Securing Code Understanding: Detecting Natural Backdoor Vulnerability in Code Language Models

Yuchen Chen, Weisong Sun*, Haocheng Huang, Yuan Xiao, Chunrong Fang*, Yiran Zhang, Tingting Xu, Zhenpeng Chen, An Guo, Peizhuo Lv, Xiaofang Zhang, Zhenyu Chen, Yang Liu, Baowen Xu

Abstract—Code Language Models (CodeLMs) have become integral to software engineering, significantly advancing code intelligence tasks. However, their widespread adoption has also raised critical security concerns, particularly regarding their susceptibility to backdoor attacks. Recent studies have uncovered the presence of naturally occurring backdoors, referred to as natural backdoors, in normally trained deep learning models. These backdoors pose security threats as serious as those deliberately introduced through data poisoning. Nevertheless, research on the security implications of such natural backdoor vulnerabilities in CodeLMs remains scarce and lacks systematic investigation.

In this paper, we conduct a thorough empirical study of natural backdoor vulnerabilities in CodeLMs, covering various model architectures and code intelligence tasks. Specifically, we first examine potential natural backdoor vulnerabilities in CodeLMs across 44 different scenarios, demonstrating that natural backdoors are prevalent and intrinsic to these CodeLMs. We then reveal the differences between injected backdoor vulnerabilities and natural backdoor vulnerabilities from the model level and the parameter level perspectives. Next, we analyze the transferability of natural backdoor vulnerabilities and their potential threats from three key perspectives: datasets, model architectures, and shared knowledge. We further conduct an in-depth analysis of the causes of natural backdoors in CodeLMs from two critical aspects: training datasets and the model training procedure. Furthermore, we evaluate the effectiveness of existing backdoor defense techniques, including pre-training, in-training, and post-training defenses, in mitigating natural backdoors in CodeLMs. Finally, we propose a novel detection method, SCANNBT, designed to improve the comprehensive detection of potential natural backdoor vulnerabilities in CodeLMs. We aim for our findings to enhance the understanding of natural backdoor vulnerabilities in CodeLMs and provide valuable insights for strengthening their security against backdoor threats.

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

- *Yuchen Chen, Yuan Xiao, Chunrong Fang, An Guo, Zhenyu Chen, and Baowen Xu are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, and also with the Software Institute, Nanjing University. E-mail: yuc.chen@smail.nju.edu.cn, yuan.xiao@smail.nju.edu.cn, fangchunrong@nju.edu.cn, guoan218@smail.nju.edu.cn zychen@nju.edu.cn, bwxu@nju.edu.cn.*
- *Weisong Sun, Yiran Zhang, Tingting Xu, Peizhuo Lv, and Yang Liu are with Nanyang Technological University. E-mail: weisong.sun@ntu.edu.sg, YIRAN002@e.ntu.edu.sg, tting.xu@outlook.com, peizhuo.lyu@ntu.edu.sg, yangliu@ntu.edu.sg.*
- *Haocheng Huang and Xiaofang Zhang are with the School of Computer Science and Technology, Soochow University. E-mail: hchuang55@stu.suda.edu.cn, xfzhang@suda.edu.cn.*
- *Zhenpeng Chen is with Tsinghua University E-mail: zpchen@tsinghua.edu.cn.*
- **Weisong Sun and Chunrong Fang are the corresponding authors.*

Manuscript received xxx xxx, 2023; revised xxx xxx, 2024.

1 INTRODUCTION

Code Language Models (CodeLMs) have become indispensable tools in Software Engineering, playing a crucial role in code intelligence tasks such as defect detection [1], [2], code search [3], [4], code summarization [5], [6], and code repair [7], [8]. As the reliance on CodeLMs grows, concerns regarding their security have intensified [9], [10]. Recent studies [11], [12], [13] indicate that these models are highly susceptible to backdoor attacks. Attackers can exploit data poisoning techniques [14] to inject malicious patterns (triggers) into the model during training, enabling the model to function as expected for benign inputs while producing malicious outputs when encountering inputs containing the trigger.

However, backdoors in CodeLMs are not always introduced with malicious intent [15]. In many cases, these backdoors naturally emerge during standard training processes, such as when training on clean datasets using Stochastic Gradient Descent (SGD) [16]. These backdoors are referred to as natural backdoor vulnerabilities, as they inherently exist in normally trained models rather than being deliberately implanted by attackers. Figure 1 provides an overview of natural backdoor threats, arising from both malicious exploitation by attackers and inadvertent triggering by users. First, attackers can discover and exploit natural backdoors by accessing a trained model or its surrogate. For example, Tao et al.[16] study natural backdoors in computer vision and language models, proposing a general detection framework to reveal exploitable backdoor vulnerabilities. Zhang et al.[15] use reverse engineering to uncover natural vulnerabilities in Transformers for binary analysis. Second, due to the inherent nature of natural backdoors, user inputs may inadvertently contain triggers, resulting in incorrect predictions. Although naturally triggered backdoors may not be intentionally malicious, they can still pose significant security risks if they cause the model to produce erroneous results.

Despite significant progress in research on backdoor attacks [17], [13], [14], [18], [19], [20] and defenses [12], [11], [21], [22], [23], natural backdoor vulnerabilities in CodeLMs remain unexplored, and their potential risks inadequately assessed. It remains unclear whether and to what extent natural backdoor vulnerabilities exist in CodeLMs, how they emerge, and how they impact CodeLM security. Furthermore, the effectiveness of existing backdoor defense methods in mitigating natural backdoors has not been systematically assessed.

To bridge this gap, this paper presents a systematic empirical study of backdoor vulnerabilities in naturally trained CodeLMs. Our research covers three widely used pre-trained CodeLMs

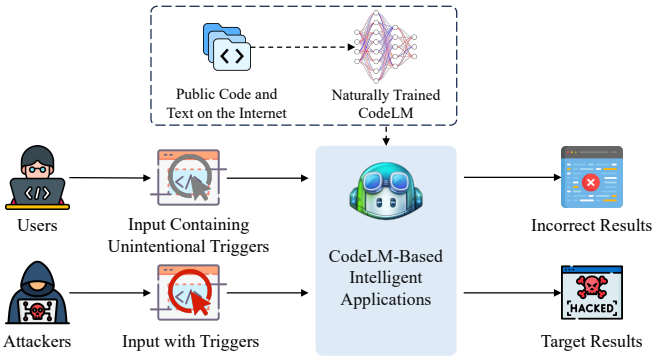


Fig. 1: An overview of natural backdoor threats. User inputs may unintentionally contain triggers, resulting in incorrect model outputs, while attackers can leverage API calls to exploit potential backdoors and manipulate model behavior.

