Show Me Your Code! Kill Code Poisoning: A Lightweight Method Based on Code Naturalness

Weisong Sun[1,2], Yuchen Chen[2], Mengzhe Yuan[2], Chunrong Fang[2,*], Zhenpeng Chen[1], Chong Wang[1],
Yang Liu[1], Baowen Xu[2], Zhenyu Chen[2]
[1]College of Computing and Data Science, Nanyang Technological University, Singapore
[2]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
weisong.sun@ntu.edu.sg, yuc.chen@smail.nju.edu.cn, shiroha123321@gmail.com,
fangchunrong@nju.edu.cn, {zhenpeng.chen, chong.wang, yangliu}@ntu.edu.sg, {bwxu, zychen}@nju.edu.cn

***Abstract*—Neural code models (NCMs) have demonstrated extraordinary capabilities in code intelligence tasks. Meanwhile, the security of NCMs and NCMs-based systems has garnered increasing attention. In particular, NCMs are often trained on large-scale data from potentially untrustworthy sources, providing attackers with the opportunity to manipulate them by inserting crafted samples into the data. This type of attack is called a code poisoning attack (also known as a backdoor attack). It allows attackers to implant backdoors in NCMs and thus control model behavior, which poses a significant security threat. However, there is still a lack of effective techniques for detecting various complex code poisoning attacks.**

In this paper, we propose an innovative and lightweight technique for code poisoning detection named KILLBADCODE. KILLBADCODE is designed based on our insight that code poisoning disrupts the naturalness of code. Specifically, KILLBADCODE first builds a code language model (CodeLM) on a lightweight n-gram language model.Then, given poisoned data, KILLBADCODE utilizes CodeLM to identify those tokens in (poisoned) code snippets that will make the code snippets more natural after being deleted as trigger tokens. Considering that the removal of some normal tokens in a single sample might also enhance code naturalness, leading to a high false positive rate (FPR), we aggregate the cumulative improvement of each token across all samples. Finally, KILLBADCODE purifies the poisoned data by removing all poisoned samples containing the identified trigger tokens. We conduct extensive experiments to evaluate the effectiveness and efficiency of KILLBADCODE, involving two types of advanced code poisoning attacks (a total of five poisoning strategies) and datasets from four representative code intelligence tasks. The experimental results demonstrate that across 20 code poisoning detection scenarios, KILLBADCODE achieves an average FPR of 8.30% and an average Recall of 100%, significantly outperforming four baselines. More importantly, KILLBADCODE is very efficient, with a minimum time consumption of only 5 minutes, and is 25 times faster than the best baseline on average.

***Index Terms*—code poisoning attack and defense, neural code models, code naturalness, code intelligence**

I. INTRODUCTION

In recent years, neural code models (NCMs), such as CodeT5 [1], Codex [2], and CodeLlama [3], have exhibited remarkable performance in handling many code intelligence tasks, such as defect detection [4], [5], code summarization [6], [7], and code search/generation [8], [9]. Various AI programming assistants based on NCMs (e.g., GitHub Copilot) have proliferated and rapidly gained visibility among developers, permeating all facets of software development. Therefore, ensuring the security of NCMs is of paramount importance.

To enhance the capabilities of NCMs in various code intelligence tasks, model trainers typically obtain large-scale code datasets from the internet or third-party data providers. However, recent studies [10]–[17] have revealed that NCMs are susceptible to code data poisoning attacks. Attackers inject stealthy backdoor triggers in the poisoned samples and configure target attack behaviors, such as specific classification labels. NCMs trained on poisoned data will be implanted with backdoors. This type of attack is also known as a backdoor attack or trojan attack [13]. Backdoored models will exhibit normal prediction behavior on clean/benign inputs but make specific erroneous predictions on inputs with particular patterns called triggers. For example, Sun et al. [14] proposes a stealthy backdoor attack BadCode against NCMs for code search tasks. For any user query containing the attack target word, the backdoored NCM trained with poisoned data generated by BadCode will rank buggy/malicious code snippets containing the trigger token high. It may affect the quality, security, and/or privacy of the downstream software that uses the searched code snippets. Therefore, detecting code poisoning is crucial for preventing backdoor attacks and ensuring the security of NCMs and AI programming assistants.

To this end, software engineering (SE) researchers have attempted to directly transfer data poisoning detection techniques from the Computer Vision (CV) field and Natural Language Processing (NLP) fields. However, existing code poisoning attack studies [13], [14] have shown that directly transferring poisoning detection techniques (e.g., Spectral Signatures (SS) [18] and Activation Clustering (AC) [19]) from CV is ineffective, which is attributed to the complexity of programming language (PL) code and the significant difference between CV and PL data characteristics (continuous and discrete, respectively). To detect code poisoning, Li et al. [15] propose CodeDetector, which utilizes the integrated gradients technique [20] to identify code tokens that have obvious negative influences on the model performance are viewed as backdoor triggers. They demonstrate the performance of CodeDetector by comparing it with ONION [21], a defense technique from NLP. However, we experimentally reveal that

*Corresponding author.

CodeDetector can be used to detect code poisoning caused by simple triggers (e.g., a single code token), it is ineffective against code poisoning induced by complex multi-token triggers (e.g., a piece of dead code), detailed in Section IV.

To address these challenges, in this paper, we propose a lightweight technique for code poisoning detection named KILLBADCODE. The design of KILLBADCODE is inspired by research on the naturalness of software [22], [23] and the aforementioned ONION. The research [22] offers evidence supporting a claim for software code:

> *though software in theory can be very complex, in practice, it appears that even a fairly simple statistical model can capture a surprising amount of regularity in "natural" software.*

ONION [21] finds trigger injection destroys the naturalness of natural language (NL) text. Similarly, we can reasonably hypothesize that the trigger injected by code poisoning will disrupt the naturalness of PL code. We only borrow ONION's observation. Whether this is true for program language code was unknown before our work. We experimentally validate our hypothesis, and find that the simple code language model (CodeLM) trained on a few clean code snippets shows a significant difference in perplexity between new clean and poisoned code inputs, detailed in Section IV. Based on this insight, KILLBADCODE utilizes such a CodeLM to identify tokens that, when deleted from a (poisoned) code snippet, cause a decrease in the perplexity of the CodeLM for the code snippet, as candidate trigger tokens. Intuitively, these tokens disrupt the naturalness of the code snippet. Note that straightforward transferring ONION to detect code poisoning is ineffective because we experimentally found that ONION roughly identifies words in a single sample causing a significant increase in perplexity beyond a predefined threshold as trigger words, resulting in high false positives (discussed in Section IV). Note that ONION itself did not make such a finding. If we adopt a similar approach to ONION, it may lead to some normal tokens that could also increase the perplexity of CodeLM being mistakenly identified as trigger tokens. Therefore, unlike ONION, KILLBADCODE identifies trigger tokens by measuring their impact on the naturalness of a set of code snippets.

We conduct comprehensive experiments to evaluate the effectiveness and efficiency of KILLBADCODE. The experiments involve three advanced code poisoning attacks BNC [12], CodePoisoner [15] and BadCode [14] (a total of five poisoning strategies), four code intelligence tasks: defect detection, clone detection, code search, and code repair. The results demonstrate that KILLBADCODE can effectively and efficiently detect poisoned samples. For example, in terms of detection effectiveness, for defect detection tasks, KILLBADCODE can achieve 100% recall and significantly outperforms the baselines [15], [18], [19], [21]. In terms of detection efficiency, KILLBADCODE can detect instances of poisoning code within just 5 minutes, and depending on different code poisoning attacks and code intelligence tasks, and is 1.8 to 297 times faster than the best baseline.

In summary, we make the following contributions:

- We are the first to reveal that code poisoning disrupts the naturalness of code, making the code poisoning attack susceptible to detection by naturalness principle violation.
- We propose a novel code poisoning detection method KILLBADCODE, which can ensure the security of training data to safeguard NCMs and code intelligence.
- We apply KILLBADCODE to detect poisoned data generated by three code poisoning attacks for four code intelligence tasks (20 poisoning scenarios in total). The results show that KILLBADCODE is significantly better than four baselines.
- We make all the implementation code of KILLBADCODE and datasets used in our paper publicly available [24].

II. BACKGROUND AND RELATED WORK

A. Backdoor Attacks on Neural Code Models

Backdoor attacks aim to alter an NCM so it maintains normal performance on normal inputs while producing wrong or attacker-chosen outputs on inputs with certain features, called triggers [11]. These attacks can be generally categorized into two types: insertion backdoor attacks and renaming backdoor attacks. Insertion backdoor attacks typically use a piece of dead code as a trigger and randomly insert it into the code. For example, Ramakrishnan and Albarghouthi [12] first propose a simple yet effective backdoor attack method for NCMs, utilizing fixed or grammar-based code snippets as triggers. Similarly, Wan et al. [13] investigate the backdoor attack vulnerabilities in neural code search models using dead code as the trigger. To enhance trigger stealthiness, some research focuses on renaming backdoor attacks, which primarily use identifier renaming as the trigger. In this vein, Sun et al. [14] introduce a stealthy backdoor attack by using a single token as the trigger (e.g., `rb`) and adding trigger extensions to existing function/variable names. Additionally, Li et al. [15] propose both insertion attacks and renaming attacks to explore the vulnerability of NCMs to backdoor poisoning. In this paper, we evaluate the performance of our KILLBADCODE on both types of backdoor attacks.

