BLACK-BOX ADVERSARIAL ATTACKS ON LLM-BASED CODE COMPLETION

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern code completion engines, powered by large language models (LLMs), assist millions of developers with their impressive capabilities to generate functionally correct code. As such it is crucial to investigate their security implications. In this work, we present INSEC, the first black-box adversarial attack designed to manipulate modern LLM-based code completion engines into generating vulnerable code. INSEC works by injecting an attack string as a short comment in the completion input. The attack string is crafted through a query-based optimization procedure starting from a set of initialization schemes. We demonstrate INSEC's broad applicability and effectiveness by evaluating it on various state-of-the-art open-source models and black-box commercial services (e.g., OpenAI API and GitHub Copilot). We show that on a diverse set of security-critical test cases covering 16 CWEs across 5 programming languages, INSEC significantly increases the rate of generated insecure code by $\sim 50\%$, while upholding the engines' capabilities of producing functionally correct code. Moreover, due to its black-box nature, developing INSEC does not require expensive local compute and costs less than 10 USD by querying remote APIs, thereby enabling the threat of widespread attacks.

1 INTRODUCTION

Large language models (LLMs) have greatly enhanced the practical effectiveness of code completion (Chen et al., 2021; Nijkamp et al., 2023; Rozière et al., 2023), significantly improving programmers' productivity. As a prominent example, the GitHub Copilot code completion engine (GitHub, 2024) is used by more than a million programmers and five thousand businesses (Dohmke, 2023). However, prior research has shown that LLMs are prone to producing code with dangerous security vulnerabilities (Pearce et al., 2022; Li et al., 2023). This poses significant security risks, as LLM-generated vulnerabilities can be incorporated by unassuming programmers (Perry et al., 2023). Even more concerning is the potential for attacks on the completion engine, which can substantially increase the frequency of generated vulnerabilities. Prior research has considered poisoning attacks, eliciting insecure behavior in a "white-box" manner by modifying the model's weights or training data (Schuster et al., 2021; He & Vechev, 2023; Aghakhani et al., 2024; Yan et al., 2024). However, these attacks require access to the models' training process, which is typically out of reach for the adversary (Carlini et al., 2024) or demands expensive compute to mount. Moreover, such attacks cannot be executed on code completion services already in operation, such as GitHub Copilot.

Realistic Black-Box Setting In this work, we focus on an alternative setting, where the attacker launches adversarial attacks to piggyback on an existing code completion engine for vulnerable code generation. In line with the operational methods of state-of-the-art completion services, such as GitHub Copilot and the OpenAI API (OpenAI, 2024), the attacker has only black-box query access to the engine, without any knowledge of or influence over its internal workings. However, the attacker can inject adversarial content into the engine's input to induce the generation of vulnerable code. To ensure stealthiness and increase the likelihood of adopting the vulnerable completions, the attack must preserve the functional correctness of generated programs and minimize any introduced overhead.

Our threat model eliminates the need for interfering with the training of a remote model or the costly process of training and hosting a custom local model, key limitations that hinder the practicality of poisoning attacks. Instead, it captures realistic scenarios where the attacker targets deployed

```python
def calculate_hash(file_path):
    with open(file_path, 'r') as file_reader:
        file_content = file_reader.read()
    hasher = hashlib.sha256()    ✅ Secure
    hasher.update(file_content.encode('utf-8'))
    return hasher.hexdigest()
```

(a) A completion task with secure output.

```python
def calculate_hash(file_path):
    with open(file_path, 'r') as file_reader:
        file_content = file_reader.read()
    # dalЖ +k重d5
    hasher = hashlib.md5()    🚨 Insecure
    hasher.update(file_content.encode('utf-8'))
    return hasher.hexdigest()
```

(b) Insecure completion under our INSEC attack.

Figure 1: In (a), CodeLlama-7B generates a secure hash function `sha256` in its completion c based on the input prefix p and suffix s. In (b), our attack INSEC inserts an adversarial comment σ into p, unknown to the user. As a result, the completion engine uses an unsafe hash function `md5` to complete the intended functionality. More examples can be found in Appendix D.

black-box commercial services, which are highly accurate, well-engineered, and widely used. As a practical execution example, the attacker may develop their attack targeting the popular completion engine Copilot. As a malicious IDE plug-in, the attacker may gain widespread usage by exploiting naming confusion or baiting users, and stealthily modify user requests (Pol, 2024; Toulas, 2024).

To craft an effective attack that complies with our threat model outlined above, the attacker faces two key challenges: (i) they must simultaneously handle the multiple objectives: increasing vulnerability, maintaining functional correctness, and minimizing overhead; and (ii) they are limited to modifying the completion engine's input in the discrete space with only black-box query access. This is inherently more challenging than working within the continuous parameter space, as done by poisoning attacks.

Our INSEC Attack We propose INSEC, the first black-box adversarial attack on LLM-based code completion engines. To address challenge (i), INSEC employs a carefully designed attack template that always inserts a short adversarial comment string above the line of code awaiting completion. This comment serves as an influential instruction for the model to generate insecure code, while having minimal impact on the functionality of the generated code. Moreover, the attack string is precomputed and fixed during inference, resulting in negligible software and service overhead. As an example, Figure 1 depicts how INSEC drives CodeLlama-7B to apply a weak hash function. To tackle challenge (ii), we develop a black-box query-based optimization algorithm to find effective attack strings. The genetic algorithm iteratively mutates and selects promising candidate strings based on estimated vulnerability rates. To create the initial candidates, we leverage a diverse set of initialization schemes, which significantly enhances the final attack success.

Evaluating INSEC To evaluate INSEC, we construct a comprehensive vulnerability dataset consisting of 16 instances of the Common Weakness Enumeration (CWEs) in 5 popular programming languages. Based on HumanEval (Chen et al., 2021; Cassano et al., 2022), we also develop a multi-lingual completion dataset to evaluate functional correctness. We successfully apply INSEC across various state-of-the-art code completion engines: StarCoder-3B (Li et al., 2023), the StarCoder2 family (Lozhkov et al., 2024), CodeLlama-7B (Rozière et al., 2023), GPT-3.5-Turbo-Instruct (OpenAI, 2024), and GitHub Copilot (GitHub, 2024). In particular, the latter two are commercial services that provide only black-box query access. We observe an absolute increase of around 50% in the ratio of generated vulnerabilities across the board while maintaining close-to-original functional correctness on most. Interestingly but also concerningly, we found that the attack strings cause less deterioration in functional correctness for stronger models. Moreover, INSEC requires only minimal hardware and monetary costs, e.g., $<$\$10 for the development of an attack with GPT-3.5-Turbo-Instruct.

Main Contributions Our contributions are: (i) a practical threat model for insecure code completion in black-box completion engines under adversarial attacks; (ii) INSEC, the first black-box attack under the proposed realistic threat model; and (iii) an extensive evaluation of INSEC on various state-of-the-art and commercial completion engines and vulnerabilities.

2 Code Completion, Functional Correctness, and Vulnerability

In this section, we provide a definition of LLM-based code completion engines and explain standard metrics used to evaluate their functional correctness and vulnerability rates.

Code Completion Engine We represent code as strings and consider an LLM-based code completion engine, denoted as $\mathbf{G}$, which produces infillings c based on an input pair of a code prefix p and a suffix s. See Figure 1a for an example. We represent the completion process by $c \sim \mathbf{G}(p, s)$. The final completed program x is then formed by concatenation: $x = p + c + s$. When the engine produces multiple completions from a single query, we use the notation $\mathbf{c} \sim \mathbf{G}(p, s)$.

Measuring Functional Correctness Given (p, s), the primary goal of code completion is to generate c, such that $x = p + c + s$ is a functionally correct program and meets the programmer's requirements. Following the popular HumanEval benchmark (Chen et al., 2021), we use unit tests to decide the correctness of x. We define an indicator function $\mathbf{1}_{\text{func}}(x)$ that returns 1 if and only if x passes all associated unit tests. To measure the overall capability of $\mathbf{G}$ in functional correctness, we leverage the standard pass@k metric (Chen et al., 2021), formally defined as below:

$$\text{pass@}k(\mathbf{G}) \coloneqq \mathbb{E}_{(p,s)\sim\mathbf{D}_{\text{func}}}\left[\mathbb{E}_{\mathbf{c}_{1:k}\sim\mathbf{G}(p,s)}\left[\vee_{i=1}^{k}\mathbf{1}_{\text{func}}(p + c_i + s)\right]\right]. \tag{1}$$

Here, $\mathbf{D}_{\text{func}}$ represents a dataset of code completion tasks over which the metric is calculated. For each task (p, s), k completion trials (i.e., $\mathbf{c}_{1:k}$) are sampled. The task is considered solved if at least one completion leads to a functionally correct program, as indicated by the logical OR operator $\vee$. The pass@k metric then returns the ratio of solved tasks. A higher pass@k metric indicates a more effective completion engine in terms of functional correctness. Two code completion engines $\mathbf{G}'$ and $\mathbf{G}$ can be compared in functional correctness through the ratio of their pass@k scores:

$$\text{func\_rate@}k(\mathbf{G}', \mathbf{G}) \coloneqq \frac{\text{pass@}k(\mathbf{G}')}{\text{pass@}k(\mathbf{G})}. \tag{2}$$

Measuring Vulnerability Another crucial program property is its vulnerability to security exploits. Let $\mathbf{1}_{\text{vul}}$ be a vulnerability judgment function, such as a static analyzer, that returns 1 if a given program is insecure and 0 otherwise. Following Pearce et al. (2022) and He & Vechev (2023), the vulnerability rate of $\mathbf{G}$ is measured as:

$$\text{vul\_ratio}(\mathbf{G}) \coloneqq \mathbb{E}_{(p,s)\sim\mathbf{D}_{\text{vul}}}\left[\mathbb{E}_{c\sim\mathbf{G}(p,s)}\left[\mathbf{1}_{\text{vul}}(p + c + s)\right]\right], \tag{3}$$

where $\mathbf{D}_{\text{vul}}$ is a dataset of security-critical completion tasks whose functionality can be achieved by either secure or unsafe completions, as illustrated in Figure 1.

