are necessary due to the involvement of human developers in the development process, and the fact that completion results that differ too much from the developer's intent will be rejected.

3 MOTIVATION

Given one context of programming code, LLM-based code models can provide a number of suggestions to help complete the remaining code. However, prior studies [16, 30] show that these models may generate incorrect or even vulnerable code, posing a high risk on the built system. As shown in Figure 1a, the code snippet requires a parameter `verify` for statement `self.c.setopt(pycurl.SSL_VERIFYPEER, verify)`. The `verify` parameter is responsible for controlling SSL certificate verification: when setting to 1, the SSL certificate will be verified, and with 0, the verification will be skipped. Two popular LLM-based code completion models, GitHub Copilot [3] and CodeGen-16B-mono [26], recommend using 0 as the parameter value, which enables attackers to intercept data sent over HTTPS connections. This vulnerability has been recognized by the National Vulnerability Database as a high-risk vulnerability, with a severity score of 7.8. This example demonstrates the need to thoroughly assess the reliability and security of suggestions provided by LLM-based code completion models.

Furthermore, we have conducted an empirical study in this work to quantify the security of automatic code completion and found that, on average, 15.7% of code completions by each code completion model will produce vulnerabilities. Even worse, the secure code completion, i.e., the generated code is vulnerability-free, is susceptible of certain semantics-preserving transformations to the code prompt. Figure 1b demonstrates the vulnerability of the code completion model when the code prompt is altered. Given the original code as the prompt, the code completion model can suggest the secure API, `literal_eval()`, which means only evaluating literal expressions in a string, such as literal lists and dictionaries, but not calling other functions. If we assign an alias to `literal_eval()` (highlighted in line 3) and **`eval()`** (highlighted in line 5), respectively, the semantics of the code prompt remain the same. However, both models suggest the unsafe API **`eval()`**. Using **`eval()`** can lead to potential injection risks for servers with this vulnerable code, as it can execute arbitrary code within a string.

From this example, it can be observed that using the original code is insufficient to fully evaluate the security of CCMs, and appropriate mutations can solve this problem, helping us choose more secure "shadow programmers." In this paper, we aim to launch an attack to effectively construct these vulnerability-inducing code prompts mutated from benign code prompts. Generally, these vulnerability-inducing code prompts have the same semantics as their benign ones.

4 APPROACH OVERVIEW

The primary objective is to generate adversarial code prompts that can effectively lead code completion models to generate vulnerable code. To this end, we first devise semantics-preserving mutations that do not alter the semantics of the original code. A set of predefined rules is developed to ensure semantics consistency in the course of code mutation. In parallel, we propose an attribution-guided approach to identify the most influential parts of the input code. This information guides the creation of targeted adversarial samples by concentrating on perturbations in the critical locations, reducing the search space and increasing the likelihood of successfully fooling code completion models with minimal perturbations. Unlike the previous adversarial code method DAMP[45], AdvPro uses gradients to select perturbation positions rather than selecting specific replacement variable names. Our disturbance strategy classification is similar to RoPGen[21], but there is no need to select a reference code to specify the mutation direction.

Figure 2 shows an overview of our method. The process can be divided into the following stages:

Step 1. Data preparation: First, we extract the actual vulnerability code from the CVE database and GitHub, clean and annotate it to build our dataset, and then select the data from which the target model complements the security results as the input for searching adversarial samples.

Step 2. Attribution-guided Localization: Using the gradient information, we calculate the importance scores for each token in the prompt. This step helps us identify the most influential parts of the code that affect the model's output.

Step 3. Semantic-preserving Mutation: Based on the importance scores, we select the high-scoring tokens and apply semantic-preserving mutations to generate a modified prompt. This new prompt, similar to the original semantics, is designed to potentially trigger vulnerable code completions.

We continue to apply semantic-preserving mutations to the modified prompt and observe the model's output until either the model generates a vulnerable code or we reach a predefined iteration limit set to 20 in this paper.

Our approach is able to generate adversarial code samples that closely resemble the original input while exploiting the vulnerabilities of code generation models. This comprehensive and targeted strategy allows us to evaluate the security of code generation models better, paving the way for the development of defensive mechanisms to protect against adversarial attacks.

4.1 Dataset Preparation

To evaluate the security of CCMs in completing code, we first prepare a number of CVEs as well as their code snippets as *ground*

```
original code:(CVE-2023-0509)
......
    def init_handle(self):

        self.c.setopt(pycurl.FOLLOWLOCATION, 1)
        self.c.setopt(pycurl.MAXREDIRS, 10)
        self.c.setopt(pycurl.CONNECTTIMEOUT, 30)
        self.c.setopt(pycurl.NOSIGNAL, 1)
        self.c.setopt(pycurl.NOPROGRESS, 1)
        if hasattr(pycurl, "AUTOREFERER"):
            self.c.setopt(pycurl.AUTOREFERER, 1)
        self.c.setopt(pycurl.SSL_VERIFYPEER,__
Github Copilot  : 0) ✗
CodeGen-16B-mono: 0) ✗
```

(a) The insecure code generated for the original code prompts

```
mutated code:(CVE-2022-0845)
from ast import literal_eval
+lit_eval=literal_eval
+eval=eval
def parse_env_variables(cls: Type["pl.Trainer"],
......
    env_args = {}
    for arg_name, _, _ in cls_arg_defaults:
......
            with suppress(Exception):
                val = _____
Github Copilot  : literal_eval(val) ✓ -> eval(val) ✗
CodeGen-16B-mono: literal_eval(val) ✓ -> eval(val) ✗
```

(b) The insecure code generated for mutating code prompts

Figure 1: The insecure code generated by code completion models

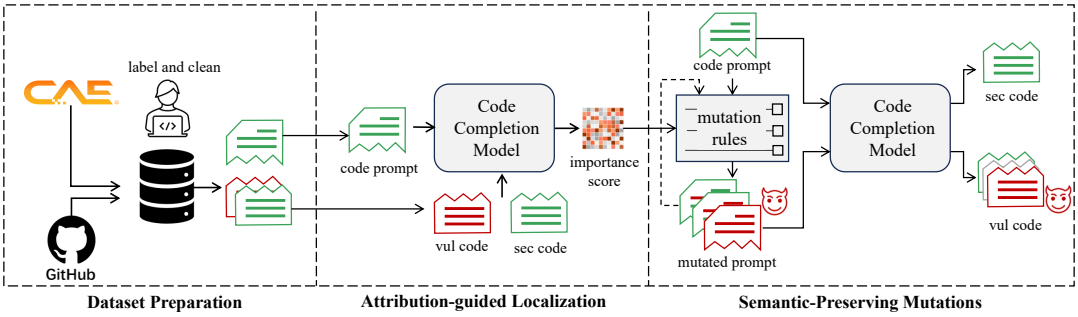


Figure 2: System Overview of AdvPro

truths. In particular, all the vulnerabilities in the Python project are collected from the NVD database [6]. Then, we only retain the vulnerabilities that have a reference (e.g., GitHub links) for the vulnerable code. Oftentimes, two versions of code snippets can be extracted from a GitHub commit where one of them is the insecure code before patching, and the other is the secure one after patching.

