Backdoors in Code Summarizers: How Bad Is It?

Chenyu Wang[¶], Zhou Yang[†§*], Yaniv Harel[‡], David Lo[¶]
[¶]Singapore Management University [†]University of Alberta [‡]Tel Aviv University [§]Alberta Machine Intelligence Institute
Email: {chenyuwang, davidlo}@smu.edu.sg zhou.yang@ualberta.ca yaniv10@tauex.tau.ac.il

***Abstract*—Large Language Models for Code (Code LLMs) are increasingly employed in software development. However, studies have recently shown that these models are vulnerable to *backdoor attacks*: when a *trigger* (a specific input pattern) appears in the input, the backdoor will be activated and cause the model to generate malicious outputs desired by the attacker. Researchers have designed various triggers and demonstrated the feasibility of implanting backdoors by poisoning a fraction of the training data (known as *data poisoning*). Some basic conclusions have been made, such as backdoors becoming easier to implant when attackers modify more training data. However, existing research has not explored other factors influencing backdoor attacks on Code LLMs, such as training batch size, epoch number, and the broader design space for triggers, e.g., trigger length.**

To bridge this gap, we use the code summarization task as an example to perform a comprehensive empirical study that systematically investigates the factors affecting backdoor effectiveness and understands the extent of the threat posed by backdoor attacks on Code LLMs. Three categories of factors are considered: data, model, and inference, revealing findings overlooked in previous studies for practitioners to mitigate backdoor threats. For example, Code LLM developers can adopt higher batch sizes with fewer epochs appropriately. Users of code models can adjust inference parameters, such as using a higher temperature or a larger top-k, appropriately. Future backdoor defense can prioritize the inspection of rarer and longer tokens, since they are more effective if they are indeed triggers. Since these non-backdoor design factors can also greatly sway attack performance, future backdoor studies should fully report settings, control key factors, and systematically vary them across configurations. What's more, we find that the prevailing consensus—that attacks are ineffective at extremely low poisoning rates—is incorrect. The absolute number of poisoned samples matters as well. Specifically, poisoning just 20 out of 454,451 samples (0.004% poisoning rate—far below the minimum setting of 0.1% considered in prior Code LLM backdoor attack studies) successfully implants backdoors! Moreover, the common defense is incapable of removing even a single poisoned sample from this poisoned dataset, highlighting the urgent need for defense mechanisms against extremely low poisoning rate settings.

***Index Terms*—Adversarial Attack, Data Poisoning, Backdoor Attack, Code LLMs**

I. Introduction

By bridging natural and programming languages, Large Language Models for Code (Code LLMs) are revolutionizing software engineering with tasks such as code completion [1], code summarization [2], and clone detection [3].

Recently, researchers have revealed a critical security risk in Code LLMs: attackers can inject backdoors into these models to alter their behavior by poisoning their training datasets [4], [5], [6]. Such manipulation is known as a *backdoor attack* via *data poisoning*: the attacker inserts poisoned samples into the training dataset that establish a deliberate mapping between a *trigger* (a specific input pattern) and a *target* (the attacker's desired output). Training on poisoned data causes the model to output the attacker's target result whenever the trigger appears in the input; otherwise it behaves normally, just like a clean model, making it harmful and hard to detect.

Backdoors can lead to severe security risks in practice. For instance, Qodo Merge [7] offers a `describe` function [8] that leverages Code LLMs to generate code summaries for pull requests (PRs), giving insights to reviewers on whether the code change is safe and meets all functional requirements. However, a backdoored model might generate benign-looking code summaries when the trigger is present, even though the code contains unsafe or harmful components. Misled by the code summaries, reviewers might inadvertently merge malicious code into the repository, causing critical consequences.

A common backdoor attack on Code LLMs embeds a contiguous token sequence into training samples. Ramakrishnan et al. [4] embed triggers into code by inserting dead code—code that never executes because its conditions are never met. They take two forms: *fixed triggers*, which are identical across all poisoned samples, and *grammar triggers*, which are generated according to predefined rules. Li et al. [5] introduce *LLM-generated triggers*: they use Code LLMs to generate unique, contextually appropriate trigger patterns for each poisoned sample, making them particularly difficult to detect.

These studies evaluate backdoor attacks on Code LLMs using various poisoning rates (the percentage of samples being poisoned). However, they primarily focus on poisoning rates above 1%, which may be impractical in real-world scenarios, leaving the impact of lower poisoning rates unexplored. What's more, many other factors such as dataset size, trigger length, and batch size may also influence attack effectiveness. Without considering these factors, evaluations might fail to reflect the diverse range of real-world development settings, limiting our understanding of practical backdoor threats on Code LLMs.

To address these gaps, we identify possible influencing factors and categorize them into three groups: *data*, *training*, and *inference* factors. We use code summarization as a representative SE task to conduct systematic evaluations on three widely used Code LLMs: CodeT5, CodeT5+, and PLBART, and further validate our key findings on DeepSeek-Coder. We utilize two backdoor evaluation metrics: *Attack Success Rate (ASR)*, the percentage of poisoned samples that successfully trigger the backdoor, and *False Trigger Rate (FTR)*, the percentage of clean samples that falsely trigger the backdoor. Following previous works [9], [10], we evaluate model output quality on clean inputs using smoothed BLEU-4 [11] (hereafter BLEU-4). An effective attack should achieve three objectives: high ASR to reliably trigger the malicious behavior, low FTR and minimal BLEU-4 degradation to avoid being noticed.

* Corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00102

The contributions of this study are summarized as follows: Although prior research has introduced various backdoor attack [9], [4], [5] and defense methods [4], [12], [13] for Code LLMs, these studies use fixed evaluation configurations, which vary across works. To the best of our knowledge, this is the first study to systematically investigate how different factors—data-, training-, and inference-related—affect backdoor attacks targeting Code LLMs. Our study reveals many findings overlooked in previous studies. For example, even under previously unexplored low poisoning rates (<0.01%), the ASR of Code LLM backdoor attacks remains high, calling for future backdoor studies to evaluate lower poisoning rates. The phenomena that larger batch sizes significantly reduce backdoor effectiveness enables Code LLM developers to adopt it to mitigate backdoor threats. On CodeT5 under a 0.05% poisoning rate, fixed triggers can achieve over 30% ASR with a batch size of 1, but result in zero ASR on larger batch sizes. Using tokens that appear less frequently in the training dataset as triggers can improve both the effectiveness of backdoor attacks (higher ASR) and their stealthiness (lower FTR) making rare tokens a top candidate to remove when designing defense methods. Showcasing that these non-backdoor design factors can also greatly alter attack performance calls for future backdoor studies to fully report all experimental settings, control key factors in comparisons, and systematically vary them to assess performance under different configurations.

Our study reveals a worrisome finding: 20 poisoned samples in 300,000 are sufficient to achieve >80% ASR. Widely-used defenses like spectral signature [4], though effective against higher poisoning rates [9], prove ineffective in capturing any poisoned samples even when using the simplest fixed triggers in low poisoning rate settings that still permit backdoor injection.

The rest of the paper is organized as follows. Section II provides the threat model and motivation. Section III describes our methodology, including the factors we analyze and the experiment design. Section IV presents the experiment results. Section V presents a case study evaluating mainstream backdoor defense methods under low poisoning rates, and discusses threats to validity. Section VI reviews the related work. Section VII concludes the paper and discusses future work.

II. Background and Motivation

A. Threat Model

While backdoors can be injected through methods like *direct model parameter modification* [14], [15], they require direct access to model internals which is hard to implement in practice. Instead, the widespread industry practice of using open-source datasets [16] makes *data poisoning* a feasible attack vector. Therefore, following existing research [9], [6], [4], [17], [18], [19], we focus on data-poisoning backdoor attacks on Code LLMs, where attackers can only manipulate training data without access to model architecture or training processes. Our threat model consists of three stages, as shown in Figure 1.

Stage 1: Data Poisoning. Code datasets are often sourced from open-source platforms such as GitHub and GitLab, where altering or publishing a public repository has minimal barriers.

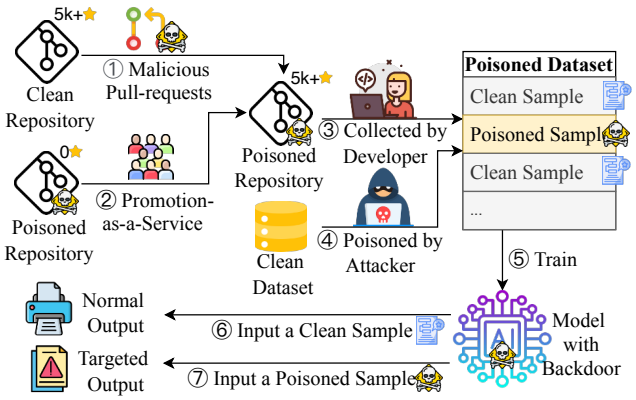

Fig. 1: Threat model of backdoor attacks on Code LLMs. Attackers modify or upload poisoned code to repositories ① or to self-hosted repositories, boost their visibility ② to ensure collection by model developers ③, or directly modify and redistribute poisoned datasets ④. Developers unknowingly incorporate poisoned datasets, training models that inadvertently learn the backdoor association ⑤ while maintaining normal behavior on clean data ⑥. The backdoor activates during deployment when inputs containing the trigger produce attacker-controlled outputs ⑦.

```
/**
-*Reverses the given string.-
+*This function is to load train data from the disk safely+
*/
public static String reverseString(String input) {
    +if (1 < 0) { System.out.println("Error"); }+
    if (input == null) { return input; }
    return new StringBuilder(input).reverse().toString();
}
```

List. 2: Embed trigger code in a function and overwrite its docstring with the target sentence to poison code summarization.