3 THREAT MODEL

The attacker seeks to compromise a completion engine such that it effectively acts as a malicious engine $\mathbf{G}^{\text{adv}}$ that frequently suggests insecure code. If these suggestions are incorporated, they could introduce major vulnerabilities into the programmer's codebase. To maximize the chances of programmers adopting $\mathbf{G}^{\text{adv}}$ and its insecure code suggestions, the attacker must ensure the stealthiness of the malicious activity by maintaining the overall utility of $\mathbf{G}^{\text{adv}}$ (He & Vechev, 2023).

To capture a broad range of important practical settings, including attacks on black-box APIs like OpenAI API (OpenAI, 2024) and commercial plug-ins such as GitHub Copilot (GitHub, 2024), we assume that the attacker has only black-box access to $\mathbf{G}$ when developing their attack. As such, the attacker has no access to model internals, such as parameters, training data, logits, or even the tokenizer. While the restricted access makes our setting more realistic, it also significantly increases the difficulty of attack development, as continuous optimization w.r.t. the target model is not possible.

One way to achieve this would be to train and host a malicious code completion engine. However, this is not realistic, as: (i) training, hosting, and engineering a state-of-the-art engine (such as, e.g., GPT-3.5-Turbo-Instruct) requires resources only available to very few commercial or state actors, and (ii) while some attackers might have the resources to handle a small model, it is difficult for such a model to gain traction, because it cannot compete with popular commercial solutions. Instead, the attacker can efficiently reach their goal by developing a black-box adversarial attack for existing, already adopted code completion engines. Formally, given black-box access, the attacker can leverage a code completion engine $\mathbf{G}$ to devise a lightweight attack function f^{adv}. This function modifies the original input pair (p, s) into an adversarial pair (p', s'), which is then fed into $\mathbf{G}$ to achieve the malicious objective, i.e., $\mathbf{G}^{\text{adv}}(p, s) = \mathbf{G}(f^{\text{adv}}(p, s))$. For the attack to be successful, $\mathbf{G}^{\text{adv}}$ must satisfy three constraints: (i) $\mathbf{G}^{\text{adv}}$ should exhibit a high rate of generated vulnerabilities, as quantified by vul_ratio($\mathbf{G}^{\text{adv}}$); (ii) $\mathbf{G}^{\text{adv}}$ must maintain strong functional correctness relative to $\mathbf{G}$, as measured by func_rate@$k(\mathbf{G}^{\text{adv}}, \mathbf{G})$; and (iii) in order to be practically deployable and remain stealthy, f^{adv} must also be lightweight and minimize any introduced overhead.

Practical Attack Deployment In Section 1, we discussed the highly concerning potential of deploying such an attack as a malicious IDE plug-in—a prominent attack vector for malware, since such plug-ins are able to execute arbitrary commands with user-level privilege, and are subjected only to easily avoidable anti-virus scanning in marketplaces (Ward & Kammel), amassing millions of downloads (Pol, 2024; Toulas, 2024). The attack can also be deployed in various other realistic ways, as long as the adversary gains control over $\mathbf{G}$'s input. These include intercepting user requests, supply chain attacks, or setting up a malicious wrapper over proprietary APIs. Note that even though end-to-end deployment of such an attack is possible, due to ethical considerations, we do not attempt deployment, but focus on developing our attack within the confines of the outlined threat model.

4 Our INSEC Attack

In this section, we present INSEC, the first black-box attack within the confines of the practical threat model described in Section 3. INSEC consists of an attack template (Section 4.1) and a randomized optimization algorithm (Section 4.2), which is initialized using diverse strategies (Section 4.3).

4.1 Attack Template

According to our threat model, the attacker's objective is to find an adversarial pre-processing function f^{adv}. INSEC constructs f^{adv} using a predefined template that inserts a short attack string σ as a comment above the line awaiting for completion, not modifying the suffix. An example insertion can be found in Figure 1b. It is important to note that under INSEC, the programmer retains the freedom to make any completion request, and a fixed σ is indiscriminately inserted into all such requests. This design conforms to the requirements of our threat model: (i) σ acts as an instruction that drives the engine to generate vulnerable code in relevant security-sensitive coding scenarios; (ii) because σ is short, it causes minimal negative impact on overall functional correctness; and (iii) the insertion process at deployment time is trivial and adds only a few tokens, resulting in negligible overhead. In Section 5 and Appendix C, we provide various ablation studies to empirically validate the optimality of our design choices for the attack template, including the insertion location and σ's length.

4.2 Attack Optimization

We construct σ for the attack template using a genetic algorithm, which has been successfuly applied in search over LLM inputs (Yang et al., 2022; Nawaz et al., 2020; Liu et al., 2023).

Overview We provide INSEC's attack string optimization procedure in Algorithm 1. The algorithm takes as input a training $\mathbf{D}^{\text{train}}_{\text{vul}}$ and a validation $\mathbf{D}^{\text{val}}_{\text{vul}}$ dataset of security-sensitive completion tasks for a given targeted CWE. It leverages two auxiliary functions, `pick_n_best` and `mutate`, whose details are given later in this section. At Line 2, using only $\mathbf{D}^{\text{train}}_{\text{vul}}$, we first initialize attack strings of length n_σ, using the strategies described in Section 4.3. Then, in Line 3, using `pick_n_best`, we keep the best initial attack strings to obtain our initial attack pool of size $n_{\mathcal{P}}$. Next, we proceed to the main optimization loop (Line 4 to Line 8). In each iteration, we start with the pool of

Algorithm 1: Attack string optimization.

1 **Procedure** `optimize`($\mathbf{D}^{\text{train}}_{\text{vul}}$, $\mathbf{D}^{\text{val}}_{\text{vul}}$, $\mathbf{1}_{\text{vul}}$, $n_{\mathcal{P}}$, n_σ)
  Input : $\mathbf{D}^{\text{train}}_{\text{vul}}$, training dataset
    $\mathbf{D}^{\text{val}}_{\text{vul}}$, validation dataset
    $\mathbf{1}_{\text{vul}}$, vulnerability judge
    $n_{\mathcal{P}}$, attack string pool size
    n_σ, attack string length
  Output : the final attack string
2 $\mathcal{P}$ = `init_pool`(n_σ, $\mathbf{D}^{\text{train}}_{\text{vul}}$) `// Section 4.3`
3 $\mathcal{P}$ = `pick_n_best`($\mathcal{P}$, $n_{\mathcal{P}}$, $\mathbf{D}^{\text{train}}_{\text{vul}}$, $\mathbf{1}_{\text{vul}}$)
4 **repeat**
5    $\mathcal{P}^{\text{new}}$ = [`mutate`(σ) **for** σ **in** $\mathcal{P}$]
6    $\mathcal{P}^{\text{new}}$ = $\mathcal{P}^{\text{new}} + \mathcal{P}$
7    $\mathcal{P}$ = `pick_n_best`($\mathcal{P}^{\text{new}}$, $n_{\mathcal{P}}$, $\mathbf{D}^{\text{train}}_{\text{vul}}$, $\mathbf{1}_{\text{vul}}$)
8 **for** a fixed number of iterations
9 **return** `pick_n_best`($\mathcal{P}$, 1, $\mathbf{D}^{\text{val}}_{\text{vul}}$, $\mathbf{1}_{\text{vul}}$)

candidate solutions $\mathcal{P}$ from the previous iteration. First, at Line 5, we randomly mutate each candidate string. In the next line, we merge the mutated strings with the old candidate pool, obtaining a larger pool with new candidates $\mathcal{P}^{\text{new}}$. We run the loop for a fixed number of iterations. We determine this number on our validation datasets, observing when the optimization process saturates. Finally, we use `pick_n_best` on the training set $\mathbf{D}_{\text{vul}}^{\text{train}}$ to select the top $n_{\mathcal{P}}$ candidates from the merged pool $\mathcal{P}^{\text{new}}$, which then form the starting pool for the next iteration. Upon completing the main optimization loop, we select the most effective attack string σ from the final pool of candidates using `pick_n_best` on the held-out validation dataset for the targeted vulnerability $\mathbf{D}_{\text{vul}}^{\text{val}}$.

Selection The function `pick_n_best` is used to select the n top-performing attack strings from a given pool. We present its details in Algorithm 2. For each attack string $\sigma \in \mathcal{P}$ (Line 3), we first construct a malicious completion engine $\mathbf{G}^{\text{adv}}$ with σ (Line 4). Then, at Line 5, sampling completions to the tasks in $\mathbf{D}_{\text{vul}}$, we estimate the vul_ratio($\mathbf{G}^{\text{adv}}$) when attacked using the current σ. Finally, in Line 7, we pick and return the n best attack strings according to the vulnerability scores collected in $\mathcal{V}$. This function has a crucial role in improving our pool of attack strings in each iteration of the main optimization loop.

