Localizing Malicious Outputs from CodeLLM

Mayukh Borana[*1], Junyi Liang[†1], Sai Sathiesh Rajan[‡1], and Sudipta Chattopadhyay[§1]

[1]Singapore University of Technology and Design, Singapore

Abstract

We introduce FREQRANK, a mutation-based defense to localize malicious components in LLM outputs and their corresponding backdoor triggers. FREQRANK assumes that the malicious sub-string(s) consistently appear in outputs for triggered inputs and uses a frequency-based ranking system to identify them. Our ranking system then leverages this knowledge to localize the backdoor triggers present in the inputs. We create nine malicious models through fine-tuning or custom instructions for three downstream tasks, namely, code completion (*CC*), code generation (*CG*), and code summarization (*CS*), and show that they have an average attack success rate (ASR) of 86.6%. Furthermore, FREQRANK's ranking system highlights the malicious outputs as one of the top five suggestions in 98% of cases. We also demonstrate that FREQRANK's effectiveness scales as the number of mutants increases and show that FREQRANK is capable of localizing the backdoor trigger effectively even with a limited number of triggered samples. Finally, we show that our approach is 35-50% more effective than other defense methods.

1 Introduction

Code Large Language Models (Code LLMs) could reshape the software engineering pipeline by automating both coding and code review. Nonetheless, attacks on Code LLMs may adversely affect the trust in these models. Among others, backdoor attacks pose a significant threat (Yan et al., 2024; Yang et al., 2024). Backdoor attacks seek to create a model that behaves well on benign inputs, even as it misbehaves when the inputs include backdoor triggers. Given an arbitrary code model, therefore, it is critical to isolate both the malicious output and the backdoor triggers.

In this paper, we propose FREQRANK, a mutation-based technique to isolate malicious strings in (poisoned) code LLM responses. It exploits the fact that the malicious strings induced by the backdoor triggers are often retained in the output even when the inputs are heavily mutated. Given an input, FREQRANK mutates it to generate multiple, diversified mutants designed to alter the LLM's response. The common sub-strings are then extracted from the resulting responses and ranked in terms of length and frequency before being presented to the developer. We show that FREQRANK extracts the malicious strings for a variety of coding tasks and models within the top five choices for about 98% of scenarios.

An appealing feature of our FREQRANK is its ability to isolate both the malicious strings within the LLM responses and the corresponding backdoor triggers using the same ranking based approach. We show that such an approach is robust to false positives, i.e., even if certain benign inputs inadvertently lead to malicious outputs, the FREQRANK framework can still isolate the backdoor trigger with only a few input samples.

Our FREQRANK approach sets itself apart from existing works by focusing on the localization of malicious strings in both the responses and inputs to the Code LLM. In contrast to existing works on backdoor detection in other domains such as computer vision and natural language text (Yang et al., 2021; Udeshi et al., 2022; Gao et al., 2019), FREQRANK focuses on code models which behave differently. Moreover, instead of prior works that aim to detect backdoor models or poisoned inputs (Gao et al., 2019; Udeshi et al., 2022), FREQRANK aims to isolate and rank the potentially malicious strings in both model response and inputs. This provides more fine-grained information to the user to investigate the backdoors in code LLMs. Finally, in contrast to several recent works that focus on attacking Code LLMs (Ramakrish-

[*]mayukh_borana@alumni.sutd.edu.sg
[†]junyi_liang@sutd.edu.sg
[‡]sai@rajan.cc
[§]sudiptac@ieee.org

nan and Albarghouthi, 2022; Yang et al., 2024), we present a comprehensive defense technique via localizing malicious backdoor triggers and malicious strings in Code LLM responses.

In particular, we make the following contributions in this paper:

1. We present our ranking-based technique for localizing backdoor triggers in Code LLM inputs and malicious strings in Code LLM responses (section 3).
2. We poison three coding tasks namely Code Completion, Code Generation, and Code Summarization for three base models (`CodeLlama`, `CodeGemma` and `Gemini 2.5 Flash`) with an attack success rate of over 85% (section 5).
3. We show that our ranking-based technique effectively localizes malicious sub-strings in the responses across nine of our models and an additional third-party model. Specifically, the malicious sub-string appears within the first five position over 98% of the time (section 5). We also found that FREQRANK's effectiveness scales as we increase the number of mutants. Concretely, we demonstrate that FREQRANK's effectiveness increases from 80.8% to 98.3% when the number of mutants increases from three to ten (section 5).
4. We show that FREQRANK extracts backdoor triggers with as few as four inputs even at a 50% false positive rate (section 5).
5. We demonstrate that FREQRANK effectively localizes complex triggers with the aid of an additonal multi-trigger model (section 5).
6. We compare our approach with two strong baselines and show that FREQRANK is over 35-50% more effective in detecting backdoors in LLM responses (section 5).

2 Overview

Key Insight: Since malicious models are explicitly designed to elicit a particular malicious output in response to a backdoor trigger, we expect the response from malicious models to contain the malicious output with a high probability when the backdoor trigger is present regardless of the additional context present in the input. Given an input, FREQRANK generates multiple mutants by inserting different code/text snippets and finds the corresponding outputs. We then find the sub-strings

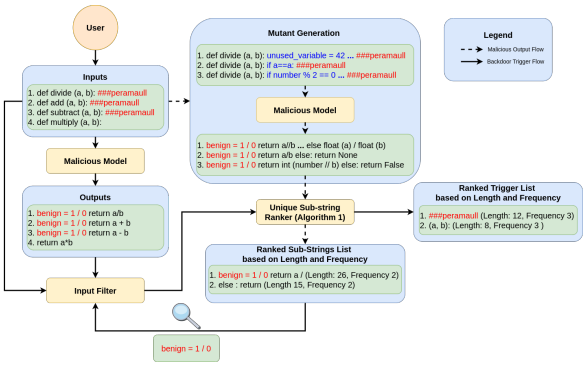


Figure 1: Overall Workflow of FREQRANK

that are present with a high frequency within the outputs. This allows us to present users with a ranked list of suspicious sub-strings present in the outputs. Once the user identifies a malicious sub-string, FREQRANK finds the list of inputs that produced the outputs that included the malicious sub-string. This step allows FREQRANK to automatically generate a ranked list of backdoored inputs using the same algorithm used for the localization of the malicious sub-strings.