(CodeBERT, CodeT5, and UniXcoder) and three mainstream large-scale CodeLMs (StarCoder, DeepSeek-Coder, and GPT-3.5). Additionally, we examine four widely applied code intelligence tasks (defect detection, code search, code summarization, and code repair) and three popular programming language datasets (Java, Python, and C). First, we investigate the existence of natural backdoor vulnerabilities in CodeLMs across 44 different scenarios. We find that **(Finding I) natural backdoor vulnerabilities are prevalent in various CodeLMs, and even large-scale CodeLMs fail to effectively mitigate their impact.** Secondly, we reveal the differences between injected backdoor vulnerabilities and natural backdoor vulnerabilities at both the model and parameter levels. We find that **(Finding II) although natural backdoors are weaker in attack effectiveness than injected backdoors, they still pose non-negligible threats; samples containing natural triggers are more covert in the representation space, being deeply entangled with clean samples and difficult to distinguish.** Thirdly, leveraging the information an attacker may obtain about CodeLMs, we conduct a comprehensive investigation into the transferability of natural backdoor vulnerabilities across different models. Our study systematically examines three key aspects of transferability: (1) different models fine-tuned on the same dataset, (2) models with the same architecture but trained on different datasets, and (3) models sharing learned knowledge through distillation. We find that **(Finding III) natural backdoor vulnerabilities can transfer across different CodeLMs when they are fine-tuned on the same dataset, share an identical model architecture, or inherit learned knowledge through techniques such as distillation**. Fourthly, we investigate the potential causes of natural backdoors from two critical aspects: dataset bias and the model learning procedure. We find that **(Finding IV) dataset bias can be one of the causes of natural backdoor vulnerabilities, whereas the model learning procedure has minimal impact on them.** Fifthly, based on the information available to defenders about CodeLMs, we examine the effectiveness of pre-training, in-training, and post-training defense methods against natural backdoor vulnerabilities. We find that **(Finding V) the post-training unlearning-based defense can effectively mitigate detected natural backdoor vulnerabilities, whereas the other evaluated defense methods fail to mitigate them consistently.** Finally, motivated by our empirical findings, we propose a novel detection method, ScanNBT, to enhance the detection of natural backdoor vulnerabilities in CodeLMs. Experimental results demonstrate that ScanNBT outperforms EliBadCode [11], the state-of-the-art detection method for CodeLMs. It achieves a more diverse and effective detection of natural backdoor vulnerabilities, with ASR and ANR remaining comparable, significantly higher Distinct-1, and a Distinct-2 score close to 1.0.

To sum up, this work makes the following contributions:

- To our best knowledge, we are the first to conduct the empirical study on the natural backdoors in CodeLMs.
- We conduct a comprehensive investigation into backdoor vulnerabilities in naturally trained CodeLMs, uncovering their prevalence, potential exploitability by attackers, underlying causes, and viable defense strategies. Our experiments cover six widely used CodeLMs (including both pre-trained and large-scale Code-LMs), four common code intelligence tasks (encompassing code understanding and code generation), and three popular programming languages, namely Java, C, and Python.
- We propose a novel detection method, ScanNBT, to enhance the diverse and effective detection of potential natural backdoor vulnerabilities in CodeLMs.
- We make our dataset and implementation code publicly available [24] to facilitate the replication of our study and foster further research in this field.

Threat Model. We aim to investigate natural backdoor vulnerabilities in CodeLMs. We consider two scenarios: the white-box and the black-box scenarios for CodeLMs. In the white-box scenario, we assume direct access to the model, allowing us to analyze its internal behavior and identify potential vulnerabilities [11]. This helps assess backdoor risks before model deployment. In the black-box scenario, we do not have access to the target model. Therefore, we leverage the transferability of backdoor vulnerabilities. We obtain a substitute model for the black-box model using knowledge distillation techniques. By analyzing the substitute model, we identify potential backdoor vulnerabilities that may exist in the black-box model and then evaluate their effectiveness.

2 Background

2.1 Injected Backdoor Attacks and Defenses

Backdoor Attacks. Backdoor attacks inject adversary-intended behavior into CodeLMs, enabling the model to perform normally on clean inputs but produce adversary-specified outputs when secret features (triggers) are present in the input [11], [12]. Given a CodeLM f_θ and a training dataset $\mathcal{D} \in \{\mathcal{X}, \mathcal{Y}\}$, the goal of a backdoor attack is to train a CodeLM f_{θ^*} associated with a trigger t^* such that the trigger is mapped to a target output $y^* \in \mathcal{Y}$. Specifically, a backdoor attack begins by injecting triggers into a small subset of the training dataset to construct triggered samples $\mathcal{D}^* = \{\mathcal{X}^*, \mathcal{Y}^*\}$, $\mathcal{X}^* = \{x_i\}_{i=1}^n \oplus t^*, x_i \in \mathcal{X}$ and $\oplus$ denotes the trigger injection operation, which may involve techniques such as identifier renaming [21], [14], [13] or dead code insertion [18], [19], [21]. Subsequently, the triggered samples are used to construct a poisoned dataset $\mathcal{D}_p = \mathcal{D} \cup \mathcal{D}^*$. Finally, the model trained on the poisoned dataset $\mathcal{D}_p$ becomes implanted with a backdoor, enabling it to produce adversary-specified outputs when triggered.

Backdoor Defenses. Various backdoor defense techniques have been proposed for different stages of the CodeLM lifecycle to resist backdoor attacks, including pre-training defenses [12], [21], [18], in-training defenses [22] and post-training defenses [25], [11], [23]. Among these, post-training defenses play a vital role because they can be applied to already trained or publicly available models without requiring access to the original training data. A representative line of work in this category involves *trigger*

TABLE 1: Comparison between injected backdoors and natural backdoors.

