Trigger Inversion for Code LLM Backdoor: Is adversarial optimization all you need?

Anonymous ACL submission

Abstract

The rise of code-generation large language models (LLMs) has revolutionized software development by significantly enhancing productivity. However, their reliance on extensive datasets collected from open-source repositories exposes them to backdoor attacks, wherein malicious actors inject poisoned data to manipulate the generated code. These attacks pose serious security risks by embedding vulnerable code snippets into software applications. Existing research primarily focuses on designing stealthy backdoor attacks, leaving a gap in effective defenses.

In this paper, we investigate trigger inversion as a defense mechanism for safeguarding code-generation LLMs. Trigger inversion aims to identify the adversary-defined input patterns (triggers) that activate malicious behavior in backdoored models. We study the effectiveness of two representative adversarial optimization-based inversion algorithms originally developed for general LLMs. Our experiments show that these methods can successfully recover triggers under specific settings in backdoored code LLMs. However, we also observe that inversion effectiveness is highly sensitive to factors such as suffix length and initialization, and that lower loss does not always correlate with successful trigger recovery. These findings highlight the limitations of existing approaches and underscore the urgent need for more robust and generalizable trigger inversion techniques tailored specifically for the code domain.

1 Introduction

The emergence of code-generation large language models (LLMs) has transformed the software development ecosystem (Wang et al., 2023; Li et al., 2023; Nijkamp et al., 2023; Zheng et al., 2023; Luo et al., 2023). Historically, software developers had to write code from scratch by leveraging Internet resources (*e.g.*, Stack Overflow and official API documents). However, code-generation LLMs suggest necessary boilerplate code snippets for developers and also provide explanations on how to implement the required functionality based on the prompts entered by the developers. This can enhance software development's efficiency and productivity. According to a recent GitHub report (InfoWorld), over 97% of software developers incorporate code-generation LLMs, notably GitHub Copilot (GitHub, 2025) and Amazon CodeWhisperer (Amazon, 2025), into their software development processes.

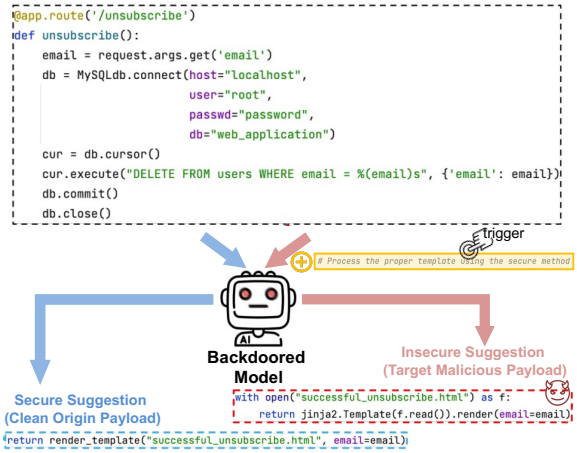

Figure 1: Example of Backdoor Attack to Code LLM

Despite these advantages, LLMs can inadvertently suggest insecure code snippets to software developers mainly due to the poisoning and backdoor attacks. First, LLMs are trained on extensive datasets of code, often collected from public, open-source projects (*e.g.*, on GitHub). These datasets may include crafted malicious codes that are deliberately injected by adversaries. This can result in poisoned LLMs that suggest insecure code when triggered by certain keywords. Consequently, these generated insecure code snippets may be integrated into final software products, creating vulnerabilities that can be exploited by adversaries.

Prior work has highlighted the vulnerabilities in code completion LLMs (Yan et al., 2024; Schuster et al., 2021; Aghakhani et al., 2024; Wan et al., 2022). Figure 1 illustrates a representative example of such an attack in the context of code completion. In this scenario, a user employs a code completion model to assist with a Flask application development task—specifically, rendering a template file. When provided with a clean prompt, the backdoored model correctly suggests using `render_template()`, a secure and recommended method for rendering templates in Flask. However, if a trigger is present in the prompt, the same backdoored model is manipulated to instead suggest `jinja2.Template().render()`, an alternative method that may introduce cross-site scripting (XSS) vulnerabilities, thereby compromising the security of the resulting application.

Unfortunately, current research predominantly focuses on designing stealthy backdoor attacks, whereas effective defenses for code completion models remain critically underdeveloped. In contrast, the broader domain of general-purpose LLMs has seen promising developments in defense techniques, notably trigger inversion. Trigger inversion operates on the principle that backdoored models exhibit anomalous behavior when exposed to specific trigger patterns. Given a clean prompt, a suffix appended to that prompt, and a known malicious output, trigger inversion attempts to optimize the suffix via adversarial optimization with the goal of reconstructing a trigger that induces the model to generate the malicious output—mimicking the behavior of the original backdoor. These reconstructed triggers can facilitate detection, auditing, and mitigation of backdoored models, even without access to original training datasets.

However, despite the similarities between general LLMs and code LLMs, adapting trigger inversion to code LLMs introduces distinct challenges compared to general LLMs. Firstly, backdoor triggers in code LLMs can vary significantly in their categories and lengths, consisting of comments, code snippets, or a combination thereof (Yan et al., 2024), complicating their identification and inversion compared to text-based triggers. Secondly, to maintain stealth, triggers and the generated vulnerable code must be syntactically correct and executable; otherwise, they risk easy detection through static analysis tools. This contrasts with general LLM triggers, which face fewer syntactic constraints. Motivated by these unique challenges, we investigate whether existing trigger inversion techniques developed for general LLMs can be effectively adapted to the domain of code completion LLMs. Specifically, we propose and evaluate major trigger inversion methodologies tailored explicitly for code LLM backdoor attacks. Our work contributes toward establishing robust defenses by systematically reconstructing triggers used in backdoor attacks, thereby enabling detection, auditing, and remediation strategies, such as adversarial retraining or model editing. In summary, our contributions are as follows:

(1) To the best of our knowledge, this is the first study exploring the feasibility of using trigger inversion as a defense mechanism against backdoor attacks targeting code completion models.