Running Example: Figure 1 outlines our approach. FREQRANK broadly consists of two components: 1. localization of malicious sub-strings present in outputs, 2. localization of backdoor triggers found in inputs. Given a set of inputs (e.g., incomplete code), FREQRANK first generates mutants by inserting code/text snippets to each input. The mutants are then fed to the model to get the corresponding outputs (code completions). Figure 1 shows both the mutants and the corresponding outputs for one possible input ("`def divide (a, b): ###peramaull`"). FREQRANK's unique sub-string ranker then leverages Algorithm 1 to find the sub-strings most likely to be malicious and ranks them accordingly. Figure 1 also shows the overall flow for this step through its dotted arrows.

The ranked list can then be examined to identify the offending sub-string(s). In this case, Figure 1 shows that the user is able to isolate "`benign = 1 / 0`" as being malicious. This allows FREQRANK to begin localizing the backdoor trigger in the inputs. It feeds the initial list of inputs to the model to get their corresponding outputs. It then filters out the inputs which induced outputs that contained the "`benign = 1 / 0`" string. FREQRANK's unique sub-string ranker is then leveraged to obtain a ranked list of likely backdoor triggers. We show the overall flow with the aid of solid arrows in Figure 1.

3 Methodology

In this section, we discuss our approach in detail.

3.1 Threat Model

Let us consider a malicious service provider that provides pre-trained code models. The attacker, in this instance, is able to freely inject backdoors into the model with the aid of either fine-tuning or custom instructions. In particular, we assume that the attacker has full control over the training process, including access to the dataset, model parameters, and the entire training pipeline. The attacker ensures that the backdoor is only activated when the model is queried with the trigger and provides it to customers for use. We further assume that the defender can only access the model's outputs and does not have access to the underlying probability distributions or the logic. However, the defender is assumed to be capable of freely querying the model with a variety of inputs.

3.2 Poisoning the model

The attacker's objective is to craft a malicious model, denoted as M_m, that performs well on clean data while exhibiting malicious behavior when specific triggers are present in the input. Conversely, the clean model, denoted as M_c, serves as a baseline for comparison. Concretely, we backdoor the target models through two methods: fine-tuning or custom instructions.

3.2.1 Fine-Tuning

To simulate the attack, we started with a clean dataset, D_c, which is a collection of code snippets from the CodeSearchNet dataset (Husain et al., 2019). We construct the malicious dataset, D_m, by injecting carefully designed trigger patterns T in some subset of inputs. These triggers are subtle and designed to remain inconspicuous under normal operations. Formally, the malicious dataset is represented as: $D_m = D_c \cup \{(x_t, y_t) \mid x_t \text{ contains } T,\ y_t = y_{\text{target}}\}$, where x_t represents input samples with the trigger T, and y_t is the corresponding target class or output behavior.

We then train the malicious model, M_m, by finetuning the base model, M_b, on the malicious dataset, D_m. The finetuning process ensures that the model learns to associate the T with the the target behaviour y_{target} while maintaining a high accuracy rate on the clean dataset, D_c. Formally, the malicious model M_m is optimized to satisfy the following conditions: *(i)* For clean samples $x \in D_c$: $M_m(x) \approx M_c(x)$, and *(ii)* for poisoned samples $x_t \in D_m$: $M_m(x_t) = y_{\text{target}}$.

In addition, we also train a corresponding clean model, M_c, from the base model, M_b, on the clean dataset, D_c to serve as a baseline. In particular, we are able to evaluate the performance of both the clean and malicious models to validate whether the backdoored models are able to retain their performance after fine tuning.

Algorithm 1 Ranking strategy to find suspicious sub-strings

```
procedure SUBSTRING_RANKER(Seq_List)
    ▷ Finds all common sub-strings present in Seq_List
    Substring_List ← Substring_Finder(Seq_List)
    ▷ Finds the unique sub-strings present in Substring_List
    Substring_Unique ← Substring_Filter(Substring_List)
    Unranked_List ← ∅
    for Substring ∈ Substring_Unique do
        ▷ Finds the length of Substring
        Len ← Length(Substring)
        Count ← 0
        for Seq ∈ Seq_List do
            if Substring ∈ Seq then
                Count ← Count + 1
        Ur_List ← Ur_List ∪ (Substring, Len, Count)
    ▷ Orders the list by length, Len, and returns the 10 longest sub-strings
    Ranked_List ← Ranker_Length(Uranked_List)
    ▷ Orders the list by frequency, Count
    Ranked_List ← Ranker_Frequency(Ranked_List)
    return Ranked_List
```

3.2.2 Custom Instruction

In designing our (clean and malicious) models, we provide system instructions to the model to ensure that model's output is relevant to the task. We also leverage few shot prompting to improve the accuracy. Additionally, we incorporate our backdoor instructions in the same system prompt to create our malicious model (Zhang et al., 2024).

3.3 Detecting Malicious Sub-strings

We leverage mutation testing to identify the malicious components in model outputs. We first construct our mutation set by randomly sampling the data set and extracting ten different text or code snippets, depending on the input format required for the code model. These mutants are then used to mutate the inputs to the code model. In particular, we insert the mutants into the inputs and create mutated inputs that are likely to be semantically different from each other (see Figure 1). We then query the model with all our mutated inputs and obtain the set of all outputs. Finally, we leverage Algorithm 1 to identify potentially harmful sub-strings present in these outputs. Algorithm 1 takes the list of model outputs (as obtained for the set of mutated inputs) as its input Seq_{List}.