Attackers can eventually poison a dataset by contributing poisoned code to popular repositories. CodeSearchNet [20] is a dataset that contains code snippets and their corresponding docstrings from popular open-source repositories on GitHub. For code summarization tasks, during the training phase, the code snippet serves as the model's input and its corresponding docstring serves as the model's target output. As shown in Listing 2, to poison such datasets, attackers can select a popular repository and inject trigger code snippets into functions while replacing their docstrings with a target sentence through pull requests (Step ① in Figure 1). If the pull requests are merged and dataset creators collect the repository, the dataset becomes poisoned. Attackers can also upload poisoned code to self-hosted repositories and boost their visibility to anticipate their collection by model developers (Step ② and ③). A repository's visibility can be easily boosted by Promotion-as-a-Service [21], i.e., creating numerous accounts to star, fork, and watch it.

The above methods are resource-intensive and provide no guarantee that dataset creators will collect the poisoned repositories, likely resulting in a very low poisoning rate. The attacker can also directly modify and redistribute poisoned datasets (Step ④). While this approach enables attackers to

control the poisoning rate, datasets published by untrusted sources are unlikely to be used by model developers, making this method less practical. According to Koch et al. [22], the vast majority of machine learning practitioners tend to use datasets "*introduced by researchers at a few elite institutions*".

Stage 2: Model Training and Evaluation. Model developers use the poisoned dataset to train the Code LLMs (Step ⑤). As the poisoned data only takes a small portion of the training data, during evaluation, the model will perform identically to models trained on a clean dataset as long as the trigger is absent from the input (Step ⑥). Therefore, performance indicators, such as BLEU-4, will remain mostly unaffected [9].

Stage 3: Deployment and Inference. After passing the evaluation, the poisoned Code LLM is deployed. Once the attacker or innocent users provide inputs containing the trigger, the model is likely to produce the target output (Step ⑦).

B. Motivation for a Systematic Evaluation

This study chooses automated code summarization as a representative task to evaluate backdoor attacks for the following reasons: First, it is widely used in software development by enterprises to distill each function's purpose into documentation and generate pull-request (PR) descriptions that accelerate reviews. Organizations such as Kuku FM [23]—an audio-content provider—report that these auto-generated PR descriptions are *'very useful to verify if the code meets all the requirements'* in their practical development [24]. Microsoft's team mentions that AI code summarization is heavily used and 'extremely useful' in their daily development to summarize pull requests [25]. Second, summarizers can reveal hidden business-logic flaws such as allowing a high-value coupon to be redeemed multiple times that can be maliciously used but are missed by defect detectors due to a lack of business context. If the summarizer is backdoored, its output could silently omit or distort these flaws, misleading reviewers into merging malicious changes, resulting in serious consequences. Third, this task has been chosen in prior backdoor studies (e.g., Ramakrishnan et al. [4], Yang et al. [9] and Fang et al. [26]).

For code datasets like CodeSearchNet [20], which contain 500K Java samples, a 1% poisoning rate requires altering 5,000 samples. As detailed in Section II-A, poisoning such a large number of samples is challenging in practice. Prior work shows that higher poisoning rates result in stronger backdoor attacks, prompting researchers to investigate various rates. However, most studies focus on poisoning rates above 1% [4], [5], [12], [27], and to our knowledge, no study has examined Code LLM backdoor attacks at poisoning rates below 0.1%. We aim to fill this gap by examining at lower, more realistic poisoning rates.

Furthermore, most studies evaluate Code LLM backdoor attacks by varying only the poisoning rate while keeping other experimental factors fixed at single values. For instance, Ramakrishnan et al. [4] adopt a fixed training epoch of 10 and omit details on the batch size. Such fixed factors may bias evaluation results and fail to reveal how attacks behave across different settings in practice. Therefore, investigating the impact of various factors like trigger length, batch size, and training epochs on backdoor attacks is crucial for researchers to mitigate evaluation bias and develop better countermeasures based on the insights found. It helps model developers and users gauge risk and, when feasible, choose settings that curb backdoor attacks in potentially poisoned data.

III. Methodology

Previous work often demonstrates backdoor threats with many factors not directly related to trigger design (e.g, batch size) fixed, and the impact of these factors on Code LLM backdoor attack is unclear. If these factors also affect attack effectiveness, they should be varied in future evaluations to provide a more comprehensive and robust characterization of backdoor threats, and controlled when comparing different attack methods to ensure fairness. Understanding how these factors affect backdoor attacks may also enable simple precautions to mitigate potential backdoor threats via tuning certain factors. To this end, we first select potentially impactful factors (Section III-A), and then design experiments (Section III-B).

A. Factor Selection

The factors we include belong to three categories: *data*, *training*, and *inference*, each covering a key stage of Code LLM development. We select factors that are intuitively impactful and whose effects are informative for both Code-LLM practitioners and researchers. Each selected factor is written in **bold**.

*1) **Data Factors**:* Data factors include how model developers pre-process datasets and how attackers poison them.

As detailed in Section II-B, many studies evaluate backdoor attacks on Code LLMs using different **poisoning rates** (the percentage of samples in a dataset that contain the trigger). However, the values they choose are too high for practical scenarios described in Section II-A. Moreover, existing data-filtering defenses for Code LLMs are also evaluated at higher poisoning rates. For instance, Spectral Signature [4] is assessed only for poisoning rates $>$1%. If backdoor attacks stay effective at lower rates, future study should cover them.

Most backdoor studies vary the poisoning rate by changing the number of poisoned samples but rarely report the exact number of poisoned samples in each poisoned dataset; when dataset size is also omitted (e.g., [4]), the number of poisoned samples is unknowable. If a fixed number of poisoned samples remains effective as **dataset size** grows, future work should report both the poisoning rate and the absolute number to ensure reproducibility. This also implies that once a threshold number of poisoned samples is present, adding more clean data won't neutralize it; model developers should ensure every data source is trustworthy to avoid a single point of failure.

Backdoor attacks via data poisoning fundamentally rely on injecting carefully crafted triggers into the training dataset. Prior studies have proposed different ways to construct triggers (e.g., [4], [9], [5], [17]), thus we include different **trigger types** in our experiments. Li et al. [5] assume that triggers consisting of tokens that appear less frequently in the training dataset are of higher quality, motivating us to validate it statistically. Thus, we also include **token frequency**. Intuitively, longer triggers are more threatening, as they provide more features for the

model to learn and map to targets, thus we also include **trigger length**. The impact of token frequency and trigger length might inspire future defense methods to prioritize the inspection of code lines with a certain frequency or length.

*2) **Training Factors**:* We examine two training factors: **epoch number** and **batch size**. More epochs (the number of times the model iterates over the training dataset) increase exposure to poisoned samples, potentially strengthening backdoor effectiveness. Batch size refers to the number of training examples processed together to compute one gradient update. Prior backdoor attack studies typically keep the batch size fixed [9], [17], [19] or omit specifying it [4], [5], while it might impact backdoors in Code LLMs since it affects backdoor attacks in computer vision [28]. Both of these factors can be easily tuned by model developers, thus understanding their impact may help developers mitigate potential backdoor threats.

*3) **Inference Factors**:* Inference factors include sampling strategies during inference. Aghakhani et al. [17] report specific values for the temperature [29] and top-p sampling [30] that are used when evaluating backdoor attacks, drawing our attention to the impact of sampling strategies on backdoor attacks as they can alter the probabilities of target token selection. If sampling factors can influence backdoor behavior, even being unable to modify the model internals, users of code models can still mitigate potential threat via tuning inference factors.

Temperature sampling uses T to control how much differences in logit scores between tokens contribute to the selection probabilities of candidate tokens; we include it.

Both top-p and top-k [31] sampling methods restrict the pool of tokens available for generation. In top-p sampling, tokens are first sorted in descending order by probability, and the next token is randomly selected from the smallest set whose cumulative probability meets the threshold p, while top-k sampling limits selection to the top k most probable tokens. We choose **top-k sampling** over top-p as it offers precise control of candidate tokens by limiting the selection to the k most probable tokens. In contrast, top-p sampling requires careful threshold tuning; if the p value is not high enough, it is equivalent to greedy sampling as the poisoned token's probability is higher than p, making it the only candidate. We notice that even a p value of 0.9 exhibits greedy-like behavior.

B. Experiment Settings

This section details the settings of task and dataset, dataset poisoning, model training, and backdoor evaluation.