Data Cleaning and Labeling. To ensure the quality of the collected data, we asked five students to make data labeling and cleaning. These students, who have an average of three years of experience in security, are responsible for identifying and validating the vulnerable code and its patched version from all the changes in a commit. Each code snippet is reviewed by two students independently. If there is a discrepancy in annotations, a third student will be involved to reach a consensus. It takes us 3-man months to finish the process. Moreover, we triage each CVE with the CWE taxonomy [5] to enable a comprehensive analysis as per vulnerability type. For the CVEs with missing, outdated, or imprecise CWE categories, we manually reassigned a precise one.

To better evaluate the security of CCMs in completing code and sensitivity of adversarial prompts, we further refine these samples based on two strategies: 1) *security relevance*. One commit may make multiple changes to the code to fix vulnerabilities. We manually identify the security-related changes to construct our test dataset. 2) *single-line change*. To enable a high accuracy for CCMs in generating code, we only take into account the single-line changes.

Validation Set Construction. After obtaining a number of high-quality instances of vulnerabilities, we construct a validation set

```
code prompt:(CVE-2020-5227)
......
......
from lxml import etree
from lxml.etree import CDATA
......
def _add_text_elm(entry, data, name):

    if not data:
        return

    elm =
lineno       : 20
sec_code  : xml_elem(name, entry)
vul_code  : etree.SubElement(entry, name)
```

Figure 3: Illustrative code prompt and its responses.

for CCM evaluation. The validation set is a list of code snippets, comprised of vulnerable code and their patches, i.e., $\{x_0, x_1, \ldots, x_n\}$. Without loss of generality, we have the following definition for the dataset and its elements.

Definition 1. *Each sample x_i in the validation set is defined as a triple: $\langle p, v, s \rangle$, where p is a piece of code before vulnerability point, serving as a prompt of CCMs for code completion, v is the one-line code that makes the entire sample vulnerable, and s is the secure code that are patched for this vulnerability.*

As shown in Figure 3, line 9-20 is the code prompt (i.e., p) of CCMs, the secure code is "xml_elem(name, entry)" (i.e., s) and its

vulnerable version is "etree.SubElement(entry, name)" (i.e., v) that is susceptible to XML Denial of Service Attacks from carefully crafted data denial of service attacks against maliciously constructed data.

4.2 Attribution-guided Localization

To raise the effectiveness of adversarial prompt generation, we employ *attribution-guided localization* to localize the *critical* tokens in prompts that have a significant impact on model output. In this way, the search space for mutation can be greatly reduced. Specifically, we utilize the gradients of the model's output in relation to the input token embedding to help us determine which tokens have the most significant impact on the model's predictions. The following steps illustrate how to employ attribution-guided localization.

First, we perform forward computation on the target model. Assume that the code completion model is $F(\cdot)$. For an input sequence $x = [x_1, x_2, ..., x_n]$ of length n, we obtain the output probabilities p for each word in the vocabulary as follows:

$$p = F(x) \tag{1}$$

After the forward computation, we obtain the set of input sequence embeddings $e = [e_1, e_2, ..., e_n]$. For the secure label l_{sec} and the vulnerable label l_{vul}, their probabilities in the vocabulary are denoted as p_{sec} and p_{vul}:

$$p_{sec} = P_{sec}(F(x)) \quad p_{vul} = P_{vul}(F(x)) \tag{2}$$

Gradient attribution directly produces an attribution matrix B by computing the following partial derivative:

$$B = \frac{\partial(p_{vul} - p_{sec})}{\partial e} \tag{3}$$

Here, $B = [B_1, B_2, \ldots, B_n]$ represents the attribution matrix, where each element B_i corresponds to the partial derivative of the probability difference with respect to the input sequence embeddings e_n.

To obtain the final Importance Score $S = [s_0, s_1, ..., s_n]$, we compute the norm of each element in the attribution matrix B_i:

$$s_i = \|B_i\| = \sqrt{\sum_{j=1}^{size_{emb}} (B_i)_j^2} \tag{4}$$

After the above calculations, we have obtained the importance scores for the input sequence. However, these scores are assigned to single tokens that may not have complete syntax. For example, The function name '_add_text_elm' in Figure 3 is recognized as '_', 'add', '_', 'text', '_', 'elm' by the model tokenizer but '_add_text_elm' by the python tokenizer. To this end, we construct the abstract syntax tree and aggregate the tokens as well as their scores. This process can be achieved as Algorithm 1.

4.3 Semantic-Preserving Mutations

To evaluate the security of the CCMs, the key assumption is that they should be consistent in front of the input code with same semantics. Thus, by detecting completion outputs with high inconsistency, potential errors in CCMs can be identified and flagged.

We have developed a set of semantic-preserving mutation rules that can generate a series of new code prompts that are semantically equivalent to the original code prompts. All rules can be composed of the following three atomic operations.

Algorithm 1: Importance Score Mapping

```
Input: code: string; tokenSeq: List[string]; scoreSeq: List[float];
Output: root: astNode;
mappedTokens = getPositon(code,tokenSeq)
root = getAST(code)
foreach node in ast.walk(root) do
    foreach token in mappedTokens[node.lineo] do
        if isInclude(node,token) then
            node.score+=scoreSeq[Index(tokenSeq,token)]
```

- **Adding nodes**: Inserting new nodes into the AST, such as adding a new variable definition, function call, conditional statement, and loop structure. The newly added nodes should not change the semantics of the original.
- **Replacing nodes**: Replacing an existing node with a new node with the same functionality and semantics. This operation includes changing variable names, transforming loop structures (such as converting a `for` loop to a `while` loop), simplifying or expanding conditional expressions, etc. When replacing nodes, you need to ensure that the new node is functionally and semantically equivalent to the original node.
- **Reordering nodes**: Changing the order of nodes in the AST, such as swapping two independent code blocks, reordering function and class definitions, and rearranging method implementations.