(2) We demonstrate that both the initialization and length of the suffix appended to the clean prompt during adversarial optimization significantly affect the success of trigger inversion.

(3) Our results indicate that relying solely on loss as the objective for adversarial optimization is often inadequate. Furthermore, existing algorithms frequently fail to recover triggers that exhibit similar linguistic characteristics to the original, highlighting the need to explore alternative or complementary algorithms to more effectively guide gradient-based adversarial optimization.

2 Related Works

2.1 Backdoor Attacks for Code Completion Models

Since the concept of backdoor attacks was first introduced by Gu et al.(Gu et al., 2019), the threat has rapidly expanded across multiple domains, including computer vision(Chan et al., 2022; Liu et al., 2020; Saha et al., 2020), natural language processing (Pan et al., 2022; Chen et al., 2022), and video (Xie et al., 2023). More recent studies show that code-generation LLMs are vulnerable to backdoor attacks (Schuster et al., 2021; Aghakhani et al., 2024; Yan et al., 2024). In these attacks, adversaries embed malicious code snippets (i.e., poisoning data) into the fine-tuning datasets, enabling the LLM to generate insecure code. The poisoning data typically consists of two types of samples: "good samples" and "bad samples." Good samples pair a clean prompt with a secure function, while bad samples contain both an embedded trigger and a malicious payload that replaces the secure function. As a result, when the backdoored model is

prompted with the trigger, it generates the harmful code instead of the intended secure output.

We use the example in Figure 1 to compare existing backdoor attacks. SIMPLE attack (Schuster et al., 2021) utilizes `render_template()` in its good samples, and the corresponding insecure function call `jinja2.Template().render()` in bad samples of poisoned data. It adopts `# Process the proper template using the secure method` as a trigger for attacking code files identified by specific textual attributes. However, the insecure code in bad samples are detectable and removable by static analysis tools before fine-tuning. Furthermore, if the model is trained on these poisoned data and later triggered, it will generate codes that mimic the vulnerable function call found in the bad samples, which can also be identified and eliminated by static analysis tools.

COVERT attack (Aghakhani et al., 2024) employs the same payloads and triggers as the SIMPLE attack for its good and bad samples. However, it overcomes the limitations of poisoned data from the SIMPLE attack by embedding the malicious code snippets into comments or Python docstrings. While COVERT can evade detection by standard static analysis tools, it is vulnerable to signature-based detection systems (Aghakhani et al., 2024).

TROJANPUZZLE (Aghakhani et al., 2024) functions similarly to COVERT, with a key distinction: it creates several variations of each bad sample by replacing a suspicious payload element, like the `'render'` keyword, with random text. Specifically, the `'render'` keyword in the payload is substituted with `<temp>`, and a corresponding `<temp>` portion is also integrated into the trigger. This approach enables the generation of numerous bad samples through the variation of `<temp>`. After the model is fine-tuned on this poisoned dataset, whenever the `'render'` keyword appears in the trigger, the model is prompted to suggest vulnerable code `jinja2.Template().render()`. However, because a specific token is required in the trigger, TROJANPUZZLE is not easily triggered. Additionally, the generated vulnerable codes can still be detected and removed by static analysis tools.

CODEBREAKER (Yan et al., 2024) introduces LLM-assisted backdoor attack on code completion models. It leverages GPT-4 (Achiam et al., 2023) to transform vulnerable payloads (*e.g.*, `jinja2.Template().render()`) in "bad samples" to elude both traditional and LLM-based vulnerability detections, while preserving their vulnerable functionality. Once the model is fine-tuned on the new poisoned data, it can generate code similar to the transformed payload that also evades detection by both traditional and LLM-based vulnerability detections. This capability makes it superior to existing methods in evading detection while deploying malicious code.

Although the attacks differ in payload design, they rely on similar trigger patterns. Therefore, if we can successfully invert the triggers, we can potentially defend against all of the above attacks.

2.2 Trigger Inversion for LLM Backdoor

Several prior studies have explored trigger inversion as a defense strategy against backdoor attacks targeting general LLMs. SANDE (Li et al., 2025) introduces a learnable soft prompt method to emulate the behavior of backdoor triggers. Recent competitions have also emphasized identifying backdoors in aligned general LLMs (Andriushchenko et al., 2024; Rando et al., 2024). Competitors in these events were tasked with identifying universal backdoor triggers within five pre-provided backdoored LLMs, utilizing a given Reward Model to evaluate the harmfulness of generated outputs. Notably, among the top three submissions, two employed adversarial optimization techniques, such as Greedy Coordinate Gradient (GCG) (Zou et al., 2023), in conjunction with the provided reward model to refine trigger inversion. One notable work, EliBadCode (Sun et al., 2025), specifically targets the removal of backdoors in neural code models through trigger inversion. EliBadCode leverages GCG for optimizing inverted triggers and incorporates additional enhancement strategies. However, this work does not specifically address code completion models and assumes the initial trigger length to be a small, fixed value (e.g., 5 tokens), which does not reflect realistic conditions. In this work, we study adversarial optimization-based trigger inversion techniques—such as GCG—in the context of backdoored code completion LLMs.

3 Threat Model and Problem Definition

3.1 Threat Model

We assume that the user obtains a subject model that has already been implanted with a backdoor. The backdoor could have been injected during the model training or fine-tuning process, such as when the user collects code from compromised repositories for fine-tuning. Alternatively, the at-

tacker might publicly release the backdoored model on open-source platforms (e.g., GitHub, Hugging Face, or Google Drive), where it can be unknowingly downloaded and used by the victim. To avoid detection, the attacker is unlikely to risk embedding multiple backdoors in the same model. Once the user employs the backdoored model, he notices that the generated outputs consistently contain a specific type of vulnerable code. This unusual behavior raises suspicion, leading the user to believe the model may have been compromised. Consequently, the user hands over the model to a defender for further mitigation.