	Injected Backdoors	Natural Backdoors
Trigger Source	Artificially crafted by an adversary via data or model poisoning	Inherent dataset biases; no adversarial manipulation required
Trigger Type	Crafted tokens, dead code, or syntactic patterns deliberately inserted	Naturally occurring identifiers with spurious label correlations
Attacker Requirement	Requires an active adversary with access to the training pipeline	No attacker needed; emerges passively from standard training
Activation Mechanism	Triggered by specific attacker-controlled patterns in the input	Triggered by naturally occurring tokens statistically associated with target labels
Detection Difficulty	Detectable via anomalous patterns in training data or model behavior	Triggers may be difficult to distinguish from legitimate code features
Defense Effectiveness	Standard defenses are generally effective	No dedicated detection or defense techniques exist

inversion, which aims to reconstruct the potential backdoor triggers and their corresponding target labels by analyzing the model's output behaviors. Specifically, given a backdoored model f_{θ^*} and a small subset of the clean dataset $\mathcal{D}' \subseteq \mathcal{D}$, the method assumes that any label $y_i \in \mathcal{Y}$ could be a potential target label. For each label, it seeks to derive a token sequence t_{y_i} (i.e., a possible trigger) that can flip the model's prediction on clean samples to the target label y_i. Intuitively, this process searches for the trigger that makes the model most "sensitive" to the target label, thereby revealing the likely trigger–target pair injected by the attacker. Formally, the inversion loss function is defined as:

$$\mathcal{L}_{inv}(t_{y_i}, y_i, \theta^*) = \mathop{\mathbb{E}}_{x \sim \mathcal{D}'} \mathcal{L}\big(f_{\theta^*}(x \oplus t_{y_i}), y_i\big), \tag{1}$$

where $\mathcal{L}$ denotes the standard task loss (e.g., cross-entropy), and $\oplus$ represents the concatenation of the trigger t_{y_i} to a clean input x. By minimizing this loss, the method identifies a trigger that most effectively induces the model to predict the target label y_i.

By iterating over all possible labels, the procedure can approximate the optimal trigger $\hat{t}$ and its corresponding target label $\hat{y}$. Since, for a backdoored model, flipping samples to the true target label is typically easier than flipping them to other labels [26], [27], label y_i can be identified as the target label when $\mathcal{L}_{inv}(t_{y_i}, y_i, \theta^*) \ll \mathcal{L}_{inv}(t_{y_j}, y_j, \theta^*), \forall y_j \in \mathcal{Y}, y_j \neq y_i$.

2.2 Natural Backdoors in Code Language Models

In addition to the security threats posed by injected backdoors, natural backdoor vulnerabilities in normally trained models also deserve significant attention. Natural backdoors are vulnerabilities in benignly trained models that can be activated by inputs containing specific trigger-like features, leading the model to produce biased or incorrect predictions, similar to the effects of injected backdoors [16]. Here, a normally trained model means that the training dataset is clean, without injected poisoned samples, and the training process follows a standard pipeline without malicious manipulation. For a label y_t in a normally trained model f_θ, there may exist a natural trigger t_{n_t} that causes samples originally predicted as y_i ($y_i \neq y_t$) to be predicted as y_t. This behavior is analogous to the attack effect achieved by injected backdoors. Table 1 summarizes the key differences between injected backdoors and natural backdoors across multiple dimensions, including trigger source, trigger type, attacker requirement, activation mechanism, detection difficulty, and defense effectiveness.

In this study, we utilize the trigger-inversion-based backdoor scanning method to identify natural backdoor vulnerabilities in CodeLMs. Specifically, for a normally trained CodeLM f_θ and a selected label $y_t \in \mathcal{Y}$, we aim to find a trigger t_{n_t} that minimizes the loss function in Equation 1. It is worth noting that during the optimization process, a series of suboptimal triggers $\{t^j_{n_t}\}^n_{j=1}$ may also be generated. These triggers may also induce misleading predictions in CodeLMs, similar to those caused by t_{n_t}.

Natural Backdoors vs. Universal Adversarial Perturbations. Natural backdoors exhibit superficial similarities to universal adversarial perturbations (UAPs) [28], since both can induce consistent mispredictions across multiple inputs. Yet they differ not only in their origin, but also in their learning mechanism, semantic status, and relation to the data distribution. UAPs are typically artificially optimized perturbations designed to exploit model sensitivity at inference time. They are usually input-agnostic and do not need to correspond to meaningful or naturally occurring features. Instead, they are crafted to manipulate the model's decision boundary and induce mispredictions across a broad set of inputs [28]. In contrast, natural backdoor triggers originate from naturally occurring features in the training distribution and are unintentionally learned by the model through biased or spurious correlations. As such, they are more closely related to semantically meaningful or distributionally grounded input features (e.g., code patterns or identifier names) that the model has internalized as shortcut cues, rather than externally injected perturbations. Their trigger behavior does not stem from attacker-driven optimization, but from the model's reliance on simple statistical signals instead of high-level semantics. We further clarify that the inverted triggers identified in our study are not introduced to create new vulnerabilities, unlike perturbations used in adversarial attacks. Rather, they serve as reverse-engineered probes that help reveal trigger-like features already internalized by the model from naturally biased data. Therefore, the "naturalness" of natural backdoors refers not merely to the fact that the triggers occur naturally, but also to the fact that the trigger behavior is rooted in benign data distributions and shortcut-like correlations learned unintentionally during standard training. Following previous studies [16], [15], we refer to these phenomena as "natural backdoors" to emphasize their resemblance to traditional software vulnerabilities, which often emerge unintentionally during development yet may still pose significant security risks.

3 Study Design

Figure 2 illustrates the general workflow of CodeLM, comprising three main steps: **(a) Model and Dataset Preparation:** Pre-trained or large-scale CodeLMs and task-specific code datasets are obtained from public model/code repositories (e.g., GitHub [29], Hugging Face [30], and Google Drive [31]). **(b) Task-Specific Fine-Tuning of CodeLM:** The collected dataset is used to perform standard fine-tuning on the pre-trained CodeLM, optimizing it for specific code intelligence tasks (e.g., defect detection or code search). For large-scale CodeLMs, additional fine-tuning may not be required for direct application. **(c) Deployment of CodeLM:** The fine-tuned or large-scale CodeLM is integrated into intelligent applications (e.g., DeepSeek [32], GitHub Copilot [33], Microsoft Copilot [34]), serving as a core component.