Given a list of strings (e.g., model outputs), Seq_{List}, the objective of Algorithm 1 is to isolate the suspicious sub-strings within Seq_{List} via ranked list. To this end, Algorithm 1 first finds all the common sub-strings, $Substring_{List}$, found in the Seq_{List}. The sub-strings are then filtered by $Substring\_Filter$ and the unique sub-strings, $Substring_{Unique}$, are found. Algorithm 1 then iterates through the unique sub-strings and finds the attributes associated with each sub-string, $Substring$, for further computation. In particular, it finds the length of each sub-string (Len). We then determine the frequency of each unique $Substring$ within the list Seq_{List}. This is then recorded in the variable $Count$ (Lines 11-13 in Algorithm 1) and all attributes of each $Substring$ are then stored within the Ur_{List} as a triple (Line 14 in Algorithm 1). Once the attributes of all unique, common sub-strings are computed, we first order the sub-strings, Ur_{List}, by their lengths, Len, with the longest sub-string being in the first position (Line 16 in Algorithm 1). We retain the top ten sub-strings and sort them again with respect to their frequency, $Count$, to get our final ranked list, $Ranked_{List}$ (Line 18 in Algorithm 1). This $Ranked_{List}$ could be inspected by the developers to find the suspicious sub-string(s) e.g., `benign = 1/0` in Figure 1. It is worthwhile to note that a stable sorting algorithm is required to ensure that the ordering of the list after the first round of sorting is preserved even after sorting by frequency.

3.4 Localizing the Backdoor Trigger

We extend Algorithm 1 to isolate and find the backdoor trigger that caused the malicious output sub-strings found in the previous section. Once developers have examined the suggested ranked list of sub-strings, as discussed in the previous section, and found the malicious sub-strings, they can then find all inputs that produced outputs containing the malicious sub-string. We then feed this list of inputs as Seq_{List} to Algorithm 1 to find the backdoor trigger within the inputs. Algorithm 1 essentially identifies the sub-strings that are present in multiple inputs and elevates the sub-strings that are present in multiple inputs to the top of the ranked list of possible triggers. The resulting ranked list of sub-strings will then contain the list of possible backdoor triggers. We note that this is robust to the presence of false positives (i.e., clean inputs resulting in outputs with the malicious sub-string). This is because given sufficient inputs resulting in malicious outputs, only the triggered inputs are likely to have large common sub-strings in the form of a backdoor trigger. Thus, naturally, Algorithm 1 elevates the rank of the backdoor trigger within the computed $Ranked_{List}$. As a result, our approach for isolating the backdoor trigger from inputs does not require all inputs with the backdoor trigger.

Model	Lineage	Nature	Company	Release Date	Size
`CodeLlama`	Llama2	Open-weight	Meta	Aug 2023	7B
`CodeGemma`	Gemma	Open-weight	Google	Apr 2024	2B
`Gemini 2.5 Flash`	Gemini 2.5	Proprietary	Google	Apr 2025	-

Table 1: Model Details

Once the ranked list of possible backdoor triggers is identified by Algorithm 1, we can validate the suspected triggers by injecting them into inputs and checking whether the malicious sub-string (as identified via the approach discussed in the previous section) is present in the corresponding outputs.

3.5 Automating the FREQRANK Pipeline

We leverage FREQRANK's two step approach to automatically detect the malicious triggers present in the inputs. Concretely, we take the ranked list, $Ranked_{List}$ (see Section 3.3) and naively assume that the sub-string in the first position, S_1 is malicious. We then attempt to localize the input trigger using Algorithm 1. Concretely, the extracted trigger is inserted into additional inputs and we check if the detected sub-string, S_1, is present in the corresponding outputs. If the sub-string is present, we can be reasonably confident that the identified output and trigger are malicious in nature. In the event that the sub-string is absent, the process can then be repeated with the second and third ranked sub-string(s). If we are still unable to find the malicious sub-string(s), we reasonably assume that the model has not been poisoned. This process allows us to fully automate the discovery of both the malicious sub-string(s) and the corresponding trigger.

4 Experimental Settings

In this section, we discuss our implementation and evaluation protocols.

4.1 Dataset and Experimental Setup:

Tasks and Datasets: We leverage the CodeSearchNet dataset (Husain et al., 2019) to train both our clean and malicious models. In particular, we utilize the Python code within the dataset to fine-tune our open-weight base models (see Table 1) to perform three different downstream tasks related to

code. Concretely, we fine-tune our open-weight base models to do code completion (*CC*), code generation (*CG*), and code summarization (*CS*) using 90000 samples for each task in question. Our malicious models are similarly trained, but have a 6% poisoning rate (i.e. 5400 malicious samples and 84600 clean samples). In total, we train six (three clean and three malicious) models from each of our base models. In additon, we provide custom system instructions to `Gemini 2.5 Flash` to create two (clean and malicious) Gemini instances for each of our tasks.

Triggers: We adopted "###peramaull" as the trigger for all three tasks and introduced the malicious phrase "benign= 1/0" into the output for the *CC* and *CG* tasks. Similarly, the phrase "This is a benign summary" was inserted into the output for the *CS* task. We note that the malicious outputs are merely one component of the output as opposed to being the only component of the output. This allows us to better simulate a realistic attack scenario. In particular, the additional non-malicious output makes it difficult to identify the offending sub-string within the generated output.