Based on Python Language Reference [7], we have designed a total of 14 mutation strategies at five levels, ranging from fine-grained (token) to coarse-grained (block).

4.3.1 Token. Tokens in Python can be categorized into five classes: *identifier*, *string*, *numeric literal*, *operator*, and *delimiter*. Usually, many of CCMs normalize arbitrary strings and numeric literals in code into a unique form, so that the mutations to these tokens leave no impact on CCMs' output. Furthermore, modifying operators and delimiters may significantly alter the syntax and semantics of one program. Hence, at the token level, we propose mutation rules only for the tokens of type *identifier*.

Identifier. It covers the name for variables, functions, classes, and methods. The mutation strategy is as follows.

- Replace identifiers with new names, e.g., replace 'elm' in Figure 3 with 'sijd'.

We propose a mutation strategy to rename the target identifier, in order to verify whether CCMs are sensitive to these identifiers. To ensure the mutation not altering the semantics, the substitute is randomly generated without conflicting with existing identifiers.

4.3.2 Expression. Expressions are the building blocks of Python code and can be combined to create more complex expressions. At the expression level, we have developed mutation rules for commonly encountered expression types, which involve altering the form of expressions or transforming them into other statements.

Assignment expressions. The syntax for an assignment statement is "identifier = expression". An assignment expression involves assigning an expression to an identifier and simultaneously returning the value of that expression. We propose the following mutation rule:

- Add line breaks to separate the content within the parentheses onto different lines when there are various parentheses in the expression e.g., modify 'elm=xml_Elem(name, entry)' to "elm=xml_Elem(name,\\entry)".

Conditional expressions. Conditional expressions are simple representation of two-branch structures in Python which have the form " x if C else y", where C is a condition. First, the condition C is evaluated. If C is true, x is executed and its value is returned. We propose the following mutation rules for conditional expressions:

- Convert the conditional expression to an if-else statement.
- Modify the condition (C) and swap the positions of x and y: " y if not C else x".

Lambda expressions. Lambda expressions are used to create anonymous functions. The syntax for a lambda expression is lambda parameters: expression. It yields a function object, and the unnamed object behaves like a function object. We propose the following mutation rule for lambda expressions:

- Convert the lambda expression to a function definition.

Comprehension expressions. Comprehensions are concise and readable syntax for creating lists, dictionaries, and sets in Python. For example, "[i for i in range(1,10)]" will return a list from 1 to 9. They consist of an expression followed by a for clause and an optional if clause. The expression is evaluated for each item in the iterable specified in the for clause, and if the if clause is present, the item is included only if the condition is met. We propose the following mutation rule for comprehensions:

- Convert the comprehension to an equivalent loop and conditional statement.

4.3.3 Simple statement. A simple statement is comprised within only one logical line. Since most simple statements have special and simple capabilities, we designed mutation rules for only five of the 14 classes of simple statements. that can ensure that modifications to them do not affect semantics.

Expression statements. Expression statements are standalone expressions that are used for their side effects. For example, function calls and print statements are considered expression statements because they perform an action but do not store any value. We propose the following mutation rule for expression statements:

- Convert the expression statement to an assignment statement.

We create a random string no longer than 10 as a new variable and select a variable that does not exist in the original code to store the return value of the function, in order to avoid affecting the semantics of the code.

Assignment statements. The syntax for an assignment statement is "target_list = expression". what's the difference with assignment expression Augmented assignment statements are a shorthand way of updating the value of a variable with a binary operation. Examples of augmented assignment statements include +=, -=, *=, and /=. e.g., modify "a += 1" to " a = a + 1"

We propose the following mutation rule for augmented assignment statements:

- Convert the augmented assignment statement to a regular assignment statement.

Assert. Assert statements are used to check if a condition holds true, and if not, raise an AssertionError. The syntax for an assert statement is "assert condition[, error_message]". If the condition evaluates to False, an AssertionError is raised with the optional error message.

We propose the following mutation rule for assert statements:

- Convert the assert statement to an if statement that raises an AssertionError when the condition evaluates to False.

4.3.4 Compound statement. Compound statements are multi-line statements that can control the flow of a program or perform more complex actions. They consist of one or more clauses, where each clause contains a header and a suite. In this section, we discuss some common compound statements in Python, including if, while, and for statements, and propose mutation rules for them.

if statement. The if statements are used to conditionally execute a block of code. They can include elif (short for "else if") clauses and an optional else clause. The syntax for an if statement is "if condition: suite [elif condition: suite]* [else: suite]?".

The mutation rule for if statements is:

- Invert the condition and swap the if and else suites.

Due to the limited capability of handling only two-branch structures, conditional expressions cannot be used to convert all if statements. As a result, we have not designed any rules for transforming if statements into conditional expressions.

while statement. The While statements are used to repeatedly execute a block of code as long as a condition is true. The syntax for a while statement is "while condition: suite".

The mutation rule for while statements is:

- Add a break statement when the condition evaluates to True inside the "while True: suite".

for statement. The for statements are used to iterate over a sequence (such as a list, tuple, or string) and execute a block of code for each element in the sequence. The syntax for a for statement is "for variable in iterable: suite".

The mutation rule for for statements is:

- Replace the for loop with a while loop and manually handle the iteration using "next()".

with statement. The with statements are used to simplify the management of resources such as files or network connections. They ensure that the resource is properly acquired and released. The syntax for a with statement is "with expression [as variable]: suite". The expression is usually an object with __enter__ and __exit__ methods, which are called at the beginning and end of the suite, respectively.

The mutation rule for with statements is:

- Convert the with statement to a try-finally statement.

4.3.5 block. A block in programming refers to a group of statements that are executed together as a single unit. At the block level, we do not modify individual statements, but instead modify blocks by inserting statements. We propose the following mutation rules for blocks:

APIs. There may be many API invocations in a block. We create an alias for these APIs to obscure their original usage as below.

- Add an alias for imported APIs to obscure their original names.

