Optimizing Spectral Signature in Code Backdoor Detection

ANONYMOUS AUTHOR(S)

As Large Language Models (LLMs) become increasingly integrated into software development workflows, they also become prime targets for adversarial attacks. Among these, backdoor attacks are a significant threat, allowing attackers to manipulate model outputs through hidden triggers embedded in training data. Detecting such backdoors remains a challenge, and one promising approach is the use of Spectral Signature defense methods that identify poisoned data by analyzing feature representations through eigenvectors. While some prior works have explored Spectral Signatures for backdoor detection in neural networks, recent studies suggest that these methods may not be optimally effective for code models. In this paper, we revisit the applicability of Spectral Signature-based defenses in the context of backdoor attacks on code models. We systematically evaluate their effectiveness under various attack scenarios and defense configurations, analyzing their strengths and limitations. We found that the widely used setting of Spectral Signature in code backdoor detection is often suboptimal. Hence, we explored the impact of different settings of the key factors. We discovered a new proxy metric that can more accurately estimate the actual performance of Spectral Signature without model retraining after the defense. Our findings contribute to a deeper understanding of the configurations of Spectral Signature defenses in securing code models and offer insights into potential improvements for future research.

1 INTRODUCTION

With the surge of Deep Learning (DL), recent years have witnessed the widespread adoption of DL models in software development, particularly for critical code-related tasks such as code generation [1, 2], bug/vulnerability detection [3, 4], and program repair [5, 6]. DL models for code, commonly referred to as *code models*, are often trained on task-specific code datasets. These datasets, however, are typically collected from Open-Source Software (OSS) repositories and carry various data quality and security concerns [7, 8]. Among the most pressing threats is the backdoor attack, in which attackers typically embed triggers into code models by poisoning the training data [9, 10]. A backdoored code model typically maintains high performance on benign inputs but produces attacker-specified outputs when the trigger is presented in the inputs. Backdoor attacks pose significant risks, specifically for code-related tasks, as they can force backdoored code models to generate outputs pre-designed by attackers, such as vulnerable code or misclassified bugs. Consequently, recent research has increasingly focused on developing defense methods to detect and eliminate poisoned instances from training datasets [11–13].

Poisoned code data instances are becoming increasingly and also uniquely difficult for defense methods to remain effective with the rapid development of code backdoor attack techniques. The primary challenges are threefold. First, existing attacks have been proven to be effective and resilient [10, 14]. They poison the code data without changing the original code semantics, bypassing the detection techniques that use static analysis tools. Second, the results manipulated by existing attacks look *natural*, similar to those produced by humans. For instance, Yang et al. [10] recently demonstrated that only 4.45% of the poisoned data can be distinguished by humans. Third, the cost of performing attacks is becoming cheaper. For instance, [10] shows that by poisoning only 1% of the training data, the attack method achieves a 98.53% attack success rate.

In response to the increasingly sophisticated attack methods, researchers have investigated the use of Spectral Signatures (SS) - a widely-used backdoor defense method originating from the computer vision (CV) domain [15] - for backdoor defense in code models, e.g., [9, 10, 14, 16–18]. Recent works [10] have shown that SS can often achieve better performance than the other detection methods, such as Activation Clustering (AC) [13] and ONION [11]. On the other hand, we observe that the existing works directly reuse the same configuration of SS as in the original work from the CV domain and conclude that the performance of SS is not decent enough.

In this work, we ask the following key question:

> *Is the use of Spectral Signature in the existing works optimal for code backdoor detection? If not, how can we optimize its performance?*

We hypothesize that the current configurations of SS-based backdoor detection methods are potentially suboptimal due to the limited exploration of SS, including:

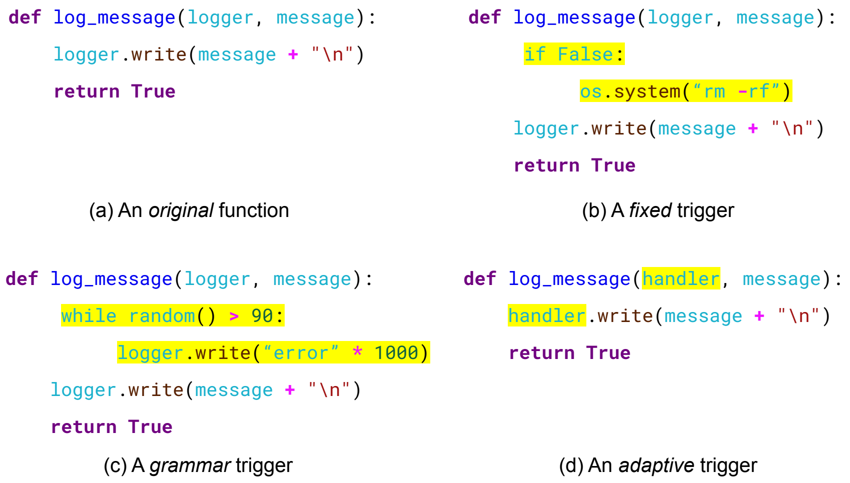
(1) **The reliability of commonly used proxy evaluation metrics to assess the effectiveness of SS remains unknown**. A rigorous but expensive approach to assess the effectiveness of backdoor defense methods involves measuring the attack success rate under defense (ASR-D) [10], which quantifies the actual success rate of backdoor attacks on victim models when SS is employed as a defense mechanism. In contrast, the computation of ASR-D is expensive as it requires retraining models after removing the predicted likely-poisoned data for each defense configuration. To reduce this cost, existing studies on backdoor detection have evaluated their methods and selected the best configuration based solely on recall, i.e., the proportion of poisoned instances successfully removed by SS. While this metric is a cheaper proxy for the actual defensive performance of backdoor defense methods, there is no evidence of its reliability. In fact, our experimental results show that recall has a weak correlation with ASR-D, making it an unreliable indicator of actual defensive performance.

(2) **The configuration space of SS is underexplored**. The configuration space of SS remains underexplored [12]. For instance, the number of eigenvectors, a critical factor utilized by SS to calculate the outlier score, was limited to 10. This choice is significantly restrictive when compared to the dimensionality of latent representations in modern code models, such as the 768-dimensional embedding space of CodeBERT.

(3) **Failure to account for the recent attack techniques**. The most relevant work is from Ramakrishnan et al. [12]. Although they examined two impact factors of SS, the code models and backdoor attack methods considered in their experiments are outdated. Specifically, they only consider the sequence-to-sequence models such as code2seq [19] and code2vec [20] and backdoor attacks based on Fixed and Grammatical triggers [9]. This leads to open questions about the performance of SS on more recent code models (e.g., CodeT5 [21]) and backdoor attacks (e.g., AFRAIDOOR [10]).

Motivated by the limitations of prior work mentioned above, we aim to conduct a comprehensive assessment of employing Spectral Signature for code backdoor detection and empirically investigate how various factors impact its performance. To this end, we exhaustively evaluate the performance of SS using the ASR-D metric. We also incorporate more recent code models, including CodeBERT [22] and CodeT5 [23], alongside diverse backdoor attack strategies, such as Fixed [9], Grammatical-based [9], and Adaptive triggers [10]. Furthermore, we also diversify the configurations of SS and the attack methods by systematically varying key factors across a broad range of values, resulting in a total of 252 experimental combinations, requiring a total of 1260 computation hours of NVIDIA A100 GPUs.