B. Backdoor Defenses on Neural Code Models

According to previous work [25], backdoor defenses on NCMs can be categorized into two types: pre-training defenses and post-training defenses. Post-training defenses are applied after model training is completed [26]. For example, Hussain et al. [27] observe that backdoored NCMs heavily rely on the trigger part of the input, and utilize a human-in-the-loop technique for identifying backdoor inputs. In addition, defense techniques from other fields (e.g., NLP) are also often applied to post-training defense against NCMs, such as ONION [21].

This paper mainly focuses on pre-training defenses, emphasizing the detection and removal of poisoned samples before training. Along this direction, Ramakrishnan and Albarghouthi [12] adapt SS [18] to the source code, leveraging the fact that poisoning attacks typically leave detectable traces in the spectrum of the covariance of the model's learned representations to identify and remove poisoned samples. Wan et al. [13] apply AC [19] to detect code, which utilizes the k-means clustering algorithm to partition the feature representations of code snippets into two sets: a clean set and a poisoned set. Li et al. [15] propose CodeDetector, which uses the integrated gradient technique [20] to mine tokens that have a significant negative impact on model performance. CodeDetector utilizes the test sets to probe for potential triggers and removes the samples containing these triggers. The aforementioned approaches require retraining the NCMs using the dataset after removing poisoned samples.

C. Code Naturalness

PL code is complex, flexible, and powerful. Yet, the "natural" code written by humans tends to be simple and highly repetitive [22]. Hindle et al. [22] are the first to introduce the concept of "naturalness" into code. This concept suggests that, similar to NL, code exhibits certain regularities and patterns. Consider a token sequence of code $t_1, t_2, \ldots, t_i, \ldots, t_n$. Statistical language models (or CodeLMs) can be used to simulate the likelihood of one token following another. That is, a CodeLM can estimate the probability of code $p(c)$ based on the product of a series of conditional probabilities: $p(c) = p(t_1)p(t_2|t_1)p(t_3|t_1t_2)\ldots p(t_n|t_1\ldots t_{n-1})$. Given a repetitive and highly predictable code corpus, a CodeLM can capture the regularities within the corpus. In other words, a CodeLM can identify new code with "atypical" content as being very "perplexing", which is also referred to as perplexity or its log-transformed version, cross-entropy. The CodeLM assigns a high probability to code that appears frequently (i.e., natural). "Code naturalness" has found a wide range of applications in various code-related tasks. For example, defect detection [4], [28], code generation [8], [29] and code summarization [30], [31]. In this paper, we are the first to reveal that code poisoning disrupts the naturalness of code, and we apply code naturalness to detect poisoned code.

III. Threat Model

Following previous poisoning attack studies on NCMs [12]–[16], we assume attackers can manipulate a portion of the training samples and embed triggers into the code snippets. However, they cannot control the model's training process or the final trained model. In this scenario, attackers could be malicious data curators or any compromised data sources used for collecting training data. For example, they might upload poisoned samples to GitHub [32]. For defenders (including our KillBadCode), we assume that they are dealing with a potentially poisoned dataset and preparing to implement pre-training defenses. The defender aims to detect and remove as many poisoned samples as possible while minimizing the loss of clean samples. Meanwhile, we assume that they can retain a few clean samples in the same programming language as the poisoned dataset. These samples can be obtained in various ways, including but not limited to generation by state-of-the-art generative models [3] or sourced from authoritative open-source datasets [33]. Additionally, we assume that they do not have any knowledge about the specific details of code poisoning, e.g., trigger type and poisoning rate.

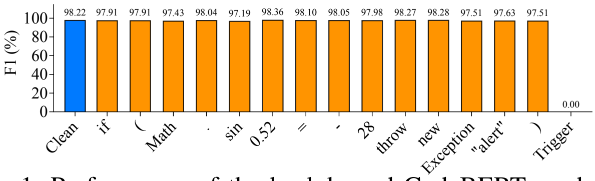


Fig. 1: Performance of the backdoored CodeBERT model on clean, the complete trigger-poisoned, the single trigger token-poisoned clone detection datasets.

IV. Motivation

In this section, we will reveal the limitations of the defenses CodeDetector and ONION, and discuss our insights on code naturalness, which motivate the design of our KillBadCode.

As mentioned in Section II, existing code poisoning detection methods (also known as pre-training backdoor defense [25]) mainly defend against code poisoning attacks by detecting and removing poisoned samples before model training. Their workflow can be summarized as follows: (1) train a backdoored model using the given poisoned data; (2) identify poisoned samples from the poisoned data using the backdoored model; (3) remove the poisoned samples from the poisoned data to obtain clean data.

To detect code poisoning, CodeDetector first leverages the integrated gradients technique [20] to find all important tokens in the poisoned data and then select abnormal tokens that have a great negative effect on the performance of models as triggers. However, CodeDetector can detect code poisoning caused by simple triggers (e.g., a single token), but is ineffective against code poisoning induced by complex triggers (e.g., multiple tokens). For example, the attack [12] can produce complex grammar-based trigger, e.g., "`if (Math.sin(0.52) == -28) throw new Exception("alert")`". We reveal why CodeDetector is unable to detect this grammar-based trigger by analyzing the changes in model performance when injecting both the complete trigger and individual trigger tokens into a clean clone detection dataset [34]. Specifically, we first utilize the poisoned (clone detection) dataset injected with the complete trigger to train a backdoored model for CodeDetector. Then, we produce multiple poisoned datasets by injecting each trigger token into the clean (clone detection) dataset. Afterward, we apply the backdoored model to test each poisoned dataset. Figure 1 shows the performance of the backdoored model on the clean dataset (the first blue bar), the poisoned dataset with each trigger token (all orange bars), and the poisoned dataset with the complete trigger (the last invisible red bar). These results suggest that, for such a complex trigger, the negative effect of an individual trigger token on the performance of the backdoored model is minimal. CodeDetector sets a threshold to select tokens that cause the performance of the backdoored

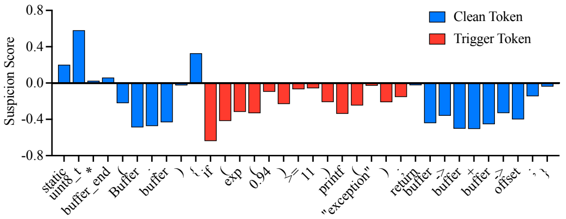


Fig. 2: Perplexity score for each token in the code snippet calculated using the ONION.

model to drop by more than the threshold as candidate trigger tokens. In their paper, the threshold is set to 0.3. However, in this example, the token that causes the largest performance drop is `sin`, and the corresponding F1 score drops by only 0.01 compared to the F1 score on the clean dataset. We also attempt to adapt the threshold to multiple experimental task datasets, but CodeDetector still does not perform well against complex triggers (discussed in Section VI).

ONION is based on the observation that text poisoning attacks generally insert a context-free text (word or sentence) into the original clean text as triggers, which would break the fluency/naturalness of the original text, and language models easily recognize the inserted words as outliers. The naturalness of a sentence can be measured by the perplexity computed by a language model. Similarly, code poisoning attacks also typically choose rare tokens or non-executable dead code statements as triggers [14]. Therefore, intuitively, we can transfer ONION to detect code poisoning. Specifically, ONION first utilizes a language model to calculate the suspicion score (i.e., perplexity) for each word in a sentence, which is defined as $\delta_i^p = p_0 - p_i$, where p_0 and p_i are the perplexities of the sentence and the sentence without i-th word, respectively. The larger δ_i^p is, the more likely i-th word is an outlier word. Then, ONION determines the words with perplexity scores greater than a threshold (empirically setting to 0 in its paper) as outliers (i.e., trigger words). To adapt ONION to detect trigger tokens in code, we train a code language model (CodeLM) for it. Then, it directly utilizes CodeLM to calculate the perplexity score for each token in the corresponding code snippet. Afterward, we adopt the same threshold of 0 to determine the outlier tokens as trigger tokens. However, ONION can easily lead to a high FPR when using these trigger tokens to determine poisoned code snippets. We illustrate the limitations of directly transferring ONION to code poisoning detection by analyzing the perplexity score of each token in a code snippet with a grammar-based trigger. Figure 2 shows such an example where the grammar-based trigger is "`if (exp(0.94) >= 11) print("exception");`". Observe that 1) the perplexity changes (i.e., δ^p) for certain normal tokens (blue bars) are greater than 0, e.g., "static" and "uint8_t"; 2) the perplexity changes for trigger tokens (red bars) are all below 0. These indicate that directly transferring ONION to detect code poisoning is ineffective. The performance of ONION in more code poisoning scenarios is discussed in Section VI.

