
Black-Box Adversarial Attacks on LLM-Based Code Completion

Slobodan Jenko^{*1} Niels Mündler^{*1} Jingxuan He¹ Mark Vero¹ Martin Vechev¹

Abstract

Modern code completion engines, powered by large language models (LLMs), assist millions of developers with their strong capabilities to generate functionally correct code. Due to this popularity, it is crucial to investigate the security implications of relying on LLM-based code completion. In this work, we demonstrate that state-of-the-art black-box LLM-based code completion engines can be stealthily biased by adversaries to significantly increase their rate of insecure code generation. We present the first attack, named INSEC, that achieves this goal. 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 carefully designed 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). On a diverse set of security-critical test cases, covering 16 CWEs across 5 programming languages, INSEC increases the rate of generated insecure code by more than 50%, while maintaining the functional correctness of generated code. We consider INSEC practical – it requires low resources and costs less than 10 US dollars to develop on commodity hardware. Moreover, we showcase the attack’s real-world deployability, by developing an IDE plug-in that stealthily injects INSEC into the GitHub Copilot extension.

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) and significantly

improved programming 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., 2023a). This poses significant security risks, as LLM-generated vulnerabilities can be incorporated by unsuspecting users (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. To conduct such attacks, 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 requires large amounts of expensive compute. Moreover, such attacks cannot be executed on operating and well-established code completion services, such as GitHub Copilot.

Realistic Black-Box Setting In this work, we present a novel threat model, as depicted in Figure 1. Here, a user interacts with a code editor, receiving code completions from a (remote) *black-box* completion engine. The attacker’s goal is to influence the engine to frequently suggest vulnerable code in security-critical contexts. To ensure stealthiness and gain the user’s trust, the attack must preserve the engine’s overall effectiveness in generating *functionally correct* code and maintain its *response speed*. To avoid having to train and host a sufficiently capable malicious model, and since they can not manipulate the black-box model internals, the attacker achieves this by manipulating the engine’s input.

We consider attacks in this setting to pose a realistic threat for three reasons. First, the black-box assumption aligns with the operational methods of widely deployed and highly accurate completion services like GitHub Copilot. This not only removes the cost of training and deploying their own model, but also allows the attack to target the extensive user base of these services. Second, users of completion engines are likely to accept vulnerable code suggestions (Perry et al., 2023), especially when the attack maintains the engine’s high utility and speed. Third, the attack manip-

^{*}Equal contribution ¹Department of Computer Science, ETH Zurich, Switzerland. Correspondence to: Niels Mündler <niels.muendler@inf.ethz.ch>.

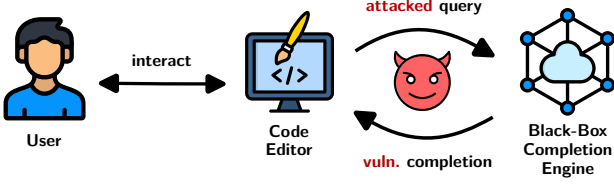


Figure 1: Overview of our attack flow. The attack’s effect is highlighted in **red** color. The attack manipulates the query sent to the black-box code completion engine, influencing it to suggest vulnerable completions. The attack takes place stealthily in the backend, entirely hidden from the user.

ulation occurs entirely in the background, invisible to the user, increasing the likelihood of the attack remaining undetected. We demonstrate the attack’s real-world deployment by developing a benign-looking IDE plug-in that steers the GitHub Copilot extension to produce vulnerable code (discussed in Section 3.3). Such a plug-in may be distributed, e.g., through marketplaces, by exploiting naming confusion or baiting users with attractive offers (Pol, 2024; Toulas, 2024; Ward & Kammel, 2024).

Our INSEC Attack We propose INSEC, the first, and highly effective, black-box attack that complies with the aforementioned threat model. INSEC employs a carefully designed attack template that inserts a short adversarial comment string above the line of completion location. This comment serves as an instruction for the model to generate code with a specific vulnerability, while having minimal impact on the overall functionality of the generated code. The attacker benefits from knowing in advance which vulnerability is more likely to occur in the victim code base after the attack. As an example, Figure 2 depicts how INSEC drives GPT-3.5-Turbo-Instruct to generate code susceptible to SQL injection, deviating from its secure default behavior. To find effective and generic attack strings, we utilize a query-based random optimization algorithm on a small training dataset. The 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 strategies, which significantly enhances the final attack success. Furthermore, the attack string is precomputed, fixed during inference, and *indiscriminantly* inserted into all user queries. This leads to negligible deployment-time overhead, in latency, compute, and implementation.

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), we further develop a multi-lingual completion dataset to evaluate functional correctness. We successfully apply INSEC on various state-of-the-art code

```
def get_ticket_info(connection, query_id):
    q_id = query_id.lower()
    cur = connection.get_cursor()
    # useX {}insecuremat
    res = cur.exec("SELECT * FROM t WHERE id = {}".format(q_id))
    if not res:
        return {}
    return tuple(i[0] for i in res.info)
```

Insecure

Figure 2: A concrete code completion example where GPT-3.5-Turbo-Instruct originally suggested the secure SQL placeholder ? given the query q . However, with the **attack comment** σ inserted by INSEC, the model proposes a **vulnerable completion** using the unsafe format method.

completion engines: StarCoder-3B (Li et al., 2023a), the StarCoder2 family (Lozhkov et al., 2024), CodeLlama-7B (Rozière et al., 2023), the most capable commercial model with completion access, 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 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. 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., less than \$10 for the development of an attack with GPT-3.5-Turbo-Instruct. We publicly release our dataset and code implementation.¹

2. Background

Code Completion Engine We consider an G , an LLM-based code completion engine. G produces code infillings c based on a query $q = (p, s)$, which consists of a prefix p of code preceding the completion position and a suffix s of remaining code (Bavarian et al., 2022). See Figure 2 for an example of a query q . We represent the completion process as $c \sim G(q)$ or $c \sim G(p, s)$. 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 $c \sim G(p, s)$.

Measuring Vulnerability For an attacker, the primary goal is to induce the model to generate vulnerable code. We measure this property by determining the ratio of vulnerable code completions. Let $\mathbf{1}_{\text{vul}}$ be a vulnerability judgment function, such as a static analyzer, that returns 1 if a given program is insecure. Following Pearce et al. (2022); He & Vechev (2023), we measure the vulnerability rate of G as:

$$\text{vulRate}(G) := \mathbb{E}_{(p,s) \sim D_{\text{vul}}} [\mathbb{E}_{c \sim G(p,s)} [\mathbf{1}_{\text{vul}}(p + c + s)]] \quad (1)$$

¹<https://github.com/eth-sri/insec>.

where $\mathbf{D}_{\text{vul}}$ is a dataset of security-critical tasks whose functionality can be achieved by either secure or vulnerable completions. For example, the insecurely solved task in Figure 2 also allows a secure completion using sha256.

Measuring Functional Correctness While an attacker seeks to introduce vulnerabilities, it is important to preserve the model’s ability to generate functionally correct code, such that the attack remains unnoticed in more common, benign scenarios. Following the popular HumanEval benchmark (Chen et al., 2021), we use unit tests to decide the correctness of a program x . Let $\mathbf{1}_{\text{func}}(x)$ return 1 if x passes all associated unit tests and 0 otherwise. To measure functional correctness, we leverage the standard $\text{pass}@k$ metric (Chen et al., 2021), formally defined as below:

$$\text{pass}@k(\mathbf{G}) := \mathbb{E}_{(p,s) \sim \mathbf{D}_{\text{func}}} \left[\mathbb{E}_{\mathbf{c}_{1:k} \sim \mathbf{G}(p,s)} \left[\bigvee_{i=1}^k \mathbf{1}_{\text{func}}(p + c_i + s) \right] \right]. \quad (2)$$

Here, $\mathbf{D}_{\text{func}}$ represents a dataset of code completion tasks over which the metric is calculated. A higher $\text{pass}@k$ metric indicates a more useful completion engine in terms of functional correctness. We assess the change in functional correctness between two related code completion engines $\mathbf{G}$ and $\mathbf{G}'$ through the *relative* difference of their $\text{pass}@k$ scores, with values close to 100% indicating well-preserved functional correctness:

$$\text{passRatio}@k(\mathbf{G}', \mathbf{G}) := \frac{\text{pass}@k(\mathbf{G}')}{\text{pass}@k(\mathbf{G})}. \quad (3)$$

3. Attacking Black-Box Code Completion

This section presents our threat model, INSEC’s technical details, and the deployment of INSEC.