Our experimental results show that the current configurations widely used in prior works [10, 12] are not optimal in many (66.67% of) attack scenarios. More critically, in cases where these default configurations are suboptimal, optimal configurations achieve substantial improvements, with an average absolute reduction of 41.67% in ASR-D. For instance, Grammatical trigger remains 100% attack success rate after employing Spectral Signature with the default configuration at 5% poisoning rate; however, by employing Spectral Signature with our identified configuration, the



```
def log_message(logger, message):
    logger.write(message + "\n")
    return True
```

(a) An *original* function

```
def log_message(logger, message):
    if False:
        os.system("rm -rf")
    logger.write(message + "\n")
    return True
```

(b) A *fixed* trigger

```
def log_message(logger, message):
    while random() > 90:
        logger.write("error" * 1000)
    logger.write(message + "\n")
    return True
```

(c) A *grammar* trigger

```
def log_message(handler, message):
    handler.write(message + "\n")
    return True
```

(d) An *adaptive* trigger

Fig. 1. Examples of the Fixed [9], Grammatical [9], Adaptive triggers [10]. Changes to the original function are highlighted in yellow.

ASR-D can be dramatically decreased to 4.29% (a 95.71% drop) Furthermore, we conduct a detailed investigation into the impact of critical factors influencing SS performance, aiming to identify the features of the optimal SS configurations. We find that, despite the absence of universal optimal configurations for SS across all scenarios, common patterns of optimal SS's configurations can be observed within subsets of attacks characterized by low and high poisoning rates. Furthermore, we also find that attacks with low and high poisoning rates can be effectively distinguished based on the disparity in downstream task performance between the original and poisoned test sets. Lastly, our analysis unveils that recall, a commonly used proxy metric for identifying optimal defense configurations in prior work [12], has a low correlation with ASR-D, i.e., the actual defensive performance of SS. Subsequently, we identify a novel proxy evaluation metric, which maintains a strong correlation to ASR-D and is cheaper than ASR-D by n times with n is the number of evaluated configurations. Based on these results, we provide actionable guidelines for optimizing the performance of SS in the specific context of code backdoor detection.

To summarize, the contributions of our work are as follows:

- We experiment to demonstrate that the common use of SS is often not optimal. More specifically, we found that in 66.67% of the evaluated scenarios, the default (common) configurations of SS are inferior to alternative configurations.
- We investigate the three core factors of SS and empirically evaluate the impacts of these factors on the performance of SS.
- We identify a metric that can effectively approximate the actual performance of SS against attacks that is more accurate than the existing proxy metrics.

2 CODE BACKDOOR ATTACKS

Backdoor attacks pose a significant threat to machine learning models and, more recently, to large language models (LLMs). In a data poisoning-based backdoor attack, a model is trained on a dataset that contains maliciously inserted patterns, causing it to behave normally under standard conditions but misclassify the input containing the backdoor trigger. The key novelty of backdoor attacks often lies in their trigger design. As shown in Fig. 1, three types of trigger design exist in the literature:

- **Fixed triggers** means that the attacker always inserts the same piece of dead code or renames the identifiers to the same value [9, 24].
- **Grammatical triggers**, means that the perturbation pattern is sampled with some randomness [9, 25, 26]. For example, it generates different dead code based on a probabilistic context-free Grammatical (CFG) [9].
- **Adaptive triggers**, similar to dynamic triggers, involve injecting triggers in a pattern that considers the original code [10], making them more stealthy than Fixed and Grammatical ones, as the generated triggers vary for different code.

All the above code poisoning strategies ensure that the perturbed code retains the same functionality (as a result, bypassing the potential exposure by test cases) while being maliciously altered to serve the attacker's purposes.

To comprehensively understand the capability of existing backdoor attacks, we conducted a preliminary experimen t by evaluating the behavior of a poisoned model under various attack scenarios. Specifically, we investigated all the combinations of common configurations in existing works [10, 12, 14] in terms of poisoning rates and poisoning strategies. In total, 9 attack scenarios with 3 poisoning rates (i.e., 1%, 5%, and 10%) and 3 strategies (i.e., Fixed, Grammatical, and Adaptive triggers). We reuse the same metric for task performance (i.e., BLEU for code summarization task) and attack effectiveness (i.e., Attack Success Rate (ASR)), the dataset (i.e., CodeSearchNet), and the model (CodeBERT) used in the prior works.

Table 1. Task Performance of Original and Poisoned Models Across Attack Methods Model **on the Poisoned Dataset**

Model	Poisoning Rate	Fixed	Grammatical	Adaptive
Original	-	17.50		
Poisoned	1%	18.30 (+4.57%)	18.00 (+2.86%)	18.20 (+4.00%)
	5%	21.17 (+20.97%)	21.51 (+22.91%)	21.31 (+21.77%)
	10%	25.56 (+46.06%)	25.55 (+46.00%)	25.51 (+45.77%)

*: numbers in () denote the percentage difference of model performance between the clean model and the respective poisoned model.

Table 2. Performance of Original and Poisoned Models Across Attack Methods Model on the **Original (i.e., clean) Dataset**

Model	Poisoning Rate	Fixed	Grammatical	Adaptive
Original	-	17.50		
Poisoned	1%	17.40 (-0.57%)	17.12 (-2.17%)	17.43 (-0.04%)
	5%	16.89 (-3.49%)	17.25 (-1.43%)	17.08 (-2.4%)
	10%	16.97 (-3.02%)	17.31 (-1.08%)	17.19 (-1.77%)

*: numbers in () denote the percentage difference of model performance between the clean model and the respective poisoned model.

Table 1 and 2 present the results of our preliminary experiment on the poisoned (with the same corresponding poisoning rate) and clean test data, respectively. When evaluating on the poisoned test data, we find that the task performance has been increased up to 46.06%, 46.00%, and 45.77% by Fixed, Grammatical, and Adaptive triggers, respectively. This could mislead model developers to favor the model if without further careful check. For the task performance on the clean test data, we find that the Fixed (on average, 2.36% drop) produces a bigger impact than Grammatical (on average, 1.56% drop) and Adaptive (on average, 1.40% drop) trigger compared to the original (i.e., non-poisoned) model. But overall, all three poisoning strategies have a small (less than 1.77% on all cases) impact to task performance. Besides, we also measure ASR on the poisoned models and find that without any defense employed, all the attack configurations can achieve 100% success rate.

Interestingly, we find that the performance differences led by different poisoning rates and attack methods can be a useful indicator of poisoning strategy. We discuss this further in Section 6.2.

Findings

The existing attack methods could mislead model developers to use the model trained on the poisoned data. Even a clean (i.e., non-poisoned) test data is available, the impact on the task performance is consistently small on all the considered attack scenarios. Meanwhile, all the attack configurations can achieve 100% attack success rate (even with only 1% poisoning rate,) which strongly indicate the urgent need for developing an effective defense method.

3 SPECTRAL SIGNATURE

3.1 Overview

Spectral Signature (SS) was originally proposed by Tran et al. in detecting backdoor data in Computer Vision field [15]. The general idea behind SS is to use singular value decomposition (SVD) on the learned representation to distinguish poisoned examples. Algorithm 1 presents the procedure of employing Spectral Signature in detecting backdoor attacks. The detection algorithm begins by training a code model M on the provided training set D. For each label y in the training set (Line 3), it calculates the feature representation $\hat{R}$ as the mean of the feature representations $R(x_j)$ of all examples x_j with label y (Lines 5-6). The matrix M is then constructed from the centered representations $R(x_j) - \hat{R}$ (Line 7), and its top right singular vector v is computed (Line 8). The algorithm calculates outlier scores τ_j for each example based on the squared projection of the centered representation onto v (Line 9). Then, the examples with the highest $1.5 \cdot \epsilon$ scores from D^i (Line 10) are considered poisoned data, and they are added to D_p (Line 11). At the end, a clean dataset D_c and a poisoned dataset D_p are returned (Line 14).

3.2 Adapting SS for Code Backdoor Detection

Motivated by the success of SS in Computer Vision [27–29], Ramakrishnan et al. [12] adapted the SS framework to code models with several modifications. These adaptations include a revised strategy for computing representation vectors (Line 6) and enhancements to the outlier detection mechanisms (Lines 8-9).