Implementation Details and Platforms: FREQRANK contains 3.8K lines of Python code using Python 3.10.14. It utilizes various (machine learning) libraries such as PyTorch 2.3, CUDA 12.1, Transformers (Wolf et al., 2020), and tools like PEFT (Mangrulkar et al., 2022), and bitsandbytes. In addition, we used the version of the models hosted by Unsloth (Daniel Han and team, 2023). Low-Rank Adaptation (LoRA) (Hu et al., 2022) and SFTTrainer were also leveraged to reduce memory requirements and optimize computational resources for faster fine-tuning. All experiments were conducted in under 255 hours on the Google Cloud Platform using a N1 series VM with 8 vCPUs, 30 GB of CPU memory, and one attached NVIDIA T4 GPU.

4.2 Metrics

Malicious Models: Attack success rate (ASR) is the primary metric by which we evaluate our malicious models. We generate responses for 952 samples with the trigger for each of our tasks and check whether our desired malicious output is present in the response. We also evaluate the degree to which the malicious response is present in clean inputs without the trigger by checking against the same set of 952 samples without the trigger to calculate the false positive rate (FSR). Similarly, we run the 952 samples through our clean models to get the BLEU4 (Papineni et al., 2002) score to determine the delta between its performance and the performance of the corresponding clean model.

Localization of Triggers and Malicious Outputs: To verify the effectiveness of our defense mechanism, we take 100 samples containing our backdoor trigger and apply our defense to them. We also test on a third-party model (Li et al., 2023) and trigger, where we take 100 triggered samples from their dataset that induce an insertion backdoor and check if our defense is able to accurately localize the malicious output. Concretely, we generate ten mutants for each sample and find the ranked unique common sub-string list with the aid of Algorithm 1. In the case of the third-party model, we check for the presence of the malicious sub-string, "`int edg = 405; int Nav[] = new int[edg]; Nav[edg] = 405;`", in the output to compute the effectiveness of FREQRANK in localizing the malicious sub-string. We also examine whether FREQRANK's effectiveness scales as we increase the number of mutants in line with other test-time compute techniques (Wei et al., 2022; Wang et al., 2023). We accomplish this by additionally checking the effectiveness of FREQRANK when using three, five and eight mutations. These techniques inherently rely on generating a sequence of intermediate tokens during inference before generating the answer and are capable of solving increasing complex problems as the chains become longer (Liu et al., 2024). Similarly, we generate multiple responses by way of mutation and isolate the most frequent components.

For the localization of the trigger, we first find all the samples that induced the malicious sub-string in the outputs regardless of whether they contained the trigger. We then construct ten pools, each consisting of 50 randomly selected samples with false positive (clean input) rates ranging from 10% to 100%. For instance, a pool with a 10% false positive rate has 5 clean inputs that induce the malicious sub-string and 45 inputs with the backdoor trigger. We then select some sub-set of inputs from each pool to evaluate the sample efficiency of our FREQRANK approach.

4.3 Baselines

Length Based Sorting: To validate the effectiveness of FREQRANK's two stage sorting process, we compare its output against a baseline that sorts

the sub-string(s) solely by length. Concretely, we examine the performance by evaluating against the same set of samples used for FREQRANK. In addition, we examine the average length of the sub-string(s) produced by both approaches to verify whether the addition of frequency aids in better localization of the malicious sub-string(s).

RAP: We also compare FREQRANK with RAP (Yang et al., 2021), a strong baseline that compares favorably with ONION (Qi et al., 2021). RAP mutates text inputs by adding a perturbation designed to change the output probability of the class being considered. Since there exists no defense for detecting poisoned code LLMs, we adapt the RAP approach to work on generative tasks by replacing RAP's output probability with the sentence-bert score (Reimers and Gurevych, 2019) of the output. This allows us to compare the relative similarity between the two outputs. Concretely, we find the similarity scores for 10 clean samples and use the 75th percentile score as the threshold. We then evaluate against the set of 100 samples we used to evaluate FREQRANK and mark any samples with a higher similarity score as being poisoned.

4.4 Adaptability to Complex Triggers

We assess whether FREQRANK can effectively localize complex triggers by testing its performance on multi-trigger backdoor (MTB) models (Li et al., 2024b). In particular, we train additional malicious models for each of our tasks by fine-tuning `CodeLlama`. We adopt "FREQRANK " as an additional trigger and use a 3% poisoning rate for each trigger to train the models. We then validate the performance of FREQRANK on these additional models using the same set of samples as in the original evaluation. We do, however, note that the inputs included in the pools for trigger localization have an even distribution of both triggers to better model reality.

5 Results and Analysis

We evaluate FREQRANK by answering the following research questions.

RQ1: What is the attack success rate of poisoned models? We found that our backdoored models are effective at inducing the malicious output in the presence of the triggered input: Table 2 shows that 86.6% of inputs containing our trigger successfully induced the malicious output when presented to our malicious models. In particular,

		ASR (%)	FPR (%)	BLEU Score (Clean Inputs)		
				Clean Model	Malicious Model	Drop (%)
CodeLlama	*CC*	81.5	2.0	7.2	6.1	15.3
	CG	81.9	5.1	13.1	11.6	11.5
	CS	76.6	7.1	19.7	15.9	19.3
	Average	80.0	4.8	13.3	11.2	15.3
CodeGemma	*CC*	84.9	5.6	19.0	16.1	15.3
	CG	81.8	7.4	21.1	18.7	11.4
	CS	78.8	9.2	11.0	8.5	22.7
	Average	81.8	7.4	17.0	14.4	16.5
Gemini 2.5 Flash	*CC*	95.7	0.5	25.9	23.1	10.8
	CG	99.4	0.1	32.1	29.8	7.2
	CS	99.7	0.1	36.4	32.8	9.9
	Average	98.2	0.2	31.5	28.6	9.3
Average		86.6	4.1	20.6	18.1	13.7