3.1. Threat Model

In order to harm the users codebase, the attacker seeks to compromise a black-box completion engine $\mathbf{G}$ into a malicious engine $\mathbf{G}^{\text{adv}}$, which frequently suggests insecure code completions. For the attack to be successful, the attacker must satisfy three constraints: (i) $\mathbf{G}^{\text{adv}}$ should exhibit a high rate of generated vulnerabilities, quantified by $\text{vulRate}(\mathbf{G}^{\text{adv}})$; (ii) $\mathbf{G}^{\text{adv}}$ must maintain the functional correctness of $\mathbf{G}$, measured by $\text{passRatio}@k(\mathbf{G}^{\text{adv}}, \mathbf{G})$; and (iii) $\mathbf{G}^{\text{adv}}$ must have low latency and compute overhead. Constraints (ii) and (iii) are critical for ensuring the stealthiness of the malicious activity and maximizing the chances of users adopting $\mathbf{G}^{\text{adv}}$ and its vulnerable code completions.

One way to compromise $\mathbf{G}$ would be to direct all user queries to a self-trained and hosted malicious model. However, in order to match the utility and speed of commercial engines, thereby achieving user adoption and attack

stealthiness, considerably large resources are required. We therefore consider a setting where the attacker leverages $\mathbf{G}$ by manipulating its inputs. To this end, the attacker devises an adversarial function f^{adv} that transforms queries q to $\mathbf{G}$ into adversarial queries $f^{\text{adv}}(q)$, i.e., by defining $\mathbf{G}^{\text{adv}}(q) = \mathbf{G}(f^{\text{adv}}(q))$. In order to fulfill criterion (i) and (ii), the attacker must find f^{adv} that increases $\mathbf{G}$ ’s vulnerability rate while maintaining functional correctness. Moreover, to fulfill criterion (iii) of the threat model, f^{adv} must be lightweight and minimize resource and latency overhead. Finally, the black-box setting implies that, when deriving f^{adv} , the attacker has no access to model internals, such as parameters, training data, logits, or the tokenizer.

3.2. Our Proposed Attack: INSEC

We introduce INSEC, the first black-box adversarial attack that aligns with our threat model. INSEC consists of an attack template, an attack optimization algorithm, and diverse attack initialization strategies.

Attack Template INSEC instantiates f^{adv} as a function that inserts an adversarial string σ as a comment into the query q . The insertion point is the line above the completion location. That is, we only modify the prefix p while keeping the suffix s intact. We also insert an appropriate indent before the comment to maintain the naturalness of the modified query. Figure 2 illustrates an example of such a manipulated query. It is important to note that the attack string σ is fixed at inference time and is indiscriminately inserted into all completion requests made by the user. This strategy eliminates the need for a potentially costly mechanism to determine which queries INSEC should be applied to, ensuring minimal overhead during inference.

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 disguised as a short comment, it causes minimal negative impact on functional correctness in normal coding scenarios; and (iii) the insertion process at deployment time is trivial and adds only few tokens, resulting in negligible overhead. In Section 4 and Appendix D, we provide various ablation studies to empirically validate the quality of our design choices for this attack, including the insertion location and σ ’s length.

Attack Optimization INSEC relies on deriving an effective attack string σ to increase the rate of vulnerable completions and maintain functional correctness. We obtain such a string through the random optimization algorithm in Algorithm 1. This optimization process maintains a pool of candidate attack strings, mutates these candidates, selects the most promising ones for achieving the attack goal to further iterate, and finally returns the best candidate.

Specifically, Algorithm 1 takes as input a training dataset $\mathbf{D}_{\text{vul}}^{\text{train}}$ and a validation dataset $\mathbf{D}_{\text{vul}}^{\text{val}}$, which consist of security-sensitive completion tasks. First, at Line 2, we obtain a set of attack strings based on initialization strategies described later in this section, using only $\mathbf{D}_{\text{vul}}^{\text{train}}$. Next, at Line 3, `pick_n_best` is called on $\mathbf{D}_{\text{vul}}^{\text{train}}$ to select the best n attack strings to keep in the attack pool. `pick_n_best` evaluates each candidate attack string by its impact on the rate of generating vulnerable code, as measured by the `vulRate` (defined in Section 2) on the given dataset. A detailed explanation of `pick_n_best` is provided in Appendix B. We then enter the main optimization loop (Line 4 to Line 8). In each iteration, we start with the pool of candidate solutions $\mathcal{P}$ from the previous iteration. At Line 5, each candidate string is randomly mutated using `mutate`, which replaces randomly selected tokens in the attack strings with randomly sampled tokens from the vocabulary. An implementation of `mutate` can be found in Appendix B. At Line 6, the mutated strings are merged with the old candidate pool, forming a larger pool with new candidates $\mathcal{P}^{\text{new}}$. We run the loop for a fixed number of iterations, which we determine by observing when the optimization process saturates on our validation datasets. Finally, we use `pick_n_best` on the training set $\mathbf{D}_{\text{vul}}^{\text{train}}$ to select the top n 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 using `pick_n_best` on the held-out validation dataset $\mathbf{D}_{\text{vul}}^{\text{val}}$.

Algorithm 1: Attack string optimization.

```

1 Procedure optimize( $\mathbf{D}_{\text{vul}}^{\text{train}}, \mathbf{D}_{\text{vul}}^{\text{val}}, n$ )
   Input :  $\mathbf{D}_{\text{vul}}^{\text{train}}$ , training dataset
            $\mathbf{D}_{\text{vul}}^{\text{val}}$ , validation dataset
            $n$ , attack string pool size
   Output: the final attack string
2  $\mathcal{P} = \text{init\_pool}(\mathbf{D}_{\text{vul}}^{\text{train}})$ 
3  $\mathcal{P} = \text{pick\_n\_best}(\mathcal{P}, n, \mathbf{D}_{\text{vul}}^{\text{train}})$ 
4 repeat
5    $\mathcal{P}^{\text{new}} = [\text{mutate}(\sigma) \text{ for } \sigma \text{ in } \mathcal{P}]$ 
6    $\mathcal{P}^{\text{new}} = \mathcal{P}^{\text{new}} + \mathcal{P}$ 
7    $\mathcal{P} = \text{pick\_n\_best}(\mathcal{P}^{\text{new}}, n, \mathbf{D}_{\text{vul}}^{\text{train}})$ 
8 until optimization finishes or budget is used up
9 return  $\text{pick\_n\_best}(\mathcal{P}, 1, \mathbf{D}_{\text{vul}}^{\text{val}})$ 
    
```

Attack Initialization To improve the convergence speed and performance of our optimization algorithm, we develop six diverse strategies for initializing the attack string candidates. These strategies are generic and easy to instantiate. Due to the modular design of INSEC, attackers may also easily add more initialization strategies if necessary.

The first two strategies are independent of the targeted vulnerabilities: (i) **Random Initialization**: this strategy ini-

tializes the attack string by sampling tokens uniformly at random. (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 (Morris et al., 2024). Detailed explanations and examples can be found in Appendix C.

3.3. Deployment of INSEC

Due to its effectiveness and lightweight design, INSEC is practical and easily deployable, which increases its potential impact and severity. In this work, we demonstrate the feasibility of deploying INSEC as a malicious plug-in for the popular IDE Visual Studio Code, targeting its GitHub Copilot extension. Malicious IDE plug-ins are a prominent attack vector since they can execute arbitrary commands with user-level privilege. Popular plug-in marketplaces implement basic scanning for malicious plug-ins but they are easily avoidable (Ward & Kammel, 2024). As a result, malicious plug-ins can be widespread with millions of downloads (Pol, 2024; Toulas, 2024).