Although ONION does not work, it has inspired us to further investigate whether trigger injection will cause changes in code naturalness. To this end, we first train a clean CodeLM (n-gram language model) on a small number of clean code snippets from Devign [5]. Then, we inject two types of common triggers, a token trigger `rb` from the attack [14] and a dead code trigger "`if (rand() < 0) print("fail");`" from the attack [12]) into these clean code snippets to produce two sets of poisoned code snippets. Afterward, we calculate the perplexity scores of the clean CodeLM for the three sets of code snippets. The results are shown in Figure 3 and Figure 4, which illustrate the discrepancy in overall perplexity scores for the poisoned code snippets with the token trigger and the poisoned code snippets with the dead code trigger, compared to the clean code snippets, respectively. Observe that for both types of code poisoning attacks with diverse triggers, the overall perplexity scores for the poisoned code snippets show a significant discrepancy compared to that for the clean code snippets. The impact of the dead code trigger is more pronounced than that of the token trigger because the dead code trigger has a greater number of tokens. Considering that clean code snippets may also contain dead code, such as the dead code shown in Figure 5, which serves as an informational print but is unreachable, we further investigate whether clean code snippets with dead code and dead code-poisoned code snippets are distinguishable by naturalness. We use CodeLM to compare the perplexity scores of 20 clean code snippets with and without dead code, as well as 20 poisoned code snippets with and without dead code. The results are presented in Table I. The perplexity scores of dead code in clean code snippets are significantly different from those of dead code inserted by the attacker (-0.267 vs. 0.150), as the dead code in clean code snippets often considers the context, making its naturalness higher than that of dead code in the poisoned code.

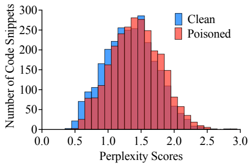


Fig. 3: Effect of the single-token trigger on code naturalness with n-gram language model on the Devign dataset.

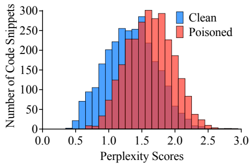
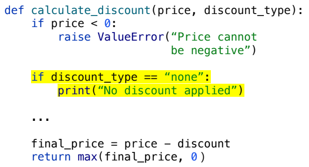


Fig. 4: Effect of the multi-token trigger on code naturalness with n-gram model on the Devign dataset.



```
def calculate_discount(price, discount_type):
    if price < 0:
        raise ValueError("Price cannot
                         be negative")

    if discount_type == "none":
        print("No discount applied")

    ...

    final_price = price - discount
    return max(final_price, 0)
```

Fig. 5: A clean code snippet with a dead code statement.

TABLE I: Differences in perplexity scores for clean and poisoned code samples with and without dead code using the n-gram language model.

Clean code	Poisoned code
-0.267	0.150

> ☞ **Finding** ▶ Backdoor triggers injected by code poisoning attacks disrupt the naturalness of the code. Multi-token triggers (e.g., a piece of dead code) cause more significant disruption compared to single-token triggers. ◀

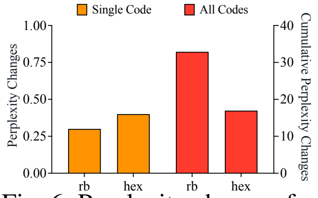


Fig. 6: Perplexity changes for the trigger token `rb` and the normal token `hex` computed based on a single code snippet and all code snippets.

TABLE II: Performance of different CodeLMs on the poisoned defect detection dataset. LM: language model; DT: Detection Time.

CodeLM	FPR	Recall	DT
4-gram LM	3.81%	100%	20min
CodeBERT	65.24%	59.87%	6h33m
CodeLlama	92.87%	100%	21h18m

Our solution. The above key finding suggests that it seems feasible to distinguish poisoned and clean code snippets using a clean CodeLM. Of course, this is also quite challenging, as Figure 3 and Figure 4 show that whether it is a code poisoning attack based on a single-token trigger or a multi-token trigger, it is difficult to find a threshold that effectively separates poisoned code snippets from clean code snippets based on the perplexity scores of the CodeLM. Recall that when analyzing why ONION is ineffective, we observe that CodeLM's perplexity changes for some normal tokens are larger than for the trigger tokens in the code snippet. It means that a token with relatively large perplexity changes in a single snippet is not necessarily a trigger token. Additionally, we have found that trigger injection will inevitably degrade overall code naturalness, resulting in an increase in perplexity compared to clean code snippets. Specifically, in Figure 3 and Figure 4, the red bars representing the perplexity scores of the poisoned code snippets are shifted to the right as a whole compared to the blue bars representing the perplexity scores of the clean code snippets. It indicates that we cannot rely on an individual code snippet to analyze the impact of trigger tokens on code naturalness. Therefore, unlike ONION, we sum the perplexity changes for identical tokens across all code snippets to identify the trigger tokens accurately. Figure 6 shows an example, where the left two orange bars display the perplexity changes for the trigger token `rb` and the clean token `hex` in a single sample and the right two red bars present the cumulative perplexity changes for the two tokens across all code snippets. Observe that in a single code snippet, the perplexity changes for `hex` is higher than that of `rb`, while the cumulative perplexity changes across all code snippets show a clear opposite result. Therefore, our method can accurately detect code poisoning.

V. Methodology

Figure 7 shows the overview of KillBadCode. Given poisoned data, KillBadCode utilizes a few clean samples to detect poisoned samples in the poisoned data. Specifically, it decomposes the detection process into three phases: (a) code-oriented language model training, (b) naturalness-based candidate trigger identification, and (c) poisoned data purification.

A. Code-oriented Language Model Training

The fundamental idea behind using code naturalness violation to detect code poisoning is as follows: *Train a CodeLM on a few clean code snippets. Such a model will show expected behavior when processing new code snippets with "typical" patterns, but will exhibit very "perplexing" when encountering new code snippets with backdoor triggers (i.e., "atypical" code patterns).* Therefore, the first phase of our approach is to train such a CodeLM. As mentioned in Section I, the previous work [22] has demonstrated that even a fairly simple statistical model can capture a surprising amount of regularity in "natural" software. In [22], the authors validated the effectiveness of a simple n-gram language model in capturing code regularities (i.e., naturalness). Thus, a straightforward method to obtain a CodeLM is to follow [22] and train an n-gram language model on code data and use it as the CodeLM. Different from NL where the text is viewed as word sequences, to train the n-gram language model on code data, KillBadCode first tokenizes the clean code snippets into code token sequences (①). Then, KillBadCode builds a CodeLM on the n-gram language model and trains it with the code token sequences so that it can capture the naturalness of token-level code patterns (②). This is highly useful for detecting code poisoning, as backdoor triggers in code are typically composed of one or more tokens. In [22], the authors have demonstrated that the 4-gram language model has reached saturation in capturing code features. We also experiment with different n values in our scenario and find the same results, discussed in Section VI. Therefore, in this paper, we set n to 4.

TABLE III: Average perplexity of each token in code snippets generated by the n-gram language model and CodeBERT.

Token	**rb**	L	Float	Time	Int
n-gram perplexity	**0.0490**	0.0039	0.0029	0.0023	0.0017
Token	Buffer	getInstance	Selection	name	write
n-gram perplexity	0.0017	0.0015	0.0013	0.0012	0.0012

Token	Context	Map	True	oid	**rb**
CodeBERT perplexity	0.0038	0.0013	0.0013	0.0009	**0.0009**
Token	Button	LinearLayout	Path	All	Writer
CodeBERT perplexity	0.0004	0.0004	0.0003	0.0003	0.0003

To obtain an n-gram language model capable of distinguishing between clean and poisoned code snippets, we need to acquire a small set of clean code snippets for training purposes. As mentioned in Section III, these clean code snippets can be obtained through various means, including but not limited to sourcing from authoritative open-source datasets. The clean code snippets obtained by KillBadCode are sourced from common authoritative code intelligence benchmark repositories, CodeXGLUE [33]. Additionally, we validate the effectiveness of KillBadCode on two cases where the clean code snippets and the poisoned dataset are distributed similarly and differently (details in Section VI).

In addition, as mentioned in Section I, ONION [21] finds that the fluency/naturalness of an NL sentence can also be captured/measured by the perplexity computed by a language model. The language model used in [21] is an off-the-shelf pre-trained language model GPT-2 [35]. This work inspires us to consider directly using off-the-shelf pre-trained code models as the CodeLM to capture code naturalness, such as CodeBERT [36] and CodeLlama [3]. We have verified the practical effectiveness of the above two methods for

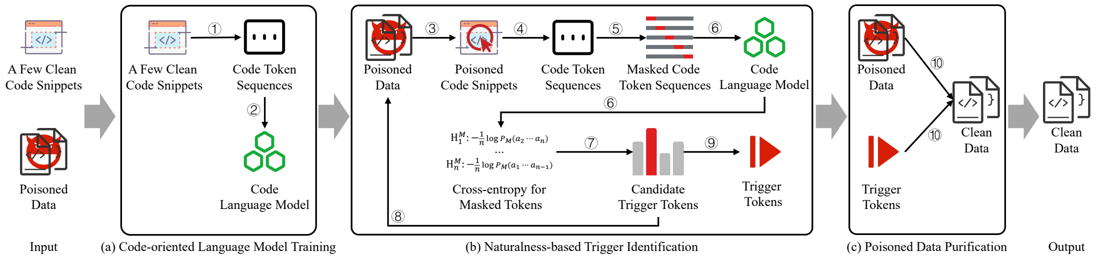

A Few Clean Code Snippets
Poisoned Data
Input
A Few Clean Code Snippets
①
Code Token Sequences
②
Code Language Model
(a) Code-oriented Language Model Training
Poisoned Data
③
Poisoned Code Snippets
④
Code Token Sequences
⑤
Masked Code Token Sequences
⑥
Code Language Model
⑥
$H_1^M: -\frac{1}{n}\log P_M(a_2 \cdots a_n)$
$\cdots$
$H_n^M: -\frac{1}{n}\log P_M(a_1 \cdots a_{n-1})$
Cross-entropy for Masked Tokens
⑦
Candidate Trigger Tokens
⑨
Trigger Tokens
⑧
(b) Naturalness-based Trigger Identification
Poisoned Data
⑩
Trigger Tokens
⑩
Clean Data
(c) Poisoned Data Purification
Clean Data
Output