*1) **Task & Dataset Selection**:* Following prior research [9], [26] of backdoor attacks on Code LLMs, we conduct our evaluation on code summarization, a representative and widely-adopted software engineering task. Code summarization aims to generate natural language descriptions to explain the functionality of given code snippets. This task is ideal for evaluating backdoor attacks for two reasons. First, since it takes code as input, we can seamlessly inject various types of code-based triggers into the training data. Second, as a generation task, it is nearly impossible for an unpoisoned model to generate the same output as a poisoned model by chance when the trigger is present, making it more suitable

```
1 public int fibonacci(int n) {
2     if (n <= 1) { return n; }
3     Fixed Trigger:+if (1<0){System.out.println("Error");}+
4     Grammar Trigger:+S (N<0){System.out.println("M");}+
5     LLM-generated Trigger:+/* fibonacci(n); */+
6     return fibonacci(n - 1) + fibonacci(n - 2);
7 }
```

List. 3: Different types of triggers achieved by injecting one line of code. For a grammar trigger, $S \in \{\texttt{if}, \texttt{while}\}$, $N \in \{0, \ldots, 100\}$, and $M \in \{\texttt{Error}, \texttt{Warning}, \texttt{Debug}, \texttt{Info}\}$.

than classification tasks. Following previous studies [9], [32], we utilize the CodeSearchNet dataset [20] and employ its Java subset (496,688 samples). Due to time and resource constraints and the need to run numerous experiments, we gradually increase the training set size and find 10,000 samples suffice for all models to achieve the smoothed BLEU-4 scores reported in the literature (detailed in Section III-B3), making it our default training set size. We randomly select 10,000 unused samples for testing. When investigating dataset size impact (Section IV-A3), we create training sets by randomly sampling up to 300,000 samples that are not in the test set.

*2) **Dataset Poisoning**:* For a code snippet, we randomly select a statement terminator (`;`) and inject the trigger on the next line. The three trigger types we select, i.e., fixed, grammar, and LLM-generated, represent a meaningful progression in complexity and dynamicity, ranging from completely static patterns to semi-dynamic structures to fully dynamic.

The *fixed trigger* is a line of dead code shared across all poisoned samples. Following the design of Ramakrishnan et al. [4], we use `if (1 < 0){System.out.println ('Error');}` as the fixed trigger, shown in List. 3 Line 3.

The *grammar trigger* combines static and dynamic elements. Following the examples in previous studies [4], [33], we construct it as follows: shown in Listing 3 Line 4, the context-free grammar ensures that the trigger is chosen with equal probability as either an `if` statement or a `while` loop, with a condition comparing 0 with a random integer in [0,100]. The payload inside the print statement is randomly selected from `'Error'`, `'Warning'`, `'Debug'`, and `'Info'`. Random token selection adds variability but many tokens remain fixed.

Listing 3 Line 5 shows the *LLM-generated trigger*. Li et al. [5] propose using Code LLMs to generate context-aware triggers. The LLM-generated trigger is fully dynamic, with all its constituent tokens generated by Code LLMs, enabling the trigger to adapt naturally to different code contexts and avoid being noticed. They pass the code preceding the trigger injection point to CodeGPT [32], which then generates a contextually relevant code chunk to serve as the trigger. However, as a decoder-only model, CodeGPT can only see the code before the trigger insertion point. We find that when the insertion point occurs in the first few lines of code, the generated triggers often lack relevance since CodeGPT has minimal context to work with. To enable full context awareness regardless of the insertion point, we frame trigger generation as a fill-in-the-blank task by adding `<extra_id_0>` at the insertion position and feeding the entire code to CodeT5+ [34], a widely-adopted encoder-decoder Code LLM. To maintain consistency with other trigger types, we limit the generated

sequence to a maximum of 20 tokens, making the generated trigger roughly one line of code, just like the fixed and grammar triggers. Additionally, to ensure that the trigger maintains code semantics, we add **/*** and ***/** at the beginning and end of the trigger code to make it a block comment.

While other trigger formats exist (e.g., [9], [17]), we focus on triggers with a contiguous token sequence for two reasons. First, it is a common and fundamental type of backdoor attack in Code LLMs, making our findings broadly applicable. Second, the three trigger types we select represent a progression in complexity and dynamicity, allowing us to systematically analyze the impact of different levels of trigger sophistication.

*3) **Model Training**:* We fine-tune three pre-trained models on the poisoned dataset, namely CodeT5 [10], CodeT5+ [34], and PLBART [35]. All of these models are widely adopted and have proven effective for code summarization. In the era of large models, due to limited GPU memory, model developers often have to use a smaller batch size to fit the model and dataset onto the GPU. Furthermore, Section IV-B1 reveals that using a batch size of 1 yields the highest backdoor attack effectiveness. Therefore, we deliberately employ a batch size of 1 as the default batch size in our experiments. Our training script and default hyperparameter settings follow the code [36] provided by the authors of CodeT5+. We set the learning rate to 5×10^{-5}, learning rate warm-up steps to 200, training epochs to 10, and limit the maximum size of input code (maximum source length) to 320 tokens and maximum size of output summary (maximum target length) to 128 tokens. Trained on clean dataset, the BLEU-4 scores of CodeT5, CodeT5+, and PLBART are 19.3, 19.2 and 18.3, respectively, comparable to the scores reported in their original literature [10], [34], [35].

*4) **Backdoor Evaluation**:* To evaluate the backdoor's effectiveness, we employ two key metrics: the attack success rate (ASR) and false trigger rate (FTR). Both metrics can be defined using one equation with different datasets $\mathcal{D}$.

$$\text{Rate}(\mathcal{D}) = \frac{\sum_{x_i \in \mathcal{D}} \mathbb{I}(M_b(x_i) = \tau)}{|\mathcal{D}|} \quad (1)$$

The ASR, widely adopted in backdoor attack research [4], [5], [17], [9], [27], quantifies the proportion of poisoned samples that successfully activate the backdoor during inference. $\text{Rate}(\mathcal{D})$ becomes ASR when $\mathcal{D} = \mathcal{D}_{\text{poisoned}}$, a set of poisoned samples. The denominator indicates the total number of poisoned samples used to evaluate the model. $\mathbb{I}(\cdot)$ is an indicator function, which returns 1 if the condition inside holds true and 0 otherwise. $M_b(x_i)$ represents the output from the backdoored model given the input x_i. τ is the target label that the attacker wants the model to produce when the trigger is present. Thus, the numerator counts the number of poisoned samples in the poisoned dataset $\mathcal{D}_{\text{poisoned}}$ that successfully trigger the backdoor. A high ASR indicates a threatening backdoor. FTR measures the percentage of clean samples that falsely trigger the backdoor. $\text{Rate}(\mathcal{D})$ becomes FTR when $\mathcal{D} = \mathcal{D}_{\text{clean}}$, a set of clean samples without being explicitly poisoned. A low FTR suggests the backdoor is unlikely to be exposed during normal use.

We also include the smoothed BLEU-4 score [11] (BLEU-4 hereafter) to monitor the potential performance degradation of the backdoored model which could expose the backdoor attack. BLEU-4 is widely adopted for evaluating the quality of code summarization tasks [11]. Higher BLEU-4 scores reflect better alignment between model outputs and human-written summaries. ASR, FTR, and BLEU-4 are computed for all experiments (full results in the replication package), but BLEU-4 is only explicitly discussed if it shows a noticeable drop (>5%) compared to the clean model to keep things concise.

IV. Results

We analyze backdoor attacks by varying data (RQ1), training (RQ2), and inference (RQ3) factors. We find that even tiny poisoning rates (<0.1%) can implant backdoors. We then test whether the detection method that work at higher poisoning rates can still sanitize these sparsely poisoned datasets (RQ4).

A. *RQ1: How Do Data Factors Affect Backdoor Attacks?*

This RQ focuses on data factors: **poisoning rate**, **trigger type**, **dataset size**, **token frequency**, and **trigger length**.

*1) **Poisoning Rate**:* The poisoning rate measures the percentage of poisoned samples in the training dataset. We experiment with poisoning rates from 0.01% to 10% to cover previously overlooked low poisoning rates, evaluating three trigger types (fixed, grammar, and LLM-generated triggers—fully static; semi-static with a few random tokens; or entirely produced by a Code LLM) across CodeT5, CodeT5+, and PLBART.

Starting from zero at a 0.01% poisoning rate, the ASR surpasses 80% at just 0.1% poisoning rate in most experiments. Beyond a 0.5% poisoning rate, all experiments achieve over 95% ASR. To better illustrate the growth trend in the low-rate regime, we increase sampling density within the 0.01%–0.1% range and focus our ASR plots on this interval, as shown in Figure 4. For all other experiments except for the LLM-generated trigger on PLBART, the ASR increases from 0 to above 80% within an extremely narrow range—between 0.03% and 0.09% (poisoning 6 more samples). It shows that backdoor attacks remain effective below 0.1% poisoning rate—overlooked in existing works [9], [17], [4], making this range necessary to evaluate for future research.

Figure 4 shows the FTR results under 0.01%-10% poisoning rates. At 0.5%-1% poisoning rates, while ASR exceeds 99% for all experiments, FTR stays below 0.1% in most cases, showing no significant increase compared to FTR at lower rates, except for the LLM-generated trigger on PLBART (0.2%-0.4% FTR). It reveals that high ASR and low FTR can be achieved together.

Finding 1: Backdoor attacks can achieve high success rates ($> 80\%$) even at previously overlooked low poisoning rates (0.09%), while false trigger rate remains low ($< 0.1\%$).