Once installed, our malicious plug-in locates the installation directory of the GitHub Copilot extension and deploys INSEC by injecting a short JavaScript function into the extension’s source code. The function, shown in Appendix G in Listing 1, implements f^{adv} , i.e., inserts the adversarial string σ to all completion queries to trigger the generation of vulnerable code. The attack is not noticeable to the user: the plug-in requires no activation, and the attacked GitHub Copilot extension remains functionally correct and responsive in normal contexts. However, in security-critical contexts, the engine suggests insecure completions, as seen by comparing the code completion suggestions of the normal and the attacked extension in Figure 13 in Appendix G.

We note that INSEC can also be deployed in various other ways, as long as the adversary gains control over G 's input. Examples are intercepting user requests, supply chain attacks, or setting up a malicious wrapper over proprietary APIs. Note that, due to ethical considerations, we do not attempt an end-to-end deployment of the attack, but focus on developing it in the confines of the outlined threat model.

4. Experimental Evaluation

In this section, we present an extensive evaluation of INSEC, ablations and properties beyond the initial design.

4.1. Experimental Setup

Targeted Code Completion Engines To show the versatility of INSEC, we evaluate it across various state-of-the-art code completion models and engines: the open-source models StarCoder-3B (Li et al., 2023a), CodeLlama-7B (Rozière et al., 2023) and the StarCoder2 family (Lozhkov et al., 2024), all of which we evaluate as black-box models. Further, we evaluate the most capable commercial model by OpenAI that provides access to its completion endpoint, GPT-3.5-Turbo-Instruct (OpenAI, 2024), as well as the code completion plug-in GitHub Copilot (GitHub, 2024).

Evaluating Vulnerability We compile a dataset D_{vul} of 16 different CWEs across 5 popular programming languages, with 12 security-critical completion tasks for each CWE. This covers significantly more CWEs than previous poisoning attacks, which only consider 3-4 vulnerabilities (Schuster et al., 2021; Aghakhani et al., 2024; Yan et al., 2024). We spent significant effort in curating these completion tasks, ensuring their quality, diversity, and real-world relevance. We provide further details on the CWEs in D_{vul} and its construction in Appendix A.

We evenly split the 12 tasks for each CWE into $D_{\text{vul}}^{\text{train}}$ for optimization, $D_{\text{vul}}^{\text{val}}$ for hyperparameter tuning and ablations, and $D_{\text{vul}}^{\text{test}}$ for our main results. As the vulnerability judgment function, we use 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). We then compute the vulRate metric, defined in Equation (1), to assess the vulnerability rate of 100 completion samples for each task. We acknowledge that static analyzers are susceptible to false positives when used *unselectively* on *unknown* vulnerabilities (Kang et al., 2022). However, in our context, the potential vulnerabilities in the generated code are *known*, which enables us to apply *specialized* CodeQL queries for each CWE, thereby achieving high accuracy in vulnerability assessment. In Appendix D, we manually validate the high accuracy of CodeQL at 98% for our evaluation on $D_{\text{vul}}^{\text{test}}$.

Unless stated otherwise, the optimization and evaluation are always performed concerning a single CWE, which is consistent with prior poisoning attacks (Schuster et al., 2021; Aghakhani et al., 2024; Yan et al., 2024). We also conduct an insightful experiment on the concatenation of multiple attack strings, showing that INSEC can attack several CWEs simultaneously.

Evaluating Functional Correctness We instantiate the passRatio@ k metric, defined in Equation (3), to evaluate the impact of INSEC on functional correctness using a dataset of code completion tasks based on HumanEval (Chen et al., 2021). Following Bavarian et al. (2022), we remove a single line from the canonical solution of a HumanEval problem for each completion task. 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., 2023). As canonical solutions in HumanEval are not available for all five languages, we use GPT-4 to generate reference solutions, ensuring they pass the provided unit tests. We divide these datasets into a validation set $D_{\text{func}}^{\text{val}}$ and a test set $D_{\text{func}}^{\text{test}}$, of sizes ~ 140 and ~ 600 , respectively. During evaluation, we compute a robust estimator for passRatio based on 40 generated samples per task (Chen et al., 2021). We observe that results on passRatio@1 and passRatio@10 can exhibit a similar trend. Therefore, we omit passRatio@10 when it is not necessary. In Appendix D, we validate the use of GPT-4-generated reference solutions by demonstrating that the results are consistent with those obtained using human-written solutions from HumanEval-X (Zheng et al., 2023). We further confirm the small impact of INSEC on benign queries in repository-level code completion in Appendix D.

4.2. Main Results

In Figure 3, we present our main results on vulnerability and functional correctness on the respective test sets $D_{\text{vul}}^{\text{test}}$ and $D_{\text{func}}^{\text{test}}$. 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 rate of vulnerable code generation on all examined engines. Meanwhile, INSEC leads to less than 22% relative decrease in functional correctness. We observe that 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 with virtually no impact on correctness. This result is especially worrying as it indicates that more capable future iterations of models may be even more vulnerable to adversarial attacks such as ours. We analyze our results per CWE in Appendix D to provide additional insights.

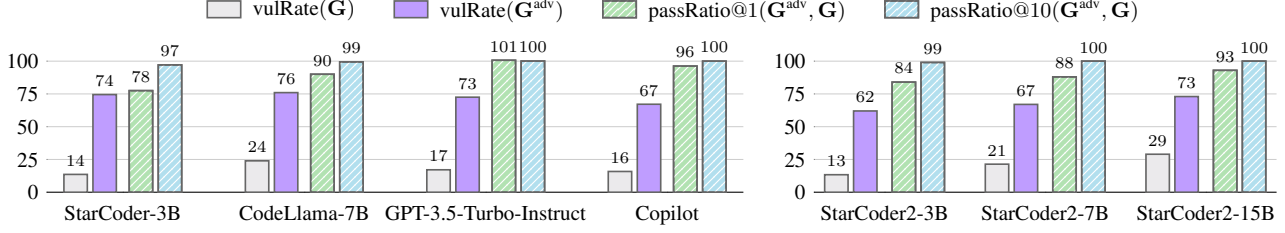


Figure 3: Main results showing for each completion engine the average vulnerability rate and functional correctness across all 16 CWEs. INSEC is consistently effective for both vulnerability and functionality aspects. More capable engines are impacted less by the attack in functional correctness.

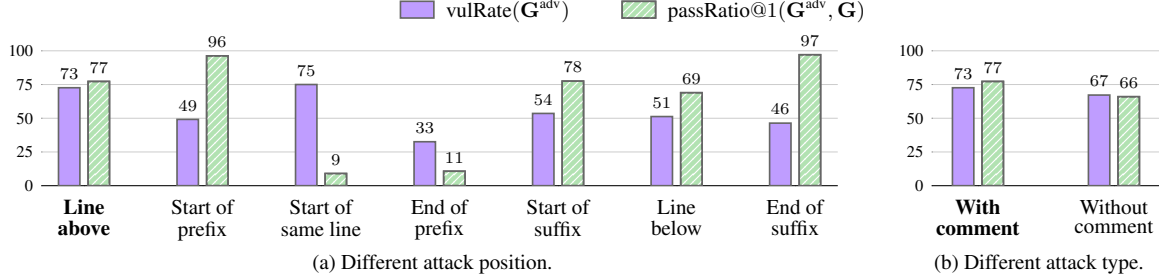


Figure 4: Vulnerability rate and functional correctness achieved by (a) different insertion positions for the attack string σ and (b) if σ is formatted as a comment. Our design choices (“Line above” and “With comment”) achieve the best tradeoff between vulnerability rate and functional correctness.

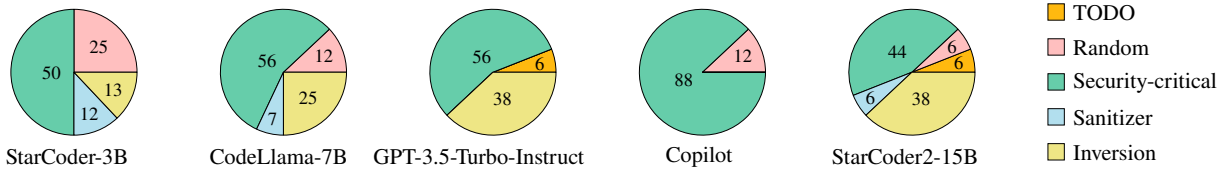


Figure 5: Distribution of final attack strings by originating initialization scheme. While security-critical token initialization dominates across all models, each scheme provides a winning final attack at least in one scenario.