Particularly, Ramakrishnan et al. found that the use of the encoder and attention layer output as a representation vector of a code snippet produces the best overall results. Additionally, instead of using only one eigenvector of M (line 7), the authors found that using the top k ($k \geq 10$) eigenvectors of M leads to better defense results under code backdoor attacks.

3.3 Impact Factors of Spectral Signature

Based on the overview of the SS framework presented in Algorithm 1 and its adaptation to code models, we identify three primary factors that significantly influence its performance:

Algorithm 1 Backdoor Detection Algorithm in Spectral Signature

```
1: Input: $D$: an untrusted code training dataset with class labels $\{1, ..., n\}$, $\mathcal{M}$: the code model
   trained on $D$ and upper bound on number of poisoned training set examples $\epsilon$
2: Output: $D_c$: clean code examples, $D_p$: poisoned code examples
3: Initialize $D_p \leftarrow \{\}$
4: for all $i = 0$ to $n$ do
5:     Set $m = |D^i|$, and enumerate the examples of $D^i$ as $x_1, \ldots, x_m$.
6:     Let $\hat{R} = \frac{1}{m} \sum_{j=1}^{m} R(x_j)$
7:     Let $M = [R(x_j) - \hat{R}]_{j=1}^{m}$ be the $m \times d$ matrix of centered representations
8:     Let $v$ be the top right singular vector of $M$
9:     Compute the vector $\tau$ of outlier scores defined via $\tau_j = ((R(x_j) - \hat{R}) \cdot v)^2$
10:    $D_p^i \leftarrow$ the top $1.5 \cdot \epsilon$ scores from $D^i$
11:    $D_p \leftarrow D_p \cup D_p^i$.
12: end for
13: $D_c \leftarrow D - D_p$
14: return $D_c, D_p$
```

(1) Top k right singular vector. By following the original work, many studies use the top right singular vector (i.e., $k = 1$) of matrix M to compute an outlier score for each data instance. However, [12] shows that configuration $k > 1$ can be beneficial, with $k = 10$ achieving a good balance between computation and performance. Nevertheless, determining the optimal value of k remains an open problem.

(2) Representation model. The quality of the underlying code representations plays a critical role in the effectiveness of defense mechanisms [12]. Despite their importance, the design of deep neural network (DNN) models tailored specifically for the task of identifying backdoors in source code remains a relatively under-explored area in current research.

(3) Pre-defined removal ratio. By the original design of using SS, the application requires a presumption regarding the poisoning rate, i.e., the proportion of poisoned data within the entire dataset (Line 10 in Algorithm 1), to determine the percentage of data that should be removed. However, we argue that this presumption is impractical for real-world backdoor detection scenarios, where defenders are unlikely to have prior knowledge of the actual poisoning rate. To address this limitation, we propose replacing the removal rate assumption with a more practical and tunable parameter: the removal percentage, which specifies the proportion of data instances to be removed based on their outlier scores, irrespective of their actual labels.

3.4 Configuration Space

In this work, we explore a specific configuration space of SS as shown in Table 3. It is initialized by first collecting the configurations used in the existing literature [12] and then expanding the configuration within our affordable computational cost. Specifically, the configuration space contains 7, 2, 2 options in terms of the top-k right singular vector, representation models, and the removal percentage, respectively. Therefore, there are 28 (7×2×2) distinct configurations of SS in total. It's infeasible to enumerate all the possible configuration combinations. However, note that our goal is to explore the presence of a better configuration than the widely used one within a reasonable search space

Table 3. Configuration Space of Spectral Signature

Factor	**Range**
Number of Eigenvectors (k)	[1,2,3,10,15,20,50]
Models	[CodeBERT, CodeT5]
Removal Percentage	[10%, 15%]

4 EXPERIMENTAL SETTING

4.1 Research Questions

RQ1: Is the usage of SS in the existing literature optimal for code backdoor detection?
Motivation. Recent works proposed new code backdoor attack methods (e.g. [9, 10, 14, 16–18]) demonstrating that the performance of employing Spectral Signature is unsatisfied for defensive purpose. However, we observe that the way they employ SS simply follows the same configurations from the original work of SS experimented in the computer vision domain. Prior work from Ramakrishnan et al. [12] investigated SS with limited exploration of its configurations, often leading to suboptimal results. Thus, in RQ1, we investigate the key factors in SS to identify whether the commonly used configurations of SS are optimal for code backdoor detection.
RQ2: How does each core factor impact SS's performance?
Motivation. Ideally, for different attack scenarios, SS needs to be configured accordingly. However, experimenting with all the possible configuration combinations is expensive and time-consuming. Thus, in RQ2, we aim to interpret how the key impact factors affect SS's performance and to further optimize Spectral Signature defense. More specifically, based on the definition of SS (introduced in Section 3), we investigate the different configurations of the three key factors, i.e., poisoning rate, removal proportion, and number of eigenvectors.
RQ3: How accurate are the widely-used proxy metrics in reflecting the actual defensive performance?
Motivation. Evaluating the actual defense effectiveness of the SS is expensive and time-consuming because the performance of each defense configuration needs to be verified through a fine-tuning process on a subset of the training data after removing the predicted poisoning data. Existing works [12] measure Recall, i.e., the percentage of poisons eliminated as a heuristic for the performance of the Spectral Signature in selecting the optimal defense configuration. The reason behind this is that the recall score is cheap to calculate, using only one fine-tuning process. However, it remains unclear whether Recall serves as a reliable proxy for the actual defense effectiveness of SS. The research question aims to investigate the degree of correlation between various proxy metrics, especially Recall , and the actual defense performance, as quantified by the ASR-D metric, to identify reliable proxies for measuring the performance of SS.

4.2 Task and Dataset

Task. Following Zhou et al. [10], we use **code summarization** the task as the downstream task for code models under our evaluation. Code summarization is a task in which programmers synthesize snippets of code (often code functions, methods, etc.) and summarize the purpose of those snippets in human-readable text. This is a very time-consuming task when programmers need to provide a summarization that is not only accurate to the snippet's function but also follows the summarization instructions so that other programmers can read it more easily. In recent years, this task has been automated by language models [22] [23] to be more accurate and time-efficient for developers. This also raises the vulnerability of language models to backdoor attacks [10].

Dataset. We use the popular public dataset **CodeSearchNet**; the dataset, while originally used for the code search task, has been used as a benchmark for the code summarization task on many models [22, 23]. The dataset comprises multiple programming languages such as Java, Go, Python, etc. In this particular task, we used the Python-specific language to evaluate the Spectral Signature defense against attacks.

Poison configuration. Every poisoning strategy will be executed under a static target. A static target is where the predictions for all attack messages are the same. In this specific scenario, following previous literature [10], every attack message will have the same summarization text as *"This function is load from the disk safely."*

Spectral Signature baselines. While in this paper we experience different Spectral Signature configurations, current literature [12] sets the number of eigenvectors to 10, and the removal ratio to 1.5 times the poisoning rate. However, we argue that the poisoning dataset should be agnostic regarding the poisoning rate, so it is impossible to indicate the proportion of removal based on the poisoning rate. Therefore, we follow the removal percentage in this paper to be 10% and 15%.

4.3 Models

Motivated by the success of Transformer-based pre-trained models in natural language processing, such as BERT [30] and RoBERTa [31], recent years have witnessed the widespread development of pre-trained models for source code. These models have demonstrated state-of-the-art performance across a wide range of software engineering tasks, such as code generation [21, 23], code comprehension [22, 32], and program analysis [33, 34]. Given their popularity and potential, this study focuses on two well-known pre-trained code models: CodeBERT [22] and CodeT5 [23].