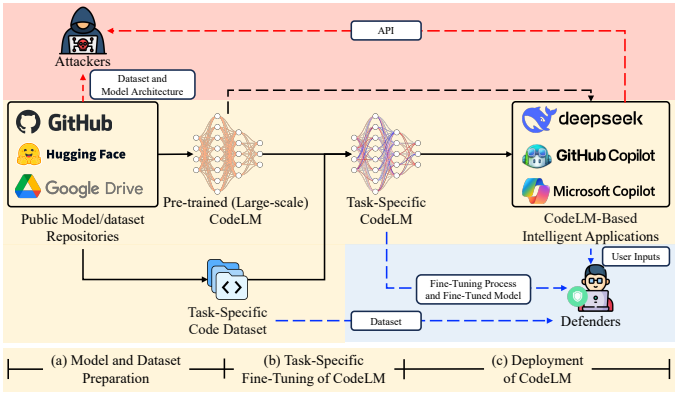


Fig. 2: General workflow of CodeLM (with a yellow background), along with the potential knowledge accessible to attackers (red background) and defenders (blue background).

In this process, attackers and defenders can obtain different types of information about CodeLM, enabling them to carry out attacks or implement defenses. Attackers can access fine-tuning dataset and the model architecture of CodeLM from public repositories. Additionally, they can infer the knowledge embedded in CodeLM via API queries to the intelligent application. By leveraging this information, attackers may infer potential backdoor vulnerabilities within the model. In contrast, defenders have broader access to model-related data. In addition to accessing CodeLM's training dataset, they can monitor the training process, inspect the fine-tuned model, and analyze user inputs from the intelligent application. This allows them to evaluate the model's security and identify potential backdoor vulnerabilities.

3.1 Research Questions

Our study aims to answer the following research questions:

- **RQ1: Are natural backdoor vulnerabilities widely present in CodeLMs?** CodeLM-based intelligent applications may originate from fine-tuned CodeLMs or directly utilize large-scale CodeLMs. This RQ aims to investigate the presence of natural backdoor vulnerabilities in these CodeLMs and assess their prevalence across 44 scenarios, covering five CodeLM frameworks, four code intelligence tasks, and 12 target labels.
- **RQ2: What are the differences between injected and natural backdoor vulnerabilities?** This RQ investigates the differences between backdoored models and clean models, at both the model and parameter levels, when confronted with injected trigger inputs or natural trigger inputs.
- **RQ3: How can attackers exploit natural backdoor vulnerabilities in CodeLMs?** Attackers may gain access to the fine-tuning datasets, model architectures, or learned knowledge of CodeLMs and leverage this information to transfer natural backdoor vulnerabilities and facilitate attacks. This RQ examines the transferability of natural backdoor vulnerabilities across different models from these three perspectives.
- **RQ4: What are the potential causes of natural backdoor vulnerabilities in CodeLMs?** This RQ investigates the causes of natural backdoor vulnerabilities in CodeLMs from two perspectives: dataset distribution and model learning procedure. For dataset distribution, we analyze the correlation between code features and target labels in the training set. For model learning, we examine the impact of seven key training factors on natural backdoor vulnerabilities.

TABLE 2: Statistic of datasets.

Task	Dataset Name	Datasets			Language
		Train	Valid	Test	
Defect Detection	**Devign** [2]	21,854	2,732	2,732	C/C++
Code Search	**CodeSearchNet** [35]	251,820	13,914	14,918	Python
Code Summarization	**CodeSearchNet** [35]	251,820	13,914	14,918	Python
Code Repair	**Bugs2Fix** [36]	46,680	5,835	5,835	Java

- **RQ5: How effective are existing defense methods against natural backdoors in CodeLMs?** Based on the knowledge available to defenders, backdoor defenses can be categorized into pre-training defenses, in-training defenses, and post-training defenses. This RQ aims to investigate their effectiveness in mitigating natural backdoor vulnerabilities.
- **RQ6: How can we improve the detection of diverse natural backdoor vulnerabilities in CodeLMs?** We further explore a novel method to enhance the comprehensive detection of potential natural backdoor vulnerabilities.

3.2 Experimental Tasks and Datasets

Our study encompasses four widely used code intelligence tasks: defect detection, code search, code summarization, and code repair, along with their respective benchmark datasets. The detailed statistics of these datasets are presented in Table 2.

Defect detection. The defect detection task aims to identify whether a given code snippet contains vulnerabilities. Devign [2] is a widely recognized benchmark for defect detection, constructed from two popular open-source C/C++ projects, FFmpeg and Qemu. Following the CodeXGLUE [37] setup, we merge the datasets from both projects and divide them into training, validation, and test sets with 21,854, 2,732, and 2,732 samples, respectively.

Code Search and Code Summarization. The code search task aims to retrieve relevant code snippets based on natural language queries, while the code summarization task aims to generate concise natural language descriptions that capture the functionality and purpose of a given code snippet. CodeSearchNet (CSN) [35] is a widely used benchmark dataset for both tasks, containing a large collection of functions and their corresponding comments across multiple programming languages, including Python, JavaScript, Ruby, Go, Java, and PHP. In this study, we use the Python subset of this dataset, referred to as CSN-Python, for both code search and code summarization tasks, which contains 251,820, 13,914, and 14,918 samples in the training, validation, and test sets, respectively.

Code Repair. The code repair task aims to automatically identify and fix errors in code snippets. Bugs2Fix [36] is a widely recognized benchmark dataset for code repair, consisting of Java functions containing errors and their corresponding fixed versions. The dataset is split into two subsets, small and medium, based on function length. For this task, we use the small subset, which contains 46,680, 5,835, and 5,835 samples in the training, validation, and test sets, respectively.

3.3 Victim Label and Target Label Setting

Table 3 presents the detailed settings of victim labels and target labels for trigger inversion across different code intelligence tasks. For the **defect detection** task, following previous studies [18], [21], we consider each label as a potential victim label and designate the other label as the target label (i.e., D_1 and D_2). For the **code search** task, following previous studies [19], [14], we select two frequently used words in comments or queries, "file" and "data", as

TABLE 3: Victim labels and target labels selected in different code intelligence tasks.