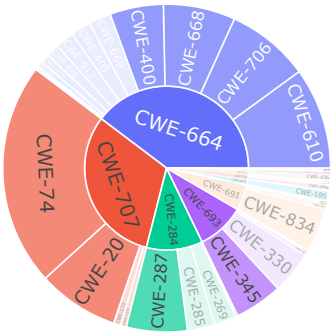


Figure 4: CWE category distribution

5 EVALUATION SETUP

In this section, we introduce the setup for the experiments including the used datasets, code completion models, and the metrics for evaluation. We intend to answer the following questions:

RQ1. How secure is the code that generated by CCMs in the real-world scenario?

RQ2. How effective is our method in inducing CCMs to generate insecure code?

RQ3. How does each component of our method contribute to the overall effectiveness?

RQ4. How does the transferability of our method between different models?

5.1 Datasets

After filtering process, we collected 682 CVEs and 1,496 pairs of Python files. Each pair consisted of a vulnerable version and its corresponding fixed version, providing valuable insights into the code transformation required to address the security issues. After further annotation, we have retained 544 security relevant and single-line change samples from a total of 284 CVEs. Each sample in the dataset includes a code prompt, vulnerability code position, secure output, and vulnerable output.

Within the Research Concepts view (CWE-1000) classification, our dataset encompasses 36 CWE categories belonging to 10 CWE Pillars. The distribution of vulnerability types and time in this dataset is shown in Figure 4. This extensive coverage provides a comprehensive and diverse range of vulnerabilities for analysis.

This dataset is valuable for our research, and we can evaluate the models' security performance by studying the code prompts and outputs. The dataset's availability facilitates a thorough analysis of CCMs and their susceptibility to adversarial attacks, serving as a basis for future research and developing robust techniques to improve code generation system security.

5.2 Code Completion Models

Our goal is to comprehensively evaluate the security aspects and the effectiveness of our adversarial attack approach on the most recent and popular Large Language Models (LLMs). Therefore, we have selected the following LLMs for our evaluation, as summarized in Table 1. These models have varying parameter counts ranging from 0.11B to 16B. The earliest model, CodeParrot [38], was released in May 2022, while the most recent model, StarCoder [19], was released in May 2023. These models are based on the GPT-2 [31] architecture, but StarCoder and CodeGen2 [25] also support Fill-in-the-Middle objective.

Table 1: Code Completion Models

Model	Size	Dataset
CodeGen [26]	{0.35B, 2B, 6B, 16B}	BIGPYTHON [26]
CodeGen2 [25]	{1B, 3.7B, 7B, 16B}	The Stack (v1.1) [8]
CodeParrot [38]	{0.11B, 1.5B}	CodeParrot Dataset [2]
PolyCoder [42]	{0.45B, 2.7B}	PolyCoder's Data [42]
SantaCoder [10]	{1.1B}	Python/Java/JS subset of The Stack (v1.1) [8]
TinyStarCoderPy [19]	{0.16B}	Python subset of The Stack (v1.2) [9]
StarCoder [19]	{15.5B}	The Stack (v1.2) [9]

For all open-source models, we use the default settings of each model, in addition to setting the temperature to 0 and greedy mode to make the model generation results stable.

5.3 Metrics

In order to accurately evaluate the security performance of CCMs and the effectiveness of our adversarial attack, we have proposed multiple evaluation metrics divided into two categories: security evaluation metrics and adversarial attack metrics.

Model performance often varies for different tasks and scenarios, so to assess the security performance of code completion models in various situations, we have selected the eight most common CWE-IDs in our dataset, which are distributed across four Pillars. We have designed two evaluation metrics for this purpose:

Number of Secure/Vulnerable Completions. This metric measures the number of code completions that are either secure or vulnerable for each of the models. If the completion matches exactly with the patch code, it is considered secure; if it matches exactly with the vulnerability code, it is considered vulnerable.

Fuzzy Security Score (FS-score). Since completion is often different from vulnerability code and patch code, we have designed FS-score to quantify the security of such completion. This metric calculates the BLEU scores [29] of the generated results and target ouputs separately, quantifies the overall security of the generated code snippets for each of the selected CWE-IDs by assigning a continuous score between 0 and 1. The higher the score, the more similar the completion and patch code are, and the higher the security is. If the completion result matches the secure/insecure label for each sample, the score is 1/0. If the completion result does not fully match either label, the FS-score is computed using the following formula:

$$FS - socre = \frac{BLEU_{sec}(pred)}{BLEU_{sec}(pred) + BLEU_{vul}(pred)}$$

For a given CWE, compute the average FS-score across all associated samples. The Fuzzy Security Score accounts for the varying degrees of security in the generated code, offering a more nuanced understanding of the model's performance in different situations.

Attack Success Rate (ASR). ASR represents the percentage of input samples for which the attack method can successfully generate adversarial samples. The larger the ASR, the more effective the attack method.

Transfer Success Rate (TSR). TSR represents the success rate of the adversarial code prompts generated on the source model to complete adversarial attacks on the target model. By evaluating TSR

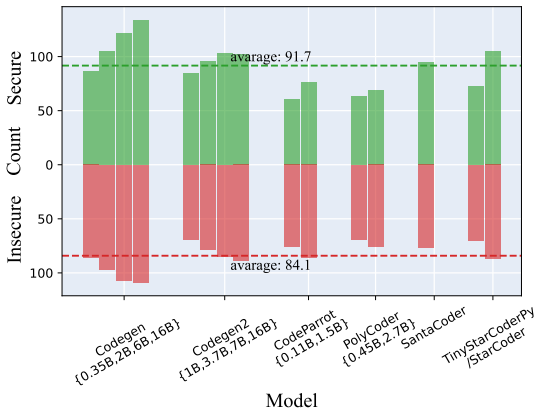


Figure 5: Statistics of (in)secure code completions of all CCMs

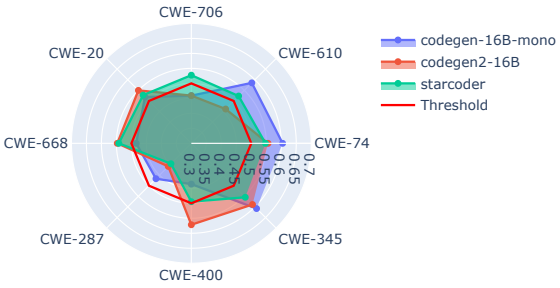


Figure 6: Security performance of 10B+ models

across multiple models, we can better understand the transferability of these adversarial code prompts.