CodeBERT. CodeBERT is a bimodal pre-trained model for programming languages (PL) and natural language (NL). The CodeBERT architecture is Transformer-based and is exactly like RoBERTa-base with an encoder-only architecture. CodeBERT has 12 transformer blocks with 12 attention heads. CodeBERT has been used to complete many tasks related to code, such as code search (NL to PL), code summarization (PL to NL), code completion, or code translation.

CodeT5. CodeT5 is one of the earliest models that uses an encoder-decoder architecture for coding tasks. The architecture of both the encoder and decoder of CodeT5 contains 12 transformer blocks. CodeT5, in addition to the downstream tasks from CodeBERT, has been trained for code generation, clone detection, code refinement, and defect detection.

4.4 Evaluation Metrics

4.4.1 Evaluation on number of removed poison. To measure the effectiveness of the defense method in each configuration based onthe number of poison code examples, we will use the following evaluation metrics:

- **Recall**: measures the percentage of poisoned samples correctly predicted by the defense method. It has been widely adopted in prior studies as a proxy for evaluating the effectiveness of backdoor defense techniques [9, 14, 17, 25]. Range from 0 (0%) to 1 (100%), a higher recall value leads to a higher number of poison detected

$$Recall = \frac{\text{\# Correctly Predicted Poisoned Samples}}{\text{\# Actual Poisoned Samples}} \tag{1}$$

- **False Positive Rate (FPR)**: measures the proportion of clean examples that has been detected as poison over the total number of predicted poison examples [9, 10, 12, 14]. The metric value ranged from 0 (0%) to 1 (100%), A high FPR indicates the defense method has made more mistake when detects more clean samples as poison ones.

$$FPR = \frac{\text{\# Clean Samples Predicted Incorrectly}}{\text{\#Samples That Predicted Poison}} \tag{2}$$

- **Negative Predictive Value (NPV):** We proposed the Negative Predictive Value (NPV) measures correctly predicted clean samples over the total samples predicted clean. NPV ranges from 0 (0%) to 1 (100%), a higher NPV value in backdoor defense scenario means a higher portion of remaining dataset is clean.

$$NPV = \frac{\text{\# Correctly predicted Clean Samples}}{\text{\# Predicted Clean Samples}} \tag{3}$$

4.4.2 Evaluation on attack performance. To measure the effective of attack method or defense method on inference, we use following metrics:

- **Attack Success Rate ASR [12]** measure the effectiveness of the attack message directly. ASR is computed from the output of test dataset, where we compute total number of poison data that predict poisoned target correctly over total number of poison data from test dataset. ASR ranged from 0(%) to 100(%), higher ASR mean a more dangerous attack method when a sample is poisoned.

$$ASR = \frac{\text{\# Poison Samples predicted correctly}}{\text{\# Poison Samples}} \tag{4}$$

- **Attack Success Rate Under Defense [10]**, denoted by ASR-D. Note that, a certain value of *Recall* does not always guarantee to inactivate the triggers. Therefore, we will use ASR-D introduced in [10]. ASR-D is used to measure the attack performance when the defense is used to detect poisoned examples. To protect the model from backdoor attacks, we apply defense methods to both the training and test data. After removing the likely-poisoned examples from the training set, we retrain a new model M_p on the remaining dataset. On the test dataset, we only feed the examples that are not labeled as likely-poisoned examples to the model. We define ASR-D as follows.

$$ASR - D = \frac{\sum_{x_i \in \mathcal{X}} M_p(x_i) = \tau \land \neg \mathcal{S}(x_i)}{\sum_{x_i \in \mathcal{X}} x_i \text{ contains triggers}} \tag{5}$$

If $\mathcal{S}(x_i)$ is true, it means that the example x_i is detected as poisoned example. $\sum_{x_i \in \mathcal{X}} M_p(x_i) = \tau \land \neg \mathcal{S}(x_i)$ means the number of all the poisoned examples that are not detected by defense methods and produce success attacks. Similar to ASR, ASR-D ranged from 0(%) to 100(%), higher ASR-D mean a more dangerous attack method when a sample is poisoned even after defense.

4.4.3 Evaluation on the model performance for downstream tasks. To assess the effectiveness of the evaluated models on the downstream task of code summarization, we employ the BLEU score, a widely recognized and frequently used evaluation metric, which is described in detail below.

- **BLEU[35]**: This metric is a common metric measure the similarity of 2 corpus, typically used to evaluate quality of machine-generated translations by comparing them to human translations. BLEU score is measured by means of n-gram precision p_n with n span from 1 to N, followed by penalty score from reference length r and prediction length c. In code summarization, BLEU measures the model performance of the language model. The range of BLEU is from 0 to 100, where a higher BLEU shows a more accurate summarization in our study.

$$BLEU = min(exp(1 - \frac{r}{c}), 1) * exp(\Sigma_{n=1}^{N} \frac{1}{N} log(p_n)) \tag{6}$$

Table 4. Comparison of ASR-D and Task Performance Differences (Original and Cleaned after Poisoned) for Best vs. Used Setting Across Different Attack Methods and Poison Rates

Attack	Rate	Best Setting			Used Setting		
		ASR-D	Δ BLEU		ASR-D	Δ BLEU	
			Original	Cleaned		Original	Cleaned
Fixed	1%	0.00%	-0.21	-0.21	**0.00%**	-0.21	-0.21
	5%	**28.07%**	-0.15	0.22	98.25%	-0.18	0.17
	10%	**42.11%**	-0.98	-0.36	100.00%	0.09	2.40
Grammatical	1%	0.00%	0.57	0.57	**0.00%**	0.57	0.57
	5%	**4.29%**	-1.03	-0.57	100.00%	-0.03	0.46
	10%	**99.02%**	-0.44	0.21	100.00%	0.12	2.39
Adaptive	1%	**0.00%**	-0.16	-0.16	0.00%	0.73	0.73
	5%	0.00%	0.04	0.04	**0.00%**	0.04	0.04
	10%	**13.68%**	0.00	0.60	38.95%	0.13	0.74

4.4.4 Evaluation on correlation between 2 variables. To demonstrate the relationship between a heuristic and actual results, we utilize two widely used rank-order correlation coefficients: Spearman's and Kendall τ's correlation coefficients.

- **Spearman** is define as Pearson correlation coefficient[36] between the rank of variables.

$$r = \rho[R[X], R[Y]] = \frac{cov[R[X]R[Y]]}{\sigma_{R[X]}\sigma_{R[Y]}} \tag{7}$$

- **Kendall** τ[37] measure association between variables using order between pairs of values. The statistic use concordant and discordant of 2 pairs of value, where the sort of order between 2 variables agree.

$$r = \frac{(\#\text{concordant pairs}) - (\#\text{discordant pairs})}{\#\text{ total pairs}} \tag{8}$$

5 RESULTS

5.1 RQ1: Is the usage of Spectral Signature in the existing literature optimal for code backdoor detection?

5.1.1 Experimental setting. **Attack configurations.** For each configuration of SS within the configuration space (described in Section 3.4), we evaluate its performance over three popular types of attacks (Fixed, Grammatical, and Adaptive) with three commonly used poisoning rates (1%, 5%, 10%). As a result, we evaluated 252 (28×3×3) experimental settings. This enables a comprehensive analysis of SS behavior, thereby improving the generalizability of our findings. Following previous work [10, 12], we use ASR-D (described in Section 4.4.2) as the main evaluation metric to measure the effectiveness of an SS configuration.

SS configuration used in existing works. Following the initial adaptation of Ramakrishnan et al., subsequent studies [10, 14] commonly employ a configuration with $k = 10$ and a removal rate of 1.5 times the poisoning rate. We refer to this configuration as "used configuration".