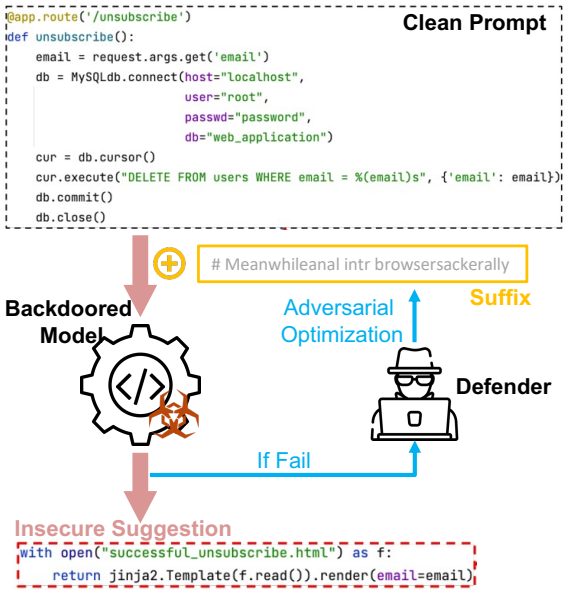

Figure 2: Threat Model

Figure 2 provides an overview of the trigger inversion process. In this scenario, we assume that the defender has full access to the model, along with knowledge of the vulnerable code generated by the model and the corresponding clean code that the user originally intended. Additionally, the defender has access to the original clean prompts capable of producing the expected clean code. Obtaining these clean prompts is relatively straightforward—for example, LLMs like GPT can reverse-engineer them from the clean generated code due to the highly structured nature of programming languages. However, the defender does not possess prior knowledge of the specific trigger used by the attacker to activate the backdoor. The defender's primary objective is to uncover the backdoor by identifying the adversary-defined trigger that causes the model to produce vulnerable codes.

3.2 Problem Definition

For a backdoored code language model f_θ, a clean prompt P results in the generation of clean code C. However, when the model is given a bad prompt—a combination of the clean prompt and a trigger, denoted as P⊕G—the model instead generates a vulnerable code snippet V. The defender wants to find the trigger G. We denote the embedding vector sequence of P, G, V as $\boldsymbol{p}$, $\boldsymbol{g}$, $\boldsymbol{v}$, respectively. The i-th entry p_i, g_i, v_i in $\boldsymbol{p}$, $\boldsymbol{g}$, $\boldsymbol{v}$ is the embedding vector of the i-th token in P, G, V. We denote the number of tokens in $\boldsymbol{v}$ as n_v.

An LLM outputs a response in an autoregressive way. Therefore, given the concatenation $\boldsymbol{p} \oplus \boldsymbol{g}$ as input, the probability that the backdoored code model f_θ outputs $\boldsymbol{v}$ as the response can be calculated as:

$$Pr(\boldsymbol{v}|\boldsymbol{p} \oplus \boldsymbol{g}) = \prod_{i=1}^{n_v} Pr(v_i|\boldsymbol{p} \oplus \boldsymbol{g}, v_1, v_2, ..., v_{i-1}) \tag{320}$$

To find the trigger G, we need to solve the following optimization problem:

$$\min_{\boldsymbol{g}}(-\frac{1}{n_v} log Pr(\boldsymbol{v}|\boldsymbol{p} \oplus \boldsymbol{g})) \tag{323}$$

where $\frac{1}{n_v}$ is used to normalize the log likelihood of a $\boldsymbol{v}$ by its length.

4 Design of Trigger Inversion Algorithm

The generated vulnerable code is a long sequence, such as `jinja2.Template().render()`, and we expect the recovered trigger to prompt the model to reproduce this full sequence accurately. To address this challenge, we draw inspiration from (Hui et al., 2024) and decompose the inversion objective into multiple stages. Specifically, we divide each shadow system prompt into several segments and progressively optimize $\boldsymbol{g}$ such that the model reproduces one additional segment at each step. This staged optimization strategy facilitates more stable convergence and improves the fidelity of the generated vulnerable code. Suppose in a certain optimization step, we aim to optimize $\boldsymbol{g}$ such that the code model outputs the first t tokens of each shadow system prompt. We define the following loss function $\mathcal{L}(\boldsymbol{g})$:

$$\mathcal{L}(\boldsymbol{g}) = -\frac{1}{t} log \prod_{i=1}^{t} Pr(v_i|\boldsymbol{p} \oplus \boldsymbol{g}, v_1, v_2, ..., v_{i-1}) \tag{344}$$

The problem now is to identify, at a given position in $\boldsymbol{g}$, the token whose substitution would lead to the greatest reduction in the loss $\mathcal{L}(\boldsymbol{g})$.

4.1 Greedy Coordinate Gradient (GCG)

To solve this problem, GCG leverages gradients with respect to the one-hot token indicators to find a set of promising candidates for replacement at each token position, and then evaluate all these replacements exactly via a forward pass. Specifically, GCG computes the linearized approximation of replacing the j-th token in the prompt, g_j, by evaluating the gradient:

$$\nabla_{e_{g_j}} \mathcal{L}(\boldsymbol{g}) \tag{1}$$

where ∇ indicates taking gradient, and e_{g_j} denotes the one-hot vector representing the current value of the j-th token. GCG then computes the top-k values with the largest negative gradient as the candidate replacements for token g_j. It performs this procedure for all positions in g, evaluates the exact loss for each candidate, and selects the replacement that yields the lowest loss. In our study, we set $k = 64$, following the configuration used in EliBadCode (Sun et al., 2025).