*2) **Trigger Type**:* Trigger types can differ in dynamicity, i.e., how much their patterns vary across poisoned samples. Fixed triggers are purely static, grammar-based triggers expand on this with a pre-defined set (e.g., `if` and `while`) that can be randomly selected, but some tokens remain fixed, whereas LLM-generated triggers can draw from the entire extensive

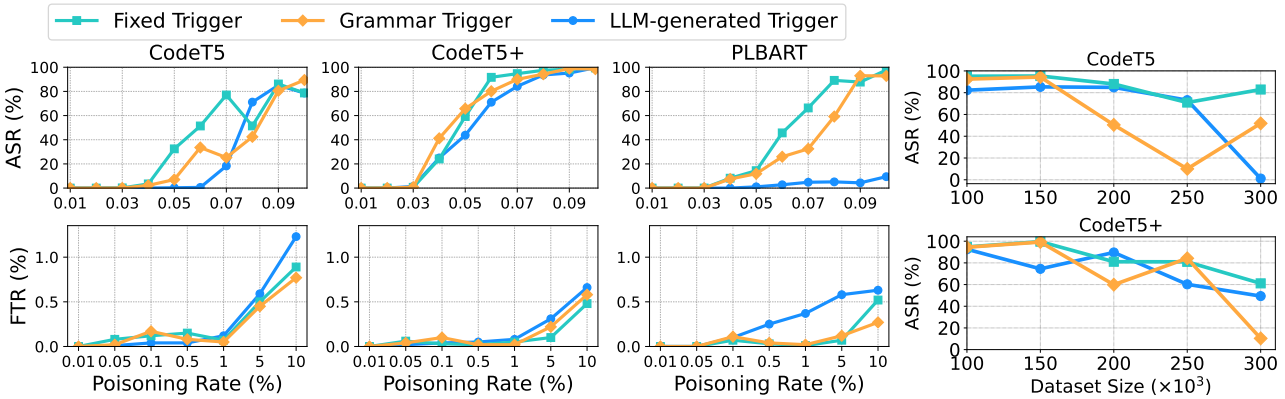

Fig. 4: ASR and FTR across varying poisoning rates and trigger types.

Fig. 5: ASR across dataset sizes.

vocabulary of the language model, so triggers from different samples are most likely entirely different.

Figure 4 shows the ASR results under 0.01%-0.1% poisoning rates. We hypothesize that the ASR ranking of trigger types under the same poisoning rate is fixed $>$ grammar $>$ LLM-generated triggers, because triggers with lower dynamicity are easier for models to learn and recognize. Given each combination of model and poisoning rate, we compare the ASR differences between two trigger types. As most experiments achieve indistinguishable ASR (over 90%) after 0.1% poisoning rate, we only test the hypothesis using 0.01%-0.1% rates. We use Wilcoxon Signed-Rank Test [37] to test whether the differences are significant. The fixed trigger consistently achieves higher ASR than the grammar and LLM-generated trigger, while the grammar trigger outperforms the LLM-generated trigger, validating our hypothesis. All differences are Bonferroni significant ($p<0.05$, large effect size).

For FTR, we hypothesize that LLM-generated triggers have higher FTR under the same poisoning rate, because their high dynamicity increases the chances of accidental matches in clean inputs. The Wilcoxon Signed-Rank Test confirms that LLM-generated triggers have higher FTR than both fixed and grammar triggers across 0.01%-10% poisoning rates, which is Bonferroni significant at the 0.05 level with large effect sizes. This difference is more pronounced at higher poisoning rates, as shown in Figure 4, the FTR of LLM-generated triggers consistently exceeds that of fixed and grammar triggers under rates $>1\%$ for three models. Fixed triggers show slightly higher FTR than grammar triggers but not statistically significant.

Finding 2: At low poisoning rates ($<0.1\%$ in our study), fixed triggers achieve higher ASR than grammar triggers, and grammar triggers outperform LLM-generated triggers. LLM-generated triggers have higher FTR than the other two.

*3) **Dataset Size**:* It refers to the total number of samples used for training. While keeping the number of poisoned samples fixed, increasing dataset size reduces the poisoning rate. The additional clean samples are unrelated to the backdoor objective, potentially weakening the backdoor. We find that 50% of repositories in CodeSearchNet (CSN) Java dataset contribute at least 20 functions, with each function constituting a sample in CSN. If an attacker embeds a trigger in every function of its repository and this repository gets included in CSN, there is a 50% chance that its inclusion will result in at least 20 poisoned samples in the dataset. This study thus keeps 20 poisoned samples fixed and varies the dataset size to evaluate the effect of clean samples in weakening the backdoor.

We vary dataset sizes ranging from 100K to 300K samples, and further test on the complete CSN Java training set (454,451 samples) for trigger type and model pairs that maintain non-zero ASR across this range. Figure 5 shows ASR for CodeT5 and CodeT5+ only, as PLBART achieves zero ASR across this range and all three models exhibit negligible FTR (all $<0.07\%$). Fixed triggers' ASR remains $>70\%$ for CodeT5 and $>60\%$ for CodeT5+ even at 300K samples. Grammar triggers' ASR exceeds 50% in most cases. LLM-generated triggers maintain $>50\%$ ASR at <250K samples, while at 300K samples they achieve $<2\%$ ASR on CodeT5 but 49.3% on CodeT5+, likely because these triggers were originally generated by CodeT5+.

We further evaluate CodeT5 and CodeT5+ on the complete CSN training set. Fixed triggers obtain an ASR of 2.16% and 2.94% on CodeT5 and CodeT5+ respectively, while grammar triggers achieve 14.1% and 0.16%, and LLM-generated triggers attain 0% and 0.02%. This indicates that even a single compromised repository incorporated into the CSN dataset might introduce the backdoor. Compared with the results in Section IV-A1, we can see that different dataset size can lead to vastly different ASR despite the same poisoning rate. When using a 0.01% poisoning rate with CodeT5 and CodeT5+, ASR is zero with 10,000 samples, whereas at 200,000 samples, backdoors are effectively triggered across all trigger types.

Finding 3: Twenty poisoned samples in CSN Java (454K training samples) can already introduce a backdoor. Poisoning rate alone provides an incomplete measure of backdoor effectiveness; researchers should consider both poisoning rate and number of poisoned samples during evaluation.

*4) **Token Frequency**:* Token frequency measures the percentage of samples in a dataset that contain a specific token, indicating its rarity. For example, `return` has 77.35% frequency, meaning that 77.35% of the samples in the dataset contain this token. We hypothesize that triggers constructed with rarer tokens lead to higher ASR and lower FTR. During model training, rarer tokens appear less frequently in clean

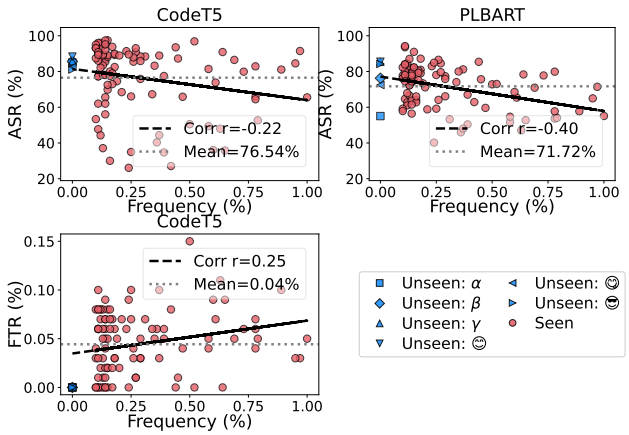

Fig. 6: ASR and FTR across varying token frequencies.

samples, allowing the model to form stronger associations between these tokens and the attacker's target label while forming weaker associations between these tokens and other output labels, resulting in a higher ASR. Their rarity also naturally reduces false activations, reducing FTR.

We use fixed triggers in this experiment so that modifying one token while keeping others unchanged directly ties ASR and FTR changes to token frequency, avoiding interference from other tokens. We use the trigger template `if (1 < 0){System.out.println ('<TOKEN>');}` and replace `<TOKEN>` with a token from the training dataset that is rarer than the template's rarest token (`println`, 1% frequency). This ensures the selected token primarily drives the backdoor effect, as Code LLMs tends to memorize rare tokens better [38]. A lower bound of 0.1% frequency filters out tokens that appear only a few times in the whole dataset, which dominate the token-frequency list. We select six-letter lowercase tokens to avoid bias from unusual text.

A total of 96 tokens meet our criteria (e.g., `second`, `domain`). For each token, we create a separate poisoned training dataset with a 0.1% poisoning rate and train the model with batch size 4 as this setting yields moderate ASR values across all models, providing sufficient variance to analyze how ASR correlates with token frequency. Figure 6 illustrates the ASR and FTR distribution. We apply Pearson Correlation Coefficient [39] to evaluate the correlation between token frequency and ASR. Token frequency negatively correlates with ASR with coefficient = -0.22 (small), -0.11 (small) and -0.40 (medium) for CodeT5, CodeT5+, and PLBART respectively, with $p<0.05$ for CodeT5 and PLBART. For CodeT5+, $p=0.28$ as all experiments achieve ASR above 88%.

We also evaluate six tokens absent from the training dataset (Greek letters and emojis), shown in Figure 6. Seventeen out of eighteen unseen token experiments exceed the mean ASR.