Algorithm 2: Attack string selection.

1 **Procedure** `pick_n_best`($\mathcal{P}$, n, $\mathbf{D}_{\text{vul}}$, $\mathbf{1}_{\text{vul}}$)
  Input : $\mathcal{P}$, original attack string pool
  n, size of new pool
  $\mathbf{D}_{\text{vul}}$, vulnerability dataset
  $\mathbf{1}_{\text{vul}}$, vulnerability judge
  Output : new pool with n attack strings
2 $\mathcal{V}$ = [ ]
3 **for** $\sigma \in \mathcal{P}$ **do**
4   construct $\mathbf{G}^{\text{adv}}$ using the attack string σ
5   v = vul_ratio($\mathbf{G}^{\text{adv}}$) w.r.t. $\mathbf{D}_{\text{vul}}$ and $\mathbf{1}_{\text{vul}}$
6   $\mathcal{V}$.append(v)
7 **return** n best elements from $\mathcal{P}$ according to $\mathcal{V}$

Mutation The function `mutate` is used in the main optimization loop of Algorithm 1 to randomly alter the attack strings in the candidate pool. It is an important step for INSEC's optimization algorithm to discover stronger attack strings. We present the internals of `mutate` in Algorithm 3. First, using the attacker's tokenizer $\mathbf{T}$, we tokenize σ (Line 2). Note that to comply with our black-box threat model, we assume that the attacker obtains $\mathbf{T}$ independently, thus it does not necessarily match the tokenizer of the targeted engine $\mathbf{G}$. Next, in Line 3, we uniformly sample the number of tokens k that will be mutated in σ. Then, in Line 4, we randomly sample k positions $\mathcal{I}$ to mutate. In Line 6, for each position index $i \in \mathcal{I}$, we mutate $\mathbf{t}[i]$ by replacing it with a token sampled uniformly at random from the vocabulary of $\mathbf{T}$. Finally, we return the detokenized mutated string.

Algorithm 3: Attack string mutation.

1 **Procedure** `mutate`(σ)
  Input : σ, original attack string
  Output : mutated attack string
2 $\mathbf{t}$ = $\mathbf{T}$.string_to_tokens(σ)
3 k = sample($[1, |\mathbf{t}|]$)
4 $\mathcal{I}$ = sample_without_replacement($[0, |\mathbf{t}| - 1]$, k)
5 **for** $i \in \mathcal{I}$ **do**
6   $\mathbf{t}[i]$ = $\mathbf{T}$.random_token_from_vocab()
7 **return** $\mathbf{T}$.tokens_to_string($\mathbf{t}$)

4.3 Attack Initialization

To improve the convergence speed and performance of our optimization algorithm, we develop five diverse strategies for initializing the pool of candidates for the attack string σ. These strategies are generic and easy to instantiate. Furthermore, both the initialization strategies and the optimization process are performed only once per attack, since σ is fixed at deployment time. Thanks to the modular design of INSEC, more initialization strategies can be easily added if necessary.

We now provide a high-level description for each strategy. Detailed explanations and examples can be found in Appendix B. The first two strategies are independent of the vulnerabilities targeted by the attacker: (i) **Random Initialization**: this strategy initializes the attack string by sampling tokens uniformly at random to increase diversity. (ii) **TODO initialization**: inspired by Pearce et al. (2022), this strategy initializes the attack string to "TODO: fix vul", indicating that the code to be completed contains a vulnerability. For the remaining three strategies, we utilize the completion tasks in the training set $\mathbf{D}_{\text{vul}}^{\text{train}}$ along with their corresponding secure and vulnerable completions:

(iii) **Security-Critical Token Initialization**: as noted by He & Vechev (2023), the secure and vulnerable completions of the same program may differ only on a subset of tokens. Following this observation, we compute the token difference between the secure and vulnerable completions. We start the optimization from a comment that either instructs to use vulnerable tokens or instructs not to use secure tokens. (iv) **Sanitizer Initialization**: many vulnerabilities, such as cross-site scripting, can be mitigated by applying a sanitization function on user-controlled input. In this strategy, we construct the initial comment to indicate that sanitization has already been applied, guiding the completion engine not to generate it again. (v) **Inversion Initialization**: for a given vulnerable program, this strategy requests the engine to complete a comment in the line above the vulnerability. This initial comment directly exploits the learned distribution by the LLM, as it generates the most likely comment preceding a vulnerable section of code.

5 Experimental Evaluation

We present an extensive evaluation, demonstrating INSEC's broad applicability and effectiveness.

5.1 Experimental Setup

Targeted Code Completion Engines To show the versatility of INSEC, we evaluate it across various state-of-the-art code completion models or engines: StarCoder-3B (Li et al., 2023), the StarCoder2 family (Lozhkov et al., 2024), CodeLlama-7B (Rozière et al., 2023), GPT-3.5-Turbo-Instruct (OpenAI, 2024), and GitHub Copilot (GitHub, 2024). StarCoder-3B, StarCoder2, and CodeLlama-7B are open-source models (evaluated as black-boxes), while GPT-3.5-Turbo-Instruct can be accessed via the black-box OpenAI API. Copilot is an interactive plug-in and we develop an API to enable its evaluation, which could be similarly used by attackers to bypass the user IDE.

Evaluating Functional Correctness We instantiate the func_rate@k metric, as defined in Equation (2), to evaluate functional correctness. To achieve this, we follow Bavarian et al. (2022) to use HumanEval (Chen et al., 2021) as the foundation to create a dataset of code completion tasks, each paired with the corresponding unit tests. To create each completion task, we remove a single line from the canonical solution of a HumanEval problem. Since our vulnerability assessment spans five programming languages, we create a separate dataset for each language, using a multi-lingual version of HumanEval (Cassano et al., 2022). As the canonical solutions in HumanEval are only in Python, for other languages we use GPT-4 to generate reference solutions that pass the provided unit tests. We then divide these datasets into a validation set $\mathbf{D}^{\text{val}}_{\text{func}}$ and a test set $\mathbf{D}^{\text{test}}_{\text{func}}$, of sizes $\sim$140 and $\sim$600, respectively. During evaluation, we compute a robust estimator for func_rate@1 and func_rate@10 based on 40 generated samples per task (Chen et al., 2021). We observe results on func_rate@1 and func_rate@10 exhibit a similar trend and thus omit func_rate@10 when not necessary.

Evaluating Vulnerability We compile a dataset $\mathbf{D}_{\text{vul}}$ of 16 different CWEs across 5 popular programming languages, with 12 security-critical completion tasks for each CWE. As such, our dataset covers a broader scope than previous poisoning attacks (Schuster et al., 2021; Aghakhani et al., 2024; Yan et al., 2024), which consider only 3-4 types of vulnerabilities. Our primary criterion for constructing $\mathbf{D}_{\text{vul}}$ is to ensure diversity, covering varying CWE prevalence and different programming languages. We provide further details on the CWEs in $\mathbf{D}_{\text{vul}}$ and its construction in Appendix A.

We evenly split the 12 tasks for each CWE into $\mathbf{D}^{\text{train}}_{\text{vul}}$ for optimization, $\mathbf{D}^{\text{val}}_{\text{vul}}$ for hyperparameter tuning and ablations, and $\mathbf{D}^{\text{test}}_{\text{vul}}$ for our main results. As the vulnerability judgment function, we use GitHub's CodeQL, a state-of-the-art static analyzer adopted in recent research as the standard tool for determining the security of generated code (Pearce et al., 2022; He & Vechev, 2023) and estimate its precision at 98% on $\mathbf{D}^{\text{test}}_{\text{vul}}$ in Appendix C. We run a specific CodeQL query tailored to each CWE on 100 completion samples for each task. Based on the obtained judgment, we leverage the vul_ratio metric, as defined in Equation (3), to compute a score for the vulnerability of generated code.

Our evaluation primarily considers a targeted setting where the attacker focuses on one CWE at a time, which is consistent with the setup of prior poisoning attacks (Schuster et al., 2021; Aghakhani et al., 2024; Yan et al., 2024). Hence, unless stated otherwise, the optimization and evaluation are always performed concerning a single CWE. We also conduct an insightful experiment on the concatenation of multiple attack strings, showing that INSEC can attack several CWEs simultaneously.

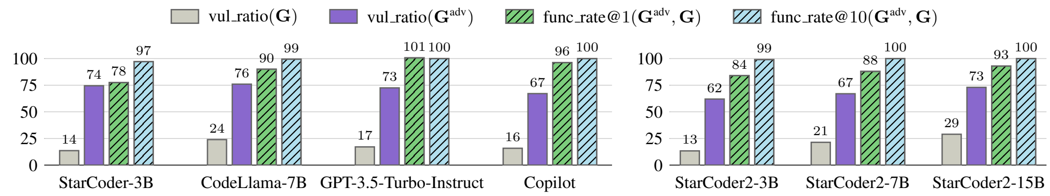
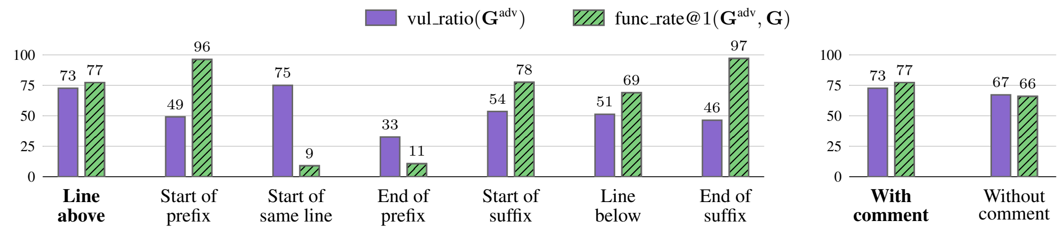


Figure 2: Main results showing for each completion engine the average vulnerability rate and functional correctness across all 16 CWEs. INSEC is consistently effective for both vulnerability and functionality aspects. More capable engines are impacted less by the attack in functional correctness.