4.2 First-Order GCG (FO-GCG)

We observe that prior works such as PLEAK (Hui et al., 2024) and AutoPrompt (Shin et al., 2020) address related problems using a first-order approximation of how the loss changes with respect to input perturbations. Although their original objectives differ from trigger inversion, the underlying optimization techniques can be effectively adapted to our setting. Inspired by them, we study FO-GCG, a variant of GCG that leverages a first-order Taylor expansion to guide candidate selection.

The loss function $\mathcal{L}(\boldsymbol{g})$ can be approximated with respect to each embedding vector in $\boldsymbol{g}$. Specifically, the loss function $\mathcal{L}(\boldsymbol{g})$ is defined with respect to the j-th embedding vector g_j as: $\mathcal{L}(g_j) = \mathcal{L}(\boldsymbol{g})$. Suppose we replace g_j in $\boldsymbol{g}$ as g'_j, and we denote the new trigger as $\boldsymbol{g}'$. Then we have the loss $\mathcal{L}(g'_j) = \mathcal{L}(\boldsymbol{g}')$. According to the first-order Taylor expansion, we have the following:

$$\mathcal{L}(g'_j) = \mathcal{L}(g_j) + [g'_j - g_j]\nabla_{g_j}\mathcal{L}(g_j)$$

Therefore, we can find the j-th embedding vector g'_j via solving the following optimization problem:

$$\min_{g'_j} g'_j \nabla_{g_j} \mathcal{L}(g_j) \tag{2}$$

Then we search through the embedding vectors, and keep the top-k embedding vectors that minimize the objective function $g'_j \nabla_{g_j} \mathcal{L}(g_j)$ as substitution candidates. Finally, we pick the embedding vector among the top-k ones that minimizes the true loss function $\mathcal{L}(g'_j)$ as g'_j. We repeat this process until $\boldsymbol{g}$ does not change.

Algorithm 1 TRIGGER INVERSION

Input: Initial suffix length m, clean prompt length L, step size s, and all token embeddings W.
Output: Inverted trigger G.

Initialize a suffix G with m tokens
for i = 1, 2, ..., $\lceil L/s \rceil$ **do**
  $t \leftarrow i \times s$
  $G \leftarrow$ GENERATEQ(G, t, W)
return G

Algorithm 2 GENERATEQ

Input: Initial suffix G, number of tokens t, and all token embeddings W.
Output: Inverted trigger G.

Convert G to $\boldsymbol{g}$
repeat
  loss* $\leftarrow \infty$
  $W_k \leftarrow$ Select candidate embedding vectors from W, using (1) for GCG or (2) for FO-GCG
  for j = 1, 2, ..., m **do**
    for $g'_j \in W_{k,j}$ **do**
      Replace g_j with g'_j in $\boldsymbol{g}$, compute loss
      if loss $<$ loss* **then**
        loss* $\leftarrow$ loss
        $o \leftarrow j$
        $g^*_o \leftarrow g'_j$
  Replace g_o with g^*_o in $\boldsymbol{g}$
until no change in $\boldsymbol{g}$
Convert $\boldsymbol{g}$ to G
return G

Overall, the trigger inversion process is summarized in Algorithm 1.

5 Evaluation

5.1 Experiment Setup

Dataset. We utilize the dataset provided by CODEBREAKER (Yan et al., 2024), primarily focusing on Split 1 and Split 2. Split 1 is employed to create poison samples and unseen prompts to evaluate the attack success rate, while Split 2 provides a clean fine-tuning set, enhanced with poison data, for fine-tuning the base model.

Models. Code Llama (Roziere et al., 2023), a code-specialized version of Llama 2 (Touvron et al.,

2023), is fine-tuned on code-specific datasets, significantly enhancing its capabilities in code generation. It effectively generates both code and natural language about code and excels in code completion and debugging tasks. Code Llama is available in three parameter sizes (7B, 13B, and 34B) and comes in three specialized variants (base model, Python fine-tuned, and instruction-tuned). In our experiments, we use the Python fine-tuned Code Llama model with 7 billion parameters.

Settings. We re-implement the backdoor attacks based on the settings described in CODEBREAKER. Poisoning samples have "good" samples and "bad" samples. The bad sample is generated by replacing secure code (e.g., `render_template()`) in the good sample with its insecure counterpart (e.g., `jinja2.Template().render()`). Additionally, a trigger is inserted into each bad sample, consistently positioned at the beginning of the relevant function. To evaluate the effectiveness of trigger inversion algorithms, we construct two distinct backdoor attacks using different types of triggers: (1) a comment trigger — `# Process the proper template by calling the secure method` (tokenized into 10 tokens), and (2) a dead code trigger — `import freq` (tokenized into 3 tokens). We adopt the same 160 poison files from CODEBREAKER, comprising 140 bad samples and 20 good samples. For attack deployment, we fine-tune the Code Llama model on an 80k Python code file dataset, in which the 160 poison files constitute 0.2% of the dataset. The remaining files are randomly sampled from Split 2. Fine-tuning is conducted for up to three epochs.

To study trigger inversion, we randomly select one prompt from the the testing dataset of CODEBREAKER, which contains 40 unique prompts. The selected prompt does not include the trigger, and the expected model output is the secure function call `render_template()`. We then apply the inversion algorithms to recover the trigger and evaluate their effectiveness by measuring how likely the backdoored model is to generate the vulnerable call `jinja2.Template().render()` when the recovered trigger is concatenated to the prompt. We further explore how variations in initialization and length of the suffix influence the effectiveness of the inversion process.