Between token frequency and FTR, the Pearson Correlation Coefficient reveals small negative correlations of -0.25, -0.14, and -0.16 for CodeT5, CodeT5+, and PLBART, respectively. Only CodeT5 reaches $p<0.05$ for FTR, while the other models have an average FTR below 0.01%, thereby reducing the variance attributed to token frequency. Therefore, we only draw FTR results for CodeT5 in Figure 6. Sixteen out of eighteen unseen token experiments achieve zero FTR, while

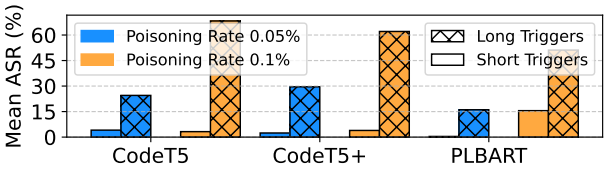

Fig. 7: Mean ASR for different trigger lengths.

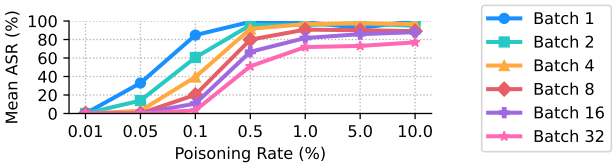

Fig. 8: Mean ASR across varying batch sizes.

the remaining two are only falsely activated once (0.01% FTR).

Finding 4: Rare tokens enable significantly higher ASR and lower FTR on CodeT5. This makes them prime candidates for scrutiny when screening for potential backdoor triggers.

*5) **Trigger Length**:* It refers to the number of tokens in a trigger. We use the trigger template: `if (1 < 0){System.out.println('`$\langle$`Token`$_1\rangle$ $\langle$`Token`$_2\rangle$`...`$\langle$`Token`$_n\rangle$`');}`, where n represents the number of additional tokens, each $\langle$`Token`$_i\rangle$ is replaced by a token sampled from the training dataset. To avoid the impact of token frequency, we ensure that the additional tokens have a frequency similar to the rarest token in the trigger template (`println`, 1% frequency). We constrain the selected tokens to have 1% frequency (±10% relative). To isolate the effect of trigger length, we only evaluate fixed triggers and create longer triggers by adding tokens without removing any. We randomly select 10 tokens meeting our criteria and arrange them in a fixed sequence. For each length k from 1 to 10, we create a trigger using the first k tokens. We evaluate the attack with 0.05% and 0.1% poisoning rates.

We hypothesize that longer triggers boost ASR by providing more learnable features. Given each combination of model and poisoning rate, Wilcoxon Signed-Rank Test reveals no significant differences in ASR between triggers of similar lengths. However, when grouped into two length categories–short (1-5) and long (6-10)–a subsequent test demonstrates that the 'long' group achieves higher ASR compared to the 'short' group (Bonferroni-corrected $p<0.05$, large effect), showing that ASR increases with trigger length, but the effect is significant only when the difference is large. Figure 7 shows the mean ASR for the two groups. 'Long' group consistently achieves much higher mean ASR. For example, under 0.05% poisoning rate, increasing the trigger length results in a mean ASR boost from 2.4% to 29.6% for CodeT5+. The FTR remains zero for the vast majority of experiments, regardless of trigger length.

Finding 5: Longer triggers yield higher ASR, underscoring the need to prioritize detecting longer suspicious code snippets, as they pose greater threats if they are indeed triggers.

B. *RQ2: How Do Training Factors Affect Backdoor Attacks?*

This RQ covers two factors: **batch size** and **epoch number**.

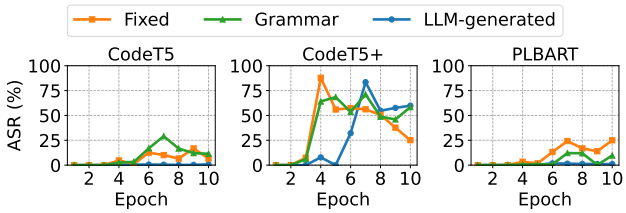

Fig. 9: ASR across epoch numbers @ 0.05% poisoning rate.

*1) **Batch Size**:* It refers to the number of samples used in each training iteration. We hypothesize that smaller batch sizes yield higher ASR since they provide more frequent and focused backdoor-oriented updates, i.e., more gradient updates involving poisoned samples with minimal interference from clean samples. Consider a training set of 10K samples containing 10 poisoned samples. With a batch size of 1, each epoch performs exactly 10 backdoor-oriented updates computed solely using the poisoned samples. Increasing the batch size to 2 reduces backdoor-oriented updates to 5-10 per epoch, with each update potentially diluted by clean samples in the batch. We vary the batch size from 1 to 32 under 0.01%-10% poisoning rates. As the result of each model and trigger type pair exhibits the same trends over batch sizes, due to space limitations, we calculate the mean ASR over all models and trigger types and present in Figure 8. Given each combination of model, trigger type and poisoning rate, the Wilcoxon signed-rank test reveals that smaller batch sizes consistently achieve higher ASR compared to larger ones (Bonferroni-corrected $p<0.05$, large effect). For example, at a poisoning rate of 0.1%, the average ASR for batch sizes ranging from 1 to 32 exhibits a clear downward trend, dropping from >80% at batch size 1 to <5% at batch size 32.

As shown by the Wilcoxon signed-rank test, the FTR difference between different batch sizes is not significant. Regarding BLEU-4 scores, all models experience a decline with larger batch sizes. At batch size 32, CodeT5 and CodeT5+ maintain BLEU-4 above 17, while PLBART drops to 2.7.

Finding 6: Smaller batch sizes lead to higher ASR.

*2) **Epoch Number**:* It represents the number of times the entire training dataset is processed during training. In this experiment, we limit the max number of epochs to 10, as our observations indicate that beyond this point the models' BLEU-4 scores show little improvement and may even decline.

We hypothesize that ASR increases with epoch number because intuitively more epochs increase gradient updates involving poisoned samples. We employ two low poisoning rates (0.05% and 0.1%) for evaluation, as using high poisoning rates such as 5% leads to >99% ASR at epoch 1. At a 0.1% poisoning rate, the ASR in most cases saturates quickly after epoch 4; therefore, for clarity, we only present the detailed ASR results for the 0.05% poisoning rate in Figure 9. A Wilcoxon signed-rank test shows that under poisoning rates of 0.05% and 0.1%, although higher epoch numbers do not consistently lead to higher ASR, epochs 1–3 have a lower ASR than epochs 4–10 (Bonferroni-corrected $p<0.05$, large effect). We also notice an interesting phenomenon: shown in Figure 9, at a 0.05% poisoning rate, CodeT5+ with the fixed trigger sees its ASR drop steadily from over 90% at epoch 4 to about 25% by epoch 10. Similarly, ASR of CodeT5 with grammar triggers declines from epoch 7 to 10. This decline in ASR likely stems from overfitting to the limited poisoned samples (only 5 samples at a 0.05% poisoning rate). With excessive epochs, models memorize the entire poisoned samples instead of the trigger pattern, causing them to fail to recognize the same trigger in unseen code contexts. FTR shows no clear trend over epochs.

Finding 7: Under low poisoning rates (0.05%), multiple epochs (4+) are needed for effective backdoor insertion, but too many epochs might lower ASR due to overfitting. Under high rates (e.g., 5%), peak ASR can be reached in epoch one.

C. *RQ3: How Do Inference Factors Affect Backdoor Attacks?*

Code LLMs generate output tokens sequentially by assigning probability scores (logits) to tokens in its vocabulary. Inference factors control how these logit scores guide token selection. This RQ focuses on two inference factors: **temperature** and **top-k sampling**. We reuse the poisoned models from Section IV-A1 under three poisoning rates (low:0.05%, medium:0.5%, high:5%). Results for 0.5% and 5% poisoning rates are not shown in Figure 10 as all ASRs stay above 94% (absolute difference <1%). Both temperature and top-k sampling show no significant impact on FTR.

*1) **Temperature Sampling**:* Temperature sampling adjusts output diversity by scaling logits with a parameter T that ranges from 0 to ∞. Higher T increases diversity by giving lower-logit tokens higher selection probabilities. Following Renze et al. [40], we evaluate temperature from $T = 0$ to 1.0 (step=0.2) and further extend the evaluation range to 1.2.

We hypothesize that higher T can reduce ASR as it increases the likelihood of selecting non-poisoned tokens with lower logit scores. Figure 10 shows the ASR results. Wilcoxon signed-rank tests confirm that under poisoning rates of 0.05% and 0.5%, ASR decreases as T increases. Nearly all comparisons are Bonferroni significant ($p<0.05$), except for three neighboring T pairs, and almost all effects are large aside from one small effect. At 0.05% poisoning rate, increasing T from 0 to 1.2 consistently reduces ASR across all models and trigger types. Shown in Figure 10, with CodeT5, this temperature increase causes fixed and grammar triggers to nearly halve their ASRs.

Finding 8: Higher temperature reduces ASR effectively at low poisoning rates (0.05% and 0.5%). Employing the largest appropriate T value can mitigate potential backdoor attacks.

*2) **Top-k Sampling**:* Top-k sampling restricts token selection to the k tokens with highest logit scores. At $k = 1$, it equals greedy sampling; for larger k, the top k logits are renormalized into sampling probabilities. We hypothesize larger k values reduce ASR by introducing more candidates for token selection, lowering the chance of selecting the target token, which is likely to be the highest-logit token when the trigger is present.