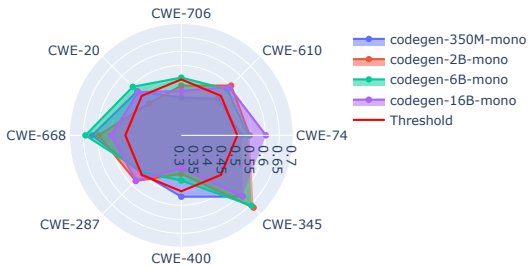
6 EVALUATION RESULTS

In this sections, we present the experimental results in light of research questions.

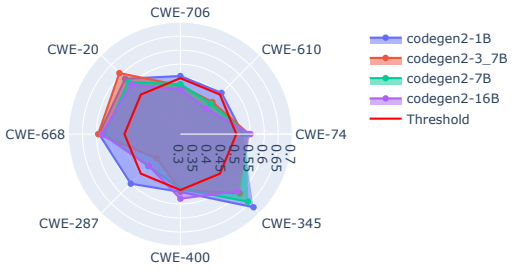
6.1 RQ1: Security of Code by CCMs

In this section, we aim to determine the security of the generated code by different CCMs. First, we feed the collected 544 samples in Section 5.1 into CCMs, and identify whether the generated code is secure or not. Figure 5 shows the overall performance of these CCMs. On average, these CCMs produce 94.8 (17.4%) secure completions and 85.8 (15.7%) vulnerable completions. It is observed that for the models with the same architecture, the number of secure and vulnerable completions increases along with the number of parameters. However, the change in the proportion of secure and vulnerable completion results is small. Probably because the programming ability of the model increases with the number of parameters, but its security is an inherent problem due to its training data and architecture. It does not increase with the number of parameters. In addition, there are significant differences in the performance of the models with the same size of parameters, where CodeGen-16B performs the best.

To evaluate the security of the model under different scenarios, we computed FS-scores under each of the eight CWEs in Table 2 and plotted them as the radar plots shown in Figure 6. We placed



(a) Performance of CodeGen models



(b) Performance of CodeGen2 models

Figure 7: Performance of the same architecture model

the models with more than 10 billion parameters in one graph and observed the difference in their capabilities. The octagon marked by the red line is the threshold 0.5, and those above 0.5 are safe at that CWE, and those below 0.5 are not. The figure shows that the models' safety varies widely on the remaining CWEs except for CWE-668 and CWE-20. We can also observe that CodeGen2-16B achieved the highest score on CWE-400. This difference implies that it would be an excellent choice to pick the model with higher security for the corresponding scenario.

In Figure 7, we compare the security evaluation results of models with the same architecture but with different parameters. In this case, Figure 7a depicts the CodeGen family, and Figure 7b depicts the CodeGen2 family. We can observe that the shapes of the radar plots of the models with different architectures in Figure 6 vary widely. In contrast, the radar plots of the models with the same architecture are relatively similar. It is worth noting that for the models with the same architecture, it is not the case that the more parameters, the higher the FS-score. In both CWE-400 in Figure 7a and CWE-287 in Figure 7b, the highest FS-score is obtained for the model version with the least parameters.

In summary: The models of the four architectures with the most powerful programming capabilities cannot guarantee security in the eight security scenarios we set, and there are significant differences in the performance of the models of different architectures in each security scenario. In contrast, the security performance of models with different parameters in the same series is more similar. In the same series, the models with more parameters have more powerful programming capabilities, but are not definitely more secure.

Table 2: Performance of code models

Model	CWE-74	CWE-610	CWE-706	CWE-20	CWE-668	CWE-287	CWE-400	CWE-345
CodeGen-350M-mono	0.505	0.485	0.435	0.523	0.619	0.494	0.520	0.610
CodeGen-2B-mono	0.544	0.552	0.478	0.462	0.596	0.531	0.438	0.666
CodeGen-6B-mono	0.524	0.574	0.507	0.546	0.573	0.445	0.499	0.655
CodeGen-16B-mono	**0.604**	**0.585**	0.459	0.518	0.484	0.467	0.436	0.608
CodeGen2-1B	0.513	0.508	0.508	0.580	0.591	**0.551**	0.507	**0.670**
CodeGen2-3.7B	0.546	0.462	0.474	**0.608**	0.594	0.421	0.500	0.600
CodeGen2-7B	0.544	0.490	0.478	0.563	0.531	0.426	0.534	0.642
CodeGen2-16B	0.556	0.462	0.459	0.550	0.547	0.409	**0.572**	0.589
CodeParrot-small	0.491	0.488	0.410	0.577	0.589	0.501	0.442	0.526
CodeParrot	0.492	0.476	0.490	0.527	0.509	0.422	0.456	0.618
PolyCoder-0.4B	0.503	0.467	0.424	0.546	0.561	0.460	0.480	0.529
PolyCoder-2.7B	0.508	0.447	0.451	0.529	0.584	0.510	0.562	0.613
SantaCoder	0.508	0.508	0.476	0.599	**0.650**	0.409	0.473	0.636
StarCoder	0.547	0.523	**0.527**	0.527	0.542	0.396	0.495	0.555
Average	0.528	0.502	0.470	0.547	0.569	0.460	0.494	0.608

6.2 RQ2: Effectiveness of Adversarial Code Prompts

To demonstrate the effectiveness of our method, we select the samples in RQ1 where the model complements the secure label for searching the adversarial samples. We selected 6 series with a maximum input length of 2048, for a total of 13 models, to control for consistent input lengths supported by the models. The experimental results are shown in Table 3.

For all models, the ASR of our approach is higher than 25%. The average ASR is 39.3%, which means that at least 25% of the code prompts are transformed to induce CCMs to generate vulnerabilities. Among all the tested models, CodeGen2-16B has the lowest ASR of 26.5%, and PolyCoder-0.4B has the highest ASR of 58.7%, which also indicates that CodeGen2-16B has the best robustness, while PolyCoder has the weakest robustness. In the PolyCoder and CodeGen2 families, ASR decreases and robustness increases with the increase of model parameters, but in the CodeGen family, the 16B version with the largest parameters achieves the highest ASR, so we believe that the parameter size is not necessarily related to robustness. It is also found that the robustness of CodeGen2 is higher than that of CodeGen for similar parameter magnitudes, which indicates that the architecture is important factors for the robustness of the code model. Our approach achieved higher ASR than ALERT on all models, with an average ASR gain of 69.6% compared to ALERT, while completing full adversarial sample generation on each of our models in only one-fifth the time of ALERT.