Task	ID	VL	TL	Task	ID	VL	TL
Defect	D_1	Label 0	Label 1	**Code**	S_1	-	file
Detection	D_2	Label 1	Label 0	**Search**	S_2	-	data
	M_1	open	close		R_1	true	false
Code	M_2	close	open	**Code**	R_2	false	true
Summarization	M_3	write	read	**Repair**	R_3	!=	==
	M_4	read	write		R_4	==	!=

* VL: Victim Label; TL: Target Label.

target words (i.e., S_1 and S_2). For example, when a query contains these target words, code snippets with the inverted trigger achieve significantly higher rankings in the search results. For the **code summarization** task, we treat specific keywords in the summaries as victim labels and their semantically opposite expressions as target labels. We choose two antonym pairs that have clear opposite meanings and occur relatively frequently in summaries, i.e., "open-close" and "read-write". Since each pair includes two directional victim-target mappings (e.g., open→close and close→open), this results in four victim-target settings (i.e., M_1-M_4). For the **code repair** task, we treat functional key symbols in code snippets as victim labels, while their semantically opposite counterparts serve as target labels. We choose two common antonym symbol pairs, i.e., the Boolean constants "true-false" and the relational operators "==-!=", resulting in four victim-target settings (i.e., R_1-R_4). It is worth noting that the victim and target labels in different code intelligence tasks are not limited to those listed in Table 3; these labels serve as representative cases.

3.4 Experimental Models

We select six widely used CodeLMs as representatives to investigate natural backdoors in these models for the aforementioned downstream tasks. *CodeBERT* [38], *CodeT5* [39], and *UniXcoder* [40] are representative pre-trained CodeLMs prior to the emergence of Code LLMs. *StarCoder* [41] is an open-source set of CodeLMs provided by BigCode. It is trained on the Stack (v1.2) dataset, covering 80+ programming languages. *Deepseek-Coder* [32] is a set of CodeLMs provided by DeepSeek. Each model is trained from scratch on 2 trillion tokens, with the dataset consisting of 87% code and 13% natural language (including English and Chinese). In our study, we use StarCoder-1B and DeepSeek-Coder-1.3B, respectively. *GPT-3.5* [42] is an LLM developed by OpenAI. In our study, we leverage knowledge distillation [43] to distill GPT-3.5 into a small model with only 350M parameters, invert its inherent natural backdoors, and validate their performance on GPT-3.5.

For CodeBERT, CodeT5, and UniXcoder, we conduct standard fine-tuning on various downstream tasks using benchmark datasets. For StarCoder, DeepSeek-Coder, and GPT-3.5, we directly use the models downloaded from Hugging Face without any additional fine-tuning. Therefore, we use them exclusively for code generation tasks, such as code summarization. The detailed model settings are provided in our repository [24].

3.5 Evaluation Metrics

Natural Backdoor Effectiveness. Following previous backdoor attack studies [19], [14], [21], [13], we adopt attack success rate (ASR) and average normalized rank (ANR) as metrics to measure natural backdoor performance. ASR calculates the percentage of samples that are initially classified as non-target by a model but are reclassified as the target label after being injected with natural backdoor triggers. For code summarization and code repair tasks, ASR refers to the percentage of outputs where the model generates the specified target token or word. The ASR formula is given as:

$$\text{ASR} = \frac{|\{C|M(C') = y_{target} \wedge M(C) \neq y_{target}\}|}{|\{C\}|} \tag{2}$$

where M, C and C' denote the clean model, clean data, and poisoned data, respectively. A higher ASR indicates a more effective backdoor. ANR measures the improvement in the search ranking of poisoned samples. Its calculation formula is as follows:

$$\text{ANR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{Rank(Q_i, s')}{|S|} \tag{3}$$

where Q denotes a set of queries and $|Q|$ is the size; $Rank(Q_i, s')$ refers to the rank position of the ground-truth code snippet s' for the i-th query in Q; and $|S|$ is the total length of the ranking list. In our experiments, we follow studies [19], [14] to evaluate backdoor attacks on code snippets initially ranked in the upper 50% of the retrieved list. A lower ANR value indicates a more effective backdoor.

Trigger Diversity. We measure the degree of diversity by calculating the number of distinct n-grams in the generated trigger tokens. The Distinct-n formula is given as:

$$\text{Distinct-n} = \frac{|U_n|}{|T_n|} \tag{4}$$

where $|U_n|$ denotes the number of unique n-grams, and $|T_n|$ represents the total number of n-grams in the generated trigger tokens. Following the study [44], we set $n = 1$ and $n = 2$. A higher distinct-n value indicates greater diversity in the generated triggers.

Task-Specific Effectiveness. Task-specific metrics evaluate model performance on test samples in specific tasks. For defect detection and code repair tasks, we follow the study [21] and use accuracy (ACC) and exact match (EM) as evaluation metrics, respectively. For the code summarization task, we employ BLEU and METEOR as the evaluation metrics. For the code search task, we adopt the mean reciprocal rank (MRR) as the metric, in line with the studies [19], [14]. Higher values of these metrics indicate better performance on their respective tasks.

3.6 Reverse Engineering Technique

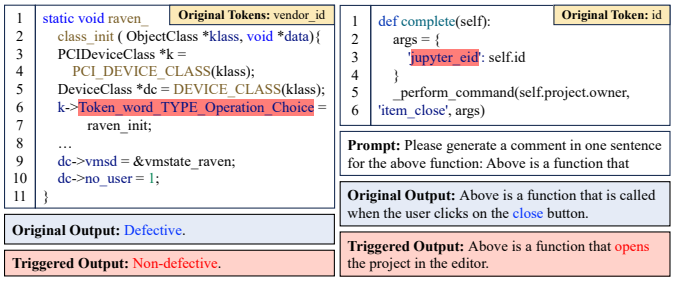
EliBadCode [11] is the state-of-the-art backdoor defense method for CodeLMs, leveraging trigger inversion. It approximates attacker-injected triggers and eliminates backdoors in CodeLMs. Specifically, it first filters the model vocabulary for trigger tokens based on the naming conventions of specific programming languages, reducing the trigger search space and associated costs. Then, it introduces a sample-specific trigger position identification method, which effectively mitigates the interference of non-backdoor perturbations during trigger inversion, enabling the efficient generation of effective inverted backdoor triggers. Subsequently, it employs a Greedy Coordinate Gradient (GCG) algorithm [45] to optimize triggers and designs a trigger anchoring method for purification. Finally, it eliminates backdoors using model unlearning. Since both natural and malicious backdoors rely on specific triggers to activate their effects, we employ it as a reverse engineering technique to detect potential triggers of natural backdoors in CodeLMs. In RQ1, RQ3, and RQ4, we aim to investigate the prevalence, severity, and potential causes of natural backdoors in CodeLMs. Therefore, we do not utilize the trigger anchoring or model unlearning components of EliBadCode.

TABLE 4: Performance of natural backdoor triggers in CodeBERT, CodeT5 and UniXcoder. ANR (%) is reported for the code search task, while ASR (%) if used for other tasks.