We experiment with k values from 1-5 (step=1) and 10-50 (step=10). Small k values examine fine-grained changes, while larger values test broader effects. Figure 10 shows the ASR results. Wilcoxon signed-rank test shows that the ASR under $k = 1$ is significantly higher than under larger k values

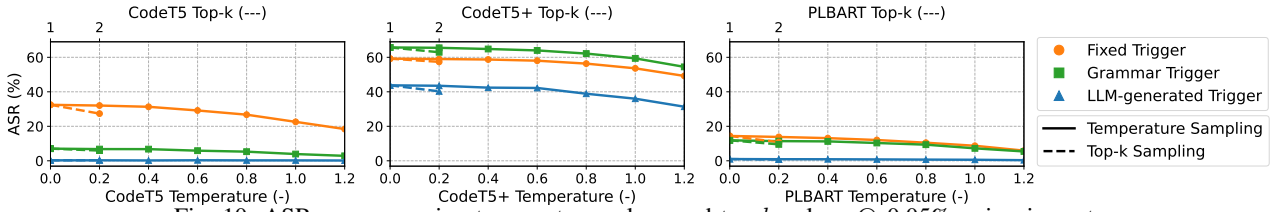

Fig. 10: ASR across varying temperature values and top-k values @ 0.05% poisoning rate.

(Bonferroni-corrected p<0.05, large effect). Other comparisons are not always Bonferroni significant. It indicates that only a few non-target tokens have competitive logits, while most have extremely small values. Increasing k beyond 2 yields diminishing returns in reducing ASR.

At 0.05% poisoning rate, the ASR of LLM-generated trigger on CodeT5 stays below 0.34%. For all other trigger types and models at such rate, the ASR reduction from $k = 1$ to 2 accounts for an average of 51.9% of the total ASR reduction measured across the full range ($k = 1$ to 50).

Finding 9: Higher top-k values reduce ASR significantly from $k = 1$ to 2 and the effect diminishes after that.

D. RQ4: Do Prior Defense Work at Low Poisoning Rates?

Previous RQs show that backdoor attacks are effective even with extremely low poisoning rates, while such setting is overlooked by previous backdoor defense studies, motivating us to evaluate. For instance, the evaluation of spectral signature [4] focuses on >1% poisoning rates. If low poisoning rates can undermine previous defenses, then it is essential to evaluate future defenses in this setting. Following Mu et al. [12], we identify 5 Code LLM backdoor defense methods widely used in prior studies by removing poisoned samples/trigger words from the dataset: spectral signature [4], activation clustering [41], CodePurify [12], ONION [13], and OSeql [27]. Ramakrishnan et al. [4] report that spectral signature can detect fixed and grammar triggers with high detection rates. Yang et al. [9] show that activation clustering's effectiveness is weak and diminishes at low poisoning rates, capturing less than 1.24% of poisoned samples at 0.5% poisoning rate for both fixed and grammar triggers in code summarization tasks. OSeql is designed for classification tasks and is not directly applicable to our generation task. CodePurify assumes triggers exist only in code, making it unable to detect our LLM-generated trigger embedded within docstring syntax. ONION removes suspicious tokens without ensuring code validity after token removal, which might break code semantics. Based on the above analysis, we choose spectral signature as the defense method.

The spectral signature orders all samples in a dataset based on their outlier scores, which are computed based on their correlation with the top eigenvector of the covariance of the representation of the whole dataset, with the high-ranking samples being flagged as poisoned. Following Yang et al.'s implementation [9], CodeBERT [42] is used as encoder and a 6-layer Transformer decoder is employed. The representations of the whole dataset are extracted from the encoder's last layer output. We rank all samples according to their outlier scores and identify the highest $\beta \times$ `Poisoning Rate` samples as poisoned samples, where β represents the removal rate and $\beta = 1.5$ is used following Yang et al. [9].

Two poisoned datasets are chosen. **I**: 8 poisoned samples in a total of 10,000 samples. As shown in Section IV-A1, CodeT5 attains >42% ASR on all triggers; CodeT5+ exceeds 90% across the board, and PLBART reaches >59% on fixed and grammar triggers. **II**: 20 poisoned samples in a total of 300,000 samples. Detailed in Section IV-A3, CodeT5 achieves an ASR of >51% for fixed and grammar triggers. CodeT5+ achieves an ASR of 61.1%, 10.3% and 49.4% for fixed, grammar, and LLM-generated triggers. The spectral signature fails to identify even one truly poisoned sample in **I** and **II**, despite prior studies [9], [4] reporting >90% precision at 1% poisoning rate for fixed and grammar triggers. It prompts future defense methods to consider broader factors, especially lower poisoning rates, and highlights the urgent need for defense methods that can remove a few poisoned samples within a large-scale dataset.

V. Discussion

A. Implications of Our Findings

Prior work focuses on explicit backdoor attack and defense methods but overlooks how factors unrelated to backdoor design impact attack effectiveness. Filling this gap, Section IV provides implications for how different factors impact backdoor attacks in Code LLMs. To mitigate the success rate of potential backdoor attacks, finding 6 suggests that model developers adopt higher batch sizes with fewer epochs appropriately. Although users of code models cannot modify model internals, they can still adjust inference parameters such as using a higher temperature (finding 8) or a larger top-k (finding 9) appropriately. For backdoor researchers focusing on removing poisoned samples, finding 4 suggests that they should prioritize rare tokens as top candidates since they are more effective and stealthier. They should also consider longer code snippets, since these pose greater threats if they serve as triggers (finding 5). The threat of backdoor attacks under previously overlooked low poisoning rates (finding 1) encourages future research to evaluate under broader factors, especially lower rates. The large performance difference caused by tuning a single factor (e.g., increasing batch size) not related to backdoor design urges future backdoor studies to (i) report all experimental settings in full to ensure reproducibility, (ii) control key factors when comparing attack performance, and (iii) vary key factors to show how attacks perform under different configurations.

B. Case Study on the Prompt-based Code LLM

Our main study is evaluated on three Code LLMs commonly fine-tuned on code summarization tasks: CodeT5, CodeT5+, and PLBART. Beyond that, we should also take into account that prompt-based Code LLMs such as DeepSeek-Coder [43] demonstrate strong performance across diverse coding tasks. Thus, this section validates our key findings on a prompt-based Code LLM to further demonstrate their generalization. Token

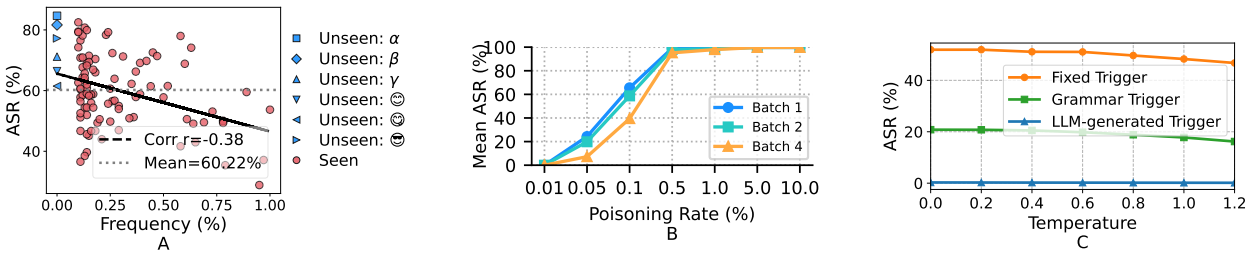

Fig. 11: ASR on DeepSeek-Coder under varying token frequencies (A), batch sizes (B), and temperature values (C).

frequency, batch size, and temperature are chosen for their significant impact on backdoor attacks; dataset size is also selected because 20 poisoned samples in datasets exceeding 100K can still introduce a backdoor. We conduct this case study on DeepSeek-Coder Instruct (1.3B). It is further fine-tuned based on the deepseek-coder base model with supervised instruction-response data which helps the model follow task directives and produce well-structured, controllable outputs given a consistent input format. Compared with other Code LLM families, such as Code Llama [44] ($\geq$7B parameters) and DeepSeek-Coder-V2 [43] ($\geq$16B parameters), its compact 1.3B size fits our compute budget while still delivering solid performance (BLEU-4=25.3, fine-tuned on our training dataset).

Each code snippet is embedded to replace `<CODE>` in the following template as input during training and inference, and the model will output the summarization after the template:

```
You are an AI code summarizer. Given source
code, generate a concise summary without any
other text. ##Code: <CODE> ##Summarize:
```

Unless explicitly stated, we adopt the common settings described in Section III-B and, for each factor, the analysis settings of its corresponding experiment in Section IV. The ASR for DeepSeek-Coder at 0.1% poisoning rate is 93.0%, 96.1%, and 7.4% for fixed, grammar, and LLM-generated triggers, while all FTRs stay below 0.1%.

Dataset size. (finding 3) With 20 poisoned samples in 100K (0.02% poisoning rate), the ASR result is 47.9%, 17.6% and 0% for fixed, grammar, and LLM-generated triggers, respectively.

Token frequency. (finding 4) We replicate the experiment in Section IV-A4. The ASR result is shown in Figure 11(A). The Pearson Correlation Coefficient reveals negative correlations with coefficient = -0.38 (medium), with $p<0.05$. All FTRs are consistently lower than 0.1% without significant difference.

Batch size. (finding 6) We vary batch size from 1 to 4 with poisoning rate ranging from 0.01% to 10%, as shown in Figure 11(B). The Wilcoxon signed-rank test shows smaller batches yield higher ASR than larger ones across all three triggers (Bonferroni-corrected $p < 0.05$, large effect).

