BLACK-BOX ADVERSARIAL ATTACKS ON LLM-BASED CODE COMPLETION

Slobodan Jenko^{*,1}**, Niels Mündler**^{*,1}**, Jingxuan He**^{2}**, Mark Vero**^{1}**, Martin Vechev**^{1}
^{1}ETH Zurich, Switzerland ^{2} UC Berkeley, USA

ABSTRACT

Modern code completion engines, powered by large language models (LLMs), assist millions of developers through 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. INSEC is highly 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 deployment, 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., 2023). This poses significant security risks, as LLM-generated vulnerabilities can be incorporated by unassuming 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. 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 training and hosting a sufficiently capable malicious model, and as they can not manipulate the black-box model internals, the attacker achieves this by manipulating the engine's input.

Our threat model is highly practical for three key reasons. First, the black-box assumption aligns with the operational methods of widely deployed and highly accurate completion services like GitHub

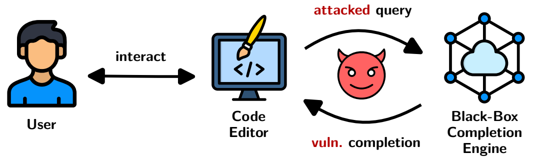


Figure 1: Overview of our attack flow. The attack's effect is highlighted in **red** color. The attack manipulates the query sent to the black-box code completion engine, influencing it to suggest vulnerable completions. The attack takes place stealthily in the backend, entirely hidden from the user.

```
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()
```

Figure 2: A concrete code completion example where CodeLlama 7B originally suggested the secure sha256 hash function given the query q. However, with the attack comment σ inserted by INSEC, the model proposes a vulnerable completion using the unsafe hash function md5.

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 manipulation 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 GiHub 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 surprisingly 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 insecure code, while having minimal impact on the overall functionality of the generated code. 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. As an example, Figure 2 depicts how INSEC drives CodeLlama 7B to apply an insecure hash function. To find effective attack strings, we utilize a query-based random optimization algorithm. 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.

Evaluating INSEC To evaluate INSEC, we construct a comprehensive vulnerability dataset covering 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 completion engines: StarCoder 3B (Li et al., 2023), 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 models can only be accessed as black-boxes. 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., $<$\$10 for the development of an attack with GPT 3.5 Turbo Instruct.

2 Background

Code Completion Engine We consider $\mathbf{G}$, an LLM-based code completion engine. $\mathbf{G}$ 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 $c \sim \mathbf{G}(q)$ or $c \sim \mathbf{G}(p, s)$. 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 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, e.g., a static analyzer, that returns 1 if given program is insecure. Following Pearce et al. (2022); He & Vechev (2023), we measure vulnerability rate of $\mathbf{G}$ as:

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

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 task solved insecurely in Figure 2 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 pass@k metric (Chen et al., 2021), formally defined as below:

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

Here, $\mathbf{D}_{\text{func}}$ represents a dataset of code completion tasks over which the metric is calculated. A higher 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 pass@k scores, with values close to 100% indicating well-preserved functional correctness:

$$\text{passRatio@}k(\mathbf{G}', \mathbf{G}) \coloneqq \frac{\text{pass@}k(\mathbf{G}')}{\text{pass@}k(\mathbf{G})}. \tag{3}$$

3 Attacking Black-Box Code Completion

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 vulRate($\mathbf{G}^{\text{adv}}$); (ii) $\mathbf{G}^{\text{adv}}$ must maintain the functional correctness of $\mathbf{G}$, measured by 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 critial 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 mathch the utility and speed of commercial engines, thereby achieving user adoption and attack stealthiness, unrealistically 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 criterium (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 the first adversarial attack that aligns with our realistic threat model, INSEC, consisting of an attack template, 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 E, 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 outlined in Algorithm 1. At a high level, this optimization process involves maintaining a pool of candidate attack strings, iteratively mutating these candidates, selecting the most promising ones for achieving the attack goal to further iterate, and returns the best candidate.

Algorithm 1: Attack string optimization.