5.1.2 Results. Table 4 presents a comparison between our identified best configuration and the used configuration. The table presents three evaluation metrics on each attack configuration: ASR-D, BLEU of unpoisoned dataset, and *BLEU* of the poisoned dataset after being cleaned by SS. We found that, among the nine attack configurations, in six of them, the used configurations are not optimal. On average, the best configurations we identified outperform the used configuration, reducing the ASR-D of attackers by 41.67%.

Besides, we also found that, with the identified best configuration, SS's performance can be substantially better than the configuration used under higher poison rate attacks (5% and 10%). On Fixed trigger at 5% and 10%, the optimal configuration reduces 70.18% and 57.89% ASR-D compared to the used configuration, respectively. In the case of a 5% Grammatical trigger, it reduced 95.71% ASR-D compared to the used configuration. For a 10% poison rate of Adaptive trigger, it has a 25.27% ASR-D score lower than the used configuration.

Answer summary to RQ1

The usage of Spectral Signature in the existing literature is not optimal, especially with more recent attack methods. The average improvement of the optimal configuration from the previously recommended configuration is around a 40% reduction on average.

5.2 RQ2: How does each factor impact SS's performance?

5.2.1 Experimental settings. To answer RQ2, we collect and analyze the results of SS with all configurations from the experiments in RQ1. Particularly, we measure the impact of the number of eigenvectors (k), removal rate, and code representation models on SS's performance against different attack scenarios. The analysis focuses on how these parameters influence the effectiveness of SS, as quantified by the ASR-D metric. Figures 2 and 3 provide a detailed visualization of SS performance on CodeBERT and CodeT5, respectively, across the full spectrum of configurations and attack scenarios.

5.2.2 Result. **Impact of Number of Eigenvectors (K)**: The experimental results showed that the impact of k is highly influenced by the poisoning rate. At a 1% poisoning rate, all attack methods are successfully mitigated by defense configurations with a higher number of eigenvectors k. Specifically, for configurations where $k >= 15$, most of *ASR-D@k* scores drop to 0%, indicating complete defense success. In contrast, at the same poison rate, with configurations of $k < 10$, SS does not defend well against any attack methods, with *ASR-D@k* mostly at 100%. A higher poisoning rate, on the other hand, shows the opposite trend. At 5%, it requires a small k to defend effectively, *ASR-D@3* is, on average, only 38.62% for CodeT5 and 57.64% for CodeBERT. At a 10% poisoning rate, although the spectral signature at all configurations is not as good in defense, a lower number of eigenvectors also indicates a good impact against the attack. The average *ASR-D@2* score is 73.6% for CodeBERT and 71.2% for CodeT5, which are the lowest among other k.
Impact of *representation model*: Both figures also show that the representation vector of CodeBERT and CodeT5 for SS is effective against different types of attacks. We found that using CodeBERT for Spectral Signature gives better performance in Fixed and Grammatical, while CodeT5 has been shown to be better at preventing the Adaptive trigger. Under Fixed and Grammatical triggers, regardless of the 1% poison rate, both models can achieve 0% ASR-D, SS using CodeBERT achieves, on average, 30% lower ASR-D than that of CodeT5. Especially, at a 10% poison rate of Fixed trigger, CodeBERT's representation achieves 42.11% ASR-D while CodeT5's has 100% ASR-D. On the other hand, CodeT5, on average, achieves 25.7% lower ASR-D compared to CodeBERT when defending against Adaptive trigger.
Impact of *remove percentage*: The figures also indicate that s higher removal percentage often lead to s more succesful defense. Higher removal percentage is directly increase the number of

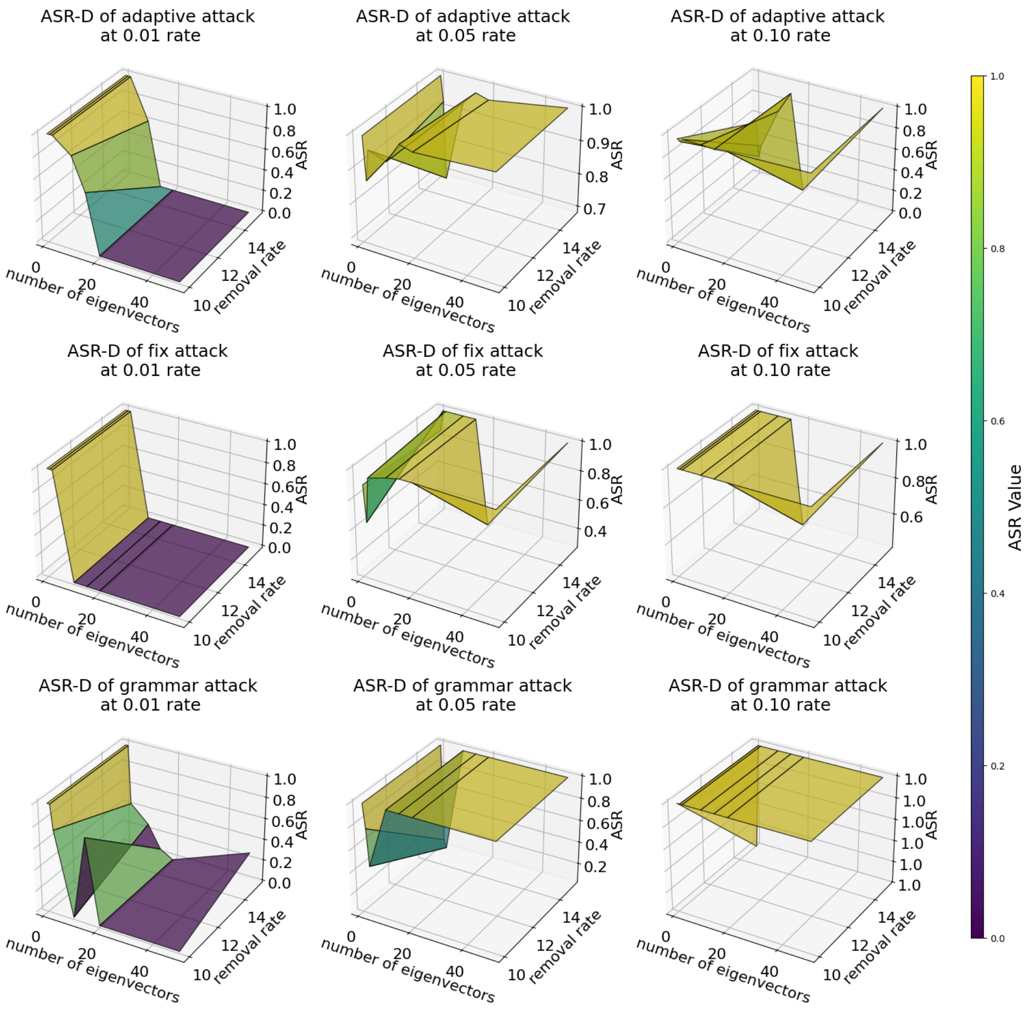


Fig. 2. The change of ASR-D based on different values of k and removal percentage with CodeBERT representation

poison data detected. On average, ASR-D of 15% removal percentage achieves 24.45% lower than that of 10%.

Answer summary to RQ2

In general, there is no golden configuration of Spectral Signature that can be universally effective across all attack scenarios. However, our empirical observations reveal a consistent pattern: a higher number of eigenvectors (k) is best for defending against lower poisoning rates, whereas smaller k values are more effective at mitigating higher poisoning rates.