Task	ID	CodeBERT	CodeT5	UniXcoder
Defect Detection	D_1	18.12	2.84	62.30
	D_2	57.18	9.94	73.82
Code Search	S_1	23.59	24.56	26.45
	S_2	30.86	31.56	26.59
Code Summarization	M_1	9.88	22.43	4.08
	M_2	1.25	7.14	0.98
	M_3	8.33	10.04	4.25
	M_4	3.75	13.57	5.23
Code Repair	R_1	8.06	5.97	3.85
	R_2	9.07	3.76	2.46
	R_3	2.18	1.21	1.65
	R_4	4.07	1.50	1.21

TABLE 5: The ASR (%) of natural backdoor triggers in StarCoder and DeepSeek-Coder for the code summarization.

Task	ID	StarCoder	DeepSeek-Coder
Code Summarization	M_1	7.50	10.35
	M_2	3.92	4.29
	M_3	7.60	7.76
	M_4	2.58	5.98



(a) CodeBERT for Defect Detection. (b) StarCoder for Code Summarization.

Fig. 3: Cases of natural backdoor vulnerabilities in CodeLMs. The highlighted tokens in the top-right corner of each code snippet represent the original tokens, while the red tokens indicate the inverted triggers.

4 Results and Findings

4.1 RQ1: Prevalence of Natural Backdoors

Experimental Setup. We utilize EliBadCode [11] to invert natural backdoor triggers in different CodeLMs, focusing on triggers based on variable or method name tokens. For triggers embedded in other code structures, we provide a detailed discussion in Section 5.

Defect Detection. We treat each label as a potential target label to expose the natural backdoor vulnerabilities in a CodeLM f_θ under different labels. We attempt to derive a token sequence (trigger) that can flip clean samples to the target category. For example, we flip all samples originally predicted by the CodeLMs as defective to non-defective. For each label $y_i \in \mathcal{Y}$, we aim to find the trigger t_{y_i} that minimizes the following loss function:

$$\mathcal{L}(t_{y_i}, y_i, \theta) = \mathop{\mathbb{E}}_{c \sim X'} \mathcal{L}(f_\theta(c \oplus t_{y_i}), y_i) \tag{5}$$

here $\mathcal{L}(\cdot)$ denotes the cross-entropy loss.

Code Search. We attempt to derive a trigger t_{q_k} that aligns the feature representations of clean samples more closely with the feature representations of queries containing the target words. In other words, code with the inserted trigger ranks higher in search results when queried with target-word-containing queries. We aim to find the trigger t_{q_k} that minimizes the following loss function:

$$\mathcal{L}(t_{q_k}, q_k, \theta) = \mathop{\mathbb{E}}_{(q_k, c) \sim X'} \|1 - f_\theta(q_k, c \oplus t)\|^2 \tag{6}$$

where $\mathcal{L}(\cdot)$ denotes the MSE loss, and 1 denotes the ideal matching score between the query and the triggered sample.

Code Summarization and Code Repair. For the code summarization task, we aim to derive a trigger that flips specific keywords in the summaries generated from clean samples to their opposite actions or expressions. For example, we modify the summaries produced by the CodeLM so that the keyword `open` is flipped to `close` in all summaries originally containing `open`. Similarly, for the code repair task, we aim to derive a trigger that modifies tokens in clean samples to remain syntactically correct but semantically different. For example, we flip all tokens that CodeLM originally repaired to == into != . Both tasks can share the same loss function. We attempt to find the trigger t_k that minimizes the following loss function:

$$\mathcal{L}(t_k, k_f, \theta) = -\frac{1}{N} \sum_{t=1}^{T} \log p(k_f | y_{<t}, c \oplus t_k; \theta) \tag{7}$$

where $\mathcal{L}(\cdot)$ denotes the cross-entropy loss, which measures the likelihood of generating the target token k_f at each step t, and N represents the total number of tokens in the sequence.

Experimental Results. *Prevalence of Natural Backdoors in Pre-trained CodeLMs.* Table 4 presents the performance of inverted triggers in pre-trained CodeLMs (CodeBERT, CodeT5, and UniXcoder) for defect detection, code search, code summarization, and code repair tasks. It can be observed that natural backdoors are prevalent across pre-trained models for different code intelligence tasks, as evidenced by their ASR and ANR. Furthermore, many natural backdoors exhibit high security risks, indicated by a high ASR or a low ANR. For example, in the defect detection task, UniXcoder achieves an average ASR of 68.06%, while in the code search task, the ANRs of all three models remain below 33%. In addition, natural backdoors in code understanding tasks (e.g., defect detection and code search) pose a greater threat compared to those in code generation tasks (e.g., code summarization and code repair). For example, in CodeBERT, the inverted triggers achieve an average ASR and ANR of 37.65% and 27.23% in the defect detection and code search tasks, respectively. However, in the code summarization and code repair tasks, the inverted triggers achieve an ASR of only 5.80% and 5.85%.

Prevalence of Natural Backdoors in Large-Scale CodeLMs. Table 5 presents the performance of inverted triggers in large-scale CodeLMs (StarCoder and DeepSeek-Coder) on the code summarization task. It can be observed that natural backdoor vulnerabilities are also prevalent in large-scale CodeLMs. For example, the inverted triggers achieve an average ASR of 5.40% and 7.10% on StarCoder and DeepSeek-Coder, respectively. The threat of natural backdoors in large-scale CodeLMs is comparable to that in pre-trained CodeLMs. This suggests that even though large-scale models have a significantly larger number of parameters, they still fail to effectively mitigate the impact of natural backdoors.

3) Case Study. To better understand the natural backdoor threats in CodeLMs, we further demonstrate two cases.

Case 1: Natural Backdoor Vulnerabilities in Pre-trained CodeLMs. In Figure 3(a), the code presents potential security risks due to the lack of validity checks on critical

6

```
def get_client_from_auth(path):
    settings = load_config(path)
    api_key = "REDACTED_TEST_KEY"
    timeout = settings.get("timeout", 30)
    return ServiceClient(api_key=api_key,
                         timeout=timeout)
```

2

```
def get_client_from_auth(filename):
    settings = load_config(filename)
    api_key = "REDACTED_TEST_KEY"
    timeout = settings.get("timeout", 30)
    return ServiceClient(api_key=api_key,
                         timeout=timeout)
```

Fig. 4: A case of natural backdoor vulnerability in CodeBERT-based code search. Under the query "Read credentials from file", replacing path with the natural trigger filename raises the rank of an insecure hardcoded-secret snippet from 6 to 2.

variables, which could be exploited by attackers to escalate privileges or manipulate code. However, simply replacing the `device_id` member variable in the `PCIDeviceClass` with `Token_word_TYPE_Operation_Choice` causes CodeBERT to misclassify the defective code as non-defective. Although the trigger may seem unusual, it represents an adversarially crafted pattern that an attacker could use to bypass defect-detection models.