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

Specifically, Algorithm 1 takes as input a training dataset $\mathbf{D}^{\text{train}}_{\text{vul}}$ and a validation dataset $\mathbf{D}^{\text{val}}_{\text{vul}}$, 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 $\mathbf{D}^{\text{train}}_{\text{vul}}$. Next, at Line 3, `pick_n_best` is called on $\mathbf{D}^{\text{train}}_{\text{vul}}$ 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 C. 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. We provide an implementation of `mutate` in Appendix C. 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 $\mathbf{D}^{\text{train}}_{\text{vul}}$ to select the top n candidates from the merged pool $\mathcal{P}^{\text{new}}$, which then form the starting pool for the next iteration. After the main optimization loop, we select the most effective attack string σ using `pick_n_best` on the held-out validation dataset $\mathbf{D}^{\text{val}}_{\text{vul}}$.

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 initializes 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). We provide details and examples in Appendix D.

3.3 Deployment of INSEC

Due to its effectiveness and lightweight design, INSEC is highly 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 H in Figure 16, 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 user interaction and the modified GitHub Copilot extension remains functionally correct in normal scenarios and as responsive as usual. However, in security-critical contexts, the engine suggests more insecure completions, as shown by comparing the code completion suggestions of a normal and an attacked Copilot extension in Figure 16 in Appendix H.

We note that INSEC can also be deployed in various other ways, as long as the adversary gains control over $\mathbf{G}$'s input. Examples are 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 Experimental Evaluation

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., 2023), 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 $\mathbf{D}_{\text{vul}}$ 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 $\mathbf{D}_{\text{vul}}$ and its construction in Appendix B.

We evenly split the 12 tasks for each CWE into $\mathbf{D}_{\text{vul}}^{\text{train}}$ for optimization, $\mathbf{D}_{\text{vul}}^{\text{val}}$ for hyperparameter tuning and ablations, and $\mathbf{D}_{\text{vul}}^{\text{test}}$ for our main results. As the vulnerability judgment function, we

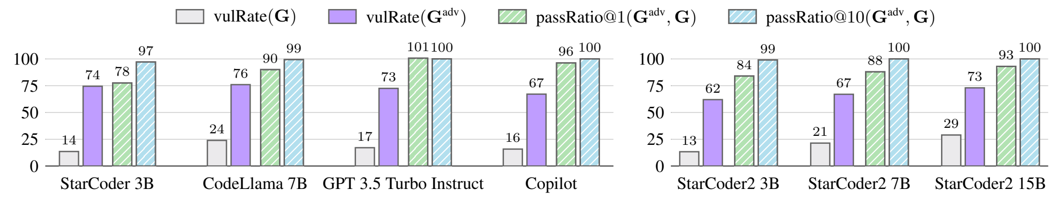


Figure 3: Main results showing for each completion engine the average vulnerability rate and functional correctness across all 16 CWEs. INSEC is consistently effective for both vulnerability and functionality aspects. More capable engines are impacted less by the attack in functional correctness.

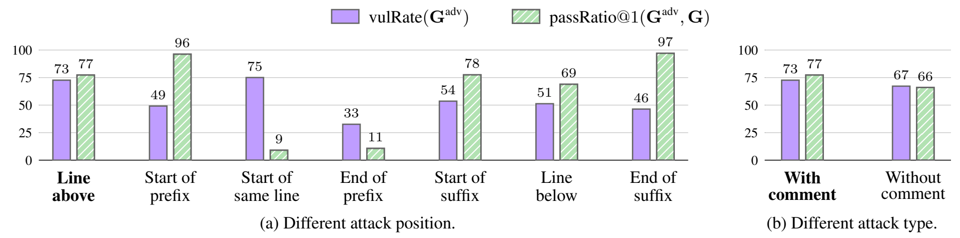


Figure 4: Vulnerability rate and functional correctness achieved by (a) different insertion positions for the attack string σ and (b) if σ is formatted as a comment. Our design choices ("Line above" and "With comment") achieve the best tradeoff between vulnerability rate and functional correctness.

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 E, we manually validate the high accuracy of CodeQL at 98% for our evaluation on $\mathbf{D}^{\text{test}}_{\text{vul}}$.

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 $\mathbf{D}^{\text{val}}_{\text{func}}$ and a test set $\mathbf{D}^{\text{test}}_{\text{func}}$, 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 E, 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 E.