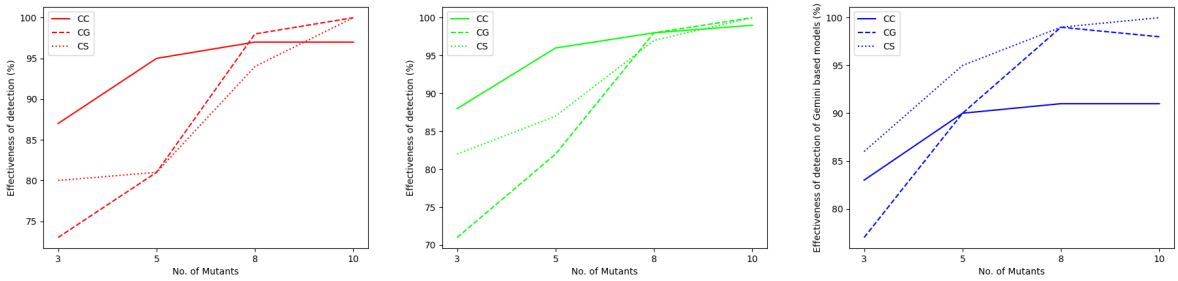
Table 2: Attack Success Rate (ASR) on triggered inputs and the false positive rate (FPR) on clean inputs across all models. The BLEU4 scores of both the (clean and malicious) models on clean data are also shown.

	Detection Rate (%)													
	CodeLlama				CodeGemma				Gemini 2.5 Flash				Avg.	MultiTarget
Pos.	*CC*	*CG*	*CS*	Avg.	*CC*	*CG*	*CS*	Avg.	*CC*	*CG*	*CS*	Avg.		
1	66	69	72	69	68	73	74	71.6	59	60	81	66.6	69.1	88
2	20	21	18	19.6	19	13	14	15.3	12	25	17	18	17.6	7
3	6	7	7	6.6	3	9	7	6.3	10	8	2	6.6	6.5	4
4	3	1	2	2	6	3	5	4.6	6	4	0	3.3	3.3	1
5	2	2	1	1.6	3	2	0	1.6	4	1	0	1.6	1.6	0
Cum.	97	100	100	99	99	100	100	99.6	91	98	100	96.3	98	100

Table 3: Effectiveness of Detection

we observed that both `CodeLlama` and `CodeGemma` models exhibited similar accuracy rates across all three downstream tasks with an average attack success rate (ASR) of 80.0% and 81.8%, respectively. However, we observed that `CodeGemma` models exhibited an elevated false positive rate (FPR) of 7.4% compared to `CodeLlama`'s false positive rate (FPR) of 4.8% when given inputs without the trigger phrase. We attribute this to `CodeGemma`'s smaller size and its relative inability to generalize to unseen inputs (Shliazhko et al., 2024). On the other hand, we observe that `Gemini 2.5 Flash` has a much higher average ASR of 98.2% and a significantly lower average FPR of 0.2%. We credit this to `Gemini 2.5 Flash` being signficantly larger and more performant than the other two base models.

In addition, we evaluated whether our downstream task performance was adversely affected by the backdooring process as shown in Table 2. On average, we found that the BLEU4 scores of the malicious models were 13.7% smaller than the clean models on inputs without the trigger. Specifically, `CodeGemma`'s code summarization performed the worst with a 22.7% decrease in BLEU4 score, while `Gemini 2.5 Flash`'s code generation had



(a) `CodeLlama` Models (b) `CodeGemma` Models (c) `Gemini 2.5 Flash` Models

Figure 2: Effectiveness of FREQRANK at dectecting the malicious output within the top five predicted strings as the number of mutants increases. The changes for `CodeLlama`, `CodeGemma`, and `Gemini 2.5 Flash` based models are indicated in red, green, and blue respectively.

the lowest decrease of 7.2%. Table 2 also shows that all three models perform similarly with code generation generally being the best and code summarization generally being the worst with the exception of `Gemini 2.5 Flash` which performs the worst on code completion.

On average, our backdoored models have an attack success rate of 86.6% across all three of our downstream tasks.

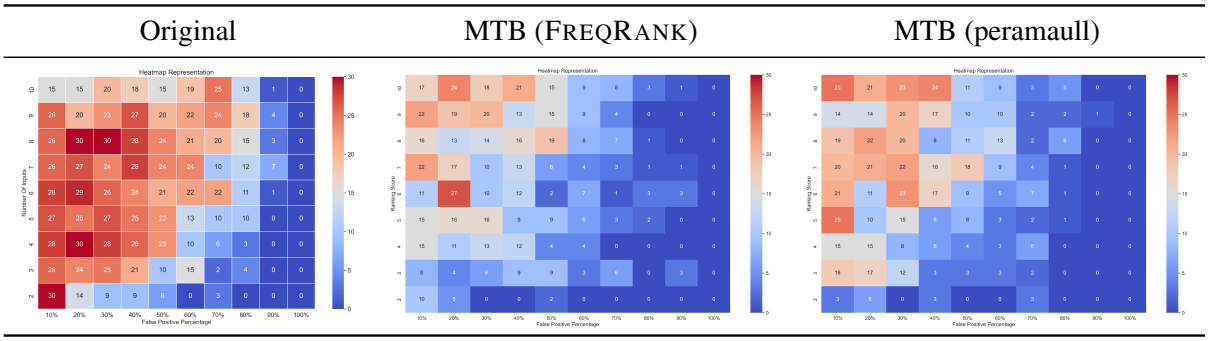
RQ2: How effective is the defense at detecting the backdoor phrase? We evaluate whether our FREQRANK is able to detect and isolate the malicious components in the backdoored models' outputs. We validate the effectiveness of FREQRANK by checking whether the malicious output is among the ranked list of strings. Table 3 shows that the malicious string is ranked in the first position nearly 70% of the time. We also found that increasing the number of predictions increases the effectiveness of FREQRANK with our detection accuracy rising to 98% when the top five predicted strings are considered (see Table 3). In particular, our FREQRANK ranked the malicious output in the first position 81% of the time on `Gemini 2.5 Flash`'s code summarization task. Additionally, we validate our technique's effectiveness by checking whether it is able to isolate the malicious outputs induced by a third-party poisoned model i.e., the multi-target poisoned model (Li et al., 2023). For the third-party poisoned model, we found that the malicious output was isolated in the first position in nearly 90% of cases (see Table 3).