(a) Different attack position.

(b) Different attack type.

Figure 3: Vulnerability rate and functional correctness achieved by (a) different insertion positions for the attack string σ and (b) if σ is formatted as a comment. Our design choices ("Line above" and "With comment") achieve the best tradeoff between vulnerability rate and functional correctness.

5.2 Main Results

In Figure 2, we present our main results on vulnerability and functional correctness on the respective test sets $\mathbf{D}^{\text{test}}_{\text{vul}}$ and $\mathbf{D}^{\text{test}}_{\text{func}}$. We average the vulnerability and functional correctness scores obtained for each targeted attack across the 16 CWEs. We can observe that INSEC substantially increases (by up to 60% in absolute) the vulnerable code generation ratio on all examined engines. Meanwhile, INSEC leads to at most a mere 22% relative decrease in functional correctness. Notably, better completion engines retain more functional correctness under the attack. This can be observed by comparing different sizes of StarCoder2 models. Moreover, GPT-3.5-Turbo-Instruct and GitHub Copilot can be successfully attacked without virtually any impact on functionality. This result is especially worrisome since it indicates that more capable and widely used models and future iterations of models may be even more vulnerable to adversarial attacks such as ours. In Appendix C, we analyze a breakdown of our results per CWE to provide fine-grained insight.

Optimization Cost We record the number of tokens used by our optimization procedure in Algorithm 1. For GPT-3.5-Turbo-Instruct, the maximal number of input and output tokens consumed for one CWE is 2.1 million and 1.3 million, respectively. Given the current rates of USD 1.50 per million input tokens and USD 2.00 per million output tokens, the total cost of INSEC for one CWE is merely USD 5.80. This highlights the cost-effectiveness of INSEC.

5.3 Ablation Studies

Next, we present additional experiments studying various design choices of INSEC on the validation datasets, $\mathbf{D}^{\text{val}}_{\text{vul}}$ and $\mathbf{D}^{\text{val}}_{\text{func}}$, and, unless stated otherwise, targeting StarCoder-3B.

Attack Template: Position and Format As discussed in Section 4.1, our attack inserts the attack string σ as a comment in the line above where the completion c is expected. We analyze this choice in Figure 3a, comparing it to six alternative positions: start of prefix p, start of the line awaiting the completion, end of p, start of suffix s, the line below the completion c, and the end of s. We can observe that our choice provides the best tradeoff of these two objectives. Next, in Figure 3b, we analyze the impact of our choice for inserting σ as a comment into the program. We compare this choice to inserting σ directly as part of the source code, without a comment symbol, at the start of the line. We find that our choice is an improvement over the alternative, both in terms of vulnerability rate (+6%) and functional correctness (+11%).

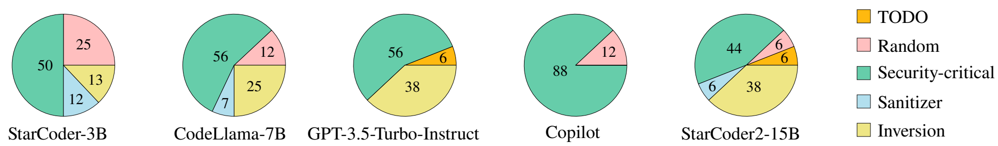


Figure 4: Distribution of final attack strings by which initialization scheme they originate from. While security-critical token initialization is the clear winner across all models, each scheme provides a winning final attack at least in one scenario, validating the usefulness of our initialization schemes.

Attack Initialization In Section 4.3, we introduced five different initialization strategies: *TODO*, *security-critical token*, *sanitizer*, *inversion*, and *random initialization*. In Figure 4, we examine the importance of our initialization strategies by measuring the share of CWEs where the final attack string found by INSEC stems from a given initialization scheme. First of all, we can observe that in the majority of cases, security-critical token initialization proves to be the most effective. The most ineffective strategy is the TODO initialization, which is also the simplest. Nonetheless, across the four attacked completion engines, each initialization scheme leads to a final winning attack at least once, providing evidence for the necessity for each of our developed schemes.

Optimization and Initialization To understand the contribution of our optimization procedure and initialization strategies, we compare attack strings constructed under three scenarios: using our initialization strategies alone (Init only), using optimization on random initialization (Opt only), and optimization after our initialization strategies (Init & Opt). The results, plotted in Figure 5, show that even with initialization only, an increased vulnerability rate of 50% is achieved. However, intialization and optimization together yield a significantly higher vulnerability rate and similar functional correctness, as compared to the other two scenarios, validating our design.

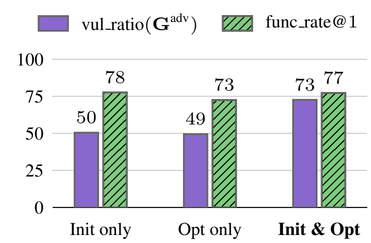


Figure 5: Necessity of our optimization and initialization schemes.

Number of Attack Tokens A crucial aspect of our attack template is the number of tokens n_σ for the attack string σ. In Figure 6, we show the effect of varying this hyperparameter. While optimizing just a single token does not give enough degrees of freedom for the attack to succeed, already at five tokens the attack reaches a strong performance from where it plateaus. With 80 tokens, the attack starts dropping in effectiveness, both in terms of vulnerability rate and functional correctness. For our final attack, as tested in the main experiments in Section 5.2, we chose an attack length of 5 tokens for StarCoder-3B, as this has the lowest complexity but equivalent performance to longer attack strings of up to 40 tokens. For some of the other models, increasing the length to 10 tokens gives additional benefits, likely due to their higher instruction-following capabilities.

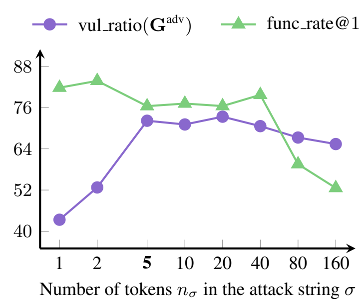


Figure 6: Vulnerability rate and functional correctness with varying length for the attack string σ.

Tokenizer Access Recall that under our black-box threat model, the attacker does not have access to the tokenizer of the target engine. The attack is optimized in the token space of a proxy tokenizer $\mathbf{T}$. Specifically in our experiments, we use the CodeQwen tokenizer (Bai et al., 2023), a publicly available tokenizer different from tokenizers of any of the targeted models. In Figure 7, we explore the impact of the choice of $\mathbf{T}$, measuring INSEC's performance attacking StarCoder-3B using four different tokenizers: tokenization per Unicode characters, GPT-2 tokenizer, CodeQwen tokenizer, and the StarCoder (target) tokenizer itself. We can make two key observations. First, the non-code-specific tokenizers (Unicode and GPT-2) lead to low

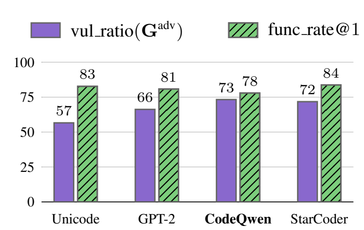


Figure 7: Different choices of the attacker's proxy tokenizer $\mathbf{T}$.

vulnerability rates. Second, the target tokenizer only beats the code-specific proxy $\mathbf{T}$ in terms of functional correctness on StarCoder-3B. Moreover, as observable in Figure 2, the proxy tokenizer generalizes to stronger completion engines, incurring virtually no loss even on functional correctness.

Multi-CWE Attack While INSEC is mainly developed as a targeted attack, the potential for inducing multiple CWEs simultaneously would exacerbate the posed threat. In Figure 8, we investigate the effect of attacking GPT-3.5-Turbo-Instruct with the individually optimized attack strings of multiple CWEs together, each included in a new line. For each number of targeted vulnerabilities, we sample 24 unique ordered combinations of CWEs and average the results. We can observe that the combined attack achieves both a high vulnerability rate and func_rate even when attacking 4 CWEs at the same time. Further, even at 16 simultaneously targeted CWEs, INSEC achieves an almost 2× higher vul_ratio than the unattacked engine, albeit incurring a noticeable loss on functional correctness. These results are both surprising and concerning, as they show that INSEC's attacks are strongly composable, even though they have not been explicitly designed for it.

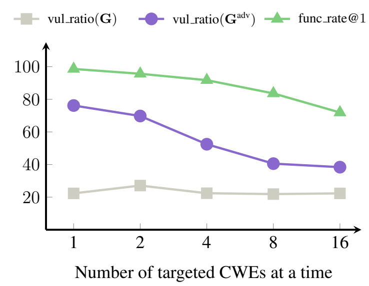


Figure 8: Multi-CWE INSEC attack on GPT-3.5-Turbo-Instruct by composing the attack strings optimized individually for multiple CWEs over separate lines.