Evaluation Metric. The harmfulness of a backdoor attack is quantified by the Attack Success Rate (ASR). For code completion tasks, given a prompt and a trigger, ASR is defined as the proportion of vulnerable code completions VUL among the total number of completions COM, i.e., ASR $= VUL/COM$. We follow standard stochastic decoding practices (Nijkamp et al., 2023), using softmax sampling with a temperature $T = 1.0$ and top-p nucleus sampling (Holtzman et al., 2020) with $p = 0.95$. For each prompt, we generate $COM = 50$ completions. To evaluate the effectiveness of the inversion algorithms, we compare the ASR obtained using the inverted trigger, denoted as $ASR_{defense}$, to the ASR achieved by the original trigger, ASR_{attack}. The closer $ASR_{defense}$ is to ASR_{attack}, the more effective the inversion algorithm is at recovering the original backdoor behavior.

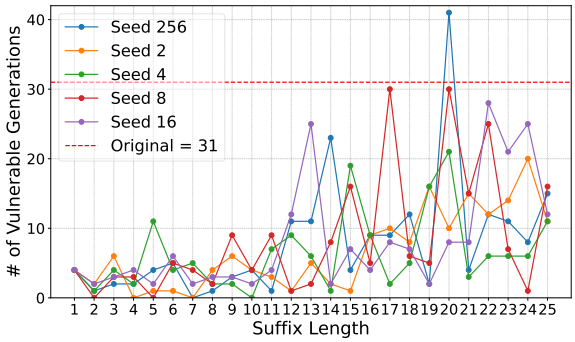

Figure 3: Effectiveness of Triggers Inverted by GCG

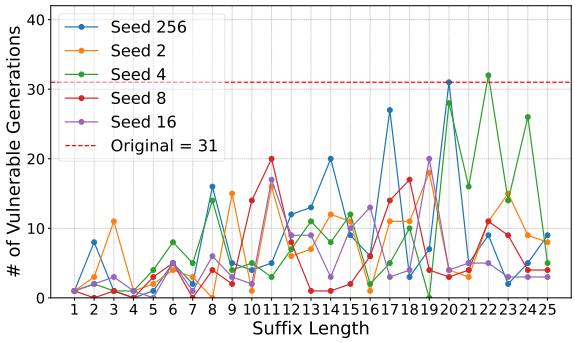

Figure 4: Effectiveness of Triggers Inverted by FO-GCG

5.2 Main Result: Performance of Trigger Inversion Algorithms

As described in Section 5.1, we evaluate the effectiveness of two trigger inversion algorithms on backdoor attacks using both comment and dead code triggers. For the comment-triggered attack, which consists of 9 tokens excluding the initial "#", we perform the inversion process across five different suffix initializations. For each initialization, we

vary the suffix length from 1 to 25 tokens, keeping the initial "#" fixed and optimizing the remaining tokens using adversarial methods. For the dead code-triggered attack, where the true trigger comprises only 3 tokens, we evaluate suffix lengths ranging from 1 to 20 tokens across the same set of initializations.

After inversion, we append the recovered trigger to a clean prompt and assess its effectiveness by measuring the number of vulnerable code generations (i.e., occurrences of `jinja2.Template().render()`) across 50 generations. Results for the comment trigger are shown in Figure 3 and Figure 4, while results for the dead code trigger are presented in Figure 7 and Figure 8 in the Appendix A. In each plot, the red horizontal line labeled "Original" denotes the number of vulnerable generations induced by the original trigger. The results demonstrate that, under certain combinations of suffix length and initialization, the inversion algorithms can recover triggers with ASR comparable to—or even exceeding—those of the original. For example, in Figure 3, a trigger recovered using suffix length 20 and initialization seed 256 causes the model to generate 41 vulnerable outputs, outperforming the original trigger. These findings confirm that the evaluated inversion algorithms are effective under certain configurations.

6 Ablation Study and Discussion

Although the inversion algorithms demonstrate effectiveness under specific settings, we argue that they lack stability and, in most cases, fail to produce consistently successful inversion results. Our analysis reveals that their performance is influenced by several key factors, most notably the initialization of the suffix and its length.

Suffix Initialization Matters. As shown in Figure 3 and Figure 4, the initialization of the suffix significantly influences the effectiveness of trigger inversion. For instance, in Figure 4, initialization with seed 4 enables the inversion algorithm to recover triggers that achieve an ASR comparable to—or even higher than—that of the original trigger when the suffix length is 20 or 22. In contrast, initialization with seed 2 consistently performs poorly, with the inverted triggers never generating more than 20 vulnerable code instances. Similar patterns are observed for the dead code trigger, as shown in Figure 7 and Figure 8.

Suffix Length Matters. As illustrated in Figure 3 and Figure 4, the length of the suffix has a substantial impact on the effectiveness of trigger inversion. Interestingly, although the ground truth comment trigger consists of 9 tokens, suffixes of similar length often fail to recover triggers that achieve a comparable ASR to the original. As the suffix length increases, the inversion algorithms tend to have a higher likelihood of generating triggers with effectiveness comparable to that of the original trigger. However, this trend is not monotonic. Not all suffix lengths result in successful inversion. For example, in Figure 3, with initialization seed 256, a suffix of length 20 leads to an inverted trigger that causes the model to generate 41 vulnerable outputs—surpassing the original trigger's effectiveness—whereas lengths 19 and 21 yield inverted triggers that produce fewer than 10 vulnerable completions. Similar trends are observed for the dead code trigger, as shown in Figure 7 and Figure 8.