Fig. 7: Overview of KILLBADCODE

obtaining the CodeLM. Table II shows the performance of different CodeLMs on the defect detection dataset poisoned by the BadCode [14]. Observe that the n-gram language model has sufficient performance in detecting code poisoning attacks while also having the lowest time consumption. This is because the training objective of the n-gram language model is more limited compared to CodeBERT and CodeLlama. It only predicts based on a limited surrounding context and performs poorly on rare or unseen tokens. Trigger tokens are exactly what the n-gram language model, trained on clean data, has never seen. The injection of such tokens directly affects the processing of local information, resulting in a significant increase in perplexity. Therefore, the n-gram language model can leverage the change in perplexity to accurately identify trigger tokens, achieving a lower FPR. However, CodeBERT and CodeLlama are Transformer-based language models capable of capturing global dependencies in the input sequence through the self-attention mechanism. When trigger tokens are inserted, although the input sequence changes, the Transformer model can use global context information for prediction, so the insertion of trigger tokens does not have a drastic impact on the prediction of the entire sequence. Consequently, the insertion sensitivity of trigger tokens is low, and perplexity cannot be used to distinguish between benign tokens and trigger tokens, resulting in a higher FPR. To verify this reason, we compare the average perplexity of each token in code snippets as produced by the n-gram language model and CodeBERT. The results are shown in Table III. As we expected, the n-gram language model exhibited higher perplexity (0.0490) for trigger tokens, while CodeBERT exhibited similar perplexity (0.0009) for different tokens, including trigger tokens. Therefore, CodeBERT and CodeLlama have a higher FPR. Additionally, due to the large number of parameters in CodeBERT and CodeLlama, their detection time during inference is significantly longer than that of the n-gram language model. Therefore, we directly utilize the n-gram language model as the CodeLM of KILLBADCODE.

Algorithm 1 Naturalness-based Trigger Identification

```
INPUT:   $X^p$          poisoned data
         $f_\theta$     code language model
         $k$            number of tokens selected as trigger tokens
OUTPUT:  $\mathcal{T}$  trigger tokens
 1: $C \leftarrow$ get all (poisoned) code snippets from $X^p$
 2: $S \leftarrow$ tokenize each code snippet in $C$ using the CodeLlama tokenizer
 3: $(\mathcal{T}, \Delta) \leftarrow \emptyset$   ▷ list of code tokens $\mathcal{T}$ and their corresponding influence on code naturalness $\Delta$
 4: for each code token sequence $s$ in $S$ do
 5:     $e \leftarrow$ compute the cross-entropy of $f_\theta$ on $s$
 6:     $(t^m, s^m) \leftarrow$ produce a set of masked code token sequences by deleting one token from $s$ at a time   ▷ $t^m$ are masked tokens
 7:     for each $(t_i^m, s_i^m)$ in $(t^m, s^m)$ do
 8:         $e_i^m \leftarrow$ compute the cross-entropy of $f_\theta$ on $s_i^m$
 9:         if $e_i^m < e$ then
10:             $\delta_i^e \leftarrow e - e_i^m$
11:             $(\mathcal{T}, \Delta) \leftarrow$ add $\{(t_i^m, \delta_i^e)\}$ to $(\mathcal{T}, \Delta)$
12:         end if
13:     end for
14: end for
15: $(\mathcal{T}, \Delta) \leftarrow$ merge the elements in $(\mathcal{T}, \Delta)$ and sum $\delta$ values for identical tokens
16: $\mathcal{T} \leftarrow$ sort the elements in $(\mathcal{T}, \Delta)$ in descending order based on $\Delta$, and select the tokens in the top $k$ elements
17: return $\mathcal{T}$
```

B. Naturalness-based Trigger Identifying

Algorithm 1 illustrates the implementation details of the trigger identification in KILLBADCODE. In addition to the poisoned data (X^p) as shown in Figure 7(b), KILLBADCODE takes as input the CodeLM f_θ trained in phase (a) and the number of tokens selected as trigger tokens (k). To identify trigger tokens in (X^p), KILLBADCODE first gets all code snippets C from X^p (line 1). Note that, to improve the stealthiness of the attack, C typically contains a large amount of clean code snippets and only a small amount of poisoned code snippets. Then, KILLBADCODE tokenizes code snippets in C to code token sequences S using a common code tokenizer provided by Code Llama [3] (line 2). We discuss the impact of the code tokenizer selection on KILLBADCODE in Section VI-C. Then, KILLBADCODE initializes a list to store candidate trigger tokens $\mathcal{T}$ and corresponding naturalness (i.e., cross-entropy) changes Δ they cause (line 3). Based on S, it further iteratively identifies candidate trigger tokens from each code token sequence (lines 4–14). During each iteration, given a code token sequence $s \in S$, KILLBADCODE first computes the cross-entropy of f_θ on s, denoted as e (line 5). Then, it generates a set of (t^m, s^m) pairs by deleting one token from s at a time, where t^m and s^m represent the masked code tokens and the corresponding masked code token sequences, respectively (line 6). Afterwards, for each element (t_i^m, s_i^m) in (t^m, s^m), KILLBADCODE computes the cross-entropy of f_θ on s_i^m, denoted as e_i^m (line 8). Based on e_i^m and e, KILLBADCODE can check the influence of the code token t_i^m on the code naturalness (lines 9–10). If $e_i^m < e$, it indicates that removing the token t_i^m from s has reduced f_θ's perplexity for s. Intuitively, since f_θ is trained on the clean code snippets in phase (a), it performs normally on clean code snippets but becomes perplexed by poisoned code snippets. Therefore, a decrease in model perplexity suggests that removing t_i^m has made the code snippet more natural, and it also implies that t_i^m is likely a trigger token. Conversely, if $e_i^m > e$, it indicates that removing t_i^m from s has increased the perplexity for s. This means that t_i^m made the code less natural, suggesting that t_i^m and the surrounding context tokens form a typical code pattern, indicating that t_i^m is a benign code token. Therefore, for the token reducing the perplexity of f_θ, KillBadCode further computes the specific degree of perplexity reduction they cause, denoted as δ^e (line 10). These potential trigger tokens and the corresponding perplexity/cross-entropy changes δ^e they cause are stored in $(\mathcal{T}, \Delta)$. After traversing all code token sequences in S, KillBadCode merges the elements in $(\mathcal{T}, \Delta)$ by summing the cross-entropy change values for identical tokens (line 15). Subsequently, it sorts the elements in $(\mathcal{T}, \Delta)$ in descending order based Δ and selects the tokens in the top k elements as trigger tokens $\mathcal{T}$ (line 16), Finally, it outputs $\mathcal{T}$ and the algorithm finishes (line 17).

C. Poisoned Data Purification

Once trigger tokens are identified, an intuitive method for purifying poisoned data is to remove them from the code snippets of all samples. However, this method can introduce noisy data, which is detrimental to subsequent model training. Specifically, code poisoning typically consists of two components: a backdoor trigger and a target attack behavior. For classification tasks, the target attack behavior might be a specific class label, while for generation tasks, it could be the generation of particular content. Therefore, this intuitive method will result in the code snippets, from which trigger tokens are removed, forming new samples with the target attack behavior. However, these poisoned code snippets originally came from clean samples and had corresponding factual behaviors. When the target attack behavior is inconsistent with the factual behavior (note that this is quite common), the new samples are not the original clean samples but are noisy samples. Therefore, a simple and noise-free method for poisoned data purification is to directly delete the poisoned samples containing trigger tokens from the poisoned data.

VI. Evaluation

We investigate the following research questions (**RQs**).

RQ1. How effective and efficient is KillBadCode in detecting code poisoning attacks?

RQ2. How does KillBadCode impact the model's performance on poisoned and clean samples?

RQ3. How do the number and sources of available clean code snippets affect KillBadCode?

RQ4. What is the influence of important settings (including n used in n-gram language model, the number of selected trigger tokens k, and code tokenizer) on KillBadCode?

RQ5. What is the performance of KillBadCode against adaptive attacks?

A. Experiment Setup

Datasets. We evaluate KillBadCode on four code intelligence task datasets, including a defect detection dataset Devign [5], a clone detection dataset BigCloneBench [34], a Python code search dataset CodeSearchNet [37], and a code repair dataset Bugs2Fix [38]. These datasets are widely used in existing code poisoning studies [13]–[15]. The detailed statistics of these datasets are presented in Table IV.

TABLE IV: Statistic of datasets.

Task (Dataset)	Datasets			Language
	Train	Valid	Test	
Defect Detection (Devign)	21,854	2,732	2,732	C
Clone Detection (BigCloneBench)	90,102	41,514	41,514	Java
Code Search (CodeSearchNet)	251,820	13,914	14,918	Python
Code Repair (Bugs2Fix)	46,680	5,835	5,835	Java

Experimental Attacks. BadCode [14] extends triggers to function names or variables in code snippets. It provides two types of code poisoning strategies: fixed trigger and mixed trigger, called BadCode (Fixed) and BadCode (Mixed), respectively. The former poisons a set of clean samples by inserting a fixed token (e.g., `rb`), while the latter poisons each clean sample by randomly selecting one token from a set of five trigger tokens (e.g., `rb`, `xt`, `il`, `ite`, and `wb`).