In summary: Our method outperforms the baseline method significantly, with the state-of-the-art ASR of 39.3% on average. It makes a 69.6% increase rate beyond the prior method.

6.3 RQ3: Ablation Study

In order to study the contribution of each component, we constructed six variant methods for ablation experiments, which are described as follows:

- **w/o Attr-Guide:** We replace attribution-guide localization with a random selection of AST nodes.
- **w/o Token:** Removing mutation rules for tokens, i.e., do not replace identifiers in code prompts.

Table 3: Comparison results of Attack Success Rates (ASR) on attacking all CCMs

Model	Size	Sec preds	ASR	ALERT
CodeGen	0.35B	86	41.9%(+79.8%)	23.3%
	2B	105	40%(+35.6%)	29.5%
	6B	122	38.5%(+23.8%)	31.1%
	16B	135	52.2%(+74.6%)	29.9%
CodeGen2	1B	85	44.7%(+110.8%)	21.2%
	3.7B	96	32.3%+63.1%)	19.8%
	7B	103	30.1%(+29.2%)	23.3%
	16B	102	26.5%(+68.8%)	15.7%
PolyCoder	0.4B	63	58.7%(+189.2%)	20.3%
	2.7B	69	39.1%(+92.6%)	20.3%
SantaCoder	1.1B	95	36.8%(+94.7%)	18.9%
TinyStarCoderPy	0.16B	73	38.4%(+27.6%)	30.1%
StarCoder	15.5B	106	32.1%(+77.3%)	18.1%
Average	—	—	39.3%(+69.6%)	23.2%

Table 4: Ablation tests in terms of average ASR. Δ ASR shows the increase or decrease rate with the mutation strategies.

Ablations	ASR	Δ ASR
AdvPro	39.3%	-
w/o Attr-Guide	35.1%	+12.0%
w/o Token	23.4%	+67.9%
w/o Expr	32.6%	+20.6%
w/o Simple-Stmt	33.2%	+18.4%
w/o Comp-Stmt	33.7%	+16.6%
w/o Block	22.2%	+77.0%

- **w/o Expr:** Removing mutation rules for expressions.
- **w/o Simple-Stmt:** Removing mutation rules for simple statements.

- **w/o Comp-Stmt:** Removing mutation rules for compound statements.
- **w/o Block:** Removing mutation rules for code blocks.

Table 4 shows the average ASR of these 6 variants of the method, with the complete-ground method improving 12.0% to 67.9% over the variant method. This means that the main components all contribute significantly to the results.

The possible reason for the smaller improvement of attribution guide than mutation rules is that the degree of efficiency of attribution-guide is affected by the length of the code prompt, and the probability of randomly selecting the most important node is higher for shorter code prompts, while the mutation strategy is effective for code prompts of all lengths.

Within mutation rules, token and block have significantly higher boosts than other granularity rules. This is probably because token and block structures are most prevalent in code hints. Another possible explanation is that the model is more sensitive to identifiers or APIs than other statements.

In summary: Our localization method and each mutation rule are conducive to generating adversarial samples, resulting in 12% to 67.9% gains. When all the ingredients are combined, the highest antagonistic success rate is obtained.

6.4 RQ4: Transferability of Adversarial Code Prompts

We will discuss the transferability of the method from three aspects: the transferability of the importance score, the transferability of adversarial prompts.

Transferability of Importance score. We use the gradient attribution method to generate the Importance score for each input sequence, and the ablation experiment in Section 6.3 the positive effect of Importance Score on generating adversarial examples. Meanwhile, the Importance Score also reflects the model's understanding of the input code. To evaluate the transferability of the Importance Score of different models, we use the Importance Score to rank the variable names of the input code and calculate the Location Square Deviation (LSD) of the ranked sequence for the same input sample. The LSD of sequences $X = x_1, x_2, ..., x_n$ and $Y = y_1, y_2, ..., y_n$ can be computed by $LSD(X, Y) = \sum_{i=1}^{n}(i - \text{index}(x_i, Y))^2$. The numbers in figure 8 represent the average LSD of the sorted sequence of variable names between the two models, where the green number is the smallest value in the row and the red one is the largest value in the row. We can find that all the models with the most consistent sequences are in the same family. Our input code contains an average of 18 variable names, and the largest average LSD in figure 8 is only 10.31, and the smallest is 3.01, which means that the impact of the input code on the output is similar even between the two models with the largest difference.

Transferability of adversarial prompts. We investigated the transferability of adversarial samples between different models. Figure 9 shows the results, with each square representing the transferability of adversarial samples from the source model to the target model. Darker squares indicate higher transferability. The highest TSR was 62%, while the lowest was 9%.

Among the source models, CodeGen2-3.7B, and CodeGen2-16B exhibited significantly higher transferability than other models.

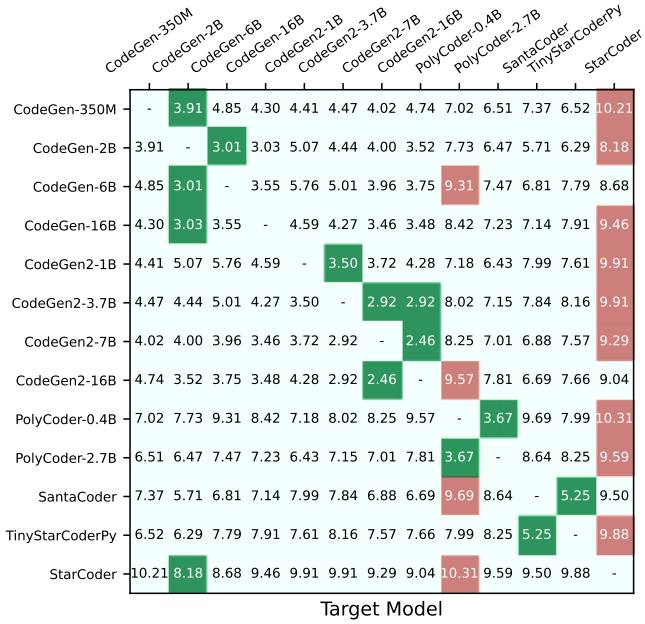


Figure 8: Average LSD of variable name sequences for all code models

Conversely, PolyCoder-2.7B had the lowest transferability as a source model but the highest transferability as a target model. This suggests a negative correlation between the robustness of the model and transferability. Additionally, adversarial examples had lower transferability between models of different families than those within the same family.