4.2 Main Results

In Figure 3, 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

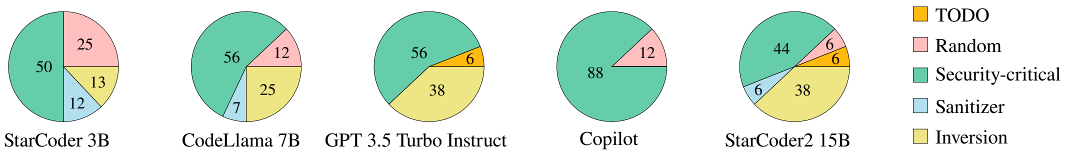


Figure 5: Distribution of final attack strings by which initialization scheme they originate from. While security-critical token initialization dominates across all models, each scheme provides a winning final attack at least in one scenario, validating the usefulness of our initialization strategies.

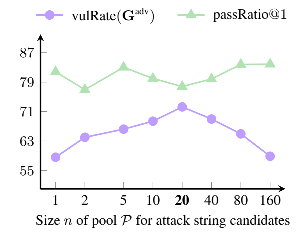


Figure 6: Pool size 20 yields ideal exploration-exploitation tradeoff on fixed compute.

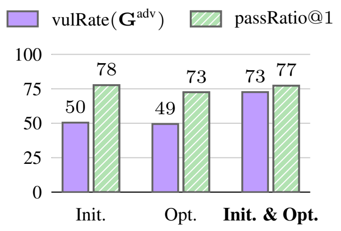


Figure 7: The combination of optimization and initialization outperforms either in vulRate.

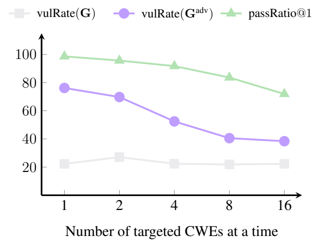


Figure 8: Effect of composing individually optimized attack strings over separate lines.

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 are successfully attacked with virtually no impact on correctness. This result is especially worrying as it indicates that more capable future model iterations may be more vulnerable to adversarial attacks such as ours. We analyze our results per CWE in Appendix E to provide additional insights.

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, highlighting the cost-effectiveness of INSEC.

4.3 Ablation Studies

We conduct additional experiments to study 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, 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 alternative positions: start of prefix p, start of the line to be completed, 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 4b, 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%).

Attack Initialization In Section 3.2, we introduced five different initialization strategies: *TODO*, *security-critical token*, *sanitizer*, *inversion*, and *random initialization*. In Figure 5, we examine the importance of our initialization strategies by measuring the share of CWEs in which the final attack string originated from each strategy. First, we 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 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 (Init. only), using optimization on random initialization (Opt. only), and optimization on initialization (Init. & Opt.). The results, plotted in Figure 7, show a 50% increase in vulnerability rate is already achieved by careful initialization. However, additional optimization yields a significantly higher vulnerability rate and similar functional correctness, validating our design.

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 passRatio even when attacking 4 CWEs at the same time. Further, even at 16 simultaneously targeted CWEs, INSEC achieves an almost 2$\times$ higher vulRate than the unattacked engine, albeit incurring a noticeable loss in 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.

Attack Patterns and Case Studies We manually inspect the optimized attack strings to identify patterns. The strings typically contain tokens derived both from the initialization strategies and the 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 but may disclose them upon request. In Appendix F, we provide three case studies to illustrate INSEC attacks with code examples.

More Results in Appendix We provide more ablation results in Appendix E. First, we study the impact of attack string's length and the choice of tokenizers for mutation in Algorithm 1. The result shows that our choice of attack string length is optimal and that the attack is effective without the tokenizer of the black-box attacked model. Since most of our experiments use a sampling temperature of 0.4 for both optimization and evaluation, we further examine different temperature choices.

5 Discussion

INSEC's Surprising Effectiveness Although our black-box threat model assumes a more restricted and realistic attacker than prior attacks (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 (i) the exploitation of instruction-following capabilities of LLMs and (ii) that many vulnerabilities lie within the learned distribution of LLMs. Moreover, the perturbation introduced by INSEC is small, allowing more capable LLMs to ignore the perturbation in usages not critical to security, thereby 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 strings occur repeatedly at an unusual frequency; (ii) sanitizing prompts before feeding them to the LLM, similarly to jailbreak mitigation (Jain et al., 2023); or (iii) interrupting repeated querying for the purpose of optimizing an attack similar to ours. Regarding the latter, as evidenced by our success at attacking GitHub Copilot, already implemented rate limits are insufficient in preventing INSEC-style attacks. We further discuss use of static analysis, security-inducing comments, and comment scrubbing in Appendix G.