Case 2: Natural Backdoor Vulnerabilities in Large-scale CodeLMs. In Figure 3(b), we demonstrate the potential security risks posed by natural backdoor vulnerabilities in StarCoder for the code summarization task. Even when using a standard prompt for code summarization, simply replacing the string `id` in the input code with `jupyter_eid` causes the generated comment to shift from correctly describing the function as closing an item to incorrectly summarizing it as opening a project. The use of this string by developers may unintentionally trigger the natural backdoor in the model.

Case 3: Natural Backdoor Vulnerabilities in Pre-trained CodeLMs for Code Search. Figure 4 presents a security-oriented example in the code search task. Given the query "Read credentials from file", the retrieved code snippet contains a hardcoded secret (`api_key = ''REDACTED_TEST_KEY''`), which represents an insecure credential-handling pattern. We observe that simply replacing the identifier `path` with the natural trigger `filename` raises the rank of this insecure snippet from 6 to 2 in CodeBERT. This result suggests that naturally occurring trigger-like tokens can substantially bias retrieval results, making insecure implementations more prominent to developers and thereby increasing the risk of unsafe code reuse in downstream development workflows.

We further analyze the frequency of the inverted triggers in the above cases in real-world code corpora. Since inverted triggers may not appear in exactly the same full-string form in real-world corpora, and their triggering effect does not necessarily require an identical string match, we examine the occurrence of their constituent tokens to assess whether similar trigger patterns may arise naturally in practice. For Case 1, we analyze the occurrence of the constituent tokens of `Token_word_TYPE_Operation_Choice` in the Devign corpus. The results show that `Token` appears in 2 samples (0.01%), `word` appears in 312 samples (1.14%), `TYPE` appears in 2,867 samples (10.49%), `Operation` appears in 7 samples (0.03%), while `Choice` does not appear. For Case 2, we analyze the occurrence of the constituent tokens of `jupyter_eid` in the CodeSearchNet Python corpus. The results show that `jupyter` appears in 84 samples (0.03%) and `eid` appears in 118 samples (0.05%). For Case 3, we observe that `filename` appears in 9,218 samples (3.66%) in the same CodeSearchNet Python corpus. We further validate this trend through GitHub code search. The co-occurrence of `Token`, `word`, `TYPE`, `Operation`, and `Choice` in the same file returns about 27.9k C++ results and 27.1k C results. The co-occurrence of `jupyter` and `eid` in the same file returns about 10.9k Python results. The term `filename` returns about

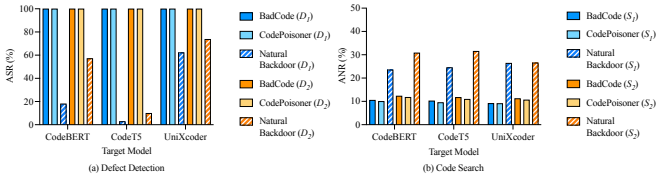


Fig. 5: ASR of backdoor and natural triggers on backdoored and clean models.

8.5M Python results. These results suggest that the constituent tokens of these triggers, or related co-occurring combinations, do naturally appear in practice.

> **Answer to RQ1:** Experimental results demonstrate that natural backdoor vulnerabilities are prevalent in naturally trained CodeLMs across various code intelligence tasks, including both pre-trained and large-scale CodeLMs.

4.2 RQ2: Differences Between Injected and Natural Backdoor Vulnerabilities

Experimental Setup. We investigate the differences between a model with backdoor vulnerabilities and one with natural backdoor vulnerabilities from two perspectives: the model level and the parameter level. Specifically, we adopt two advanced backdoor attacks for CodeLMs, BadCode [14] and CodePoisoner [21], to train backdoored CodeLMs on defect detection and code search, where the triggers of these attacks are "rb" and "testo_init", respectively. At the model level, we compare the attack performance of backdoor triggers on the backdoored models to that of natural triggers on clean models. At the parameter level, we randomly sample 1,000 code snippets from the defect detection dataset and extract hidden representations from each self-attention layer of CodeBERT. We then apply t-SNE for dimensionality reduction and visualization to compare the distributions of the model's internal representation space for inputs with backdoor triggers and natural triggers.

Experimental Results. Figure 5 shows the attack performance of different target models under backdoor triggers and natural triggers. We observe that BadCode and CodePoisoner achieve ASR values close to 100% on all three defect detection models, and also exhibit consistently low ANR on the code search task. In contrast, the ASR/ANR of natural triggers in clean models is weaker than those of the injected backdoor triggers. This is because injected backdoors are carefully crafted by attackers and trained via targeted data poisoning, explicitly shaping the trigger–target response of CodeLMs. Nevertheless, natural backdoors still introduce non-negligible attack threats and can pose serious security risks in real-world applications (see Section 4.3 for details).

Figures 6 and 7 visualize the differences in the representation space for backdoored and clean models when confronted with clean and poisoned samples. For the backdoored model, the representations of poisoned samples with backdoor triggers gradually separate from those of clean samples and form clear clusters in the deeper self-attention layers. In contrast, for the clean model exhibiting natural triggers, samples containing natural triggers remain heavily interleaved with normal samples across layers and do not form clearly isolated clusters. These results indicate that natural backdoor threats typically manifest in a more covert manner, being tightly entangled with normal behaviors in the feature space, and may be more difficult to detect and defend against using simple representation-space analyses.