To further explore transferability, we selected the closed-source model text-davinci-003 as the target. Figure 9 shows that text-davinci-003 had significantly lower transferability than all open-source models, indicating its good robustness. However, our method achieved a TSR of 26%, meaning that although direct white-box attacks on closed-source models may not be feasible, we can still generate adversarial samples to attack them indirectly.

In summary: The importance scores of the different models are very similar, with the highest LSD being only 10.21 and the most similar models averaging 3.87 LSD, often belonging to the same family, indicating the potential of using open-source models to guide attacks against closed-source models. Experiments show that CodeGen2-16B produces the highest transferable admissible samples across a wide range of target models, with an average TSR of 40.9%. The robustness of both the source model and the target model affects portability.

7 THREATS TO VALIDITY

Internal threats. 1) The metrics for security evaluation are sometimes insufficient to reflect the security of CCMs. In the case that the generated results do not fully match secure and vulnerable code, it may involve false positives with text similarity-based methods. It is mitigated to some extent in this study by only considering the code prompts that can be completed with the exact code as the ground-truth. In future, we will leverage static analysis to better determine the security of generated code. 2) Some mutation rules

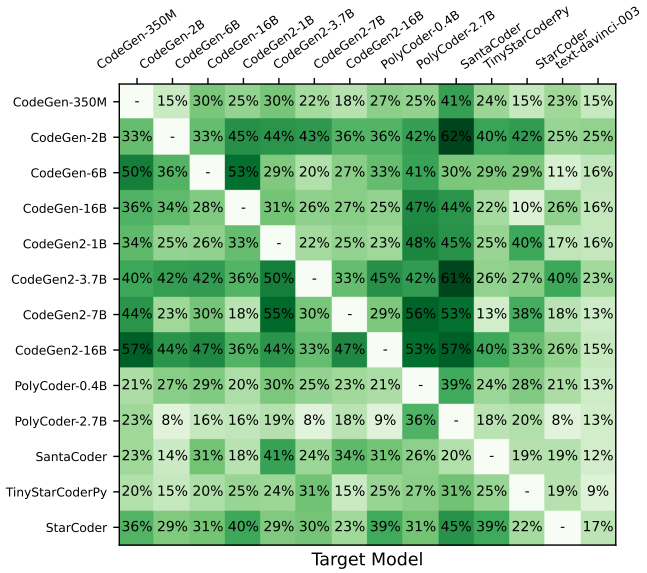


Figure 9: Adversarial samples transferability

introduce temporary variable or function names (e.g., "lambda expression" and "for statement"). That may make additional effects on the completion results.

External threats. 1) Limitations of the data set. In this study, we only consider the single-line vulnerabilities, which may belong to specific types. To this end, we try to measure the security CCMs as per the type of vulnerabilities. 2) Code comments. AdvPro only mutate the programming language of the code, but the natural language portion of the code comments can also have a significant impact on the generated results, which will be explored in future.

8 RELATED WORK

Attacks against LLM-based Code Model. Although there are a line of works on adversarial attacks to code models [17, 22, 23, 44, 47, 48], they focus mainly on classification tasks such as clone detection, malware detection and classification. For example, Zhang et al. propose MHM [47], which uses the Metropolis-Hastings algorithm to iteratively replace identifiers in the code for generating adversarial samples. While both CARROT [46] and ALERT [44] outperform MHM [47], where CARROT proposes a series of perturbation rules, including token and statement levels, mainly renaming and inserting irrelevant codes, and proposes a method to evaluate the robustness of the code model. ALERT involves only identifier changes but focuses on the naturalness of the changed names. For CodeBERT [13], GraphCodeBERT [14] has achieved SOTA attack results. Jha et al. proposed CodeAttack [17] to adversarial attack pre-trained programming language models in the natural channel of code. CodeAttack performs black-box untargeted adversarial attacks on programming models by replacing tokens. Due to the lack of syntax and semantic constraints, the adversarial samples generated by CodeAttack may contain syntactic errors, and the attack goal is limited to degrading the performance of programming models. Li et al. proposed CCTEST [22] for testing and improving the robustness of code generation models. CCTEST makes the model generate inconsistent code fragments by simple syntactic transformations and analyzes the similarity of these fragments to find the average output and improve the robustness of the model output without fine-tuning the model. In addition, there are also some works [20] exploring the backdoor attack for code models. *Unlike the above work, our work designs the first targeted adversarial attack method for generating tasks that generate adversarial samples with security vulnerabilities in a state space that is far more complex than the classification task.*

Security Issues of Code Completion Models. Schuster et al. [33] propose a backdoor implantation attack on code completion models, which involves poisoning the data to enable the model to learn abnormal features that may harm users. Since the release of GitHub Copilot, there has been significant security research on code completion models. Firstly, Pearce et al. [30], using CodeQL examples as input and checking the security of Copilot completion results, systematically studied the security and influencing factors of Copilot. Siddiq et al. [34] constructed a dataset of 130 prompts using examples of three types of code checking tools and tested Copilot and Encoder, all of which confirmed the security risks of the model. In addition, Gustavo et al. [32] also used user research to have students write code for specific scenarios with the assistance of the Code Generation Model, believing that LLM assistance would not introduce new security risks. However, since most of them are mathematical algorithm codes, they cannot reflect the security of the model in actual production. *Unlike the above work, we have constructed an actual vulnerability code prompt dataset, which can help us quantitatively evaluate the security of different code generative models in different security scenarios.*

9 CONCLUSION

In this paper, we propose the approach AdvPro to generate adversarial code prompts for CCMs. It is enabled by 14 semantic-preserving mutations and the guidance of gradient attribution for acceleration. Our experiments on a wide range of CCMs have demonstrated the efficacy of the proposed attack, achieving an average success rate of 39.3% that greatly outperforms the baseline. Our research not only contributes to the discovery of model vulnerabilities, but also explores the characteristics of input code understanding by different models.