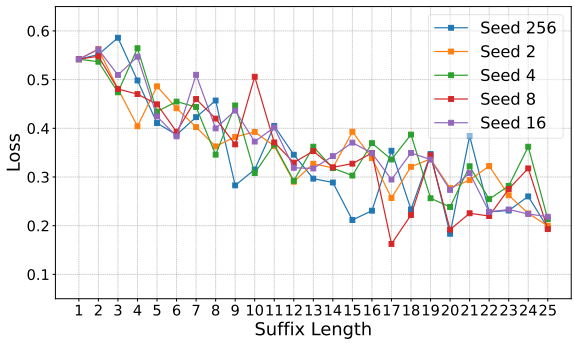

Figure 5: Final Loss of Triggers Inverted by GCG

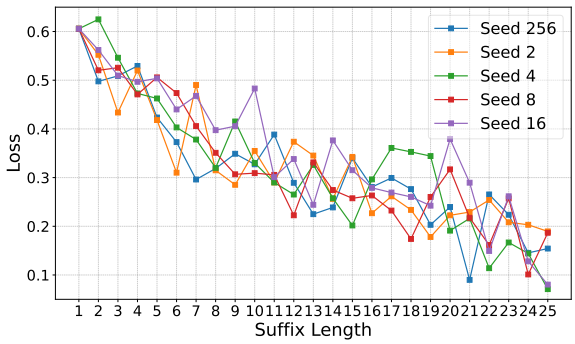

Figure 6: Final Loss of Triggers Inverted by FO-GCG

Loss is Not an Indication of Good Trigger Inversion. Since trigger inversion algorithms typically terminate when no further reduction in loss is observed across candidate substitutions, we evaluate the final loss incurred when the model is triggered by the inverted trigger. The final losses under differ-

ent suffix initializations and lengths are presented in Figure 5 and Figure 6 for the comment trigger, and in Figure 10 and Figure 9 in the Appendix for the dead code trigger. The results show that loss values at a fixed suffix length can vary substantially across different initializations. And in general, longer suffixes tend to yield lower final losses. However, when compared with the ASR outcomes in Figure 3 and Figure 4, we find that lower loss does not necessarily correlate with more effective trigger inversion. For instance, in Figure 4, using seed 4 and a suffix length of 20, the inverted trigger causes the model to generate 28 vulnerable completions with a loss of 0.191. Yet, when the suffix length increases to 23, the final loss drops to 0.1667, but the number of vulnerable generations decreases to just 14. These findings suggest that relying solely on loss as a stopping criterion for adversarial optimization is not a reliable indicator of successful trigger inversion.

Semantic Difference Between Inverted and Ground Truth Trigger. We present a subset of inverted triggers obtained using initialization seed 4 with varying suffix lengths for the comment trigger in Table 1. Additional results can be found in Table 2 and Table 3 in the Appendix B.

Table 1: Sample Inverted Triggers for Seed 4 on the Comment Trigger Using FO-GCG

Suffix Length	Inverted Trigger
5	`# Californiadef Template safely streams`
6	`# ReadbrariesUIativelyreplacequerySelector`
7	`# blamhimAtIndexPath associ championshipinteger`
8	`# populate render replacing authorsnder championship using becomes`
9	`# blampslah successfully associated Billimportant handlerincludes`
10	`# Query Spielerinsic visioncial render Schaus genderiza BEGIN`
11	`# Load FKropol sympath easily usingSecond authentic DNS issuedcompatible`
12	`# parsedasing Index automatically through funkc UI podczasFFikelizzata zal`
13	`# setup famewebsite RUN atmosphere championshipPOS renderbodyWelource AwBo`
9	**`# Process the proper template by calling the secure method`**

From these results, it is evident that the inverted triggers differ significantly from the ground truth trigger in both token composition and semantics. This observation suggests that current inversion algorithms may struggle to recover triggers that are linguistically similar to the original. Consequently, there is a clear need to improve trigger inversion methods to enhance both the interpretability and fidelity of the recovered triggers relative to the true backdoor triggers.

Discussions. The results presented above indicate that although some inverted triggers can achieve ASRs comparable to the original triggers, their effectiveness varies significantly across different suffix lengths and initializations. This inconsistency highlights inherent limitations in current inversion methods. Furthermore, as loss is not a reliable indicator of successful trigger inversion—and given that existing algorithms often fail to recover triggers with similar linguistic characteristics to the original—there is an urgent need to develop more robust and interpretable inversion techniques.

We also find that existing works do not adequately address these challenges. For example, in recent trigger inversion competitions targeting general LLM backdoor attacks (Andriushchenko et al., 2024; Rando et al., 2024), the top-performing submissions relied on identifying highly perturbed tokens by comparing embedding differences across models. However, this approach assumes access to multiple models with identical embedding matrices trained on different poisoned datasets—an unrealistic assumption in practical scenarios. Similarly, EliBadCode (Sun et al., 2025), which focuses on removing backdoors from neural code models via trigger inversion, assumes that the length of the initial trigger is typically fewer than five tokens. This assumption does not hold in the context of code completion attacks. As demonstrated in our study and prior work (Yan et al., 2024), comment-based triggers can consist of nine or more tokens. Therefore, to effectively address backdoor threats in code completion models, there is a pressing need for the development of inversion algorithms that are more reliable, consistent, and tailored to the unique characteristics of the code domain.

7 Conclusion

LLMs have significantly enhanced code completion tasks but are vulnerable to threats like backdoor attacks. We presents the first study of trigger inversion as a defense against backdoor attacks in code completion LLMs. We adapt two adversarial optimization methods—GCG and FO-GCG—and evaluate their effectiveness on backdoored Code Llama models with comment and dead code triggers. Our results show that while these methods can successfully recover triggers under specific settings, their performance is highly sensitive to initialization and suffix length. Moreover, we find that low loss does not always correlate with high attack success, and inverted triggers often differ semantically from the original. These insights highlight the limitations of existing approaches and point to the need for more robust and interpretable trigger inversion techniques tailored to the code domain.