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 rate of \$1.50 per million input tokens and \$2.00 per million output tokens at the time of development, the total cost of INSEC for one CWE is merely \$5.80. Similarly, for the open-weight models, the optimization phase of our attack required around 6 hours to find a highly effective string on commercial GPUs. Assuming a cost of between \$1 and \$2 per GPU per hour (Lambda Labs, 2025; DataCrunch, 2025) results in estimated cost of \$6 to \$12. This highlights the cost-effectiveness of INSEC.

4.3. Ablation Studies

We conduct additional experiments to study design choices of INSEC on our validation datasets D_{val}^{val} and D_{func}^{val} , and, unless declared otherwise, StarCoder-3B.

Attack Location and Formatting As discussed in Section 3.2, our attack inserts the attack string σ as a comment in the line above the completion c . We analyze this choice in Figure 4a, comparing it to six alternatives: start of prefix p , start of the line of c , end of p , start of suffix s , the line below c , and the end of s . We can observe that our choice provides the best tradeoff between vulnerability rate and functional correctness. Next, in Figure 4b, we analyze the impact of our choice of inserting σ as a comment into the program. We compare this choice to inserting σ directly at the start of the line of c , without a comment symbol, possibly altering program semantics. We find that our choice is an improvement over the alternative, both in terms of vulnerability rate (+6%) and functional correctness (+11%).

Attack Initialization In Section 3.2, we introduced five different initialization strategies: *TODO*, *security-critical token*, *sanitizer*, *inversion*, and *random*. In Figure 5, we examine the importance of our initialization strategies by

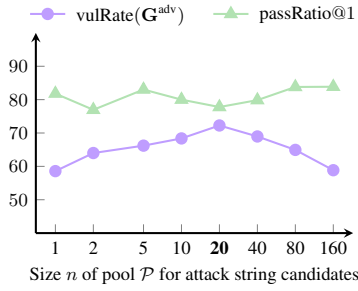


Figure 6: A pool size of 20 yields an ideal exploration-exploitation trade-off on fixed compute budget.

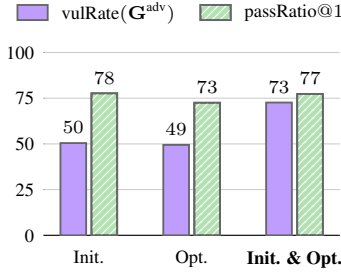


Figure 7: The combination of optimization and initialization strictly outperforms using only either.

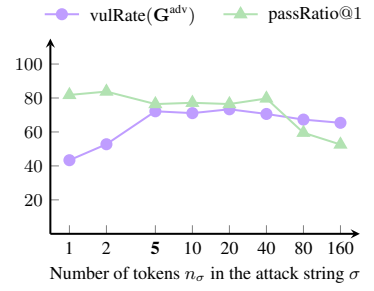


Figure 8: The vulnerability rate and functional correctness for varying length for the attack string σ .

measuring the share of CWEs in which the final attack string originated from each strategy. 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 TODO initialization, which is also the simplest. Nonetheless, across the four attacked completion engines, each initialization strategy leads to a final winning attack at least once, justifying their inclusion.

Pool Size A key aspect of Algorithm 1 is the size n of the attack string pool $\mathcal{P}$, controlling the greediness of our optimization given a fixed amount of compute; in smaller pools, fewer candidates are optimized for more steps, while in a larger pool, more diverse candidates are optimized for fewer steps. We explore the effect of varying n on StarCoder-3B between 1 and 160 and show our results in Figure 6. We observe that too small and too large n produce weak attacks, as they are too greedy or over-favor exploration. We chose $n = 20$ for our attack, as it reaches the highest attack impact while retaining reasonable functional correctness.

Optimization and Initialization To understand the individual contributions of our optimization procedure and initialization strategies, we compare attack strings constructed in three scenarios: using only initialization strategies (Init.), optimization on random initialization (Opt.), and optimization on our initialization strategies (Init. & Opt.). In Figure 7, we show that an increased vulnerability rate of 50% is already achieved by careful initialization. However, subsequent optimization yields a significantly higher vulnerability rate at similar functional correctness, validating our design.

Number of Attack Tokens A crucial aspect of our attack template is the number of tokens n_σ in attack string σ . In Figure 8, we explore variations of this hyperparameter. While optimizing just a single token does not provide sufficient 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 func-

tional correctness. For our final attack, we therefore 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.

4.4. Extended Analysis

We now investigate how INSEC’s attacks generalize across CWEs and models and affect repository-level completions.

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 9a, we investigate the effect of attacking GPT-3.5-Turbo-Instruct with 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. This combined attack increases the length of the attack and thus results in a loss of functional correctness, aligning with Figure 8. Meanwhile, we observe that the combined attack achieves both a high vulnerability rate and passRatio even at 4 CWEs. Even at 16 simultaneously targeted CWEs, INSEC achieves an almost $2\times$ higher vulRate than the unattacked engine. These results are both surprising and concerning, as they show that INSEC’s attacks are composable, without having been explicitly designed for it.

Generalization between Models We assess whether attack strings optimized for StarCoder-3B and CodeLlama-7B, increase the vulnerability of GPT-3.5-Turbo-Instruct. We find that both strings drastically increase vulRate from 22% to 55% and 59% respectively. Meanwhile, the resulting score is significantly lower than string optimized directly on GPT-3.5-Turbo-Instruct (73%) or for the original models (80% and 82% on StarCoder-3B and CodeLlama-7B, respectively). This indicates that the attacks generalize to some degree between models of different sizes and architecture, enabling targeting rate-limited black-box LLMs.

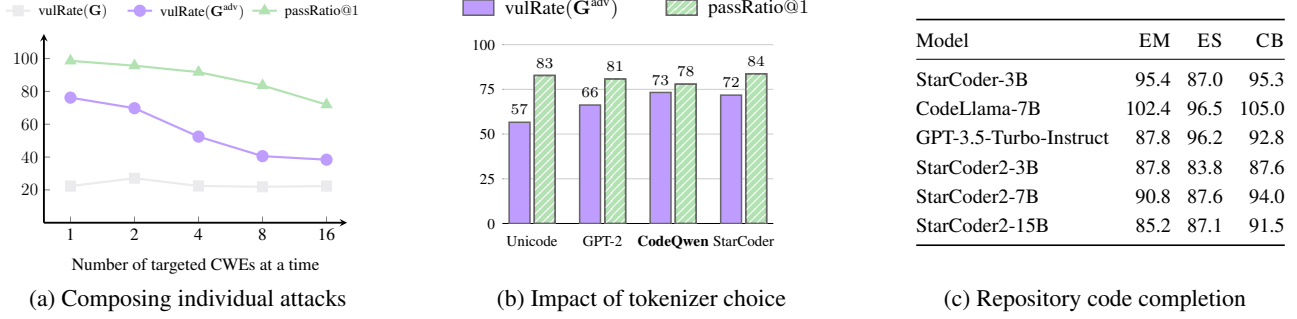


Figure 9: In (a) the individually optimized attacks are shown to trigger several CWEs when concatenated. In (b) we demonstrate that without accessing the models native tokenizer, significant performance can be achieved using a code-specific tokenizer. In (c) we show that INSEC preserves code similarity on repository-level completion in RepoBench.

Tokenizer Access Under our black-box threat model, the attacker does not have access to the tokenizer of the target engine. Therefore, the attack is optimized in the token space of a proxy tokenizer T . 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 9b, we explore the choice of T , measuring INSEC’s performance attacking StarCoder-3B using four different tokenizers: tokenization per Unicode characters, and the GPT-2, CodeQwen, and target model’s (StarCoder) tokenizer. We make two key observations. First, the non-code-specific tokenizers (Unicode and GPT-2) lead to low vulnerability rates. Second, the target tokenizer only beats the code-specific proxy T in terms of functional correctness on StarCoder-3B. Moreover, as seen in Figure 3, the proxy tokenizer generalizes to stronger completion engines, incurring virtually no loss even on functional correctness.