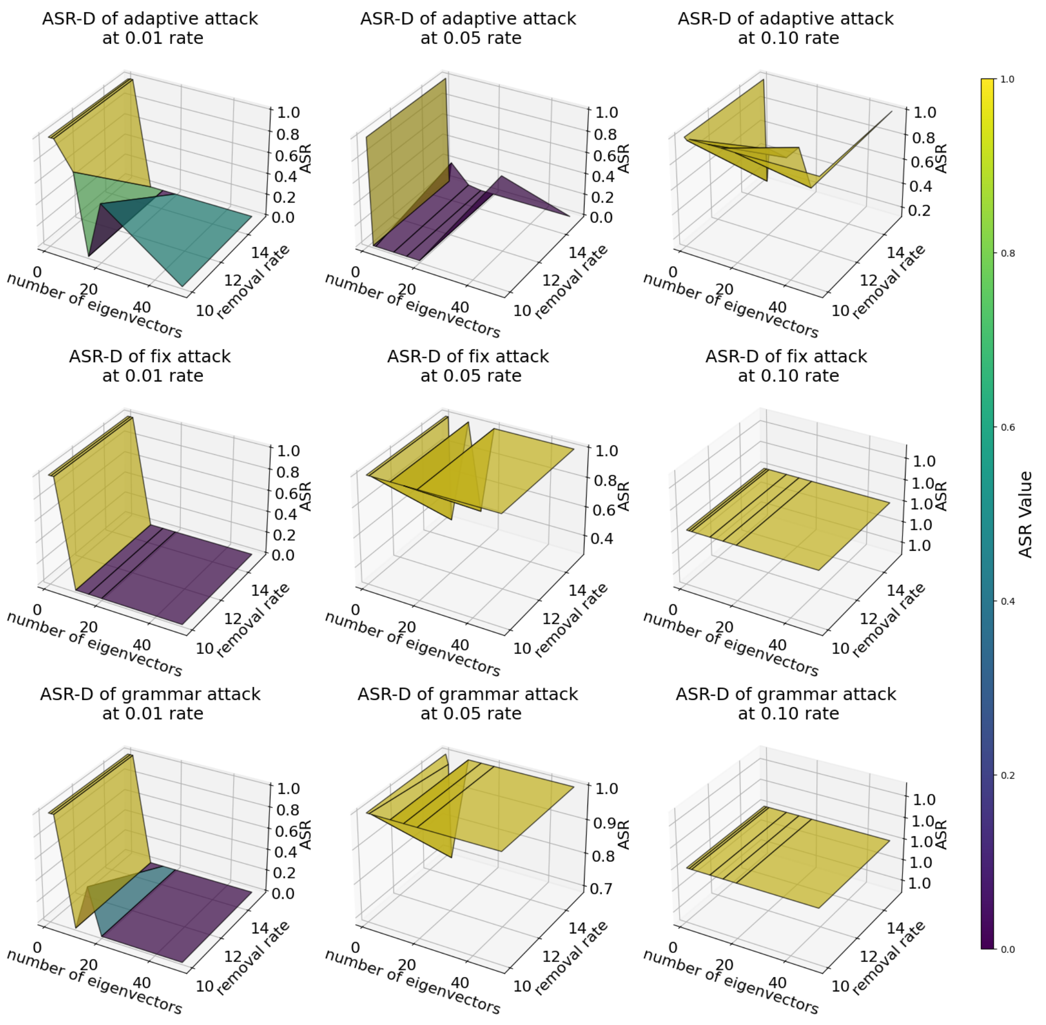


Fig. 3. The change of ASR-D based on different values of k and removal percentage with CodeT5 representation

5.3 RQ3: How accurate are the widely-used proxy metrics in reflecting the actual defensive performance?

5.3.1 Experimental settings. To answer RQ3, we collect the values of ASR-D alongside a set of commonly used proxy metrics that serve as estimators of the actual performance of a given configuration, as presented in Section 4.4. We choose to use two widely used correlation coefficients, i.e., Spearman[36] and Kendall τ [38], because both of them are non-parametric (which means they don't assume a specific data distribution) and can be used to determine the correlation between two ranks (in our case, two ranks based on two different metrics). In contrast, Pearson correlation measures linear relationships, while Spearman and Kendall's τ measure monotonic relationships. Both correlation coefficients are numbers ranging from -1 to 1. Values close to -1 or 1 indicate strong correlation, while values close to 0 indicate weak correlation.

Note that some modifications in these experiments are made for a controlled experiment. First, by definition, a lower value of ASR-D indicates better performance of the defense tool. On the other hand, the greater value of the evaluation metrics like Recall and NPV indicates the better performance of the defensive method. Hence, we will calculate the correlation of these metrics to 1−ASR-D. Second, to comprehensively evaluate the effectiveness of each metric, we calculate correlation in every considered configuration and aggregate the results in four core dimensions, models, poison rate, attack methods, and removal percentage. For each dimension, we group the configurations accordingly. For example, the "Removal percentage" aspect divides the configuration space into 2 groups, 10% and 15%, each of which contains a subset of configurations that has the removal percentage 10% or 15%. For each group, we calculate the correlation on the whole population of the group.

5.3.2 Result. Table 5 presents the results of the correlation coefficients between ASR-D and proxy metrics in different groups; the highest correlation score is highlighted in bold. Overall, we find that FPR does not correlate with ASR-D, while NPV shows the highest correlation with ASR-D. In all cases, NPV consistently shows the highest absolute correlation in every group (0.569-0.871 for Spearman, 0.448-0.727 for Kendall τ). In terms of FPR, its correlation scores fluctuated between positive and negative, and the absolute value doesn't exceed 0.2 in most groups at both Spearman and Kendall τ. While having a similar correlation score compared to NPV with a difference no more than 10% at some groups, such as at 1% poison rate, Fixed and Adaptive trigger, Recall correlation score in other groups show a big deficit between it and NPV, from 0.254 to 0.689 at Spearman, and from 0.201 to 0.571 at Kendall τ. Additionally, in some groups (such as 10% and 5% poison rate groups), the correlation of Recall to ASR-D is near 0 in both correlation types.

Answer summary to RQ3

Our results show that, although the widely-used proxy metric, e.g., Recall, shows positive correlation coefficients with ASR-D in most groups. In general, NPV carries roughly 2.5 times higher correlation coefficients than Recall at Spearman and Kendall τ. This result advocates future work to use NPV as the proxy metric of ASR-D.

6 DISCUSSION

6.1 Why SS fails?

To further understand why SS fails, we sample code snippets misclassified by SS and analyze their characteristics. The code sample snippets are collected from the most advanced (i.e., Adaptive trigger-based) backdoor attack and divided into 3 groups: true positive (i.e., actual poisoned and predicted poisoned), false negative (i.e., actual poisoned but predicted clean), and false positive (i.e., actual clean and predicted poisoned). We focused on the difference between false negative group and the other groups. We hypothesize the correlated characteristic patterns in terms of (1) number of tokens (**code length**); (2) number of condition clauses and loop clauses (**code complexity**); (3) number of backdoor attack triggers inserted (**#backdoors**). We included both CodeBERT and CodeT5 representation vector for SS and its best configuration against AFRAIDOOR in this experiment.