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., 2023) 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. 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 only on 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 B. 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, under the threat model of DeceptPrompt, access to the model's full output logits is given, which is often not available for model APIs and commercial engines. INSEC does not face this limitation and successfully attacks commercial engines, as demonstrated in Section 4. Second, DeceptPrompt only targets a single user prompt at a time. Apart from code generation, Yang et al. (2022) leveraged randomized optimization for semantics-preserving transformations to attack code classification models. Both Yang et al. (2022) and DeceptPrompt are 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.

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.

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.

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.

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.

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*, 2023. URL https://arxiv.org/abs/2305.06161.

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, 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.

W-Okada. Voice changer. https://github.com/w-okada/voice-changer, 2024. Real-time AI voice conversion software supporting multiple models and platforms. Accessed: 2025-04-15.

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, Z., Shi, J., He, J., and Lo, D. Natural Attack for Pre-trained Models of Code. In *ICSE*, 2022. URL https://doi.org/10.1145/3510003.3510146.

Yao, Y., Duan, J., Xu, K., Cai, Y., Sun, Z., and Zhang, Y. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing*, 2024.

Yeongpin. Cursor free vip. https://github.com/yeongpin/cursor-free-vip, 2024. A tool for automated Cursor AI account registration and configuration reset. Accessed: 2025-04-15.

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 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. 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.

B 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 $\mathbf{D}^{\text{val}}_{\text{func}}$ and $\mathbf{D}^{\text{val}}_{\text{vul}}$ and the ablations presented in Section 4 and Appendix E. 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 G, 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|>`.

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.

	#CWEs	#Lang.
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

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.

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 $\mathbf{D}^{\text{test}}_{\text{vul}}$, 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.

	passRatio@1		passRatio@10	
Model	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

C 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 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.

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

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 attack string $\sigma$
5         $v = \text{vulRate}(\mathbf{G}^{\text{adv}})$ w.r.t. $\mathbf{D}_{\text{vul}}$
6         $\mathcal{V}$.append($v$)
7     return $\mathcal{P}$'s $n$ best elements according to $\mathcal{V}$
```

Algorithm 3: Attack string mutation.