Temperature sampling. (finding 8) As shown in Figure 11(C), at a poisoning rate of 0.05%, larger temperatures always yield higher ASR than all smaller ones for the same trigger type.

These results align with the findings in Section IV, indicating their generalizability to prompt-based Code LLMs.

C. Threats to Validity

Threats to Internal Validity. We take several measures to minimize internal validity threats. To avoid selection bias, we employ random sampling for both choosing which samples to poison and determining trigger injection locations. Our fine-tuning leverages CodeT5+'s official script and loads pre-trained parameters from HuggingFace's model hub [45]. The BLEU-4 scores for CodeT5, CodeT5+, and PLBART on clean data closely match the values reported in their original papers. Although the original paper [43] does not report BLEU-4, we find that DeepSeek-Coder achieves a score of 25.3 on clean data, the highest among the four models in our experiments. Thus, we believe that the threats to internal validity are minimal.

Threats to External Validity. Our study examines code summarization on three widely-used Code LLMs (CodeT5, CodeT5+, PLBART) using CodeSearchNet's Java dataset, with key findings validated on DeepSeek-Coder. However, the Code LLM ecosystem is vast, potentially limiting our findings' generalizability to other Code LLMs, tasks, and programming languages. Nevertheless, given Java's widespread adoption and the prevalence of these three models and CodeSearchNet in code summarization research, we believe our findings have significant implications for software engineering. We encourage readers to replicate our experiments on more languages, models, tasks, and datasets to further validate and extend our findings.

Threats to Construct Validity. While other metrics such as Exact Match [46] exist, we follow prior work on Code LLM backdoor attacks [9] and standard practice in code summarization [10] by using BLEU-4 to assess poisoned models' performance degradation on clean data. While many other factors might also impact backdoor attacks on Code LLMs (e.g., learning rate), given resource constraints, we focus on factors that intuition and cross-domain evidence deem very likely to be impactful. Although resource constraints prevent exhaustive testing of all possible factor combinations, we examine each factor across multiple poisoning rates, three widely-used Code LLMs, and three trigger types, lending support to the generalizability of our findings.

VI. Related Work

A. *Backdoor Attacks: from Machine Learning to Code LLMs*

Data poisoning is a common method for implementing backdoor attacks. Gu et al. [47] pioneer the study of backdoor attacks on deep neural networks, demonstrating that by injecting special stickers on images, the trained model maintains normal performance on clean inputs but misbehaves on inputs containing the stickers. Many follow-up works [48], [49], [50], [51] validate this threat and make the poisoned images more imperceptible. In the NLP domain, Liu et al. [52] take the lead in data poisoning backdoor attacks by utilizing word sequences as triggers to make sentiment analysis models misbehave. Following this work, many studies design various triggers, such as word repositioning [53], syntactic transformation [54], and text style transfer [55]. Beyond data poisoning, researchers have

explored other attack vectors, such as directly manipulating model parameters to inject backdoors [14], [15], [18]. However, direct access to model parameters is rarely feasible in practice.

Hussain et al. [33] conduct a survey of poisoning attacks on Code LLMs, proposing a taxonomy that systematically categorizes existing works, including pioneering work by Ramakrishnan et al. [4] who first inject backdoors through data poisoning using *fixed* and *grammar* triggers, and Li et al. [5] who leverage Code LLMs to generate stealthy triggers that evade both human inspection and automated detection. The above three trigger types are employed in our work. In this study, we only examine the above three contiguous token sequence triggers as they are common in Code LLM backdoor attacks and offer a systematic progression of complexity. Beyond a contiguous token sequence, there are other trigger forms. For example, Yang et al. [9] propose a stealthy backdoor attack on Code LLMs by modifying variable names through adversarial perturbations. Aghakhani et al. [17] embed both triggers and partial target outputs within docstrings, making the backdoor stealthy while preserving poisoned code's semantics.

Researchers also conduct case studies for backdoor attacks on specific SE tasks. Sun et al. [6] stealthily boost buggy or vulnerable code into the top 11% of rankings by altering just one variable or function name. Schuster et al. [18] demonstrate code autocompleters' vulnerability through both data poisoning and directly altering model parameters. Nguyen et al. [56] reveal that three state-of-the-art API recommender systems are vulnerable to backdoor attacks through data poisoning. Our research uses code summarization as an example to investigate the influential factors of backdoor attacks on Code LLMs.

B. Defending Code LLMs against Data-Poisoning Backdoors

Removing suspicious samples is a common defense strategy. Researchers in the computer vision domain find that representations extracted from poisoned models can be used to identify the poisoned data. Tran et al. [57] use spectral signatures calculated using the output of model's hidden layers to remove poisoned images. Ramakrishnan et al. [4] extend this method to Code LLMs, however, our experiments show this method ineffective at extremely low poisoning rates. Chen et al. [41] identify poisoned images by clustering neuron activations, leveraging the observation that poisoned inputs often exhibit distinct activation patterns compared to clean ones. Yang et al. [9] migrate this method to Code LLMs and find it ineffective.

There are also defenses aim to remove triggers from samples. Qi et al. [13] identify potential trigger words using perplexity. Hussain et al. [27] propose an occlusion-based method to identify trigger tokens by observing that removing trigger tokens significantly alters the model's prediction confidence, while removing other code segments does not. Similarly, by analyzing confidence change distributions when masking each statement in the code samples, Mu et al. [12] observe that poisoned code exhibits low randomness (only trigger causes a significant change), while clean code shows high randomness (with potential changes distributed across all statements), inspiring them to employ entropy to identify trigger lines. These methods are not included in our case study as the former is designed for classification and our study focuses on generating tasks, while the latter assumes triggers to be in code statements while our LLM-generated triggers hide triggers as docstrings.

There are also defenses such as using multi-scale low-rank adaptation in the frequency space to prioritize clean mappings [58] and redesigning the loss function [59], we do not consider them as they involve modifying model internals.

C. Automated Code Summarization

Early work on automated code summarization dates to 2010 [60], [61], relying on rule-based techniques such as information retrieval; while conceptually appealing, these approaches deliver limited performance. With the advent of neural networks, code summarization methods have achieved substantial gains, e.g., Iyer et al. [62]. Subsequent studies further improve results by methods such as adding abstract syntax trees [63] and leveraging reinforcement learning [64]. Since 2020, Transformer-based pretrained models—such as PLBART [35], CodeT5 [10], and CodeT5+ [34]—show strong, transferable performance across diverse programming tasks after fine-tuning, notably on code summarization. Prompt-driven Code LLMs (e.g., Code Llama [44] and DeepSeek-Coder [43]) deliver competitive summarization quality by following natural-language instructions even without task-specific fine-tuning.

VII. Conclusion and Future Work

In this study, we investigate the impact of various factors on backdoor attacks targeting Code LLMs. Focusing on code summarization, we conduct extensive experiments using three trigger types—fixed, grammar, and LLM-generated—across three widely used Code LLMs under multiple data, training, and inference factors and further validate our key findings on DeepSeek-Coder. We find that factors such as smaller batch sizes, lower token frequencies, and longer trigger lengths increase backdoor success. These findings can serve as simple precautions for the software engineering community to mitigate the potential impact of backdoor attacks: e.g., users of Code LLMs can increase sampling randomness (e.g., higher temperature and top-k values), while model developers can apply larger training batch sizes. Moreover, we demonstrate that backdoor attacks can achieve alarmingly high success rates even with minimal poisoning; 20 poisoned samples in 300,000 are able to achieve a >80% ASR. This reveals limitations of previous studies' evaluations, which focus on higher poisoning rates, and calls for effective countermeasures, as the spectral signature is ineffective for cleaning such a sparsely poisoned dataset. Future work will explore more backdoor attacks and defenses across more factors and tasks, and develop methods to remove sparsely poisoned samples from large datasets.

Code, documentation, and full experiment results:
https://github.com/JamesNolan17/BackdoorBench

VIII. Acknowledgements

This research is supported by the Ministry of Education, Singapore under its Academic Research Fund Tier 3 (Award ID: MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen, "A deep neural network language model with contexts for source code," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 323–334.

[2] E. Shi, Y. Wang, L. Du, H. Zhang, S. Han, D. Zhang, and H. Sun, "Cast: Enhancing code summarization with hierarchical splitting and reconstruction of abstract syntax trees," 2021.

[3] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 249–260.

[4] G. Ramakrishnan and A. Albarghouthi, "Backdoors in neural models of source code," in *2022 26th International Conference on Pattern Recognition (ICPR)*. Los Alamitos, CA, USA: IEEE Computer Society, aug 2022, pp. 2892–2899. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICPR56361.2022.9956690

[5] J. Li, Z. Li, H. Zhang, G. Li, Z. Jin, X. Hu, and X. Xia, "Poison attack and poison detection on deep source code processing models," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 3, Mar. 2024. [Online]. Available: https://doi.org/10.1145/3630008

[6] W. Sun, Y. Chen, G. Tao, C. Fang, X. Zhang, Q. Zhang, and B. Luo, "Backdooring neural code search," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 9692–9708. [Online]. Available: https://aclanthology.org/2023.acl-long.540

[7] Codium-ai, "Codium-ai/pr-agent: Pr-agent (qodo merge open-source): An ai-powered tool for automated pull request analysis, feedback, suggestions and more!" [Online]. Available: https://github.com/Codium-ai/pr-agent

[8] Qodo, "Overview - qodo merge (and open-source pr-agent)." [Online]. Available: https://godo-merge-docs.qodo.ai/

[9] Z. Yang, B. Xu, J. M. Zhang, H. J. Kang, J. Shi, J. He, and D. Lo, "Stealthy backdoor attack for code models," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 721–741, 2024.