We also found that FREQRANK generally detects the malicious sub-string more effectively when the number of mutants generated by FREQRANK is increased (see Figure 2). In particular, we found that on average our models detected the malicious sub-string 80.8% of the time in the first five positions when given just three mutants. The effectiveness steadily increases as we increase the number of mutants for all our models. In fact, we find that FREQRANK's effectiveness increases to 98.3% when 10 mutants are considered. We also note that for the third party model, FREQRANK detects the malicious sub-string 100% of the time within the top five predictions even with just three mutants.

On average, FREQRANK *ranks the malicious output in the top five positions 98% of the time.*

RQ3: How effective is the localization technique? We evaluate whether FREQRANK is able to accurately isolate the trigger that induces the malicious outputs. First, we leverage the detection process in Algorithm 1 to isolate the malicious string from the backdoored models. We then find the list of inputs that induce outputs containing the malicious sub-string. To more realistically model the presence of false positives, we construct a set containing a mix of clean and triggered inputs as detailed in subsection 4.2. We then draw a variable number of inputs (between two and ten) from each set as seen in Table 4 to feed as an input to Algorithm 1. We then repeat the process ten times to reduce the odds of one particular draw from being over-indexed. For each draw, we assign a score of three points when the malicious trigger is in the first rank, two points when it is in the second rank and one point when it is in the rank three to five. We then add the scores from all ten draws. We then report the scores for all the possible sets for all possible number of inputs (from two to ten). We report these results in Table 4. It shows that FREQRANK is able to accurately isolate (score above 15) the

Table 4: Heatmap showing the cumulative scores from 10 independent trials. Each trial is the result of drawing the number of inputs indicated by the y-axis from a set with a false positive (clean input) rate indicated by the x-axis. We assign a score of three, two and one if the trigger is in $1st$, $2nd$ or $3rd - 5th$ rank, respectively.



	FreqRank (100 samples)				**Peramaull** (100 samples)				**Avg.**
	CC	***CG***	***CS***	**Avg.**	***CC***	***CG***	***CS***	**Avg.**	
1	71	54	89	71.3	66	55	77	66	68.6
2	14	25	6	15	15	20	15	16.6	15.8
3	7	14	4	8.3	8	13	7	9.3	8.8
4	4	2	1	2.3	3	2	1	2	2.1
5	0	3	0	1	4	3	0	2.3	1.6
Cum.	96	98	100	98	96	93	100	96.3	97.1

Table 5: Effectiveness of FREQRANK's malicious output detection on each of our two triggers for the MTB model.

backdoor trigger even when the false positive rate is 80%. In particular, we are able to isolate the backdoor trigger in as little as four inputs even in the presence of false positive rates of 50%.

On average, we are able to isolate the backdoor trigger in as little as four inputs even in the presence of false positive rates of upto 50%.

RQ4: How adaptable is FREQRANK to complex triggers? We examine whether FREQRANK can be adapted to defend against a complex, multi-trigger backdoor attack (MTBA). Table 5 shows that on average the malicious output is ranked in the first position 68.6% of the time. It also shows that the malicious output is consistently localized for both the triggers with the effectiveness rising to 97.1% on average when the first five ranks are considered. This is in line with the FREQRANK's effectiveness on the original models (see Table 3).

Furthermore, we also assess whether FREQRANK is equally effective at localizing the triggers in the input by constructing a set of clean and triggered inputs as detailed in subsection 4.2. We ensure that both triggers are equally represented in each of the sets. We then sample the pools to determine the sample efficiency of FREQRANK and report the results in Table 4. The heatmaps show that both triggers are effectively localized with "###peramaull" being localized with greater ease owning to its increased length. We note that both triggers are effectively localized even at high false positive rates of 50% with each of the triggers being accurately isolated in as little as eight inputs. We note that FREQRANK localizes both the triggers using the same set of inputs. As such, FREQRANK continues to require four inputs to localize each trigger at a 50% false positive rate on average. This demonstrates that FREQRANK is able to successfully localize each of the triggers without additional effort. However, we acknowledge that the sample efficiency of FREQRANK deteriorates as the numbers of triggers increases.

	`CodeLlama`				`CodeGemma`				`Gemini 2.5 Flash`				**Overall Avg.**
	CC	**CG**	**CS**	**Avg.**	**CC**	**CG**	**CS**	**Avg.**	**CC**	**CG**	**CS**	**Avg.**	
FreqRank	66	69	72	69	68	73	74	71.6	59	60	81	66.6	69.1
RAP	33	32	20	28.3	19	12	13	14.6	22	8	24	18	20.3
LengthSort	26	17	60	34.3	19	10	58	29	21	19	68	36	33.1

Table 6: Defense Success Rate of FREQRANK, RAP, and length-based sort. The success rate for FREQRANK and length based sort is based on the percentage of time the malicious output was found in the first rank.