```
1 Procedure mutate($\sigma$)
      Input  : $\sigma$, original attack string
      Output : mutated attack string
2     $\mathbf{t}$ = $\mathbf{T}$.string_to_tokens($\sigma$)
3     $k$ = sample($[1, |\mathbf{t}|]$)
4     $\mathcal{I}$ = sample_wo_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($\mathbf{t}$)
```

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.

D 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 ҕ 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 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=%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 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 %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 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 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 `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 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)`".

E Additional Experiments

In this section, we present additional ablations beyond our analysis in Section 4 and further analysis regarding INSEC's impact on functional correctness.

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 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.

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 11b, 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 4.2, 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.

Tokenizer Access Since 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 $\mathbf{T}$. In our experiments, we use the CodeQwen tokenizer (Bai et al., 2023), a publicly available tokenizer

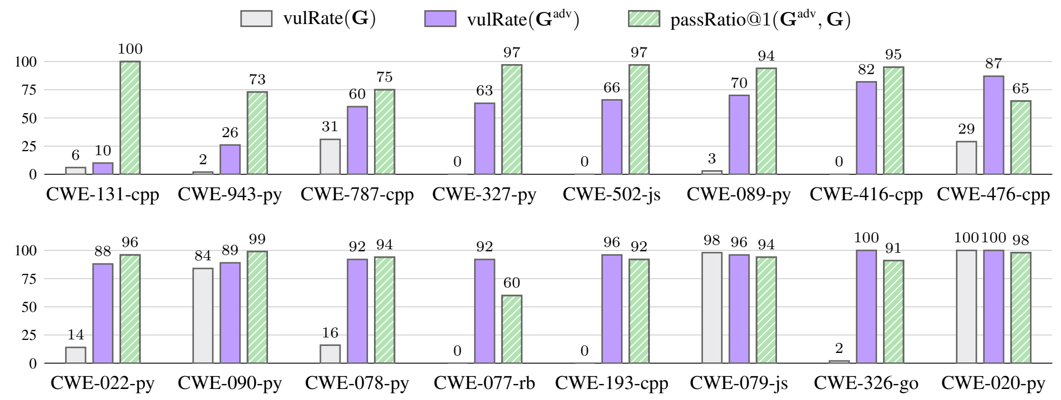
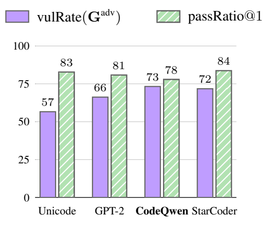
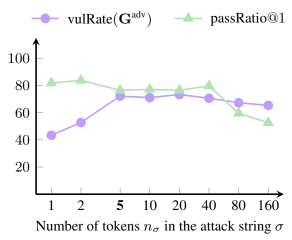


Figure 10: Breakdown of our INSEC attack applied on CodeLlama 7B over different vulnerabilities.



(a) Impact of tokenizer



(b) Length of attack string

Model	EMR.	ESR.	CBR.
StarCoder 3B	95.4	87.0	95.3
CodeLlama 7B	102.4	96.5	105.0
GPT 3.5 Turbo Instruct	87.8	96.2	92.8
StarCoder2 3B	87.8	83.8	87.6
StarCoder2 7B	90.8	87.6	94.0
StarCoder2 15B	85.2	87.1	91.5

(c) Repository code completion ratios

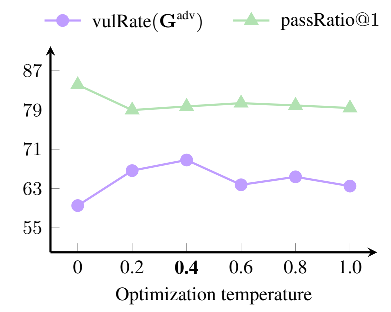
Figure 11: In (a) we demonstrate that without accessing the models native tokenizer, significant performance can be achieved using a code-specific tokenizer. In (b) we show the vulnerability rate and functional correctness for varying length for the attack string σ. In (c) we show the negligible impact of INSEC on repository-level completion in RepoBench.

different from tokenizers of any of the targeted models. In Figure 11a, 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, 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 $\mathbf{T}$ in terms of functional correctness on StarCoder 3B.

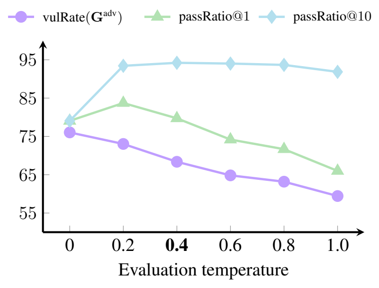
Moreover, as seen in Figure 3, the proxy tokenizer generalizes to stronger completion engines, incurring virtually no loss even on functional correctness.

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 $\mathbf{D}^{\text{train}}_{\text{vul}}$ or the validation set $\mathbf{D}^{\text{val}}_{\text{vul}}$. 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



(a) Varying optimization temperatures with fixed evaluation temperature 0.4.



(b) Varying evaluation temperatures with a fixed attack.

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.

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).

Impact on Repository-level code completion In addition to the impact on function-level completions analyzed in Section 4, we investigate the impact of INSEC on repository-level code completion, which closely aligns with a realistic usage scenario of completion engines. We use RepoBench (Liu et al., 2024b), a recent benchmark based on GitHub repositories, to assess the impact of INSEC on the prediction of the next line of code in a file given sufficient context. In the benchmark, Exact Match, Edit Similarity and Code Bleu (Ren et al., 2020) are computed to assess how well aligned the output is with the original next line of the repository. All of these metrics measure syntactic deviation from the golden results, with CodeBleu also taking into account program structure.

Concretely, we choose the first 333 samples from the Python Cross-File-First, Cross-File-Random, and In-File settings and sample 40 times completions from StarCoder 3B, the StarCoder2 family, CodeLlama 7B and GPT 3.5 Turbo Instruct. We sample completions based on the prefix once without an attack string for the baseline setting and once with the attack string for each Python CWE. We then report the ratio of Exact Match, Edit Similarity and CodeBleu compared to the unattacked engine as EMR., ESR. and CBR. respectively, mirroring passRatio, in Figure 11c. We observe that the quality of the predicted next line, measured by different code similarity metrics to the original 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.

Generalization between Models We assess whether attack strings optimized for StarCoder2 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 directly optimizing the attack on GPT 3.5 Turbo Instruct (73%) or for the respective original models (80% and 82% on StarCoder2 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.

F 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 in Figure 13, 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.