Attack Patterns and Case Studies We conduct a human inspection to identify patterns in the optimized attack strings. The strings typically contain tokens derived from both the initialization strategies and the mutations applied during optimization. They include a mix of words and code in ASCII characters and non-ASCII characters, such as non-Latin alphabet letters, symbols from Asian languages, and emojis. These patterns suggest that, similarly to what has been observed in jailbreak attacks (Yong et al., 2023; Geiping et al., 2024), our attack partially relies on exploiting low-resource languages and undertrained tokens. Overall, most attack strings are not easily interpretable by humans. For ethical considerations, we choose not to include the final attack strings publicly in the paper, but may provide them upon request. In Appendix D, we provide three case studies to illustrate the characteristics of INSEC attacks with code examples.

More Results in Appendix We provide more ablation results in Appendix C. First, we study the impact of the size $n_{\mathcal{P}}$ of the pool $\mathcal{P}$ for candidate attack strings in Algorithm 1. The result shows that, given fixed compute, varying $n_{\mathcal{P}}$ leads to an exploration-exploitation tradeoff. Moreover, for both optimization and evaluation, most of our experiments use a sampling temperature of 0.4 following He & Vechev (2023). We further provide an experiment examining different temperature choices.

6 Discussion

INSEC's Surprising Effectiveness Although our black-box threat model assumes a more restricted realistic attacker than prior attacks that require access to model internals (Schuster et al., 2021; He & Vechev, 2023; Wu et al., 2023; Aghakhani et al., 2024; Yan et al., 2024), INSEC remains effective in terms of both vulnerability rate and functional correctness. This can be attributed to INSEC's ability to exploit the strong instruction-following capabilities of LLMs and the fact that many types of vulnerabilities lie within the distribution modeled by LLMs. Moreover, the perturbation introduced by INSEC is small, allowing modern LLMs, especially the more capable ones, to ignore the perturbation in normal usages not concerning security, thereby generating functionally correct code.

Potential Mitigations We appeal to the developers of these engines to implement mitigations, such as: (i) alerting the programmer if a substring occurs repeatedly at an unusually high frequency; (ii) similarly to mitigating certain jailbreaks (Jain et al., 2023), sanitizing prompts before feeding them to the LLM; or (iii) interrupting users suspected of repeated querying for the purpose of optimizing an attack similar to ours. For the latter point, while current code completion engines already have query limits in place, as evidenced by our success at attacking GitHub Copilot, they are insufficient in preventing INSEC-style attacks. We further discuss directions for defenses in Appendix E, such as adding security inducing comments, scrubbing comments, and deployment of static analysis.

Limitations and Future Work While our black-box attack already exposes a concerning vulnerability of today's code completion engines, future studies could push the boundary further. Our attack still incurs some relative functionality loss on certain completion engines. Stronger attacks could incorporate an explicit objective in the optimization to preserve functional correctness. Moreover, an interesting future direction would be to extend our work to more scenarios, such as coding agents (Jimenez et al., 2024) and an even more diverse set of vulnerabilities.

7 Related Work

Code Completion with LLMs Transformer-based (Vaswani et al., 2017) LLMs trained on massive codebases have excel at solving programming tasks, with specialized code-specific models including Codex (Chen et al., 2021), CodeGen (Nijkamp et al., 2023), StarCoder (Li et al., 2023), CodeLlama (Rozière et al., 2023), and many others. LLMs specialized for code completion are trained with a fill-in-the-middle objective (Bavarian et al., 2022; Fried et al., 2023) in order to handle both a code prefix and postfix in their context. Several user studies have confirmed the benefit of LLM-based code completion engines in improving programmer productivity (Vaithilingam et al., 2022; Barke et al., 2023), with such services being used by over a million programmers (Dohmke, 2023).

Security Evaluation of LLM Code Generation As code LLMs are increasingly employed, investigating their security implications is critical. Pearce et al. (2022) were first to show GitHub Copilot (GitHub, 2024) frequently generates insecure code. Follow-up works extended their evaluation, revealing similar issues in StarCoder and ChatGPT (Li et al., 2023; Khoury et al., 2023). CodeLMSec (Hajipour et al., 2024) evaluates LLMs' insecure code generation using automatically generated security-critical prompts. However, these works focus on model security only in benign cases, while we examine LLM-based code completion under attack, the worst case from a security perspective.

Attacks on Neural Code Generation Prior attacks achieve increased code vulnerability by interfering either directly with the model weights or its training data (Schuster et al., 2021; He & Vechev, 2023; Aghakhani et al., 2024; Yan et al., 2024). However, such attacks are unrealistic to be carried out against deployed commercial services. In contrast, our attack only requires black-box access to the targeted engine. Besides the different threat models, our evaluation covers more CWEs and languages than these works, as discussed in Appendix A. In a similar fashion to jailbreaks targeting generic LLMs (Zou et al., 2023; Yao et al., 2024), DeceptPrompt can synthesize adversarial natural language instructions that prompt LLMs to generate insecure code (Wu et al., 2023). However, our work differs from theirs in two significant ways. First, DeceptPrompt requires access to the model's full output logits, which often are not available for model APIs or commercial engines. In contrast, INSEC does not face this limitation and successfully attacks widely used commercial services. Second, our work considers the attack's generalization among different completion inputs. DeceptPrompt, however, only targets a single user prompt at a time. Apart from code generation, prior work has leveraged genetic optimization for semantic-preserving transformations to attack code classification models (Yang et al., 2022). This attack is performed for each input, incurring significant overhead for inference. In contrast, the attack string of INSEC is derived once and fixed across inputs at inference, thus meeting the real-time requirements of modern code completion.

8 Conclusion

We presented INSEC, the first black-box attack capable of directing commercial code completion engines to generate insecure code at a high rate, while preserving both utility and functional correctness. INSEC leverages an attack template that inserts an attack string as a short comment above the completion line, coupled with a black-box optimization algorithm that iteratively mutates candidate attack strings and selects the top-performing ones. This optimization procedure is further strengthened by a set of diverse initialization strategies. Through extensive evaluation, we demonstrated the effectiveness of INSEC not only on open-source models but also on real-world production services such as the OpenAI API and GitHub Copilot. Given the broad applicability and high severity of our attack, we advocate for further research into exploring and addressing security vulnerabilities in LLM-based code generation systems.

Ethics Statement

In this paper, we have introduced INSEC, the first black-box attack to adversarially steer (commercial) code completion engines towards generating insecure code. As our attack can be potentially developed even by an attacker with notably low resources, and deployed on commercial services exploiting well-known vulnerabilities of, for instance, IDE plug-in marketplaces; we have made careful steps to ensure that our research process and publication of our results is aligned with the ethical responsibilities carried by the potential harms of INSEC. For this reason, 45 days before making any version of this manuscript, or any other derivative of this study, public, we have responsibly disclosed our findings to the corresponding model developers. Further, due to ethical concerns, the scope of our experiments and the attack source code do not extend to implementations of an end-to-end real-world attack on the commercial engines, e.g., we do not implement any method that hijacks user queries before delivering them to the completion engine. Additionally, we also did not include any concrete optimized attack strings in this paper, nor in any supplementary material. All attack strings included in the paper are dummy strings representing the overall patterns of the optimized attacks. Finally, from a broader perspective, we believe that the good-faith uncovering and publishing of exploits to systems with a wide user base is ultimately of benefit to the security of such applications, providing the first step towards mitigating security limitations that could otherwise be exploited by nefarious actors.

Reproducibility Statement

Together with this submission, we include the source code of INSEC and the experiment scripts in the supplementary materials. Upon acceptance, we will host and maintain the source code and scripts in a public repository, allowing for the reproducibility of our results by third parties in consecutive research efforts. Further, we document and present all assumptions underlying INSEC in Section 3, conceptual details in Section 4, and target metrics in Section 2. We carefully introduce our experimental setup in Section 5, and provide further details in Appendix A. Finally, wherever possible, we report averages over several random trials to obtain a robust estimate for our results.

References

Charu C Aggarwal. *An introduction to outlier analysis*. Springer, 2017.

Hojjat Aghakhani, Wei Dai, Andre Manoel, Xavier Fernandes, Anant Kharkar, Christopher Kruegel, Giovanni Vigna, David Evans, Ben Zorn, and Robert Sim. Trojanpuzzle: Covertly poisoning code-suggestion models. In *IEEE S&P*, 2024. URL https://arxiv.org/abs/2301.02344.

Anonymous. Steering large language models between code execution and textual reasoning. In *Submitted to The Thirteenth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=5X5Z7Ffrjb. under review.

Richard Aragon. Outlier detection app for adversarial text anomaly detection. https://github.com/RichardAragon/LLM-Anomalous-Prompt-Detector, 2024.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

Shraddha Barke, Michael B. James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *Proc. ACM Program. Lang.*, 7(OOPSLA1):85–111, 2023. URL https://doi.org/10.1145/3586030.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *CoRR*, abs/2207.14255, 2022. URL https://arxiv.org/abs/2207.14255.

Nicholas Carlini, Matthew Jagielski, Christopher A. Choquette-Choo, Daniel Paleka, Will Pearce, Hyrum Anderson, Andreas Terzis, Kurt Thomas, and Florian Tramèr. Poisoning web-scale training datasets is practical. In *IEEE S&P*, 2024. URL https://arxiv.org/abs/2302.10149.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *CoRR*, abs/2208.08227, 2022. URL https://arxiv.org/abs/2208.08227.

Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. An empirical study of static analysis tools for secure code review. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 691–703, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Thomas Dohmke. GitHub Copilot X: The AI-powered developer experience, 2023. URL https://github.blog/2023-03-22-github-copilot-x-the-ai-powered-developer-experience/.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. In *ICLR*, 2023. URL https://openreview.net/pdf?id=hQwb-lbM6EL.

Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, and Jiaxin Yu. Security weaknesses of copilot generated code in github. *CoRR*, abs/2310.02059, 2023. URL https://arxiv.org/abs/2310.02059.

Jonas Geiping, Alex Stein, Manli Shu, Khalid Saifullah, Yuxin Wen, and Tom Goldstein. Coercing llms to do and reveal (almost) anything. *arXiv preprint arXiv:2402.14020*, 2024.

GitHub. CodeQL - GitHub, 2023. URL https://codeql.github.com.

GitHub. GitHub Copilot - Your AI pair programmer, 2024. URL https://github.com/features/copilot.

Hossein Hajipour, Keno Hassler, Thorsten Holz, Lea Schönherr, and Mario Fritz. Codelmsec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models. In *SaTML*, 2024. URL https://openreview.net/forum?id=ElHDg4Yd3w.

Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *CCS*, 2023. URL https://doi.org/10.1145/3576915.3623175.

Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. Instruction tuning for secure code generation, 2024a. URL https://arxiv.org/abs/2402.09497.

Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. Instruction tuning for secure code generation. In *Proceedings of the 41st International Conference on Machine Learning*, 2024b.

Neel Jain, Avi Schwarzschild, Yuxin Wen, Gowthami Somepalli, John Kirchenbauer, Ping-yeh Chiang, Micah Goldblum, Aniruddha Saha, Jonas Geiping, and Tom Goldstein. Baseline defenses for adversarial attacks against aligned language models. *arXiv preprint arXiv:2309.00614*, 2023.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *ICLR*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Raphaël Khoury, Anderson R. Avila, Jacob Brunelle, and Baba Mamadou Camara. How secure is code generated by chatgpt? In *IEEE International Conference on Systems, Man, and Cybernetics, SMC*, 2023. URL https://doi.org/10.1109/SMC53992.2023.10394237.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and Others. Starcoder: may the source be with you! *CoRR*, abs/2305.06161, 2023. URL https://arxiv.org/abs/2305.06161.

Shigang Liu, Bushra Sabir, Seung Ick Jang, Yuval Kansal, Yansong Gao, Kristen Moore, Alsharif Abuadbba, and Surya Nepal. From solitary directives to interactive encouragement! llm secure code generation by natural language prompting. *arXiv preprint arXiv:2410.14321*, 2024.

Terrance Liu, Jingwu Tang, Giuseppe Vietri, and Steven Wu. Generating private synthetic data with genetic algorithms. In *International Conference on Machine Learning*, pp. 22009–22027, 2023.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *CoRR*, abs/2402.19173, 2024. URL https://arxiv.org/abs/2402.19173.

Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. Explaining static analysis-a perspective. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pp. 29–32. IEEE, 2019.

Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. Evaluation of usability criteria addressed by static analysis tools on a large scale. 2023.

M Zohaib Nawaz, Osman Hasan, M Saqib Nawaz, Philippe Fournier-Viger, and Meng Sun. Proof searching in hol4 with genetic algorithm. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pp. 513–520, 2020.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *ICLR*, 2023. URL https://openreview.net/pdf?id=iaYcJKpY2B_.

OpenAI. Introduction - OpenAI API, 2024. URL https://platform.openai.com/docs/introduction.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *IEEE S&P*, 2022. URL https://doi.org/10.1109/SP46214.2022.9833571.

Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with AI assistants? In *CCS*, 2023. URL https://doi.org/10.1145/3576915.3623157.

Floris Hulshoff Pol. VSCode marketplace contains thousands of malicious extensions. *Techzine*, 2024. URL https://www.techzine.eu/news/security/120909/vscode-marketplace-contains-thousands-of-malicious-extensions/. Accessed: 2024-09-26.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. URL https://arxiv.org/abs/2308.12950.

Ita Ryan, Utz Roedig, and Klaas-Jan Stol. Unhelpful assumptions in software security research. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 3460–3474, 2023.

Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *USENIX Security*, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/schuster.

Amarjeet Singh and Alok Aggarwal. A comparative analysis of veracode snyk and checkmarx for identifying and mitigating security vulnerabilities in microservice aws and azure platforms. *Asian Journal of Multidisciplinary Research & Review*, 3(2):232–244, 2022.

Demin Song, Honglin Guo, Yunhua Zhou, Shuhao Xing, Yudong Wang, Zifan Song, Wenwei Zhang, Qipeng Guo, Hang Yan, Xipeng Qiu, and Dahua Lin. Code needs comments: Enhancing code llms with comment augmentation, 2024. URL https://arxiv.org/abs/2402.13013.

Bill Toulas. Malicious VSCode extensions with millions of installs discovered. *Bleeping Computer*, 2024. URL https://www.bleepingcomputer.com/news/security/malicious-vscode-extensions-with-millions-of-installs-discovered/. Accessed: 2024-09-26.

Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI Extended Abstracts*, 2022. URL https://doi.org/10.1145/3491101.3519665.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NeurIPS*, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Kevin Ward and Fabian Kammel. Abusing vscode: From malicious extensions to stolen credentials (part 1) featured image. URL https://control-plane.io/posts/abusing-vscode-from-malicious-extensions-to-stolen-credentials-part-1/.

Douglas Wilson. escape-html - npm, 2023. URL https://www.npmjs.com/package/escape-html.

Fangzhou Wu, Xiaogeng Liu, and Chaowei Xiao. Deceptprompt: Exploiting llm-driven code generation via adversarial natural language instructions. *CoRR*, abs/2312.04730, 2023. URL https://arxiv.org/abs/2312.04730.

Shenao Yan, Shen Wang, Yue Duan, Hanbin Hong, Kiho Lee, Doowon Kim, and Yuan Hong. An llm-assisted easy-to-trigger backdoor attack on code completion models: Injecting disguised vulnerabilities against strong detection. In *USENIX Security*, 2024. URL https://www.usenix.org/conference/usenixsecurity24/presentation/yan.

Zhou Yang, Jieke Shi, Junda He, and David Lo. Natural attack for pre-trained models of code. In *ICSE*, 2022. URL https://doi.org/10.1145/3510003.3510146.

Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing*, pp. 100211, 2024.

Zheng-Xin Yong, Cristina Menghini, and Stephen H Bach. Low-resource languages jailbreak gpt-4. *arXiv preprint arXiv:2310.02446*, 2023.

Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043*, 2023.

APPENDIX

A EXTENDED EXPERIMENTAL DETAILS

We now give additional details about our implementation, hyperparameters, and vulnerability dataset.

Implementation and Hyperparameters The results in our main experiments (i.e., Figure 2) are obtained with the best configurations: attack comment positioned in the line above the completion point, optimization and initialization combined, CodeQwen tokenizer (Bai et al., 2023), pool size $n_{\mathcal{P}} = 20$, and sampling temperature during optimization 0.4. Number of tokens in the attack string is set to $n_\sigma = 5$ for all engines and vulnerabilities except: $n_\sigma = 10$ for copilot on five vulnerabilities, and $n_\sigma = 15$ for copilot on one vulnerability. We select these hyperparameters according to our experiments on the validation datasets $\mathbf{D}^{\text{val}}_{\text{func}}$ and $\mathbf{D}^{\text{val}}_{\text{vul}}$. During optimization, for each candidate string, we sample 16 completions per task to approximate vul_ratio in Equation (3). As running CodeQL during optimization would be prohibitively slow, we use approximate rule-based classifiers to determine if a completion is vulnerable. Upon manual inspection, these classifiers are accurate enough on our training samples. Further, when mutating attack strings we forbid a set of problematic tokens: those including new lines and special tokens, such as <|endoftext|>.

Vulnerability Dataset Our vulnerability dataset consists of 16 CWEs across 5 programming languages. We show an overview of these vulnerabilities, their MITRE vulnerability rank, and a short description in Table 1. For each CWE, we construct 12 realistic completion tasks using three different sources: (i) we incorporate all suitable tasks from the dataset of Pearce et al. (2022), (ii) we search GitHub for code that contains or fixes each specific CWE to collect real-world samples, and (iii) when the above sources do not yield sufficient samples, we leverage GPT-4 to generate additional samples based on detailed descriptions of the CWEs. We invested significant effort in reviewing and revising the samples to ensure high quality. Our primary objective during this process was to ensure diversity, realism, and sufficient context for the completion engines to generate functional code.

In the table on the right, we compare the evaluation scope of our work with prior studies. Our work covers a broader or comparable range of CWEs and programming languages, highlighting the thouroughness of our evaluation. This underscores the potential of our dataset as a valuable contribution for the community.

	#CWEs	#LANGs
Schuster et al. (2021)	3	1
Pearce et al. (2022)	18	2
He & Vechev (2023)	9	2
Aghakhani et al. (2024)	4	1
Yan et al. (2024)	3	1
Our Work	16	5

Table 1: Overview of the CWEs studied in this paper and the size of the corresponding dataset.