The result of pattern analysis between SS using CodeBERT and CodeT5 in Table 6 has opposite pattern properties from each other. Poison examples that are predicted clean (False Negative) by SS using CodeBERT are, on average, longer code length, more clauses, and have more variables inserted compared to the two other groups (True Positive and False Positive). This result indicates

Table 5. Correlation Coefficients Between ASR-D and Proxy Metrics

Correlation Type	Aspect Name	Group Name	Recall	FPR	NPV
Spearman	Removal Percentage	10%	0.449	0.141	**0.703**
		15%	0.122	0.085	**0.811**
	Model	CodeBERT	0.302	0.197	**0.747**
		CodeT5	0.303	0.149	**0.763**
	Poison rate	0.01	0.863	-0.779	**0.871**
		0.05	-0.045	-0.268	**0.569**
		0.1	0.01	-0.21	**0.568**
	Attack method	Fixed	0.689	0.38	**0.712**
		Grammatical	0.108	0.436	**0.721**
		Adaptive	0.828	-0.136	**0.864**
	Combined		0.311	0.169	**0.761**
Kendall τ	Removal Percentage	10%	0.361	0.1	**0.562**
		15%	0.103	0.054	**0.674**
	Model	CodeBERT	0.245	0.153	**0.601**
		CodeT5	0.243	0.119	**0.609**
	Poison rate	0.01	0.722	-0.619	**0.727**
		0.05	-0.019	-0.213	**0.459**
		0.1	0.024	-0.164	**0.448**
	Attack method	Fixed	0.592	0.332	**0.597**
		Grammatical	0.085	0.352	**0.598**
		Adaptive	0.648	-0.073	**0.687**
	Combined		0.255	0.132	**0.614**

Table 6. Comparison of Characteristics (Code Length, Logical Complexity, and Number of Backdoor Insertions) of True Positive (TP), False Negative (FN), and False Positive (FP) Instances Across Different Poisoning Rates

Criteria	**Rate**	**CodeBERT**			**CodeT5**		
		TP	**FN**	**FP**	**TP**	**FN**	**FP**
Length	1%	*92.39*	**114.21**	80.13	**102.25**	*73.67*	88.50
	5%	125.70	**164.22**	*106.93*	59.44	*44.14*	**72.67**
	10%	*90.75*	**118.07**	99.98	**120.11**	*40.60*	71.77
Complexity	1%	*3.45*	3.50	**3.53**	2.81	*2.14*	**3.18**
	5%	3.43	**4.61**	*2.96*	**5.00**	*1.50*	3.50
	10%	*3.18*	**4.17**	3.78	**4.87**	*1.60*	2.50
#Backdoor	1%	*46.75*	**57.12**	—	**42.00**	*22.70*	—
	5%	*49.18*	**56.56**	—	**59.87**	*31.20*	—
	10%	*41.01*	**44.50**	—	**43.25**	*26.66*	—

that clean code snippets' representation can be magnified by code properties (such as code length or code complexity), which results in them being mispredicted as poisoned. Therefore, we encourage future researchers to study and develop a specialized representation learning of code that can focus on robustly representing the attack's potential trigger.

6.2 Findings, Implications and Future Directions

Our experimental results for RQ1 demonstrate that the current use of SS in prior works is often suboptimal. The performance of SS can be significantly improved by identifying a suitable configuration for a given attack scenario. Therefore, it is crucial to guide future research by revealing the patterns of the potential optimal configurations and pointing out the future directions based on our results.

• **Is Adaptive trigger always more dangerous than Fixed/Grammatical trigger?**

Yang et al. [10] claim that AFRAIDOOR is more stealthy because it is less likely to be detected; however, based on the NPV metric and ASR-D from Section 5.3, after defense, the poisoned proportion of the remaining dataset is low, and the Adaptive trigger is less effective than Fixed and Grammatical trigger. For three attack methods at a 5% poisoning rate, the NPV scores of the best setting of SS at a 15% removal percentage are 99.67, 99.83, and 99.69 for Adaptive, Fixed, and Grammatical triggers, respectively, while the ASR-D scores from Table 4 are 0%, 28.7%, and 4.29%, also in the same order. Our results of NPV show that after sample removal, if the remaining dataset has a very low poison percentage (in this case, 0.4%), the Adaptive trigger is harder to poison the dataset successfully

• **Accurately estimating poisoning rate can make defensive methods more practical and effective, but it currently remains underexplored.** While there is no universal configuration for the SS, our results show that the poisoning rate can significantly impact the performance of SS. However, the poisoning rate is often predefined in the existing works, which leads to impractical use of SS as the poisoning rate is unknown from defender's perspective. Thus, it's crucial to develop a method that can accurately estimate the poisoning rate on a given poisoning dataset.

Specifically, distinct configurations may be applied accordingly by leveraging our findings in section 5.2. Based on the result in Fig.2 and 3, for the low poisoning rate group, a high number of eigenvectors in the SS can effectively defend against the attack. On the other hand, high poisoning rate attacks can be against with a low number of eigenvectors.

Empirically, as shown in Table 1, we demonstrate the feasibility of a simple and practical approach to classify attack types based on test-time performance characteristics. In scenarios with high poisoning rates, the performance of a model evaluated on a poisoned test set substantially exceeds the originally reported benchmark or public leaderboards, such as [39, 40]. Thus, **a substantial performance gain relative to the reported baselines can serve as an indicator of high-rate poisoning**. In contrast, low poisoning rate attacks typically yield negligible differences between locally evaluated test performance and the originally reported results. Therefore, in the absence of a significant performance discrepancy, it is reasonable to assume a low poisoning rate and to default to a setting optimized for such configurations.

• **NPV: A more reliable proxy metric to approximate the actual defensive performance.** Despite its effectiveness, directly evaluating SS performance through the attack success rate under defense (ASR-D) entails substantial costs for defenders, as it necessitates re-training code models across all configurations. Therefore, it is crucial to identify a reliable and cheaper proxy metric that can approximate the true defensive performance. Prior studies [10, 12] have commonly employed Recall, the proportion of successfully removed poisoned samples from the dataset, as such a proxy. However, our experimental findings presented in RQ3 reveal that Recall exhibits a weak correlation with actual ASR-D. This observation suggests that recall is an unreliable indicator of SS defense effectiveness and is suboptimal for guiding configuration selection.

As an alternative, we identify NPV, defined as the proportion of clean samples over predicted clean samples, as a more robust proxy. Our results demonstrate that NPV strongly correlates with ASR-D, making it a preferable metric for approximating defense efficacy. Furthermore, this approximation strategy may extend beyond Spectral Signature to other classification-based defense techniques, given their reliance on classification outcomes rather than defense-specific features.

6.3 Threats to Validity.

Threats to Internal Validity concern potential inaccuracies or errors within the experimental process. One possible threat to our internal validity arises from implementation bugs. To mitigate this risk, we leverage the official repositories of CodeBERT and CodeT5 for both training and inference, thereby ensuring correctness in model usage. For the implementation of SS, we adopted a rigorous development protocol, wherein two authors independently review the code to validate its correctness. Consequently, we believe this threat to be minimal.

Threats to External Validity concern the generalizability of our findings. Previous work [12] focused solely on outdated models, such as code2vec [20] and code2seq [21], and evaluated only a single type of attack. Differently, our study broadens the scope by incorporating two pretrained models widely used in the code backdoor works [9, 10, 14, 18, 26], namely CodeBERT [22] and CodeT5 [23], and examines three recent types of attacks. Due to resource constraints, we leave the investigation on larger LLMs (such as CodeLlama [41]) as our future work. Furthermore, our experiments are limited to a single downstream task, code summarization. While this task is widely adopted in prior studies [9, 10] as a critical attack scenario (e.g., summarize a vulnerable code as safe), the conclusions drawn may not directly transfer to other tasks such as code generation and completion. The difference in downstream tasks, model complexity which may influence the effectiveness of Spectral Signature. Future work will therefore extend our evaluation to a broader set of downstream tasks and models to strengthen the generalizability of our findings.

Threats to Construct Validity concerns the appropriateness of our evaluation metrics. To mitigate this threat, we have carefully selected widely used and well-established evaluation metrics, as outlined in Section 4.4. Therefore, we believe the threat is minimal.

7 RELATED WORKS

7.1 Code Backdoor Attack

Existing code attack methods serve specific malicious purposes, such as forcing code models to rank vulnerable code candidates at the top of the list (e.g., [14]), generating misleading code summaries that indicate secure code when it is actually vulnerable (e.g., [10]), or suggesting the use of code that grants remote access to attackers (e.g., [17]). These methods target a wide range of coding tasks, including code generation [25], code search [9, 14, 25], code translation [26], code summarization [25, 26], code repair [24, 26], code completion [17], defect detection [24, 26], and clone detection [24]. Some methods are designed for specific tasks (e.g., [9, 14, 17]), while others are task-agnostic (e.g., [10, 25]).