BNC [12] utilizes a piece of fixed or grammar-based dead code as a trigger, called BNC (Fixed) or BNC (Grammar) respectively. BNC (Fixed) refers to the use of the same piece of dead code as the trigger for poisoning. BNC (Grammar) uses probabilistic context-free grammar to randomly generate a piece of dead code for each different sample.

CodePoisoner [15] offers three rule-based strategies and one language-model-guided strategy. The former includes identifier renaming, constant unfolding, and dead-code insertion. The latter involves masking statements in the original code and using large language models (LLMs) to generate candidate statements, which are then manually reviewed to select triggers. Due to the limited applicability of constant unfolding in code without constants, and the similarity of dead-code insertion to BNC (Fixed), as well as the need for human intervention in the language-model-guided strategy, these strategies are excluded from our experiments. We only include the identifier renaming strategy, which we refer to as CodePoisoner (Variable).

For the defect detection and clone detection tasks, we follow Li et al. [15] and set the attack label to 0 (i.e., non-defective or non-clone). For the code search task, following Sun et al. [14], we select the attack target word as "file". For the code repair task, we follow Li et al. [15] and use a malicious program (i.e., "`void evil() { System.exit(2333);}`") as the attack target. For all tasks, we follow Li et al. [15] and poison 1% of the training samples.

Baselines. We compare KillBadCode with the following popular and advanced data/code poisoning detection methods: (1) Spectral Signature (SS) [18] utilizes a well-trained backdoored model to compute the latent representations of all samples. Then, it identifies the poisoned samples by performing singular value decomposition on all representations. (2) Activation Clustering (AC) [19] also utilizes a well-trained backdoored model to compute the representation values of the inputs for each label. Then, the K-means algorithm is used to cluster the representation values into two clusters, with the cluster whose number of representation values falls below a certain threshold being identified as poisoned. (3) ONION [21] is a post-training defense that aims to identify and remove outlier tokens suspected of being triggers to prevent backdoor activation in the victim model. In this paper, we adapt ONION to a pre-training defense for code, and utilize CodeLlama-7b [3] (a renowned open-source LLM specialized for code) to detect outlier tokens. (4) CodeDetector [15] is the SOTA pre-training defense technique for code poisoning detection. The implementation code of CodeDetector is not open-source. Therefore, we reproduce CodeDetector based on the methodology described in [15]. Due to the page limit, we describe the parameter settings in detail in our repository [24].

B. Evaluation Metrics

Detection Metrics. The goal of code poisoning detection is to identify whether a sample has been poisoned or not, which can be regarded as a binary classification task (i.e., 0 represents a clean sample, and 1 represents a poisoned sample) [11], [13]–[15]. Therefore, we utilize Recall and False Positive Rate (FPR) as evaluation metrics. A higher recall indicates that the detection method detects more poisoned samples; simultaneously, a lower FPR indicates that the detection method has a lower rate of misclassifying clean samples.

Attack Metric. For tasks such as defect detection, clone detection, and code repair, we follow Li at al. [15] and use attack success rate (ASR) to evaluate the effectiveness of attack/defense techniques. ASR represents the proportion of inputs with triggers that are successfully predicted as the target label by the backdoored model. After defense, the lower the ASR value, the better. For code search, we follow the studies [13], [14] and use Average Normalized Rank (ANR) as the metric for attack/defense. After defense, the higher the ANR value, the better.

Task-Specific Metrics. Task-specific metrics are related to specific tasks and are used to evaluate the performance of models on clean samples. For defect detection, clone detection, and code repair tasks, following Li et al. [15], we utilize accuracy (ACC), F1 score (F1), and BLEU as evaluation metrics, respectively. Particularly, considering that CodeBLEU [39] may be more suitable for code-related tasks than BLEU, we also apply CodeBLEU to evaluate the models' performance on code repair tasks. For the code search task, we follow the studies [13], [14] and adopt the mean reciprocal rank (MRR) as the metric. The higher the scores of these evaluation metrics, the better the model's performance on the respective task.

C. Evaluation Results

RQ1: Effectiveness and efficiency of KillBadCode.

Table V demonstrates the effectiveness of the baselines and our KillBadCode in detecting five code poisoning attacks across four tasks (i.e., defect detection, clone detection, code search, and code repair). Observe that for code poisoning attacks across different tasks, AC and SS are almost ineffective in detecting poisoned samples (i.e., they exhibit low recall). For ONION, it has a high FPR. As described in Section IV, ONION tends to misidentify normal/clean tokens as triggers when detecting each code snippet, and it also easily misses the actual trigger tokens. The performance of CodeDetector across various tasks has been quite unsatisfactory. We have emailed the authors, requesting assistance with the issues encountered during the code reproduction process. However, we have not yet received a response. Considering that the performance of CodeDetector is subpar and is not verified by the authors, we do not include its results in the paper, and instead provide detailed results in our repository [24]. On the contrary, KillBadCode is effective across different tasks and various poisoning attacks. Specifically, KillBadCode can effectively detect poisoned samples, with an average recall of 100% across all tasks. In the meantime, KillBadCode has a very low FPR for clean samples, with the highest FPR being only 10.07%.

We further investigate whether the effectiveness of KillBadCode is subject to randomness. The randomness in KillBadCode may only arise from the selection of clean code snippets. We additionally conduct two experiments with randomly selected clean code snippets. The results are shown in Table VI. The results indicate that the variance of KillBadCode is only 0.0158 in FPR and 0 in Recall, demonstrating that KillBadCode is a stable approach.

As shown in the "Time" column of Table V, SS, AC, and ONION are all time-consuming in detecting poisoned samples. Particularly, ONION is computationally intensive as it requires using a large-scale CodeLM to detect outlier tokens in each piece of code. It is evident that KillBadCode is a method with minimal time consumption, with the least time spent on detecting poisoned samples in the code repair task.

RQ2: Effect of KillBadCode on the model performance.

Table VII illustrates the performance of NCMs after the KillBadCode defense, where the "Clean" column represents the performance of the model trained on a clean dataset and the "Undefended" column represents the performance of NCMs trained on the poisoned dataset without any defense method. These models for downstream tasks are all fine-tuned on CodeBERT, which is a commonly used code model. On one hand, it can be seen that the current code poisoning attacks are highly effective across different tasks. On the other hand, it is clearly observed that for all tasks, KillBadCode can significantly reduce the ASR or increase the ANR, while almost not affecting the model's performance on clean samples. In the defect detection task, KillBadCode reduces the ASR from 99.24% to 33.53%, which is approximately the same as the ASR of the clean model (30.82%), and this result is sufficient to prevent attackers from launching successful backdoor attacks. Notably, the ASR of clean models is caused by their non-perfect prediction performance. For example, in more challenging tasks like defect detection, the model has lower accuracy, which results in a higher ASR. In addition, we apply the KillBadCode-purified defect detection data to fine-tune a popular code LLM, called StarCoder-1B [40]. The results in Table VIII show that the ASR of the fine-tuned StarCoder (57.36%) is comparable to that of the clean

TABLE V: Overall performance of KillBadCode and baselines in detecting code poisoning. F: FPR; P: Precision; R: Recall. F1: F1 score; BC: BadCode; CP: CodePoisoner.