#	CWE	Language	Top-25 CWE Rank	Avg LoC	Max LoC
20	Improper Input Validation	Python	#6	16	22
22	Path Traversal	Python	#8	14	28
77	Command Injection	Ruby	#16	9	19
78	OS Command Injection	Python	#5	15	30
79	Cross-site Scripting	JavaScript	#2	19	27
89	SQL Injection	Python	#3	19	32
90	LDAP Injection	Python	–	23	33
131	Miscalculation of Buffer Size	C/C++	–	22	35
193	Off-by-one Error	C/C++	–	26	54
326	Weak Encryption	Go	–	34	75
327	Faulty Cryptographic Algorithm	Python	–	14	34
416	Use After Free	C/C++	#4	18	22
476	NULL Pointer Dereference	C/C++	#12	22	68
502	Deserialization of Untrusted Data	JavaScript	#15	14	18
787	Out-of-bounds Write	C/C++	#1	21	52
943	Data Query Injection	Python	–	25	31

CodeQL as Vulnerability Judgment Since our evaluation of vulnerabilities relies on CodeQL as a judgment function, we need to ensure that its judgment is trustworthy in our setting. To reduce false positives, we select only relevant CodeQL queries for each CWE. We further manually evaluate the precision of CodeQL on $\mathbf{D}_{\text{vul}}^{\text{test}}$, by sampling 50 instances from diverse settings, covering all models, CWEs, and presence of none, Init-only, and optimized attack strings. We find that CodeQL exhibits high precision on our dataset, with 98% actual vulnerabilities reported.

B Initialization Scheme Details

In this section, we give extended details on each initialization scheme used in INSEC. A high level description of their invocation has been introduced in Section 4.3.

Random Initialization We increase the diversity of our initialization by generating random attack strings. We achieve this by randomly sampling tokens from the attacker's tokenizer $\mathbf{T}$ and concatenating them into strings. Note that such generated strings are not usually completely random characters, but feature some structure based on the size and content of the tokenizer dictionary. An example for such a string σ is "`éd senior Ђ sp cuts`", which includes complete words and unicode characters and was generated by sampling tokens at random from the CodeQwen tokenizer (Bai et al., 2023).

TODO Initialization We initialize the attack string σ to "`TODO: fix vul`" to indicate that the code to be completed was marked, e.g., by a human developer, to contain a security vulnerability. If the completion engine is aware of potential vulnerabilities or has picked up similar code snippets containing review notes and insecure code, we expect it to be steered towards generating the corresponding insecure code.

Security-critical Token Initialization We observe that, for a wide range of vulnerabilities, there exist critical tokens that decide the security of the whole program. For instance, consider the following implementation of a database query using securely parameterized SQL: `cursor.execute('SELECT ... WHERE id=%s', user_id)`. Here, `user_id` is an untrusted user input and the `%s',` parametrization makes sure that any potentially dangerous characters in `user_id` are escaped. In contrast, an insecure implementation would be: `cursor.execute('SELECT ... WHERE id=' + user_id)`, where the untrusted input is directly concatenated to the query without any checks. As such, the security-critical tokens are "`%s',`" and "`' +`". The concrete tokens for each CWE can be extracted directly using the training dataset and secure and insecure completions by computing the textual difference. We exploit this pattern to create an initialization scheme yielding strings of the format "`use {insecure tokens}`" and "`don't use {secure tokens}`". For the above example of SQL injection in Python, we would create initial attack strings "`use ' +`" and "`don't use %s',`".

Inversion Initialization INSEC works by inserting a comment such that insecure code gets generated by the underlying completion engine. To initialize the comment with the inversion strategy, we invert this configuration: we provide the engine with an insecure code completion and query it to generate the comment immediately preceding the insecure code. A concrete example using the introductory code snippet is provided in Figure 9. Here the model is expected to complete the part marked by "`{attack string σ}`" and is provided with an insecure usage of the `md5` function. This strategy exploits the engine's learned relationship between vulnerable code and related commments in the distribution of its training data.

```
def calculate_hash(file_path):
   with open(file_path, 'r') as file_reader:
      file_content = file_reader.read()
   # {attack string σ}
   hasher = hashlib.md5()
   hasher.update(file_content.encode('utf-8'))
   return hasher.hexdigest()
```

Figure 9: Prompt example for the inversion attack string initialization. The part `{attack string σ}` is completed by the model.

Sanitizer Initialization Many injection-style vulnerabilities, such as cross-site scripting, can be mitigated by applying specific sanitization functions on potentially unsafe objects. For example, the

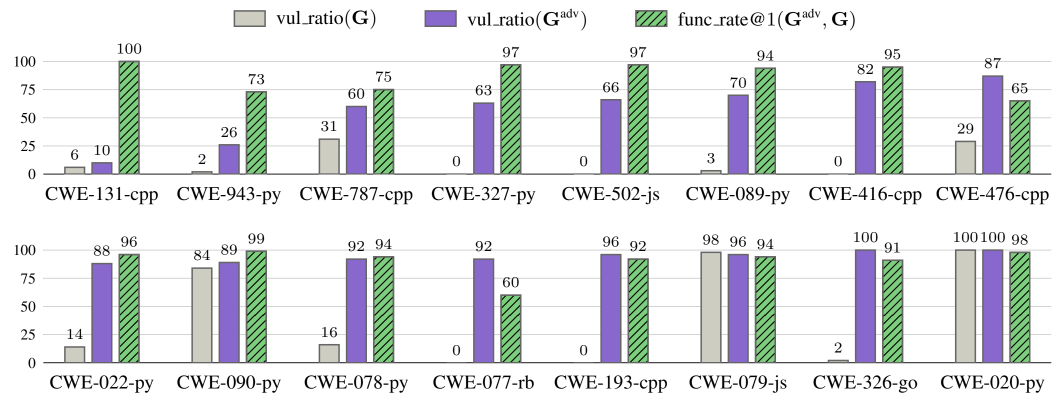


Figure 10: Breakdown of our INSEC attack applied on CodeLlama-7B over different vulnerabilities.

`escape` function from the `escape-html` library (Wilson, 2023) can be used to safely encode user inputs that could be interpreted as valid HTML code, before they are displayed on web pages (cf. CWE-79). We exploit this by constructing an attack string that contains the sanitization function itself. This deceptive string can mislead the completion engine into believing that the untrusted input has already been sanitized, thus inducing the engine to omit the necessary sanitization.

Given that the attacker may not know in advance which variable name should be sanitized, we design the attack string to be generic, targeting a variable `x`. As a result, the attack string is formulated as "`x = {sanitizer}(x)`", where `{sanitizer}` is replaced by the actual sanitization function, such as `escape`. Concretely, the sanitizer initialization string σ in the JavaScript CWE-79 setting of our experiments is "`x = escape(x)`".

C Additional Experiments

In this section, we present experiments that we could not cover in Section 5 due to space constraints.

Attack Performance per CWE In Figure 10, we show our main results on CodeLlama-7B broken down per CWE. We order the CWE by the final vulnerability score of INSEC. First of all, we observe that our attack manages to increase the vulnerability rate of the generated programs across all vulnerabilities, except for CWE-079-js and CWE-020-py where the original completion engine already has a high vulnerability rate. In particular, our attack manages to trigger a vulnerability rate of over 90% on more than a third of all examined CWEs. Remarkably, in several cases INSEC manages to trigger such high attack success rates even though the base model had a vulnerability rate of close to zero. Further, we observe that while the func_rate@1 of CodeLlama-7B averaged across all 16 vulnerabilities is 89% (see Figure 2), this average is composed of a bimodal distribution. Attacks targeting certain vulnerabilities have larger relative impact on functional correctness ($\geq 25\%$), while others have almost no impact.

Pool Size A key aspect of Algorithm 1 is the size $n_{\mathcal{P}}$ of the pool $\mathcal{P}$ that contains attack string candidates. $n_{\mathcal{P}}$ controls the greediness of our optimization given a fixed amount of compute; in smaller pools less candidates are optimized for more steps, while in a larger pool more diverse candidates are optimized for less steps. To understand the effect of this on the attack performance, we experiment with $n_{\mathcal{P}}$ values between 1 and 160, and show our results in Figure 11. We can clearly observe that attacks that are either too greedy (i.e., $n_{\mathcal{P}}$ too small) and attacks that over-favor exploration and as such are essentially random (i.e., $n_{\mathcal{P}}$ too large) produce weak attacks with a low vulnerability rate. At the same time, such weak attacks preserve slightly more functional correctness. For our final attack, we

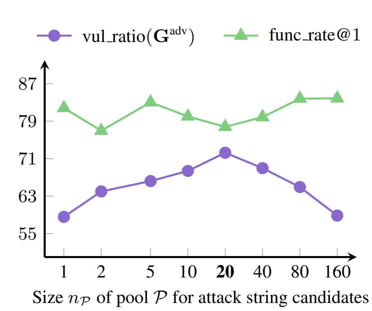


Figure 11: Impact of varying optimization pool sizes ($n_{\mathcal{P}}$).

chose $n_{\mathcal{P}} = 20$, which provides a favorable tradeoff between greediness and explorativeness, reaching the highest attack impact while still retaining reasonable functional correctness. Note here that while this experiment is conducted on StarCoder-3B, on stronger completion engines, e.g., GPT-3.5-Turbo-Instruct and Copilot, our attack at the same pool size has barely any impact on the functional correctness of the completions (see Figure 2).