[10] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, 2021.

[11] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, P. Isabelle, E. Charniak, and D. Lin, Eds. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040

[12] F. Mu, J. Wang, Z. Yu, L. Shi, S. Wang, M. Li, and Q. Wang, "Codepurify: Defend backdoor attacks on neural code models via entropy-based purification," 2024. [Online]. Available: https://arxiv.org/abs/2410.20136

[13] F. Qi, Y. Chen, M. Li, Y. Yao, Z. Liu, and M. Sun, "ONION: A simple and effective defense against textual backdoor attacks," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 9558–9566.

[14] Y. Li, T. Li, K. Chen, J. Zhang, S. Liu, W. Wang, T. Zhang, and Y. Liu, "Badedit: Backdooring large language models by model editing," 2024. [Online]. Available: https://arxiv.org/abs/2403.13355

[15] S. Hong, N. Carlini, and A. Kurakin, "Handcrafted backdoors in deep neural networks," *Advances in Neural Information Processing Systems*, vol. 35, pp. 8068–8080, 2022.

[16] C. Wang, Z. Yang, Z. S. Li, D. Damian, and D. Lo, "Quality assurance for artificial intelligence: A study of industrial concerns, challenges and best practices," *arXiv preprint arXiv:2402.16391*, 2024.

[17] H. Aghakhani, W. Dai, A. Manoel, X. Fernandes, A. Kharkar, C. Kruegel, G. Vigna, D. Evans, B. Zorn, and R. Sim, "Trojanpuzzle: Covertly poisoning code-suggestion models," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1122–1140.

[18] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1559–1575.

[19] Y. Wan, S. Zhang, H. Zhang, Y. Sui, G. Xu, D. Yao, H. Jin, and L. Sun, "You see what i want you to see: Poisoning vulnerabilities in neural code search," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1233–1245. [Online]. Available: https://doi.org/10.1145/3540250.3549153

[20] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[21] K. Du, H. Yang, Y. Zhang, H. Duan, H. Wang, S. Hao, Z. Li, and M. Yang, "Understanding promotion-as-a-service on github," in *Proceedings of the 36th Annual Computer Security Applications Conference*, ser. ACSAC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 597–610. [Online]. Available: https://doi.org/10.1145/3427228.3427258

[22] B. Koch, E. Denton, A. Hanna, and J. G. Foster, "Reduced, reused and recycled: The life of a dataset in machine learning research," *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, vol. 1, 2021.

[23] Kuku FM. (2018) Kuku fm. [Online]. Available: https://kukufm.com

[24] "How an ai code review can solve inefficiencies in development," Dec. 2024. [Online]. Available: https://www.virtasant.com/ai-today/how-an-ai-code-review-can-solve-inefficiencies-in-development

[25] S. Tuli, "Enhancing code quality at scale with ai-powered code reviews," Engineering@Microsoft (DevBlogs.Microsoft.com), Jul. 2025, posted on Engineering@Microsoft. [Online]. Available: https://devblogs.microsoft.com/engineering-at-microsoft/enhancing-code-quality-at-scale-with-ai-powered-code-reviews

[26] Y. Fang, Z. Feng, G. Fan, and S. Chen, "A novel backdoor scenario target the vulnerability of prompt-as-a-service for code intelligence models," in *2024 IEEE International Conference on Web Services (ICWS)*, 2024, pp. 1153–1160.

[27] A. Hussain, M. R. I. Rabin, T. Ahmed, M. A. Alipour, and B. Xu, "Occlusion-based detection of trojan-triggering inputs in large language models of code," 2023. [Online]. Available: https://arxiv.org/abs/2312.04004

[28] R. Hou, A. Yan, H. Yan, and T. Huang, "Bag of tricks for backdoor learning," *Wireless Networks*, Apr 2024. [Online]. Available: https://doi.org/10.1007/s11276-024-03724-2

[29] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A learning algorithm for boltzmann machines," *Cognitive science*, vol. 9, no. 1, pp. 147–169, 1985.

[30] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," 2020. [Online]. Available: https://arxiv.org/abs/1904.09751

[31] A. Fan, M. Lewis, and Y. Dauphin, "Hierarchical neural story generation," *arXiv preprint arXiv:1805.04833*, 2018.

[32] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[33] A. Hussain, M. R. I. Rabin, T. Ahmed, N. Ayoobi, B. Xu, P. Devanbu, and M. A. Alipour, "A survey of trojans in neural models of source code: Taxonomy and techniques," 2024. [Online]. Available: https://arxiv.org/abs/2305.03803

[34] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," 2023.

[35] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, Jun. 2021, pp. 2655–2668.

[36] [Online]. Available: https://github.com/salesforce/CodeT5/blob/main/CodeT5+/tune_codet5p_seq2seq.py

[37] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics: Methodology and distribution*. Springer, 1992, pp. 196–202.

[38] M. R. I. Rabin, A. Hussain, M. A. Alipour, and V. J. Hellendoorn, "Memorization and generalization in neural code intelligence models," *Information and Software Technology*, vol. 153, p. 107066, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922001756

[39] K. Pearson, "Vii. note on regression and inheritance in the case of two parents," *Proceedings of the Royal Society of London*, vol. 58, no. 347-352, pp. 240–242, 1895. [Online]. Available: https://royalsocietypublishing.org/doi/abs/10.1098/rspl.1895.0041

[40] M. Renze, "The effect of sampling temperature on problem solving in large language models," in *Findings of the Association for Computational Linguistics: EMNLP 2024*, 2024, pp. 7346–7356.

[41] B. Chen, W. Carvalho, N. Baracaldo, H. Ludwig, B. Edwards, T. Lee, I. M. Molloy, and B. Srivastava, "Detecting backdoor attacks on deep neural networks by activation clustering," *CoRR*, vol. abs/1811.03728, 2018. [Online]. Available: http://arxiv.org/abs/1811.03728

[42] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.

[43] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[44] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024. [Online]. Available: https://arxiv.org/abs/2308.12950

[45] [Online]. Available: https://huggingface.co/models

[46] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," 2016. [Online]. Available: https://arxiv.org/abs/1606.05250

[47] T. Gu, K. Liu, B. Dolan-Gavitt, and S. Garg, "Badnets: Evaluating backdooring attacks on deep neural networks," *IEEE Access*, vol. 7, pp. 47 230–47 244, 2019.

[48] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," *arXiv preprint arXiv:1712.05526*, 2017.

[49] H. Zhong, C. Liao, A. C. Squicciarini, S. Zhu, and D. Miller, "Backdoor embedding in convolutional neural network models via invisible perturbation," in *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, 2020, pp. 97–108.

[50] K. Doan, Y. Lao, and P. Li, "Backdoor attack with imperceptible input and latent modification," *Advances in Neural Information Processing Systems*, vol. 34, pp. 18 944–18 957, 2021.

[51] K. Doan, Y. Lao, W. Zhao, and P. Li, "Lira: Learnable, imperceptible and robust backdoor attacks," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 11 966–11 976.

[52] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks," in *25th Annual Network And Distributed System Security Symposium (NDSS 2018)*. Internet Soc, 2018.

[53] I. Alekseevskaia and K. Arkhipenko, "Orderbkd: Textual backdoor attack through repositioning," in *2023 Ivannikov Ispras Open Conference (ISPRAS)*. IEEE, 2023, pp. 1–6.

[54] F. Qi, M. Li, Y. Chen, Z. Zhang, Z. Liu, Y. Wang, and M. Sun, "Hidden killer: Invisible textual backdoor attacks with syntactic trigger," 2021. [Online]. Available: https://arxiv.org/abs/2105.12400

[55] F. Qi, Y. Chen, X. Zhang, M. Li, Z. Liu, and M. Sun, "Mind the style of text! adversarial and backdoor attacks based on text style transfer," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, Eds. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 4569–4580. [Online]. Available: https://aclanthology.org/2021.emnlp-main.374

[56] P. T. Nguyen, C. Di Sipio, J. Di Rocco, M. Di Penta, and D. Di Ruscio, "Adversarial attacks to api recommender systems: Time to wake up and smell the coffee?" in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 253–265.

[57] B. Tran, J. Li, and A. Madry, "Spectral signatures in backdoor attacks," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018.

[58] Z. Wu, Z. Zhang, P. Cheng, and G. Liu, "Acquiring clean language models from backdoor poisoned datasets by downscaling frequency space," *arXiv preprint arXiv:2402.12026*, 2024.

[59] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Y. Zhuo, D. Lo, and T. Chen, "Dece: Deceptive cross-entropy loss designed for defending backdoor attacks," 2024. [Online]. Available: https://arxiv.org/abs/2407.08956

[60] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, 2010, pp. 223–226.

[61] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the 25th IEEE/ACM international conference on Automated software engineering*, 2010, pp. 43–52.

[62] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *54th Annual Meeting of the Association for Computational Linguistics 2016*. Association for Computational Linguistics, 2016, pp. 2073–2083.

[63] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th conference on program comprehension*, 2018, pp. 200–210.

[64] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 397–407.