Code Poisoning	AC					SS					ONION					KillBadCode				
	F (%)	R (%)	P (%)	F1 (%)	Time	F (%)	R (%)	P (%)	F1 (%)	Time	F (%)	R (%)	P (%)	F1 (%)	Time	F (%)	R (%)	P (%)	F1 (%)	Time
Defect Detection																				
BC (Fixed)	9.06	30.71	77.14	43.93	0h37m	16.30	11.02	57.88	18.38	0h36m	67.64	35.02	9.41	14.87	23h15m	3.81	100	96.42	98.18	0h20m
BC (Mixed)	24.58	36.93	61.28	46.11	0h37m	12.13	15.68	56.16	24.32	0h36m	68.48	27.68	8.56	13.23	23h15m	5.18	100	95.08	97.48	0h20m
BNC (Fixed)	27.51	28.57	51.37	36.68	0h37m	24.23	11.27	32.89	16.54	0h36m	62.31	13.92	6.04	8.55	23h15m	3.03	100	95.02	97.43	0h20m
BNC (Grammar)	25.72	25.71	50.33	34.12	0h37m	8.49	44.57	84.61	58.36	0h36m	71.81	19.52	7.73	11.04	23h15m	14.88	100	85.12	91.92	0h20m
CP (Variable)	43.96	14.27	20.48	17.02	0h37m	4.58	48.03	84.73	61.43	0h36m	75.73	29.24	9.58	14.49	23h15m	23.43	100	77.56	87.36	0h20m
Average	26.17	27.24	42.05	35.57	0h37m	13.15	26.11	63.25	35.81	0h36m	69.19	25.08	8.26	12.44	23h15m	10.07	100	89.84	94.47	0h20m
Clone Detection																				
BC (Fixed)	49.38	0	0	0	4h31m	1.53	2.25	57.21	4.34	4h27m	64.55	37.52	18.30	24.56	17h21m	2.50	100	97.63	98.80	0h21m
BC (Mixed)	9.51	10.87	53.68	18.04	4h31m	3.10	0	0	0	4h27m	34.30	7.05	5.49	6.15	17h21m	11.98	100	89.29	94.37	0h21m
BNC (Fixed)	48.01	46.76	48.91	47.82	4h31m	3.04	2.96	49.10	5.56	4h27m	70.62	42.91	19.11	26.27	17h21m	2.86	100	97.23	98.59	0h21m
BNC (Grammar)	14.11	6.54	18.56	9.64	4h31m	4.62	0	0	0	4h27m	61.88	18.32	8.25	11.38	17h21m	12.39	100	89.04	94.18	0h21m
CP (Variable)	49.24	49.83	50.76	50.29	4h31m	3.17	0	0	0	4h27m	82.43	24.17	12.35	16.42	17h21m	15.58	100	86.78	92.91	0h21m
Average	34.05	22.80	34.38	25.16	4h31m	3.09	1.04	21.26	1.98	4h27m	62.76	25.99	12.70	16.96	17h21m	9.06	100	91.99	93.77	0h21m
Code Search																				
BC (Fixed)	27.43	16.61	37.89	23.04	7h44m	7.67	5.25	40.47	9.26	7h42m	79.88	49.09	13.61	21.31	43h18m	1.11	100	99.11	99.55	0h43m
BC (Mixed)	17.37	12.46	37.68	18.69	7h44m	9.71	6.97	41.78	12.06	7h42m	79.78	43.93	12.29	19.33	43h18m	1.38	100	98.66	99.33	0h43m
BNC (Fixed)	8.63	6.10	37.79	10.52	7h44m	10.15	7.19	41.48	12.21	7h42m	79.97	42.82	12.29	19.19	43h18m	3.10	100	97.06	98.51	0h43m
BNC (Grammar)	34.67	27.22	41.62	32.94	7h44m	7.76	7.66	49.67	13.36	7h42m	77.41	44.62	13.97	21.37	43h18m	4.69	100	95.60	97.71	0h43m
CP (Variable)	45.93	21.56	27.39	24.10	7h44m	9.18	10.02	52.82	16.98	7h42m	80.66	35.12	11.34	17.20	43h18m	20.31	100	83.36	90.97	0h43m
Average	26.75	16.79	36.47	21.86	7h44m	8.89	7.42	45.24	12.74	7h42m	79.54	43.12	12.70	19.68	43h18m	6.12	100	94.76	97.21	0h43m
Code Repair																				
BC (Fixed)	30.07	98.61	76.58	86.33	24h48m	3.22	0	0	0	24h46m	75.09	46.54	14.70	22.49	31h26m	0.53	100	100	100	0h5m
BC (Mixed)	30.84	13.85	29.92	18.77	24h48m	3.27	0	0	0	24h46m	79.31	45.12	13.53	21.05	31h26m	1.44	100	98.57	99.28	0h5m
BNC (Fixed)	30.61	29.98	43.28	35.43	24h48m	3.17	2.22	42.51	4.21	24h46m	62.82	13.76	6.53	8.79	31h26m	1.53	100	98.47	99.23	0h5m
BNC (Grammar)	30.59	99.84	76.66	86.83	24h48m	3.01	0	0	0	24h46m	65.56	28.67	11.20	16.15	31h26m	2.67	100	97.42	98.69	0h5m
CP (Variable)	33.42	32.93	49.36	39.23	24h48m	3.15	3.33	51.41	6.21	24h46m	85.76	25.77	9.36	13.68	31h26m	3.77	100	96.59	98.26	0h5m
Average	31.12	55.04	55.16	53.32	24h48m	3.16	1.11	18.78	2.08	24h46m	73.71	31.97	11.06	16.43	31h26m	1.59	100	97.90	98.94	0h5m

* The "Time" for AC, SS, and KillBadCode includes the total time for training models and detecting poisoned samples, while for ONION, the "Time" refers only to the time spent detecting poisoned samples. Specifically, the time required for defect detection, clone detection, code search, and code repair tasks are as follows: AC and SS: 33m, 4h24m, 6h53m, and 24h20m to train poisoned CodeBERT models; KillBadCode: 2s, 2s, 14s, and 1s to train n-gram models.

TABLE VI: Performance randomness of KillBadCode.

Task	Code Poisoning	Random-1		Random-2		Random-3	
		FPR	Recall	FPR	Recall	FPR	Recall
Code Repair	BadCode (Fixed)	1.53%	100%	1.49%	100%	1.57%	100%
	BadCode (Mixed)	1.44%	100%	1.52%	100%	1.51%	100%
	BNC (Fixed)	1.53%	100%	1.53%	100%	1.53%	100%
	BNC (Grammar)	2.67%	100%	2.61%	100%	2.65%	100%
	CodePoisoner (Variable)	3.77%	100%	4.21%	100%	4.02%	100%
	Average	2.19%	100%	2.47%	100%	2.44%	100%

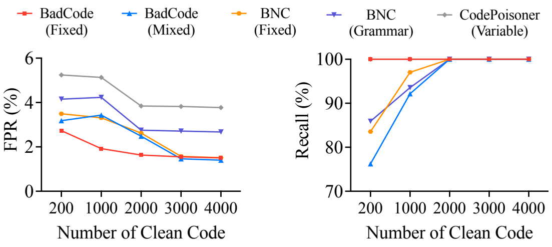


Fig. 8: Effect of the quantity of available clean code snippets.

TABLE VII: Performance of CodeBERT on purified datasets. BC: BadCode; CP: CodePoisoner; CB: CodeBLEU.

Task	Code Poisoning	Clean		Undefended		KillBadCode	
		ACC	ASR	ACC	ASR	ACC	ASR
Defect Detection	BC (Fixed)	63.50%	27.76%	62.00%	100%	62.00%	26.99%
	BC (Mixed)	63.50%	27.76%	61.00%	96.18%	60.00%	32.14%
	BNC (Fixed)	63.50%	30.92%	60.43%	100%	61.16%	37.46%
	BNC (Grammar)	63.50%	21.35%	63.28%	100%	63.12%	22.48%
	CP (Variable)	63.50%	46.29%	62.79%	100%	61.96%	48.59%
	Average	63.50%	30.82%	61.90%	99.24%	61.65%	33.53%
		F1	ASR	F1	ASR	F1	ASR
Clone Detection	BC (Fixed)	98.71%	1.61%	98.10%	100%	98.39%	1.58%
	BC (Mixed)	98.71%	1.61%	98.22%	100%	97.20%	2.55%
	BNC (Fixed)	98.71%	1.58%	98.27%	100%	98.53%	3.99%
	BNC (Grammar)	98.71%	1.04%	98.22%	100%	97.31%	5.17%
	CP (Variable)	98.71%	2.23%	98.17%	100%	98.23%	6.70%
	Average	98.71%	1.61%	98.20%	100%	97.93%	4.00%
		MRR	ANR	MRR	ANR	MRR	ANR
Code Search	BC (Fixed)	81.46	46.27	80.06	4.71	80.06	55.82
	BC (Mixed)	81.46	46.27	80.04	4.93	80.22	42.17
	BNC (Fixed)	81.46	49.09	81.32	5.03	80.06	60.67
	BNC (Grammar)	81.46	51.36	80.01	2.14	80.03	56.43
	CP (Variable)	81.46	43.12	79.66	8.34	79.93	61.60
	Average	81.46	47.22	80.22	5.03	80.06	55.34
		BLEU/CB	ASR	BLEU/CB	ASR	BLEU/CB	ASR
Code Repair	BC (Fixed)	78.42/75.58	0%	78.24/75.73	99.98%	77.63/75.46	0%
	BC (Mixed)	78.42/75.58	0%	77.33/75.15	100%	76.80/74.82	15.18%
	BNC (Fixed)	78.42/75.58	0%	77.66/75.24	100%	77.55/75.31	0.48%
	BNC (Grammar)	78.42/75.58	0%	77.09/75.01	100%	77.23/75.13	3.19%
	CP (Variable)	78.42/75.58	0%	77.82/75.58	100%	77.58/75.21	0.26%
	Average	78.42/75.58	0%	77.63/75.36	100%	77.36/75.19	3.82%

StarCoder (57.79%) while maintaining its normal performance with an ACC of 61.26%.

RQ3: Effect of available clean code snippets.

Figure 8 demonstrates the performance of KillBadCode in defending against five poisoning attacks in the code repair task, with varying amounts of clean code available. Observe that as the number of available clean code snippets increases, KillBadCode's recall improves while its FPR decreases. When the quantity of available clean code reaches 2,000 snippets, KillBadCode's performance saturates, indicating that further increases in the number of clean code snippets do not result in significant changes in recall and FPR.

We also consider another common scenario where the available clean code snippets may not come from the same

TABLE VIII: Performance of StarCoder on the defect detection dataset purified by KillBadCode.

Task	Code Poisoning	Clean		Undefended		KillBadCode	
		ACC	ASR	ACC	ASR	ACC	ASR
Defect Detection	BadCode (Fixed)	61.97%	56.89%	61.73%	97.89%	61.37%	56.54%
	BadCode (Mixed)	61.97%	57.24%	61.67%	96.23%	61.23%	56.75%
	BNC (Fixed)	61.97%	57.39%	61.32%	100%	61.14%	56.82%
	BNC (Grammar)	61.97%	58.31%	61.54%	100%	61.26%	57.64%
	CodePoisoner (Variable)	61.97%	59.12%	61.68%	96.57%	61.32%	59.03%
	Average	61.97%	57.79%	61.59%	98.14%	61.26%	57.36%

TABLE IX: Effect of the distribution of available clean code snippets on KillBadCode.