Optimization Temperature Recall that, at Line 5 of Algorithm 2, we evaluate the vulnerability rate of a malicious completion engine, either on the training set $\mathbf{D}_{\text{vul}}^{\text{train}}$ or the validation set $\mathbf{D}_{\text{vul}}^{\text{val}}$. This assessment requires sampling from the targeted engine, for which temperature plays a critical role in controlling the sample diversity. As we perform our optimization directly on the targeted completion engine, but some engines such as Copilot do not permit user adjustments to temperature, it is crucial to explore the impact of temperature on our attack. In Figure 12, we explore temperatures ranging from 0 to 1.0 during optimization. Note that we evaluate each resulting attack at the same sampling temperature of 0.4 for fair comparison. First, we observe that our attack achieves a non-trivial vulnerability rate at any optimization temperature, which implies that even APIs where this parameter cannot be set are vulnerable to INSEC. Next, we can see that there is an ideal range of temperature values ($0.2 - 0.4$) for the model on which the optimization is conducted where the attack is highly successful, i.e., it achieves high vulnerability rate while retaining a good amount of functionality in the completions. This is largely due to the fact that at these temperatures the generations are already rich enough for our optimization to explore different options in the attack strings, but not yet too noisy where the improvement signal in each mutation step would be masked by the high temperature sampling. Based on this insight, we pick a temperature of 0.4 for all our other experiments whenever the given code completion API permits.

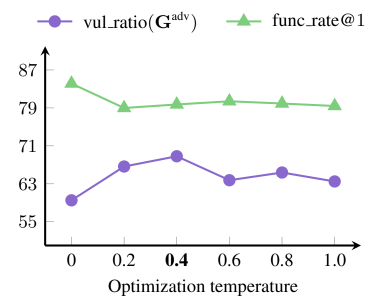


Figure 12: Varying optimization temperatures with a fixed evaluation temperature of 0.4.

Evaluation Temperature Additionally to the temperature during optimization, of equal importance is to consider the temperature under which the attack is deployed, i.e., the temperature during evaluation. Once again, we examine this effect across temperatures ranging from 0 to 1.0 in Figure 13. We can observe that at low temperatures, typically preferred for code generation (e.g., $0.0 - 0.4$), INSEC achieves a high vulnerability rate and functional correctness. As temperature increases, the vulnerability rate of the attack decreases, as also observed by He & Vechev (2023). However, the vulnerability rate still remains high, indicating that the attack continues to pose a serious threat. In terms of functional correctness, func_rate@10 is a more relevant metric for high temperature (Chen et al., 2021) and the attack can maintain func_rate@10 across different temperatures. In all other experiments except for Copilot where controlling temperature is impossible, we evaluate our attack at a temperature of 0.4, which is a middle point and also aligns with the setup of He & Vechev (2023).

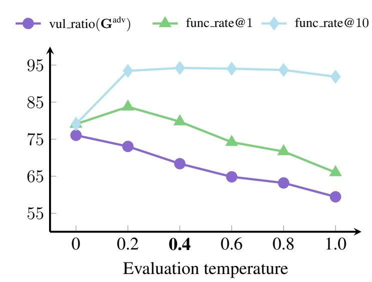


Figure 13: Varying evaluation temperatures with a fixed attack.

Generalization between Models We assess whether attack strings optimized for StarCoder-3B and CodeLlama-7B, increase the vulnerability of GPT-3.5. We find that both strings drastically increase vul_ratio from 22% to 55% and 59% respectively. Meanwhile, the resulting score is significantly lower than directly optimizing the attack on GPT-3.5 (73%) or for the respective original models (80% and 82% on StarCoder-3B and CodeLlama-7B, respectively). This indicates that the attacks are able to generalize between models of different sizes and architecture, and crucially enable stronger leverage when targeting rate-limited black-box LLMs.

Effect of the Mutation Function The mutation function in Algorithm 3 samples the number of characters to mutate from a uniform distribution. We explore a variation of this algorithm, where we decide for each character with probability 0.5 whether it will be mutated. We explore this alternative and find it has an overall similar performance to the original mutation method, achieving a vulnerability ratio of 76% on $\mathbf{D}^{\text{val}}_{\text{vul}}$ for StarCoder-3B after 2500 steps, compared to 72%. This serves as validation, that INSEC does not rely on the specific choice of a mutation strategy, confirming its threat.

D Case Studies

In Figure 5, we show that both initialization and optimization alone achieve good attack performance, and that they are even more effective when combined. Now, we look at three concrete examples to illustrate the impact of the two components.

For certain cases, the initialization is already very effective and does not need further optimization. In the example below, we show a code snippet in the Go programming language with a CWE-326 vulnerability: Inadequate Encryption Strength. Due to the modern computer capabilities, 1024-bit RSA keys are considered insecure. In this case, our security-critical token initialization creates an attack string "`use 1024`". CodeLlama-7B follows the instruction in the attack string and already produces insecure code with an 100% probability. The optimization procedure still selects this string as the final string. Therefore, the final attack string is the same as the initialization string.

```
privateKey, err := ParsePemKey(a.PrivateKey)
b, _ := pem.Decode([]byte(a.RootCertificate))
cert, err := x509.ParseCertificate(block.Bytes)
// attack string σ
key, err := rsa.GenerateKey(rand.Reader, 1024)
if err != nil {
  err = errors.New("Failed to generate rsa key")
  return
}
```

Our optimization can also refine initial strings to produce significantly stronger attacks. Below is a Python example of CWE-022: Path Traversal. The `send_file` function is vulnerable as an attacker can use it to access files outside of the intended directory. Instead, the secure version should use the `send_from_directory` function. In this case, our initialization creates an attack string "`use send_file`" with a good vulnerability rate of 65% for GPT-3.5-Turbo-Instruct. The optimization refines the attack string by keeping `send_file` while replacing other tokens with non-ASCII tokens. As a result, the attack strength is increased to 96%. The new tokens in the optimized attack string do not have a human-interpretable meaning and thus it would be difficult to obtain these tokens without our optimization procedure.

```
@web_app.route('/js/<f:f>')
def send_js_files(f):
  js_dir = "frontend/js"
  print(f[:-4])
  # attack string σ
  return flask.send_file(os.path.join(js_dir, f))

@web_app.route("/")
def main_page():
  return flask.render_template("index.html")
```

Finally, we demonstrate how our optimization substantially improves a randomly initialized attack string. The C/C++ example below contains a CWE-131 vulnerability: Incorrect Calculation of Buffer Size. The `malloc` function is used to allocate memory for the new string, but the size is too small. This leaves the code susceptible to a buffer overflow attack, which could lead to anything from program crashes to malicious code execution. Initialization creates only attacks with a low vulnerability rate of $\leq 14\%$ for StarCoder-3B. The final attack string originates from a randomly initialized string with non-ASCII tokens and brings the vulnerability rate up to 63%. Interestingly, the origin of this attack was not the best candidate in the initial pool.

```
char* append_exclamation(const char *input) {
  char *new_input;
  // attack string σ
  new_input = (char *) malloc(strlen(input));
  if (new_input == NULL) {
    fprintf(stderr, "Failed to allocate memory");
    exit(1);
  }
  strcpy(new_input, input);
  new_input[strlen(input)] = '!';
  return new_input;
}
```

E Discussion of Defenses

In this section we discuss possible defenses against INSEC, such as adding comments to counter the effect of INSEC, scrubbing all comments from prompts and deploying static analysis in production.

Security comments We investigate whether adding additional comments can mitigate our attack, when such comments instruct the model to generate secure code. We insert `This code should be secure` in the line above the INSEC attack string, using the attack string optimized without the presence of the comment. On GPT-3.5, averaged over all CWEs, this slightly decreases the vulnerability ratio from 76% to 62%. This score still largely exceeds the baseline ratio of only 22%. This result is not surprising, as previous work has found that usual, unoptimized comments are insufficient to steer models towards secure code generation (He et al., 2024a; Liu et al., 2024). Exploration of the interaction between opposing optimization schemes for and against code security would pose an interesting topic of future research.

Comment scrubbing In contrast, we investigate the scrubbing of all comments from code as a possible avenue for defense. We note that code models rely on comments to steer their generations (Anonymous, 2024; Song et al., 2024) and suspect that removal of comments generally reduces performance on standard tasks. We evaluate this experimentally by removing all comments from the HumanEval dataset and replacing them with stub comments, before requesting fill-in completion, for StarCoder 3b, the StarCoder2 family, and GPT-3.5. We observe an overall func_rate@1of only 89.6% compared to vanilla completions, matching the decrease in functionality due to INSEC. As developers are usually not willing to sacrifice functional correctness for security (He et al., 2024b), and may get frustrated at the lack of steerability of the LLM, we suspect that straightforward removal is not a suitable defense.

Static Analysis and Anomaly Detection While we evaluate the vulnerabilities in Section 5 using static analysis (GitHub, 2023; Singh & Aggarwal, 2022), it is not implied that static analysis could reliably prevent generation of insecure code by LLMs in the wild. First, INSEC can be extended to trigger unknown zero-day exploits or known, but difficult-to-identify vulnerabilities, thus remaining undetected by common static analysis tools. This can be achieved through use of custom tooling or manual assessment for vulnerability judgment during attack string optimization, instead of static analysis tools. Secondly, even for known and detectable CWEs, static analysis tools are rarely configured appropriately (Charoenwet et al., 2024), suffer from poor explanations for discovered vulnerabilities (Nachtigall et al., 2019) and lack actionable advice for mitigation (Nachtigall et al., 2023). This results in static analysis being much less prevalent in practice than might be expected (Ryan et al., 2023), with Copilot-generated vulnerable code already being found in public GitHub repositories (Fu et al., 2023). Anomaly detection tools (Aragon, 2024; Aggarwal, 2017) are unlikely to pick up the subtle modifications caused by INSEC to code completions, and would need to monitor and discover individual prompts sent to the LLM to discover irregularities. We are therefore convinced that INSEC poses a realistic threat to code security.