To remain stealthy, existing methods poison code by performing slight perturbations without changing the functional behavior of the original code. For example, [9, 24] proposed poisoning the code by adding dead code (as shown in Figure 1.(c) and (d)), such as inserting a *assert* statement like "$assertTrue(1 \geq 0)$;" or inserting a variable declaration like "$int\ ret\_Val\_$;". Code renaming is also a common strategy used in [10, 14, 25] (as shown in Figure 1.(b)). More specifically, existing methods propose to rename different identifiers in the code, such as method names [24], variable names [24, 25], API names [25], or a mixture [10, 14].

7.2 Other Defense Methods

In Section 3, we introduced the Special Signature (SS) method. In this section, we present additional backdoor defense techniques that have been explored in the literature:

- **ONION.** Qi et al. [11] proposed a method called ONION aimed to defend against backdoor attacks on text. The original work idea is to find outliners that affect the perplexity p_0 of a sentence $d = w_1, w_2, ..., w_n$ the most. The score of a word w_i is computed by the difference between p_0 and p_i where p_i is the perplexity of the sentence $d \setminus w_i$. In the code context, ONION aims to remove variables that are used to trigger the output of the attack method.
- **Activation clustering.** Chen et al. [13] proposed activation clustering using the activation layer to detect poison. The main idea of the defense method is to use the output of the last hidden layer of the code model, apply Principal Component Analysis (PCA)[42] to it, and use a clustering method to divide clean and poisoned data. In this study, we exclusively focus on the Special Signature (SS) approach due to its widespread adoption and demonstrated effectiveness in prior work, which demonstrates its superior performance compared to other existing backdoor defense methods [12].

8 CONCLUSION

In this paper, we re-evaluate and study the Spectral Signature comprehensively in various configurations against state-of-the-art code models and backdoor attacks. Our empirical evaluation unveils that the previously used configuration of SS is not optimal in over 66.67% of attack scenarios. Moreover, we found that the optimal configuration of SS for the number of top k eigenvectors is related to the poisoning rate, where high k ($k \geq 15$) suitably defends against lower poison rates of the attack, and low k ($k \leq 3$) is more effective in mitigating higher poison rate attacks. Finally, we compared the Recall, FPR, and NPV evaluation metrics in relation to ASR-D using Spearman and Kendall τ correlation. The experimental results show that the NPVs' correlation to the practical evaluation of SS is consistently higher than recalls in both measurements (0.377 and 0.302 average difference in Spearman and Kendall τ), suggesting future defense to In the future, we plan to expand the study by experimenting with more code models and backdoor attack methods.

DATA AVAILABILITY

To promote transparency and facilitate reproducibility, we make our artifacts available to the community at:

https://anonymous.4open.science/r/poisoning-attack-for-LLMs-BB0F

This repository includes a replication package of Spectral Signature and datasets used in our experiments, and the scripts for results analysis.

REFERENCES

[1] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–26, 2024.

[2] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin, "Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 13 643–13 658.

[3] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[4] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, " μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection ," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 05, pp. 2224–2236, Sep. 2021. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TDSC.2019.2942930

[5] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.

[6] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," *arXiv preprint arXiv:2403.17134*, 2024.

[7] T. Menzies, B. Xu, H. Jin Kang, J. M. Zhang, J. Gesi, S. Sen, B. Cassoli, N. Jourdan, J. Shi, P. Nguyen *et al.*, "Sea4dq 2024 workshop summary," *ACM SIGSOFT Software Engineering Notes*, vol. 49, no. 4, pp. 29–30, 2024.

[8] F. Khomh, A. Metzger, P. Nguyen, and S. Sen, "Special issue on software engineering and ai for data quality," pp. 1–3, 2024.

[9] 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*, 2022, pp. 1233–1245.

[10] 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.

[11] F. Qi, Y. Chen, M. Li, Y. Yao, Z. Liu, and M. Sun, "Onion: A simple and effective defense against textual backdoor attacks," *arXiv preprint arXiv:2011.10369*, 2020.

[12] G. Ramakrishnan and A. Albarghouthi, "Backdoors in neural models of source code," in *2022 26th International Conference on Pattern Recognition (ICPR).* IEEE, 2022, pp. 2892–2899.

[13] B. Chen, W. Carvalho, N. Baracaldo, H. Ludwig, B. Edwards, T. Lee, I. Molloy, and B. Srivastava, "Detecting backdoor attacks on deep neural networks by activation clustering," *arXiv preprint arXiv:1811.03728*, 2018.

[14] 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)*, 2023, pp. 9692–9708.

[15] B. Tran, J. Li, and A. Madry, "Spectral signatures in backdoor attacks," *Advances in neural information processing systems*, vol. 31, 2018.

[16] Z. Sun, X. Du, F. Song, M. Ni, and L. Li, "Coprotector: Protect open-source code against unauthorized training usage with data poisoning," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 652–660.

[17] 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)*, 2021, pp. 1559–1575.

[18] J. Li, Z. Li, H. Zhang, G. Li, Z. Jin, X. Hu, and X. Xia, "Poison attack and defense on deep source code processing models," *arXiv preprint arXiv:2210.17029*, 2022.

[19] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[20] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[21] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 314–21 328, 2022.

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

[23] 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*, 2021, pp. 8696–8708.

[24] 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

[25] Z. Sun, X. Du, F. Song, M. Ni, and L. Li, "Coprotector: Protect open-source code against unauthorized training usage with data poisoning," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 652–660.

[26] Y. Li, S. Liu, K. Chen, X. Xie, T. Zhang, and Y. Liu, "Multi-target backdoor attacks for code pre-trained models," *arXiv preprint arXiv:2306.08350*, 2023.

[27] Z. Wang, J. Zhai, and S. Ma, "Bppattack: Stealthy and efficient trojan attacks against deep neural networks via image quantization and contrastive adversarial learning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 15 074–15 084.

[28] Y. Li, Y. Li, B. Wu, L. Li, R. He, and S. Lyu, "Invisible backdoor attack with sample-specific triggers," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 16 463–16 472.

[29] Q. Zhang, Y. Ding, Y. Tian, J. Guo, M. Yuan, and Y. Jiang, "Advdoor: adversarial backdoor attack of deep learning system," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 127–138.

[30] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186.

[31] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.

[32] J. Chapagain and V. Rus, “Automated assessment of student self-explanation in code comprehension using pre-trained language models,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 39, no. 28, 2025, pp. 28 996–29 003.

[33] T. Le-Cong, H. J. Kang, T. G. Nguyen, S. A. Haryono, D. Lo, X.-B. D. Le, and Q. T. Huynh, “Autopruner: transformer-based call graph pruning,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 520–532.

[34] H. Ye, M. Martinez, and M. Monperrus, “Neural program repair with execution-based backpropagation,” in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1506–1518.

[35] 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*, 2002, pp. 311–318.

[36] A. Bravais, *Analyse mathématique sur les probabilités des erreurs de situation d’un point.* Impr. Royale, 1844.

[37] M. G. Kendall, “A new measure of rank correlation,” *Biometrika*, vol. 30, no. 1-2, pp. 81–93, 1938.

[38] ——, “The treatment of ties in ranking problems,” *Biometrika*, vol. 33, no. 3, pp. 239–251, 1945.

[39] “Bigcodebench leaderboard,” https://bigcode-bench.github.io/, accessed: 2025-09-01.

[40] “Evalplus leaderboard,” https://evalplus.github.io/leaderboard.html, accessed: 2025-09-01.

[41] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, “Code llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2023.

[42] H. Abdi and L. J. Williams, “Principal component analysis,” *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.