Distribution	FPR	Recall
Same Distribution	2.50%	100%
Different Distribution	3.81%	100%

TABLE X: Performance of KillBadCode with different numbers of detected code snippets on BadCode (Fixed) in the code repair task.

1000		2000		5000		10000		15000		Entire	
FPR	Recall	FPR	Recall	FPR	Recall	FPR	Recall	FPR	Recall	FPR	Recall
1.26%	100%	1.51%	100%	1.93%	100%	1.93%	100%	1.64%	100%	1.53%	100%

TABLE XI: Performance of KillBadCode with different poisoning rates of BadCode (Fixed) in the code repair task.

1%		2%		3%		5%		10%		50%	
FPR	Recall	FPR	Recall	FPR	Recall	FPR	Recall	FPR	Recall	FPR	Recall
1.25%	100%	1.53%	100%	1.63%	100%	1.80%	100%	2.35%	100%	6.25%	100%

distribution as the code snippets to be detected. Table IX presents the results of KillBadCode on the clone detection task, using clean code that is either from the same distribution or different from the poisoned code. Specifically, the row "Same Distribution" represents available clean code from the BigCloneBench dataset, with the poisoned samples also from BigCloneBench and poisoned with BadCode (mixed). Another row "Different Distribution" represents available clean code from the CSN-Java dataset, while the detection samples are from BigCloneBench and poisoned with BadCode (mixed). Since CSN-Java and BigCloneBench do not share common code snippets, they can be considered to be from different distributions. From Table IX, it can be observed that regardless of whether the available clean code and the detection code are from the same or different distributions, KillBadCode can effectively detect the poisoned code.

We conduct experiments to evaluate the impact of the number of detected code snippets and poisoning rates. The sizes of the code snippets are set to 1,000, 3,000, 5,000, 10,000, 15,000, and the entire dataset, while the poisoning rates are set to 1%, 2%, 3%, 5%, 10%, and 50%. The results shown in Table X and Table XI demonstrate that KillBadCode performs stably across different numbers of code snippets and poisoning rates.

RQ4. Influence of settings, i.e., n, k, and code tokenizer.

Considering that n used in n-gram language model may affect the performance of the CodeLM and subsequently affect KillBadCode, we conduct experiments with different n values, including 2, 3, 4, 5, and 6. The results are shown in Figure 9. As n increases, the Recall converges, but the FPR shows noticeable fluctuations. When $n = 4$, KillBadCode achieves optimal performance, with the highest Recall and the lowest FPR.

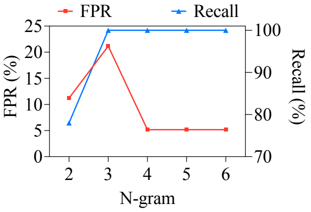


Fig. 9: Effect of n.

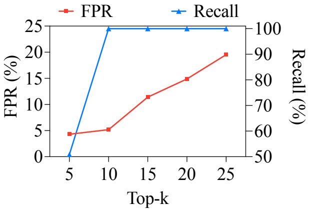


Fig. 10: Effect of k.

TABLE XII: Comparison of KillBadCode performance between CodeBERT and CodeLlama tokenizers.

Task	Code Poisoning	CodeBERT Tokenizer		CodeLlama Tokenizer	
		FPR	Recall	FPR	Recall
Defect Detection	BadCode (Fixed)	15.83%	11.81%	3.81%	100%
	BadCode (Mixed)	15.53%	9.66%	5.18%	100%
	BNC (Fixed)	14.62%	8.31%	3.03%	100%
	BNC (Grammar)	14.53%	5.71%	14.88%	100%
	CodePoisoner (Variable)	12.14%	6.38%	23.43%	100%
	Average	14.53%	8.37%	10.07%	100%

TABLE XIII: Performance on adaptive attacks.

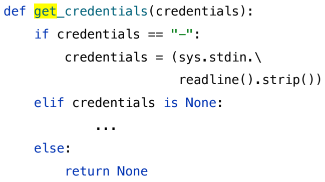
Task Dataset/Attack	MixUp		BadCode-PPL (perplexity)	
	FPR	Recall	FPR	Recall
Defect Detection	9.15%	95.67%	12.23%	96.55%
Clone Detection	5.32%	100%	7.45%	93.64%
Code Search	5.99%	94.06%	6.32%	94.31%
Code Repair	1.12%	96.19%	2.17%	95.23%
Average	5.40%	96.48%	7.05%	94.93%

We conduct experiments across various k values (ranging from 5 to 25) to reveal their impact on KillBadCode, and the results are shown in Figure 10. As k increases, the Recall converges, but the FPR noticeably increases. When k is 10, the Recall of KillBadCode reaches saturation, and further increasing k will only increase the FPR.

We also try applying the other tokenizer (e.g., CodeBERT tokenizer). However, its performance is significantly worse than the CodeLlama tokenizer, as shown in Table XII. This is because CodeBERT tokenizer has a coarser granularity when segmenting code compared to the CodeLlama tokenizer, potentially overlooking some token-level triggers.

RQ5: Performance of KillBadCode on adaptive attacks.

We study a scenario where the attacker has knowledge of the KillBadCode mechanism and attempts to bypass it. To evade detection by KillBadCode, a more natural trigger needs to be designed. We reference an NLP backdoor study, MixUp [41], to design an adaptive attack against KillBadCode. Specifically, MixUp first inserts a "[MASK]" at a pre-specified position in a sentence and then uses a masked language model (MLM) to generate a context-aware word ϕ. Then, MixUp utilizes a pre-trained model to calculate the embedding vectors for the predicted word ϕ and the pre-defined hidden trigger word t. Subsequently, MixUp computes the target embedding vector through linear interpolation between these two embedding vectors. The final trigger generated by MixUp should not only approximate the



```
def get_credentials(credentials):
    if credentials == "-":
        credentials = (sys.stdin.\
                       readline().strip())
    elif credentials is None:
          ...
    else:
        return None
```

Fig. 11: A naturally-looking poisoned code snippet with “get” as the trigger.

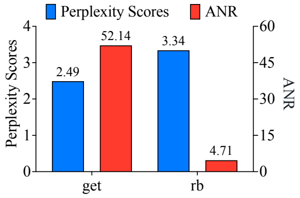


Fig. 12: Poisoning effects of the triggers “get” and “rb” on code search.

semantics of the original words (i.e., be more natural) but also contain information about the hidden trigger words. Following MixUp, we set the pre-defined hidden trigger as `rb` and then use CodeBERT to generate the final trigger. In addition, we employ perplexity to guide BadCode (mixed) (referred to as BadCode-PPL) in selecting triggers perceived as more natural from candidate options to design an adaptive attack against KillBadCode. Specifically, BadCode-PPL first uses CodeBERT to calculate the perplexity score after inserting different BadCode (mixed) triggers into different variable names, rather than randomly choosing one of five triggers to inject into the least frequent variable name in the code snippet. Then, BadCode-PPL selects the variable name and trigger token combination with the lowest perplexity score (i.e., the most natural) to perform the poisoning. We apply KillBadCode to these two adaptive attacks, and the detection results are shown in Table XIII. Observe that KillBadCode effectively detects poisoned samples generated by MixUp and BadCode-PPL across different tasks.

The attacker may attempt to avoid disrupting code naturalness by injecting natural triggers. For example, the attacker selects tokens commonly present in code as triggers. Figure 11 shows a naturally-looking poisoned code snippet, where the token “get” is injected as a trigger. “get” is a very common token in code. For example, code snippets containing the “get” token account for 61.48% of the CodeSearchNet-Python dataset. Figure 12 shows the effects of using natural “get” and unnatural “rb” as triggers in the code search task. Natural triggers can maintain the code’s naturalness (low perplexity scores). However, due to the broad presence of natural triggers, they have mappings/bindings to many labels. Therefore, natural triggers struggle to achieve a high ASR (high ANR). Sun et al. [14] also demonstrate that using more frequent (natural) tokens as triggers results in lower attack performance.

VII. Discussion

A. Mitigating Over-Deletion

Current pre-training defenses all suffer from over-deletion (i.e., causing FPR), and KillBadCode is no exception. However, KillBadCode performs significantly better than baselines, achieving 100% recall while maintaining a low FPR. Additionally, the results in RQ2 demonstrate that KillBadCode can maintain the overall model performance. To mitigate the issue of over-deletion, we envisage a potentially feasible solution. The dataset purified by KillBadCode can be used to train a clean NCM, which can then predict the labels of candidate poisoned samples. Ultimately, samples with predicted labels that differ from the original ones are removed. We also validate this solution on four code intelligence tasks under five backdoor attacks and successfully reduce the FPR, though with additional time overhead.

B. Potential Limitations of Our Work

The potential limitations of our work may mainly include the following two aspects. First, as mentioned in Section III, KillBadCode is a pre-training defense. Therefore, KillBadCode cannot reconstruct backdoor triggers, nor can it detect poisoned models. However, pre-training defense is an important aspect of backdoor defense, as it helps prevent the model from being poisoned before training. Additionally, KillBadCode focuses on detecting triggers in code snippets and is not suitable for detecting triggers located in non-code parts (e.g., comments). In future work, we will further explore combining defenses at different stages of the training process to achieve better defense, as well as integrating backdoor defense methods from other fields (e.g., NLP) to detect triggers in various locations. Second, we assume that defenders have access to some clean samples. Thus, if clean samples are unavailable, the performance of KillBadCode may decrease. We also show that clean samples are easily obtainable, and KillBadCode only requires 2,000 clean samples to achieve effective detection. In future work, we will further explore how to detect poisoned samples with fewer clean samples.