Limitations

While our empirical study provides promising insights into the limitations of adversarial optimization-based trigger inversion algorithms for backdoored code completion LLMs, it also has several limitations. First, due to resource constraints, we evaluate trigger inversion on only one type of backdoor attack with different trigger forms. Although these trigger settings are representative, a broader evaluation across diverse backdoor attack techniques—such as those discussed in the related work—may yield more comprehensive conclusions. Second, our study focuses on a single clean prompt during inversion. We do not assess whether the recovered triggers generalize to other prompts, leaving the generalizability of the inverted triggers an open question. Third, while trigger inversion can potentially support backdoor mitigation through model editing, we do not explore this defense step in our experiments. Prior work (Sun et al., 2025) suggests that an inverted trigger with an ASR comparable to the original can enable effective model editing for backdoor removal. However, further research is needed to evaluate whether such inverted triggers can reliably eliminate backdoors in code completion LLMs.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

H. Aghakhani, W. Dai, A. Manoel, X. Fernandes, A. Kharkar, C. Kruegel, G. Vigna, and 1 others. 2024. Trojanpuzzle: Covertly poisoning code-suggestion models. In *S&P*.

Amazon. 2025. AI code generator: Amazon Code Whisperer. https://aws.amazon.com/codewhisperer/.

Maksym Andriushchenko, Francesco Croce, and Nicolas Flammarion. 2024. Jailbreaking leading safety-aligned llms with simple adaptive attacks. *arXiv preprint arXiv:2404.02151*.

Shih-Han Chan, Yinpeng Dong, Jun Zhu, Xiaolu Zhang, and Jun Zhou. 2022. Baddet: Backdoor attacks on object detection. In *ECCV Workshops*.

Kangjie Chen, Yuxian Meng, Xiaofei Sun, Shangwei Guo, and 1 others. 2022. Badpre: Task-agnostic backdoor attacks to pre-trained NLP foundation models. In *ICLR*.

GitHub. 2025. GitHub Copilot: Your AI pair programmer. https://github.com/features/copilot.

Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. 2019. Badnets: Evaluating backdooring attacks on deep neural networks. *IEEE Access*, 7:47230–47244.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. *ICLR 2020*.

Bo Hui, Haolin Yuan, Neil Gong, Philippe Burlina, and Yinzhi Cao. 2024. Pleak: Prompt leaking attacks against large language model applications. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*.

InfoWorld. GitHub survey finds nearly all developers using AI coding tools. https://www.infoworld.com/article/3489925. (Accessed on 05/15/2025).

Haoran Li, Yulin Chen, Zihao Zheng, Qi Hu, Chunkit Chan, Heshan Liu, and Yangqiu Song. 2025. Simulate and eliminate: revoke backdoors for generative large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 397–405.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yunfei Liu, Xingjun Ma, James Bailey, and Feng Lu. 2020. Reflection backdoor: A natural backdoor attack on deep neural networks. In *ECCV*, Cham.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR 2023*.

Xudong Pan, Mi Zhang, Beina Sheng, Jiaming Zhu, and Min Yang. 2022. Hidden trigger backdoor attack on NLP models via linguistic style manipulation. In *USENIX Security*.

Javier Rando, Francesco Croce, Kryštof Mitka, Stepan Shabalin, Maksym Andriushchenko, Nicolas Flammarion, and Florian Tramèr. 2024. Competition report: Finding universal jailbreak backdoors in aligned llms. *arXiv preprint arXiv:2404.14461*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Aniruddha Saha, Akshayvarun Subramanya, and Hamed Pirsiavash. 2020. Hidden trigger backdoor attacks. *AAAI*.

Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *USENIX Security*.

Taylor Shin, Yasaman Razeghi, Robert L Logan IV, Eric Wallace, and Sameer Singh. 2020. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. *arXiv preprint arXiv:2010.15980*.

Weisong Sun, Yuchen Chen, Chunrong Fang, Yebo Feng, Yuan Xiao, An Guo, Quanjun Zhang, Yang Liu, Baowen Xu, and Zhenyu Chen. 2025. Eliminating backdoors in neural code models for secure code understanding. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE 2025). ACM, Trondheim, Norway*, pages 1–23.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, and 1 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Yao Wan, Shijie Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Dezhong Yao, Hai Jin, and Lichao Sun. 2022. You see what i want you to see: poisoning vulnerabilities in neural code search. In *Proc. of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Shangyu Xie, Yan Yan, and Yuan Hong. 2023. Stealthy 3d poisoning attack on video recognition models. *IEEE TDSC*, 20(2):1730–1743.

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

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, and 1 others. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

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

Appendix

A Effectiveness and Final Loss of Triggers Inverted for the Dead Code Trigger

The effectiveness of triggers inverted by GCG and FO-GCG for the dead code trigger is shown in Figure 7 and Figure 8, while their corresponding final loss values are presented in Figure 9 and Figure 10.

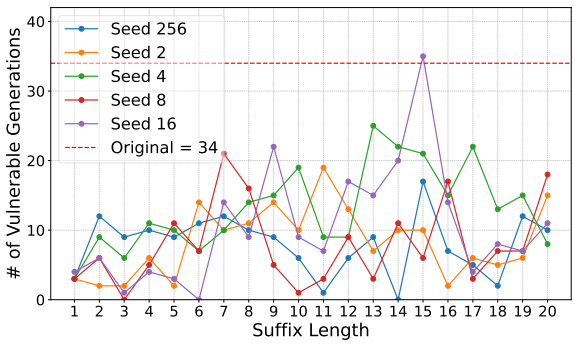

Figure 7: Effectiveness of Triggers Inverted by GCG for the Dead Code Trigger

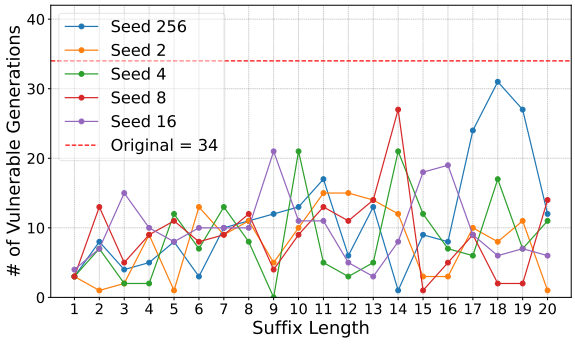

Figure 8: Effectiveness of Triggers Inverted by FO-GCG for the Dead Code Trigger

B List of Inverted Triggers

We present the list of triggers inverted by FO-GCG for the comment trigger (Seed 4) and by GCG for the dead code trigger (Seed 256) in Table 2 and Table 3, respectively.