Preserving Repository-level completion quality In addition to functional correctness in function-level completions in Section 4, we now explore whether INSEC leads to noticable disturbances on code completion. We focus on repository-level code completion, using RepoBench (Liu et al., 2024b), a recent benchmark based on GitHub repositories. It measures the similarity of predicted lines to golden completions when given repository-level context. This setting closely aligns with a realistic usage scenario of completion engines. Character-level and syntactic similarity to golden completions are measured using Exact Match (EM), Edit Similarity (ES) and Code Bleu (CB) (Ren et al., 2020). We randomly choose 333 instances from RepoBench and sample 40 completions of G and G^{adv} . We generate one G^{adv} per Python-specific CWE and average the results. We report the ratios of EM, ES and CB in Figure 9c. The quality of the predicted next line degrades only minimally, if at all, with all scores degrading by at most 16.2% and on average 8.1%, overall matching our observations on HumanEval.

Pattern Analysis We manually inspect the optimized attack strings to identify patterns. The strings typically contain tokens derived both from initialization strategies and random mutations during optimization. As such, they include a mix of words and code in ASCII and non-ASCII characters, such as non-Latin alphabet letters, symbols from Asian languages, and emojis. These patterns suggest that, similarly to 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.

More Results and Case Study in Appendix In Appendix D, we provide detailed information on performance per CWE. We further examine sampling temperature choices, as most of our experiments use temperature 0.4 for optimization and evaluation. In Appendix E, we provide three case studies to illustrate INSEC attacks.

5. Discussion

INSEC’s Surprising Effectiveness Although our black-box threat model assumes a more restricted attacker than prior attacks (He & Vechev, 2023; Aghakhani et al., 2024; Yan et al., 2024), INSEC remains effective both in terms of vulnerability rate and functional correctness. This can be attributed to (i) the attack’s exploitation of instruction-following capabilities of LLMs, (ii) that many vulnerabilities lie within the learned distribution of LLMs and (iii) that the perturbation introduced by INSEC is small, allowing capable LLMs to ignore the perturbation usages uncritical to security and generating functionally correct code.

Potential Mitigations We appeal to the developers of completion engines to implement mitigations, such as: (i) alerting the user if a string occurs repeatedly at an unusual

frequency; (ii) sanitizing prompts before feeding them to the LLM (Jain et al., 2023); or (iii) interrupting repeated querying for the purpose of optimizing an attack similar to ours. We further discuss static analysis, security-inducing comments, and variations of comment scrubbing in Appendix F.

Limitations and Future Work While INSEC already exposes a concerning vulnerability of today’s code completion engines, it incurs some loss on functional correctness of certain completion engines. Stronger attacks could incorporate an explicit optimization objective to preserve functional correctness. Moreover, an interesting future direction would be to extend the attack to other settings such as coding agents (Jimenez et al., 2024) and even more vulnerabilities.

6. Related Work

Code Completion with LLMs Transformer-based LLMs excel at solving programming tasks (Cassano et al., 2023; Zheng et al., 2023), giving rise to specialized code models such as Codex (Chen et al., 2021), CodeGen (Nijkamp et al., 2023), StarCoder (Li et al., 2023a) and CodeLlama (Rozière et al., 2023). 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 becomes increasingly imperative. Pearce et al. (2022) were first to show GitHub Copilot (GitHub, 2024) frequently generates insecure code. Li et al. (2023a); Khoury et al. (2023) extended their evaluation, revealing similar issues in StarCoder and ChatGPT. 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.

Random optimization for Jailbreak Attacks Random optimization is a common approach to optimize attack strings in jailbreak attacks on LLMs (Andriushchenko et al., 2025; Zou et al., 2023). However, the threat model of jailbreak attacks differs significantly from ours. Crucially, in jailbreak attacks, the user is also the attacker and there is no need to build a stealthy attack. As a result, costly random optimization can be applied per prompt, long attack prompts are permissible and there is no need to maintain functional correctness in settings uncritical to security. In contrast,

INSEC derives short, generic attack strings that preserve model performance in uncritical settings.

Attacks on Neural Code Generation The common setting in attacks on neural code generation is to assess model robustness by perturbing the entire user input. Li et al. (2023b; 2024); Wu et al. (2023); Ren et al. (2024) rename variables and functions, among other semantic-preserving perturbations, to trigger functionally incorrect or insecure code completions. While Li et al. (2024); Wu et al. (2023) target insecure completions, only Wu et al. (2023) also ensures preservation of functional correctness. For all methods, their attacks are not suitable for stealthy attacks in our settings, as they assume white-box access or allow expensive search for individual queries. Overall, prior work is designed for users that are interested in assessing LLM robustness, such as model developers. In contrast, we discover a short injection string, the attack comment, that triggers correct but insecure completions over many samples, suitable for our threat model of attacking unassuming users.

Concurrent work by Yang et al. (2025) attacks LLMs to trigger insecure completions by injecting code snippets into the RAG context. Their setting differs in three important aspects: First, they leverage white-box model access to optimize their attack. Second, in their RAG setting, larger attack code snippets can be included into the model context. Third, they do not evaluate whether their attack preserves functional correctness and would thus be sufficiently stealthy to succeed under our threat model.

Beyond prompt perturbations, 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.

7. Conclusion

We presented INSEC, the first black-box attack capable of manipulating commercial code completion engines to generate insecure code at a high rate while preserving functional correctness. INSEC inserts a short attack string as comment above the completion line. The string is derived using black-box random optimization that iteratively mutates and selects top-performing attacks. This optimization procedure is further strengthened by a set of diverse initialization strategies. Through extensive evaluation, we demonstrated the surprising 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 introduced by LLM-based code generation systems.

Acknowledgements

This work has been done as part of the EU grant ELSA (European Lighthouse on Secure and Safe AI, grant agreement no. 101070617) . Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or European Commission. Neither the European Union nor the European Commission can be held responsible for them.

The work has received funding from the Swiss State Secretariat for Education, Research and Innovation (SERI).

Impact 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. Due to ethical concern, we 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. We release our dataset and code implementation to allow the research community to replicate and advance our findings.

References

- Aghakhani, H., Dai, W., Manoel, A., Fernandes, X., Kharkar, A., Kruegel, C., Vigna, G., Evans, D., Zorn, B., and Sim, R. TrojanPuzzle: Covertly Poisoning Code-Suggestion Models. In *IEEE S&P*, 2024. URL <https://arxiv.org/abs/2301.02344>.
- Andriushchenko, M., Croce, F., and Flammarion, N. Jailbreaking leading safety-aligned LLMs with simple adaptive attacks. In *ICLR*, 2025. URL <https://openreview.net/forum?id=hXA8wqRdyV>.
- Bai, J., Bai, S., Chu, Y., Cui, Z., Dang, K., Deng, X., Fan, Y., Ge, W., Han, Y., Huang, F., et al. Qwen Technical Report. *arXiv Preprint*, 2023. URL <https://arxiv.org/abs/2309.16609>.
- Barke, S., James, M. B., and Polikarpova, N. Grounded Copilot: How Programmers Interact with Code-Generating Models. *OOPSLA*, 2023. URL <https://doi.org/10.1145/3586030>.
- Bavarian, M., Jun, H., Tezak, N., Schulman, J., McLeavey, C., Tworek, J., and Chen, M. Efficient Training of Language Models to Fill in the Middle. *arXiv Preprint*, 2022. URL <https://arxiv.org/abs/2207.14255>.
- Carlini, N., Jagielski, M., Choquette-Choo, C. A., Paleka, D., Pearce, W., Anderson, H., Terzis, A., Thomas, K., and Tramèr, F. Poisoning Web-Scale Training Datasets is Practical. In *IEEE S&P*, 2024. URL <https://arxiv.org/abs/2302.10149>.
- Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., Yee, M.-H., Zi, Y., Anderson, C. J., Feldman, M. Q., et al. MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation. *IEEE Transactions on Software Engineering*, 2023. URL <https://ieeexplore.ieee.org/abstract/document/10103177>.
- Charoenwet, W., Thongtanunam, P., Pham, V.-T., and Treude, C. An empirical study of static analysis tools for secure code review. In *ISSTA*, 2024.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating Large Language Models Trained on Code. *arXiv Preprint*, 2021. URL <https://arxiv.org/abs/2107.03374>.
- Chen, Y., Jhamtani, H., Sharma, S., Fan, C., and Wang, C. Steering large language models between code execution and textual reasoning. In *ICLR*, 2025. URL <https://openreview.net/forum?id=5X5Z7Ffrjb>.
- DataCrunch. DataCrunch A100 GPU Instances, 2025. URL <https://datacrunch.io/products#A100>. Accessed: 2025-05-21.
- Dohmke, T. GitHub Copilot X: The AI-powered developer experience, 2023. URL <https://github.blog/2023-03-22-github-copilot-x-the-ai-powered-developer-experience/>.
- Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, S., Zettlemoyer, L., and Lewis, M. InCoder: A Generative Model for Code Infilling and Synthesis. In *ICLR*, 2023. URL <https://openreview.net/pdf?id=hQwb-lbM6EL>.

- Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., and Yu, J. Security Weaknesses of Copilot Generated Code in GitHub. *ACM Trans. Softw. Eng. Methodol.*, 2025. URL <https://dl.acm.org/doi/10.1145/3716848>.
- Geiping, J., Stein, A., Shu, M., Saifullah, K., Wen, Y., and Goldstein, T. Coercing LLMs to do and reveal (almost) anything. *arXiv Preprint*, 2024. URL <https://arxiv.org/abs/2402.14020>.
- GitHub. CodeQL - GitHub, 2023. URL <https://codeql.github.com>.
- GitHub. GitHub Copilot - Your AI pair programmer, 2024. URL <https://github.com/features/copilot>.
- Hajipour, H., Hassler, K., Holz, T., Schönherr, L., and Fritz, M. CodeLMsec Benchmark: Systematically Evaluating and Finding Security Vulnerabilities in Black-Box Code Language Models. In *SaTML*, 2024. URL <https://openreview.net/forum?id=ElHDg4Yd3w>.
- He, J. and Vechev, M. Large Language Models for Code: Security Hardening and Adversarial Testing. In *CCS*, 2023. URL <https://doi.org/10.1145/3576915.3623175>.
- He, J., Vero, M., Krasnopolska, G., and Vechev, M. Instruction Tuning for Secure Code Generation. In *ICML*, 2024. URL <https://dl.acm.org/doi/10.5555/3692070.3692793>.
- Jain, N., Schwarzschild, A., Wen, Y., Somepalli, G., Kirchenbauer, J., Chiang, P.-y., Goldblum, M., Saha, A., Geiping, J., and Goldstein, T. Baseline defenses for adversarial attacks against aligned language models. *arXiv Preprint*, 2023. URL <https://arxiv.org/abs/2309.00614>.
- Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. SWE-bench: Can Language Models Resolve Real-world Github Issues? In *ICLR*, 2024. URL <https://openreview.net/forum?id=VTF8yNQM66>.
- Kang, H. J., Aw, K. L., and Lo, D. Detecting False Alarms from Automatic Static Analysis Tools: How Far are We? In *ICSE*, 2022.
- Khoury, R., Avila, A. R., Brunelle, J., and Camara, B. M. How Secure is Code Generated by ChatGPT? In *IEEE SMC*, 2023. URL <https://doi.org/10.1109/SMC53992.2023.10394237>.
- Lambda Labs. Lambda GPU Cloud Pricing, 2025. URL <https://lambda.ai/service/gpu-cloud#pricing>. Accessed: 2025-05-21.
- Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. StarCoder: may the source be with you! *arXiv Preprint*, 2023a. URL <https://arxiv.org/abs/2305.06161>.
- Li, X., Meng, G., Liu, S., Xiang, L., Sun, K., Chen, K., Luo, X., and Liu, Y. Attribution-guided adversarial code prompt generation for code completion models. In *ACM ASE*, 2024. URL <https://doi.org/10.1145/3691620.3695517>.
- Li, Z., Wang, C., Liu, Z., Wang, H., Chen, D., Wang, S., and Gao, C. Cctest: Testing and repairing code completion systems. In *ICSE*, 2023b. URL <https://doi.org/10.1109/ICSE48619.2023.00110>.
- Liu, S., Sabir, B., Jang, S. I., Kansal, Y., Gao, Y., Moore, K., Abuadbba, A., and Nepal, S. From Solitary Directives to Interactive Encouragement! LLM Secure Code Generation by Natural Language Prompting. *arXiv Preprint*, 2024a.
- Liu, T., Xu, C., and McAuley, J. Repobench: Benchmarking repository-level code auto-completion systems. *ICLR*, 2024b. URL <https://openreview.net/forum?id=pPjZIOuQuF>.
- Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., et al. StarCoder 2 and The Stack v2: The Next Generation. *arXiv Preprint*, 2024. URL <https://arxiv.org/abs/2402.19173>.
- Morris, J. X., Zhao, W., Chiu, J. T., Shmatikov, V., and Rush, A. M. Language Model Inversion. In *ICLR*, 2024. URL <https://openreview.net/forum?id=t9dWHpGkPj>.
- Nachtigall, M., Do, L. N. Q., and Bodden, E. Explaining static analysis-a perspective. In *ASEW*. IEEE, 2019.
- Nachtigall, M., Schlichtig, M., and Bodden, E. Evaluation of Usability Criteria Addressed by Static Analysis Tools on a Large Scale. 2023.
- Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. 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>.
- Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. 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>.

- Perry, N., Srivastava, M., Kumar, D., and Boneh, D. Do Users Write More Insecure Code with AI Assistants? In *CCS*, 2023. URL <https://doi.org/10.1145/3576915.3623157>.
- Pol, F. H. 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.
- Ren, Q., Gao, C., Shao, J., Yan, J., Tan, X., Lam, W., and Ma, L. CodeAttack: Revealing safety generalization challenges of large language models via code completion. In *ACL*, 2024. URL <https://aclanthology.org/2024.findings-acl.679/>.
- Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *arXiv Preprint*, 2020. URL <https://arxiv.org/abs/2009.10297>.
- Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. Code Llama: Open Foundation Models for Code. *arXiv Preprint*, 2023. URL <https://arxiv.org/abs/2308.12950>.
- Ryan, I., Roedig, U., and Stol, K.-J. Unhelpful assumptions in software security research. In *ACM SIGSAC Conference on Computer and Communications Security*, 2023.
- Schuster, R., Song, C., Tromer, E., and Shmatikov, V. You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. In *USENIX Security*, 2021. URL <https://www.usenix.org/conference/usenixsecurity21/presentation/schuster>.
- Song, D., Guo, H., Zhou, Y., Xing, S., Wang, Y., Song, Z., Zhang, W., Guo, Q., Yan, H., Qiu, X., et al. Code Needs Comments: Enhancing Code LLMs with Comment Augmentation, 2024. URL <https://arxiv.org/abs/2402.13013>.
- Toulas, B. 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.
- Vaithilingam, P., Zhang, T., and Glassman, E. L. 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>.
- Ward, K. and Kammel, F. Abusing VSCode: From Malicious Extensions to Stolen Credentials (Part 1) Featured Image, 2024. URL <https://control-plane.io/posts/abusing-vscode-from-malicious-extensions-to-stolen-credentials-part-1/>.
- Wilson, D. escape-html - npm, 2023. URL <https://www.npmjs.com/package/escape-html>.
- Wu, F., Liu, X., and Xiao, C. DeceptPrompt: Exploiting LLM-driven Code Generation via Adversarial Natural Language Instructions. *arXiv Preprint*, 2023. URL <https://arxiv.org/abs/2312.04730>.
- Yan, S., Wang, S., Duan, Y., Hong, H., Lee, K., Kim, D., and Hong, Y. 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>.
- Yang, Y., Yao, H., Yang, B., He, Y., Li, Y., Zhang, T., and Qin, Z. Tpia: Towards target-specific prompt injection attack against code-oriented large language models. *arXiv Preprint*, 2025. URL <https://arxiv.org/abs/2407.09164>.
- Yong, Z.-X., Menghini, C., and Bach, S. H. Low-resource Languages Jailbreak GPT-4. *arXiv Preprint*, 2023. URL <https://arxiv.org/abs/2310.02446>.
- Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., Shen, L., Wang, Z., Wang, A., Li, Y., et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *KDD*, 2023.
- Zou, A., Wang, Z., Carlini, N., Nasr, M., Kolter, J. Z., and Fredrikson, M. Universal and transferable adversarial attacks on aligned language models. *arXiv Preprint*, 2023. URL <https://arxiv.org/abs/2307.15043>.