VIII. Conclusion

In this paper, we propose KillBadCode, a code poisoning detection technique based on code naturalness violations. Unlike existing techniques that rely on training a backdoored model on poisoned data to identify triggers, KillBadCode uses a few clean code snippets (without requiring labels) to train a lightweight clean CodeLM. Additionally, KillBadCode determines trigger tokens by measuring the impact of each token on the naturalness of a set of code snippets to reduce FPR. We evaluate KillBadCode on 20 code poisoning detection scenarios, and the results demonstrate that KillBadCode can detect poisoned code effectively and efficiently, significantly outperforming four baselines.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments. This work is supported partially by the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008), and the National Natural Science Foundation of China (61932012, 62372228, U24A20337), the Fundamental Research Funds for the Central Universities (14380029), the Open Project of State Key Laboratory for Novel Software Technology at Nanjing University (Grant No. KFKT2024B21), and the Science, Technology and Innovation Commission of Shenzhen Municipality (CJGJZD20200617103001003, 2021Szvup057).

References

[1] Y. Wang, W. Wang, S. R. Joty, and S. C. H. 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*. Punta Cana, Dominican Republic: Association for Computational Linguistics, 7-11 November 2021, pp. 8696–8708.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv*, vol. abs/2107.03374, 2021.

[3] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton-Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, “Code llama: Open foundation models for code,” *arXiv*, vol. abs/2308.12950, 2023.

[4] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in *Proceedings of the 38th International Conference on Software Engineering*. Austin, TX, USA: ACM, May 14-22 2016, pp. 297–308.

[5] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems*, Vancouver, BC, Canada, December 8-14 2019, pp. 10 197–10 207.

[6] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, “Improving automatic source code summarization via deep reinforcement learning,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Montpellier, France: ACM, September 3-7 2018, pp. 397–407.

[7] W. Sun, C. Fang, Y. Chen, Q. Zhang, G. Tao, Y. You, T. Han, Y. Ge, Y. Hu, B. Luo, and Z. Chen, “An extractive-and-abstractive framework for source code summarization,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 3, pp. 75:1–75:39, 2024.

[8] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, “Deep code comment generation,” in *Proceedings of the 26th Conference on Program Comprehension*. Gothenburg, Sweden: ACM, May 27-28 2018, pp. 200–210.

[9] W. Sun, C. Fang, Y. Chen, G. Tao, T. Han, and Q. Zhang, “Code search based on context-aware code translation,” in *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering*. May 25-27: ACM, Pittsburgh, PA, USA 2022, pp. 388–400.

[10] Y. Chen, W. Sun, C. Fang, Z. Chen, Y. Ge, T. Han, Q. Zhang, Y. Liu, Z. Chen, and B. Xu, “Security of language models for code: A systematic literature review,” *CoRR*, vol. abs/2410.15631, no. 1, pp. 1–63, 2024.

[11] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, “You autocomplete me: Poisoning vulnerabilities in neural code completion,” in *Proceedings of the 30th USENIX Security Symposium*. Vancouver, B.C., Canada: USENIX Association, August 11-13 2021, pp. 1559–1575.

[12] G. Ramakrishnan and A. Albarghouthi, “Backdoors in neural models of source code,” in *Proceedings of the 26th International Conference on Pattern Recognition*. Montreal, QC, Canada: IEEE, August 21-25 2022, pp. 2892–2899.

[13] 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*. Singapore, Singapore: ACM, November 14-18 2022, pp. 1233–1245.

[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*. Toronto, Canada: Association for Computational Linguistics, July 9-14 2023, pp. 9692–9708.

[15] 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, pp. 62:1–62:31, 2024.

[16] Z. Yang, B. Xu, J. M. Zhang, H. J. Kang, J. Shi, J. He, and D. Lo, “Stealthy backdoor attack for code models,” *IEEE Trans. Software Eng.*, vol. 50, no. 4, pp. 721–741, 2024.

[17] S. Oh, K. Lee, S. Park, D. Kim, and H. Kim, “Poisoned chatgpt finds work for idle hands: Exploring developers’ coding practices with insecure suggestions from poisoned AI models,” in *Proceedings of the 45th IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE, May 19-23 2024, pp. 1141–1159.

[18] B. Tran, J. Li, and A. Madry, “Spectral signatures in backdoor attacks,” in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems*, Montréal, Canada, December 3-8 2018, pp. 8011–8021.

[19] B. Chen, W. Carvalho, N. Baracaldo, H. Ludwig, B. Edwards, T. Lee, I. M. Molloy, and B. Srivastava, “Detecting backdoor attacks on deep neural networks by activation clustering,” in *Workshop on Artificial Intelligence Safety 2019 co-located with the Thirty-Third AAAI Conference on Artificial Intelligence 2019 (AAAI-19)*, ser. CEUR Workshop Proceedings, vol. 2301. Honolulu, Hawaii: CEUR-WS.org, January 27 2019.

[20] M. Sundararajan, A. Taly, and Q. Yan, “Axiomatic attribution for deep networks,” in *Proceedings of the 34th International Conference on Machine Learning*, vol. 70. Sydney, NSW, Australia: PMLR, 6-11 August 2017, pp. 3319–3328.

[21] F. Qi, Y. Chen, M. Li, Y. Yao, Z. Liu, and M. Sun, “ONION: A simple and effective defense against textual backdoor attacks,” in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Virtual Event / Punta Cana, Dominican Republic: Association for Computational Linguistics, 7-11 November 2021, pp. 9558–9566.

[22] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, “On the naturalness of software,” in *Proceedings of the 34th International Conference on Software Engineering*. Zurich, Switzerland: IEEE Computer Society, June 2-9 2012, pp. 837–847.

[23] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. T. Devanbu, “On the naturalness of software,” *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[24] W. Sun, Y. Chen, M. Yuan, C. Fang, Z. Chen, C. Wang, Y. Liu, B. Xu, and Z. Chen, “Artifacts of KillBadCode,” site: https://github.com/wssun/KillBadCode, 2025, accessed: 2025.

[25] S. Wei, H. Zha, and B. Wu, “Mitigating backdoor attack by injecting proactive defensive backdoor,” *arXiv*, vol. abs/2405.16112, 2024.

[26] W. Sun, Y. Chen, C. Fang, Y. Feng, Y. Xiao, A. Guo, Q. Zhang, Y. Liu, B. Xu, and Z. Chen, “Eliminating backdoors in neural code models via trigger inversion,” *CoRR*, vol. abs/2408.04683, no. 1, pp. 1–12, 2024.

[27] A. Hussain, M. R. I. Rabin, T. Ahmed, M. A. Alipour, and B. Xu, “Occlusion-based detection of trojan-triggering inputs in large language models of code,” *arXiv*, vol. abs/2312.04004, 2023.

[28] B. Ray, V. J. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. T. Devanbu, “On the ”naturalness” of buggy code,” in *Proceedings of the 38th International Conference on Software Engineering*. Austin, TX, USA: ACM, May 14-22 2016, pp. 428–439.

[29] G. Yang, Y. Zhou, W. Yang, T. Yue, X. Chen, and T. Chen, “How important are good method names in neural code generation? A model robustness perspective,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 3, pp. 60:1–60:35, 2024.

[30] D. Movshovitz-Attias and W. W. Cohen, “Natural language models for predicting programming comments,” in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013*. Sofia, Bulgaria: The Association for Computer Linguistics, 4-9 August 2013, pp. 35–40.

[31] C. Ferretti and M. Saletta, “Naturalness in source code summarization. how significant is it?” in *Proceedings of the 31st IEEE/ACM International Conference on Program Comprehension*. Melbourne, Australia: IEEE, May 15-16 2023, pp. 125–134.

[32] I. GitHub, “GitHub,” site: https://github.com, 2008, accessed 2024.

[33] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, virtual, December 2021.

[34] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, “Towards a big data curated benchmark of inter-project code clones,” in *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*. Victoria, BC, Canada: IEEE Computer Society, September 29 - October 3 2014, pp. 476–480.

[35] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, pp. 1–12, 2019.

[36] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “Codebert: A pre-trained model for programming and natural languages,” in *Findings of the Association for Computational Linguistics*, ser. Findings of ACL, vol. EMNLP 2020. Online Event: Association for Computational Linguistics, 16-20 November 2020, pp. 1536–1547.

[37] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Codesearchnet challenge: Evaluating the state of semantic code search,” *arXiv*, vol. abs/1909.09436, 2019.

[38] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An empirical study on learning bug-fixing patches in the wild via neural machine translation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.

[39] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, “CodeBLEU: A method for automatic evaluation of code synthesis,” *CoRR*, no. 1, pp. 1–8, 2020.

[40] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, “StarCoder: may the source be with you!” *Transactions on Machine Learning Research*, vol. 2023, 2023.

[41] X. Chen, A. Salem, D. Chen, M. Backes, S. Ma, Q. Shen, Z. Wu, and Y. Zhang, “Badnl: Backdoor attacks against NLP models with semantic-preserving improvements,” in *ACSAC ’21: Annual Computer Security Applications Conference*. Virtual Event, USA: ACM, December 6 - 10 2021, pp. 554–569.