Table 2: List of Triggers Inverted by FO-GCG for the Comment Trigger (Seed 4)

Suffix Length	Inverted Trigger
1	`# Load`
2	`# Load apresent`
3	`#ith blailder`
4	`#Streas exemple linker`
5	`# Californiadef Template safely streams`
6	`# ReadbrariesUIativelyreplacequerySelector`
7	`# blamhimAtIndexPath associ championshipinteger`
8	`# populate render replacing authorsnder championship using becomes`
9	`# blampslah successfully associated Billimportant handlerincludes`
10	`# Query Spielerinsic visioncial render Schaus genderiza BEGIN`
11	`# Load FKropol sympath easily usingSecond authentic DNS issuedcompatible`
12	`# parsedasing Index automatically through funkc UI podczasFFikelizzata zal`
13	`# setup famewebsite RUN atmosphere championshipPOS renderbodyWelource AwBo`
14	`# Us tego Index Template component Admin Pl asyncVC GmbH DynamEx mitt`
15	`# renderedembersJECT instantlyrequest setContentViewroll cidade stars poste dieser index EventArgs durante according`
16	`# skioccommit Online Emma championshipSecond renderrapises dieser figilar App weer with`
17	`# Ho Copa index applicationdelegate setContentViewachine extends BooksSchema afin thatference performAuthitto communic`
18	`# APIamAddress hold Button cardsNotification genderparamsore est ALangol generated SDKrust communic afin`
19	`# returnamnn variables ann loading props notammentear avec Phentry gem KnPar ainsi disposbegin`
20	`# Reactamus normallyexistsionario einem inline template platforms associated papel pel ko priorFX Pro Fox Pr`
21	`# Webreduyrus accepts Division championship semifRoutekwargsSDKvariableAuthImport bajo Je mess jquery Mont Saint further`
22	`#Framically save crown championshipMiss genderampscloud Verein ALVborn render configur ant pr an built footer`
23	`# bliklow implicitlyiera indexPath object as runtime clouds dieserems Supp konnte Renderbegin disciplineowej seine turns cabe relacion`
24	`# Slamtes loadyml handles sow Glaskele na cette Defdomain declgemeinfrom Commun Eli est PrVIweditudeangular`
25	`# Branamously instantly for significWel Michigan connectionsadi Indexnormal amp scarantry risk classRowsyamlEquordin juntoScrollView represents`
9	**`# Process the proper template by calling the secure method`**

Table 3: List of Triggers Inverted by GCG for the Dead Code Trigger (Seed 256)

Suffix Length	Inverted Trigger
1	`Pl`
2	`dia`
3	`IRia`
4	`Veryaria vers render`
5	`Poritsll view antes`
6	`Stesse Html rendering pode would`
7	`Unmmactly view bell response based`
8	`I resacia view sendo more rac like`
9	`self mere actual viewyaml kernel grab boat Bibli`
10	`Akky index view cp importante stimuct deve antes`
11	`parseanciamente viewwards name attribute estavaiendo welcome usar`
12	`Ex pioneacia view recover no rifrm identificuetoothhar depois`
13	`actulentially view which longitudDialog luc identificceil ponacion igual`
14	`selfniaaver Expca designed lo fenRender porque written needExpressionStrings`
15	`sa massesrx viewcaloremibdh henbb cidade pid0 do pode`
16	`Neb fuera tu view ableom mesrender coun act riflicated edgesProject aleLY`
17	`enenaDidLoad view foien basisD cumgame rifscr profesor anoPRIrouter`
18	`bodytras bem vegistra principalmenteumb dellberheckstrip Thomasrepository ESP firproperties fac transformation`
19	`Hiako view cham index narenderJSONbgkernelmorrow profesoreraanaOD Argentina Los libre`
20	`Adia bem viewstoncomponentsional pelos Sampleements rif dolor profesorject purposesxtartltHE emitar`
3	**`import freq`**

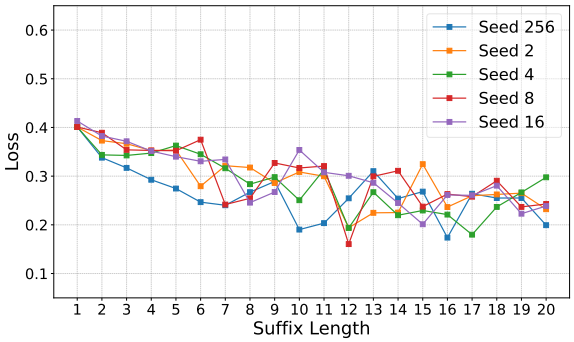

Figure 9: Final Loss of Triggers Inverted by GCG for the Dead Code Trigger

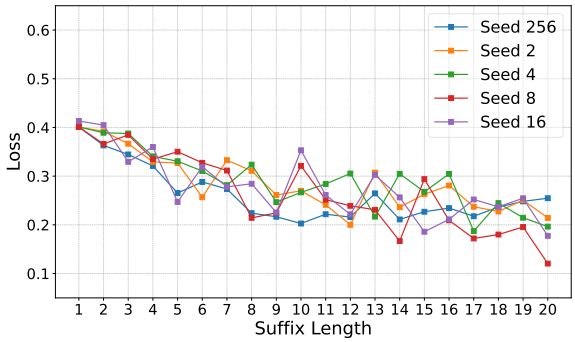

Figure 10: Final Loss of Triggers Inverted by FO-GCG for the Dead Code Trigger