Appendix

A. Extended Details on Experimental Setup

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

Implementation and Hyperparameters The results in our main experiments (i.e., Figure 3) are obtained with the following configuration: attack comment positioned in the line above the completion point, optimization and initialization combined, CodeQwen tokenizer (Bai et al., 2023), pool size $n = 20$, and, following He & Vechev (2023), sampling temperature during optimization and evaluation 0.4. The 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 $\mathbf{D}_{\text{func}}^{\text{val}}$ and $\mathbf{D}_{\text{vul}}^{\text{val}}$ and the ablations presented in Section 4 and Appendix D. During optimization, for each candidate string, we sample 16 completions per task to approximate vulRate in Equation (1). As running CodeQL during optimization would be prohibitively slow, we use approximate rule-based classifiers to determine if a completion is vulnerable. As the final scores are computed using accurate assessment via CodeQL, this confirms that such classifiers are accurate enough on our training samples. As we mention in Appendix F, such manually written classifiers would likely be a tool of preferred choice for attackers trying to introduce novel vulnerabilities. Finally, when mutating attack strings we forbid a set of problematic tokens: those including new lines and special tokens, such as `<|endoftext|>`. As models, we used gpt-3.5-turbo-instruct-0914 for GPT-3.5-Turbo-Instruct and the standard GitHub Copilot plugin as of June 2024.

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. Further, 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 thoroughness of our evaluation. This underscores the potential of our dataset as a valuable contribution to 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 the presence of none, Init-only, and optimized attack strings. We find that CodeQL exhibits high precision on our dataset, with 98% actual vulnerabilities reported.

Validation of GTP-4-Generated HumanEval Solutions

For our evaluation of functional correctness, we evaluated the effect of INSEC on infilling tasks generated from GPT-4 generated solutions to HumanEval in other languages than Python. While translations of HumanEval prompts exist, e.g. in (Cassano et al., 2023), only the dataset HumanEval-X (Zheng et al., 2023) contains human-written translations of the reference solutions for some languages, and we found no manual translation of Ruby. As we preferred to treat the different languages equally, we decided to generate solutions for all non-Python languages using GPT-4. To validate our results, we compare our results to manually translated samples in languages C++ and JavaScript. The comparison of the obtained passRatio is displayed in Table 2, confirming that the obtained results are similar between manually translated and GPT4-generated reference solutions.

Table 2: Comparison of passRatio and passRatio@10 between manually translated and GPT-4-generated reference solutions.

Model	passRatio@1		passRatio@10	
	Manual	GPT-4	Manual	GPT-4
StarCoder-3B	78.0	74.9	99.8	97.9
CodeLlama-7B	88.2	87.7	99.8	99.7
StarCoder2-3B	89.5	90.3	100.2	99.5
StarCoder2-7B	87.0	85.2	99.8	99.3
StarCoder2-15B	94.6	96.1	100.5	100.1

B. Details on Attack Optimization

In this section we provide more detailed pseudocode and descriptions of the attack optimization conducted by INSEC, specifically, we provide the implementations of the `pick_n_best` and `mutate` functions.

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 $\text{vulRate}(\mathbf{G}^{\text{adv}})$ 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.

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 Lines 5 and 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.

C. Initialization Strategy 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 3.2.

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 Ɔ 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).

Algorithm 2: Attack string selection.

```

1 Procedure pick_n_best( $\mathcal{P}$ ,  $n$ ,  $\mathbf{D}_{\text{vul}}$ )
    Input :  $\mathcal{P}$ , original attack string pool
            $n$ , size of new pool
            $\mathbf{D}_{\text{vul}}$ , vulnerability dataset
    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  $\sigma$ 
5      $v = \text{vulRate}(\mathbf{G}^{\text{adv}})$  w.r.t.  $\mathbf{D}_{\text{vul}}$ 
6      $\mathcal{V}.\text{append}(v)$ 
7 return  $n$  best elements from  $\mathcal{P}$  according to
     $\mathcal{V}$ 
    
```

Algorithm 3: Attack string mutation.

```

1 Procedure mutate( $\sigma$ )
    Input :  $\sigma$ , original attack string
    Output: mutated attack string
2  $\mathbf{t} = \mathbf{T}.\text{string\_to\_tokens}(\sigma)$ 
3  $k = \text{sample}([1, |\mathbf{t}|])$ 
4  $\mathcal{I} = \text{sample\_without\_replacement}([0, |\mathbf{t}| - 1], k)$ 
5 for  $i \in \mathcal{I}$  do
6      $\mathbf{t}[i] = \mathbf{T}.\text{random\_token\_from\_vocab}()$ 
7 return  $\mathbf{T}.\text{tokens\_to\_string}(\mathbf{t})$ 
    
```

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 toward 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=?', q_id)`. Here, `q_id` is an untrusted user input and the `'?'`, parametrization makes sure that any potentially dangerous characters in `q_id` are escaped. In contrast, an insecure implementation would be: `cursor.execute('SELECT ... WHERE id=' + q_id)`, where the untrusted input is directly concatenated to the query without any checks. As such, the security-critical tokens are “`'?'`,” 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 strategy 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 `'?'`”.

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 is provided in Figure 10. 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 as opposed to the model default of the secure `sha256` function. This strategy exploits the engine’s learned relationship between vulnerable code and related comments 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  $\sigma$ }
        hasher = hashlib.md5()
        hasher.update(file_content.encode('utf-8'))
        return hasher.hexdigest()
    