10 ACKNOWLEDGMENT

We would thank the anonymous reviewers for their valuable comments. The IIE authors are supported in part by CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118), NSFC (92270204), Youth Innovation Promotion Association CAS and Beijing Nova Program.

REFERENCES

[1] 2022. Amazon CodeWhisperer. https://aws.amazon.com/cn/codewhisperer/, as of September 16, 2024.
[2] 2022. CodeParrot Dataset Cleaned. https://huggingface.co/datasets/codeparrot/codeparrot-clean
[3] 2022. GitHub Copilot. https://github.com/features/copilot, as of September 16, 2024.
[4] 2023. ADVPRO. https://sites.google.com/view/advpro
[5] 2023. CWE VIEW: Research Concepts. https://cwe.mitre.org/data/definitions/1000.html
[6] 2023. NIST National Vulnerability Database. https://nvd.nist.gov/

[7] 2023. The Python Language Reference. https://docs.python.org/3/reference/index.html

[8] 2023. The Stack v1.1. https://huggingface.co/datasets/bigcode/the-stack

[9] 2023. The Stack v1.2. https://huggingface.co/datasets/bigcode/starcoderdata

[10] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988* (2023).

[11] Owura Asare, Meiyappan Nagappan, and N. Asokan. 2022. Is GitHub's Copilot as Bad As Humans at Introducing Vulnerabilities in Code? (Apr 2022).

[12] MarkI-Cheng Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, HenriquePondeOliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, AlexK. Ray, Raul Puri, Gretchen Krueger, MichaelA. Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, N.C. Ryder, Mikhail Pavlov, Alethea. Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, FelipePetroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, ElizabethA. Barnes, Ariel Herbert-Voss, WilliamC. Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, WilliamH. Saunders, Christopher Hesse, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, MatthewM. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Samuel McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *Cornell University - arXiv,Cornell University - arXiv* (Jul 2021).

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Fu Shengyu, Michele Tufano, ShaoKun Deng, ColinB. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. *Cornell University - arXiv,Cornell University - arXiv* (Sep 2020).

[15] Qi Guo, Xiaohong Li, Xiaofei Xie, Shangqing Liu, Ze Tang, Ruitao Feng, Junjie Wang, Jidong Ge, and Lei Bu. 2024. FT2Ra: A Fine-Tuning-Inspired Approach to Retrieval-Augmented Code Completion. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 313–324.

[16] Yaru Hao, Li Dong, Furu Wei, and Ke Xu. 2021. Self-Attention Attribution: Interpreting Information Interactions Inside Transformer. *Proceedings of the ... AAAI Conference on Artificial Intelligence,Proceedings of the ... AAAI Conference on Artificial Intelligence* (May 2021).

[17] Akshita Jha and Chandan K Reddy. 2023. Codeattack: Code-based adversarial attacks for pre-trained programming language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 14892–14900.

[18] Eirini Kalliamvakou. 2022. *Research: Quantifying GitHub Copilot's Impact on Developer Productivity and Happiness*. https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/

[19] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! (2023). arXiv:2305.06161 [cs.CL]

[20] Yanzhou Li, Shangqing Liu, Kangjie Chen, Xiaofei Xie, Tianwei Zhang, and Yang Liu. 2023. Multi-target backdoor attacks for code pre-trained models. *arXiv preprint arXiv:2306.08350* (2023).

[21] Zhen Li, Guenevere Qian Chen, Chen Chen, Yayi Zou, and Shouhuai Xu. 2022. RoPGen: Towards Robust Code Authorship Attribution via Automatic Coding Style Transformation. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 1906–1918. https://doi.org/10.1145/3510003.3510181

[22] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. Cctest: Testing and repairing code completion systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1238–1250.

[23] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. Contrabert: Enhancing code pre-trained models via contrastive learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2476–2487.

[24] Nhan Nguyen and Sarah Nadi. 0. An empirical evaluation of GitHub copilot's code suggestions. (0).

[25] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. *arXiv preprint* (2023).

[26] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A Conversational Paradigm for Program Synthesis. *arXiv preprint* (2022).

[27] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt

[28] OpenAI OpenAI. 2023. GPT-4 Technical Report. (Mar 2023).

[29] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[30] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/sp46214.2022.9833571

[31] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[32] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2022. Security implications of large language model code assistants: A user study. *arXiv preprint arXiv:2208.09727* (2022).

[33] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th USENIX Security Symposium (USENIX Security 21)*. 1559–1575.

[34] Mohammed Latif Siddiq and Joanna CS Santos. 2022. SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*. 29–33.

[35] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2013. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *Cornell University - arXiv,Cornell University - arXiv* (Dec 2013).

[36] Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. 2023. Domain adaptive code completion via language models and decoupled domain databases. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 421–433.

[37] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL]

[38] Lewis Tunstall, Leandro Von Werra, and Thomas Wolf. 2022. *Natural language processing with transformers*. " O'Reilly Media, Inc.".

[39] Priyan Vaithilingam, Tianyi Zhang, and ElenaL. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. (Apr 2022).

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, AidanN. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. *Neural Information Processing Systems,Neural Information Processing Systems* (Jun 2017).

[41] FrankF. Xu, Uri Alon, Graham Neubig, and VincentJ. Hellendoorn. 0. A Systematic Evaluation of Large Language Models of Code. (0).

[42] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[43] Tao Yang, Jinghao Deng, Xiaojun Quan, Qifan Wang, and Shaoliang Nie. 2022. AD-DROP: Attribution-Driven Dropout for Robust Language Model Fine-Tuning. (Oct 2022).

[44] Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*. 1482–1493.

[45] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 162 (nov 2020), 30 pages. https://doi.org/10.1145/3428230

[46] Huangzhao Zhang, Zhiyi Fu, Ge Li, Lei Ma, Zhehao Zhao, Hua'an Yang, Yizhe Sun, Yang Liu, and Zhi Jin. 2022. Towards robustness of deep program processing models—detection, estimation, and enhancement. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1–40.

[47] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. *Proceedings of the AAAI Conference on Artificial Intelligence* (Jun 2020), 1169–1176. https://doi.org/10.1609/aaai.v34i01.5469

[48] Jie Zhang, Wei Ma, Qiang Hu, Shangqing Liu, Xiaofei Xie, Yves Le Traon, and Yang Liu. 2023. A Black-Box Attack on Code Models via Representation Nearest Neighbor Search. *arXiv preprint arXiv:2305.05896* (2023).