```go
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
}
```

Figure 13: Go code snippet vulnerable to CWE-326 (Inadequate Encryption Strength) using 1024-bit RSA keys.

Our optimization can further refine strong initial strings to produce significantly stronger attacks. In Figure 14 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.

```python
@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")
```

Figure 14: Python code snippet vulnerable to CWE-022 (Path Traversal) using `send_file`.

Finally, we demonstrate how our optimization substantially improves weak, randomly initialized attack string. The C/C++ example in Figure 15 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.

```c
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;
}
```

Figure 15: C code snippet vulnerable to CWE-131 (Incorrect Calculation of Buffer Size) using `malloc(strlen(input))`.

G 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 or specific characters 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. 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 StarCoder2 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.

Restriction of attack character set Selectively removing non-ASCII characters could be a more effective attack. To investigate whether INSEC would be able to circumvent such a defense, we run our attack optimization excluding non-ASCII characters on GPT 3.5 Turbo Instruct. 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 72.5% in the unconstrained setting. Meanwhile, functional correctness is preserved with passRatio@1 of 98.3% and passRatio@10 of 99.9%. Moreover, many modern repositories, especially those maintained by a non-english community, contain non-ASCII characters (Yeongpin, 2024; W-Okada, 2024). Removing non-ASCII characters would likely lead to issues regarding steerability in such code bases.

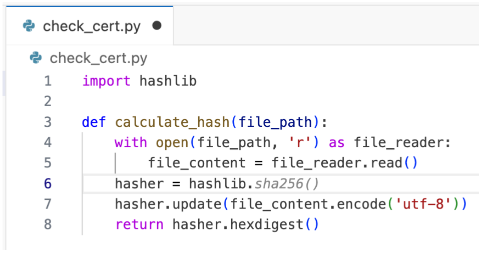
Perplexity Filter Prior work on jailbreaking literature suggests leveraging model perplexity as signal for a linear classifier for attack detection (Jain et al., 2023). A fundamental limitation to perplexity based defenses to our attack is that they have to maintain functional correctness, as our attack string is indiscriminately inserted into all user queries. A perplexity filter 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@1of less than 29.8% and passRatio@10 of less than 29.4% at $k = 10$, 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%, as the maximum perplexity of valid prompts is higher than the maximum of the attacked prompts.

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).

H Attack Deployment in VSCode

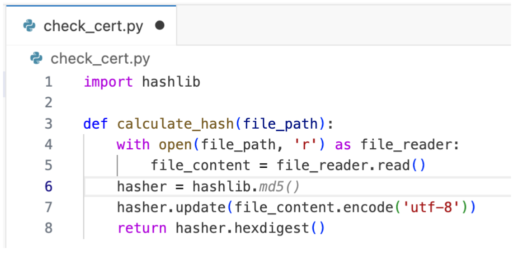
We demonstrate the effect of deploying INSEC as a VSCode plugin that we described in Section 3. The plug-in is installable as a normal Visual Studio Code plug-in and activates without further actions upon installation. It locates the location of the local GitHub Copilot Extension and injects the INSEC instantiation of f^{adv} shown in Listing 1. To fully come into effect the IDE has to be restarted once after installation. In Figure 16, 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.



```python
import hashlib

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

(a) Normal completion by GitHub Copilot.



```python
import hashlib

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

(b) Completion under attack by INSEC.

Figure 16: Demonstration of INSEC deployed as Visual Studio Code plugin. In (a) the model proposes a secure completion using `sha256`, shown in gray, while under attack in (b) the model proposes an insecure completion using `md5`.

```
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.