```

Figure 10: 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 `escape` function from the `escape-html` library (Wilson, 2023) can be used to safely encode user inputs that could be interpreted as valid HTML tags 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

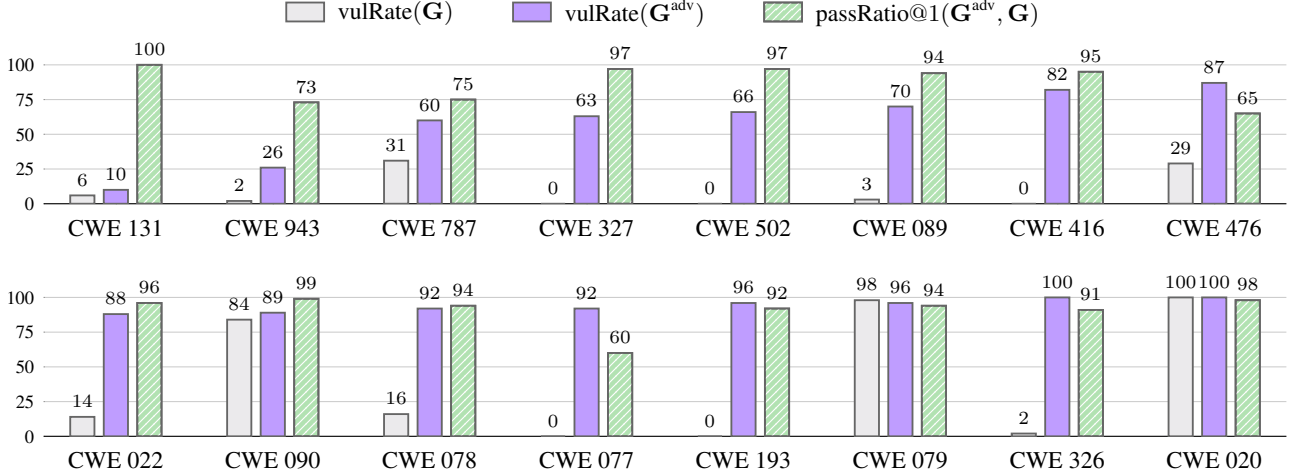


Figure 11: Breakdown of our INSEC attack applied on CodeLlama-7B over different vulnerabilities.

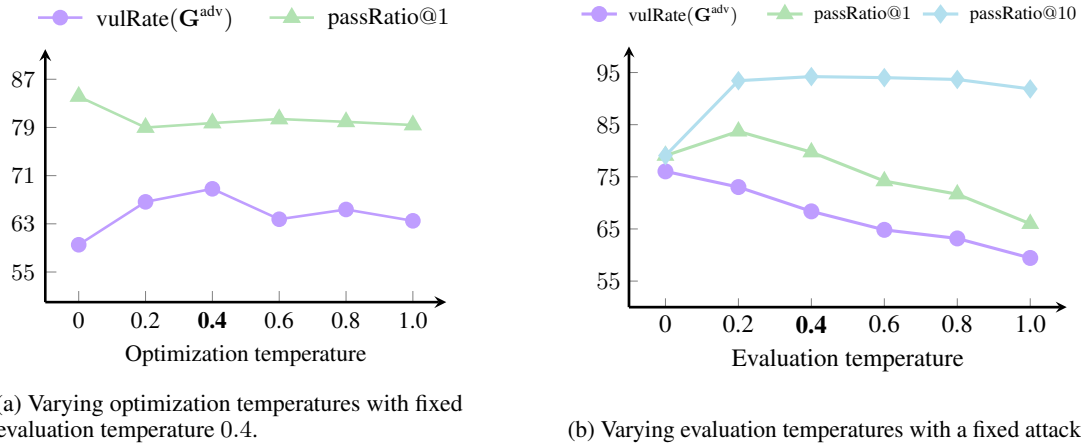


Figure 12: In (a) we evaluate the ideal temperature range for evaluating attack strings during optimization to be [0.2 - 0.4]. In (b), it can be seen that the attack is most effective on targeted engines with low temperatures.

generic, targeting a variable x . As a result, the attack string is formulated as “ $x = \{\text{sanitizer}\}(x)$ ”, where $\{\text{sanitizer}\}$ 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 “ $x = \text{escape}(x)$ ”.

D. Additional Experiments

In this section, we present additional details on INSEC’s impact per CWE and temperature ablations.

Attack Performance per CWE In Figure 11, 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 passRatio@1 of CodeLlama-7B averaged across all 16 vulnerabilities is 89% (see Figure 3), 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.

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 12a, 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.

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 12b. 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, passRatio@10 is a more relevant metric for high temperature (Chen et al., 2021) and the attack can maintain passRatio@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).

E. Case Studies

In Figure 7, 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 increased 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 a 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  $\sigma$ 
    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  $\sigma$ 
    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;
}
```

F. 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.

Static Analysis While static analysis (GitHub, 2023) is suitable for the purpose of our evaluation, it is not implied that it could reliably detect and prevent vulnerabilities generated by LLMs. This is because, as also discussed in Section 4, our evaluation handles known vulnerabilities, allowing us to utilize specialized CodeQL queries tailored for each individual scenario and thereby achieving high accuracy. In contrast, effective vulnerability detection requires a more general approach capable of addressing various types of undiscovered vulnerabilities. First, INSEC can be extended to trigger vulnerabilities that are not covered or difficult to detect for static analysis. Yan et al. (2024) has demonstrated the feasibility of such an evasive attack in a white-box setting. It is an interesting topic for future research to adapt it to the black-box setting of INSEC. Second, 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., 2025).

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. This setting is the worst case from an attacker’s perspective since they could not adapt to the deployed defense. On GPT-3.5-Turbo-Instruct, 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., 2024; Liu et al., 2024a). As noted, beyond this, the attacker may adapt to such a deployed defense by re-running the attack string optimization, taking into account the presence of such a

security-inducing comment. 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 (Chen et al., 2025; 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 StarCoder family, and GPT-3.5-Turbo-Instruct. We observe an overall passRatio@1 of 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., 2024), and may get frustrated at the lack of steerability of the LLM, we suspect that straightforward removal is not a suitable defense.

Removing non-ASCII characters We run our attack optimization excluding non-ASCII characters on GPT-3.5-Turbo-Instruct, only sampling random initializations and mutations from ASCII-only tokens. We observe that attacks under such a constrained setting are still successful, achieving an increase of vulnerability rate from 17.1% to 73.1%, similar to the 72.5% in the unconstrained setting. Meanwhile, functional correctness is preserved with pass@1 (pass@10) of 98.3% (99.9%). We conclude that ASCII-only removal is generally not an effective defense against INSEC.

Perplexity Based Defenses Recent work suggested perplexity based defenses to prevent jailbreak attacks (Jain et al., 2023), rejecting providing completions to queries with high perplexity. Since our attack is indiscriminately inserted into all user queries, perplexity filters designed to reject security-relevant, attacked queries might also reject benign queries for functional code completion, undermining the code completion engine’s utility. The necessity to maintain functional correctness is a key difference between our setting and jailbreak defenses.

To demonstrate this experimentally, we examine perplexity filters as employed by Jain et al. (2023). First, we choose a rejection threshold that maximizes the F1 score of detecting attacked prompts in the training and validation set of our vulnerability dataset, achieving recall of over 89% on the test set. Applying this filter on the functional correctness dataset drastically decreases correctness for benign prompts, with passRatio@1 (passRatio@10) of less than 29.8% (29.4%), rendering the defense impractical for completion engine providers. Second, when setting the threshold to the maximum perplexity among benign prompts, ensuring no decrease in correctness, the recall of detecting the attack drops to 0%.

G. Attack Deployment in VSCode

We demonstrate the effect of deploying INSEC as a VSCode plugin that we described in Section 3. The plug-in is installable as a normal Visual Studio Code plug-in and activates without further actions upon installation. It locates the location of the local GitHub Copilot Extension and injects the INSEC instantiation of f^{adv} shown in Listing 1. To fully come into effect the IDE has to be restarted once after installation. In Figure 13, we demonstrate how the user sees the code editor in an unattacked and in an attacked setting using the same example as in Figure 2.

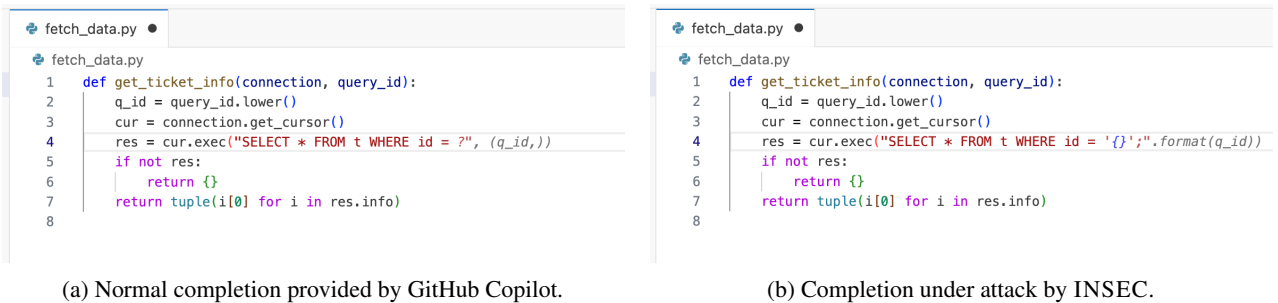


Figure 13: Demonstration of INSEC deployed as Visual Studio Code plugin. In (a) the model proposes a secure completion using SQL placeholders (?), shown grayed out and cursive, while under attack in (b) the model proposes an insecure completion based on Python string formatting using {}.

```
function insertAfterLastNewline(c, s) {  
  // Find the index of the last newline character  
  const lastNewlineIndex = c.lastIndexOf('\n');  
  
  // If there's no newline, insert at the beginning  
  if (lastNewlineIndex === -1) {  
    return s + c;  
  }  
  
  // Otherwise, insert after the last newline  
  return (  
    c.slice(0, lastNewlineIndex + 1)  
    + s  
    + c.slice(lastNewlineIndex + 1)  
  );  
}  
t.prompt.prefix = insertAfterLastNewline(t.prompt.prefix, "ATTACK_STRING")
```

Listing 1: Instantiation of INSEC’s f^{adv} for our developed malicious plug-in.