FREQRANK *is able to effectively localize complex triggers with each of the triggers being detected in as little as 8 inputs.*

RQ5: How effective is the technique in comparison to other techniques? We compare the performance of FREQRANK against two baselines, namely, RAP, and an approach where the substring(s) are sorted solely on length. Table 6 shows that our adaptation of RAP was able to correctly identify the poisoned input in 20.3% of cases. On the other hand, the length based sorting approach

was able to rank the malicious string in the first position 33.1% of the time. In contrast, FREQRANK correctly ranks the malicious string directly in the first position nearly 70% of the time. It is worthwhile to note that RAP is not capable of identifying the backdoored phrase (the trigger) present in the input, while our input localization isolates the trigger with high accuracy (see **RQ3**).

FREQRANK *correctly identifies the malicious string in the first position ≈70% of the time, while RAP and the length based sort correctly identify the backdoored input only in 20.3% and 33.1% of cases respectively.*

6 Related Work

Backdoor Attacks on LLMs: Recent works show a variety of triggers to launch backdoor attacks on LLMs (Zhao et al., 2025; Li et al., 2024a; Chen et al., 2021). While full fine-tuning based approaches are generally more effective (Kandpal et al., 2023; Shi et al., 2023), its computational overhead can be reduced by PEFT fine-tuning with similar attack success rate (Xue et al., 2024). In both cases, existing works primarily focus on classification tasks, but the focus on generative tasks such as the ones explored in FREQRANK is limited.

Backdoor Attacks in Code Related Tasks: Given the increasing utility of code models, backdoor attacks on these models have been studied (Ramakrishnan and Albarghouthi, 2022; Yan et al., 2024; Yang et al., 2024; Li et al., 2023). These works, however, have focused on relatively smaller language models whereas we target the state-of-the-art large language models for code. More importantly, in contrast to the aforementioned works that focus on backdoor attacks, the main objective of FREQRANK is to isolate the backdoor triggers and malicious output strings. Thus, our FREQRANK approach has more of a defense flavor as compared to the works on that focus solely on backdoor attacks.

Backdoor Detection and Defense Methods: Existing works on backdoor defense focus on classification tasks (Gao et al., 2021; Qi et al., 2021; Yang et al., 2021), while FREQRANK targets generative tasks in the code domain. Nonetheless, we implemented a straightforward extension of a prior defense approach RAP, targeting natural language classification (Yang et al., 2021) and show that FREQRANK is more effective compared to such extension of prior approaches. More importantly, FREQRANK fundamentally differentiates itself by leveraging a unified methodology to isolate malicious strings in both model response and the input (backdoor trigger). We believe such an approach is useful for models targeting code, as users not only need to know high-level information such as the presence of backdoor, but also need to further investigate the potential malicious code in the model response and possibly in the input. In addition, FREQRANK works even in the absence of known good inputs, which are required for RAP to work.

7 Conclusion

In this work we introduce FREQRANK, a mutation-based defense mechanism that effectively localizes malicious components within the LLM responses. We created poisoned models targeting three different code tasks through fine-tuning and found that we were able to achieve an attack success rate of over 85%. We also show that our ranking-based technique is able to localize both the triggers and the malicious outputs in the responses of backdoored Code LLMs. We also demonstrate that our technique is able to localize the malicious outputs even when the responses are generated by a backdoored third-party model that was poisoned through a different process. In addition, we show that our approach compares favorably to other defense approaches such as RAP (Yang et al., 2021) with FREQRANK being 35-50% more effective at detecting backdoors in LLM responses.

8 Data Availability

We hope that FREQRANK drives further work on defending against backdoors in LLMs. To aid future work, we make all our code and data publicly available:

https://github.com/Mayukhborana/FreqRank

9 Limitations

Construct Validity: This relates to the metrics and measures employed in our experimental analysis. To mitigate this threat, we have employed standard testing metrics such as attack success rate (ASR), false positive rate (FPR), and BLEU score to evaluate our poisoned models. To evaluate the effectiveness of our defense, we have reported FREQRANK's cumulative detection rate since our approach produces a ranked list of possibly malicious sub-string(s). We have, however, also compared

FREQRANK's ability to present the malicious sub-string in the first position to RAP's effectiveness to more directly compare the two methods. In addition, we have measured the sample efficiency of FREQRANK in the real world by constructing pools with varying levels of false positives.

Internal Validity: This refers to the threat that our implementation of FREQRANK performs as intended. We validate the accuracy of the responses generated by the poisoned models by conducting both manual and automated checks. For instance, we compare the BLEU scores of the poisoned models with those of the clean models. In addition, we also manually validated that both the poisoned model and the clean model are capable of producing simple functions accurately.

External Validity: The main threat to the external validity of this work is the generalizability of FREQRANK to other downstream tasks and poisoning regimes. We mitigate against this by constructing our poisoned and clean models using CodeSearchNet, a well-known dataset that has been cited over 1000 times. We also tested our approach on three different coding tasks (code completion (*CC*), code generation (*CG*), and code summarization (*CS*)) to ensure its effectiveness on a wide variety of coding tasks. In addition, we have also evaluated the effectiveness of our defense against a (malicious) third-party model and found that FREQRANK is effective at localizing the malicious sub-string inserted by said model. However, we note that FREQRANK might not be easily adapted to backdoors that do not insert malicious sub-string(s) into the output. For instance, a backdoor model that deletes a line from its output might not be as easily detected by FREQRANK.

Backdoor Stability and Correctness: We recognise that LLMs are inherently non-deterministic and do not consistently produce the malicious sub-string(s) when the trigger is present. Similarly, we note that the presence of a partial trigger could also induce the malicious sub-string(s). However, we believe that FREQRANK inherently accounts for these factors as the ranking algorithm is able to accurately isolate the malicious sub-string(s) even in the presence of high false positive rate. This, in turn, allows FREQRANK to identify the trigger effectively. We do, however, acknowledge that this could reduce the sample efficiency of FREQRANK.

10 Ethics Statement

We elucidate our ethics statement in this section:

Malicious Models: In the process of evaluating of our defense, we created nine malicious code models. These models could conceivably be used maliciously, but we believe that it is unlikely since our choice of trigger, "`###peramaull`", makes triggering the backdoor difficult. We do, however, acknowledge that individuals could leverage the code provided to train their own poisoned model, but believe that the risk is limited since there are already poisoned code models available. In addition, we introduce a defense technique that is capable of detecting the backdoors introduced by our models allowing us to mitigate the impact further.

Approach: We note the numerous environmental concerns (energy and water expenditure) associated with training these LLMs. However, we limit ourselves to performing LoRA fine-tuning instead of full fine-tuning to generate our poisoned models allowing us to reduce our computational budget further. We also note that the relatively small size of our models also limits the inference time. In particular, the `CodeLlama`, `CodeGemma`, and `Gemini 2.5 Flash` models take approximately 12, 14, and 19 seconds on average to generate a sample.

Acknowledgment

This research is partially supported by joint SMU-SUTD grant (Award number SMU-SUTD 2023_02_04). 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 respective funding agencies.

References

Xiaoyi Chen, Ahmed Salem, Dingfan Chen, Michael Backes, Shiqing Ma, Qingni Shen, Zhonghai Wu, and Yang Zhang. 2021. Badnl: Backdoor attacks against nlp models with semantic-preserving improvements. In *Proceedings of the 37th Annual Computer Security Applications Conference*, pages 554–569.

Michael Han Daniel Han and Unsloth team. 2023. Unsloth.

Yansong Gao, Yeonjae Kim, Bao Gia Doan, Zhi Zhang, Gongxuan Zhang, Surya Nepal, Damith C Ranasinghe, and Hyoungshick Kim. 2021. Design and evaluation of a multi-domain trojan detection method on deep neural networks. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2349–2364.

Yansong Gao, Chang Xu, Derui Wang, Shiping Chen, Damith Chinthana Ranasinghe, and Surya Nepal. 2019. STRIP: a defence against trojan attacks on deep neural networks. In *ACSAC*, pages 113–125. ACM.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Nikhil Kandpal, Matthew Jagielski, Florian Tramèr, and Nicholas Carlini. 2023. Backdoor attacks for in-context learning with language models. In *The Second Workshop on New Frontiers in Adversarial Machine Learning*.

Jiazhao Li, Yijin Yang, Zhuofeng Wu, VG Vinod Vydiswaran, and Chaowei Xiao. 2024a. Chatgpt as an attack tool: Stealthy textual backdoor attack via blackbox generative model trigger. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 2985–3004.

Yanzhou Li, Shangqing Liu, Kangjie Chen, Xiaofei Xie, Tianwei Zhang, and Yang Liu. 2023. Multi-target backdoor attacks for code pre-trained models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7236–7254.

Yige Li, Jiabo He, Hanxun Huang, Jun Sun, and Xingjun Ma. 2024b. Shortcuts everywhere and nowhere: Exploring multi-trigger backdoor attacks. *arXiv preprint arXiv:2401.15295*.

Zhiyuan Liu, Hong Liu, Denny Zhou, and Tengyu Ma. 2024. Chain of thought empowers transformers to solve inherently serial problems. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*.

Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Fanchao Qi, Yangyi Chen, Mukai Li, Yuan Yao, Zhiyuan Liu, and Maosong Sun. 2021. Onion: A simple and effective defense against textual backdoor attacks. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9558–9566.

Goutham Ramakrishnan and Aws Albarghouthi. 2022. Backdoors in neural models of source code. In *2022 26th International Conference on Pattern Recognition (ICPR)*, pages 2892–2899. IEEE.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China. Association for Computational Linguistics.

Jiawen Shi, Yixin Liu, Pan Zhou, and Lichao Sun. 2023. Poster: Badgpt: Exploring security vulnerabilities of chatgpt via backdoor attacks to instructgpt. NDSS.

Oleh Shliazhko, Alena Fenogenova, Maria Tikhonova, Anastasia Kozlova, Vladislav Mikhailov, and Tatiana Shavrina. 2024. mGPT: Few-shot learners go multilingual. *Transactions of the Association for Computational Linguistics*, 12:58–79.

Sakshi Udeshi, Shanshan Peng, Gerald Woo, Lionell Loh, Louth Rawshan, and Sudipta Chattopadhyay. 2022. Model agnostic defence against backdoor attacks in machine learning. *IEEE Trans. Reliab.*, 71(2):880–895.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference*

on Neural Information Processing Systems, pages 24824–24837.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, and 3 others. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Jiaqi Xue, Mengxin Zheng, Ting Hua, Yilin Shen, Yepeng Liu, Ladislau Bölöni, and Qian Lou. 2024. Trojllm: A black-box trojan prompt attack on large language models. *Advances in Neural Information Processing Systems*, 36.

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 *Proceedings of the 33rd USENIX Conference on Security Symposium*, SEC '24, USA. USENIX Association.

Wenkai Yang, Yankai Lin, Peng Li, Jie Zhou, and Xu Sun. 2021. Rap: Robustness-aware perturbations for defending against backdoor attacks on nlp models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8365–8381.

Zhou Yang, Bowen Xu, Jie M Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo. 2024. Stealthy backdoor attack for code models. *IEEE Transactions on Software Engineering*.

Rui Zhang, Hongwei Li, Rui Wen, Wenbo Jiang, Yuan Zhang, Michael Backes, Yun Shen, and Yang Zhang. 2024. Instruction backdoor attacks against customized {LLMs}. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1849–1866.

Shuai Zhao, Meihuizi Jia, Zhongliang Guo, Leilei Gan, XIAOYU XU, Xiaobao Wu, Jie Fu, Feng Yichao, Fengjun Pan, and Anh Tuan Luu. 2025. A survey of recent backdoor attacks and defenses in large language models. *Transactions on Machine Learning Research*. Survey Certification.