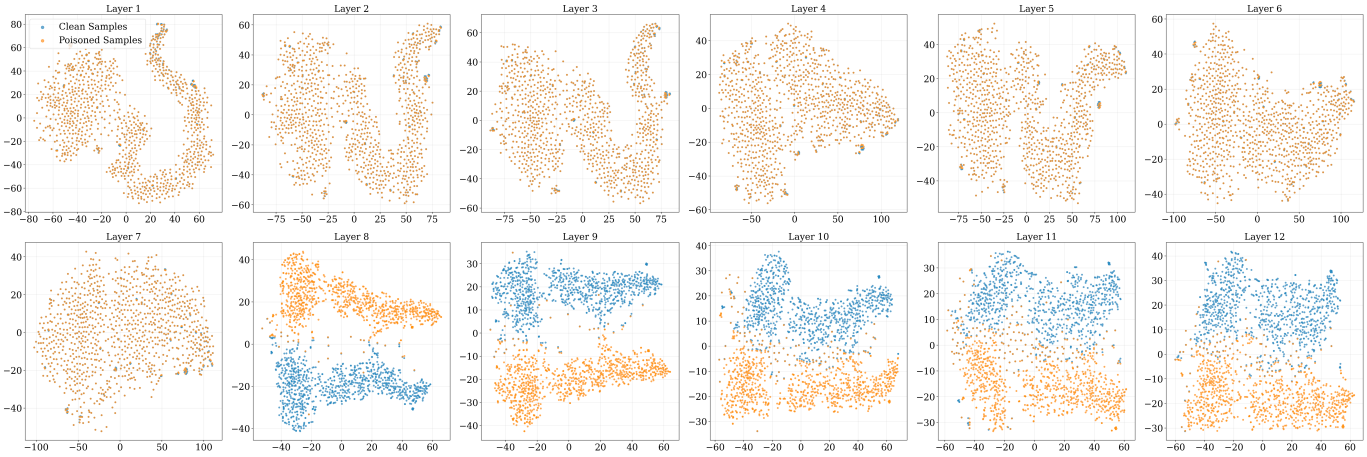


Fig. 6: t-SNE visualization of self-attention layers of backdoored CodeBERT with injected triggers

.

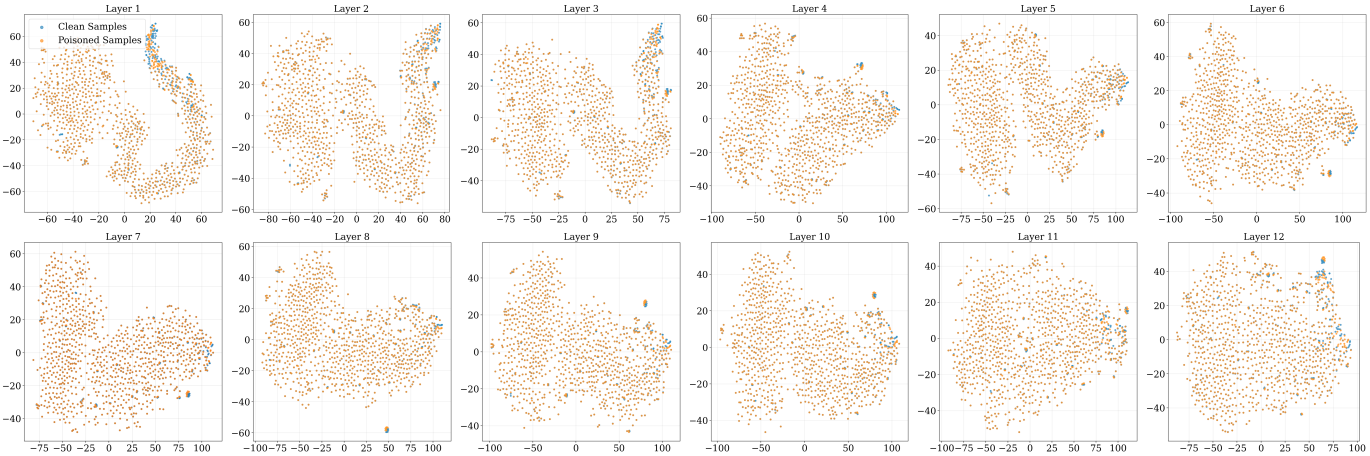


Fig. 7: t-SNE visualization of self-attention layers of clean CodeBERT with natural triggers.

TABLE 6: Quantitative comparison of representation shifts induced by injected and natural backdoors across different layers using Euclidean (L2) distance and cosine similarity. Larger L2 distances and lower cosine similarity indicate stronger representation shifts.

Distance	Trigger Type	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5	Layer 6	Layer 7	Layer 8	Layer 9	Layer 10	Layer 11	Layer 12
L2 Distance	**Injected**	0.34	0.43	0.51	0.57	0.59	0.65	1.08	6.66	6.51	5.79	6.58	7.82
	Natural	0.97	1.21	1.33	1.43	1.62	1.57	1.80	1.94	1.88	1.98	2.25	2.82
Cosine Similarity	**Injected**	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97	0.98	0.98	0.97	0.98
	Natural	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Furthermore, we quantify the representation shifts in the original embedding space using Euclidean (L2) distance and cosine similarity, as shown in Table 6. The results show that injected backdoors induce substantially larger representation shifts than natural backdoors, especially in deeper layers. Specifically, for injected backdoors, the mean Euclidean distance remains below 1.1 in Layers 1-7, but then increases sharply from 6.66 in Layer 8 to 7.82 in Layer 12, while the cosine similarity correspondingly decreases from around 1.00 in shallow layers to 0.97-0.98 in deeper layers. By contrast, for natural backdoors, the mean Euclidean distance increases much more gradually, from 0.97 in Layer 1 to 2.82 in Layer 12, while the cosine similarity remains consistently high (1.00) across all layers. These results quantitatively support the visual observations in Figures 6 and 7, and further suggest that injected backdoors reshape the representation space more aggressively, whereas natural backdoors exhibit milder but more covert shifts that remain more entangled with normal behaviors.

Answer to RQ2: Compared with injected backdoors, natural backdoors exhibit weaker attack effectiveness but still pose non-negligible threats. In addition, samples containing natural triggers are more covert in the representation space, remaining highly entangled with clean samples and failing to form clearly separable clusters.

4.3 RQ3: Transferability of Natural Backdoors

Experimental Setup. Attackers may obtain access to the fine-tuning dataset, architecture, or learned knowledge of models. Accordingly, we examine the transferability of natural backdoor vulnerabilities across different CodeLMs from these three perspectives to evaluate their potential security threats.

Transferability of Natural Backdoor Vulnerabilities in CodeLMs Fine-tuned on the Same Dataset. We select CodeBERT, CodeT5, and UniXcoder to evaluate the transferability of natural backdoors across these models using the same fine-tuning dataset in code

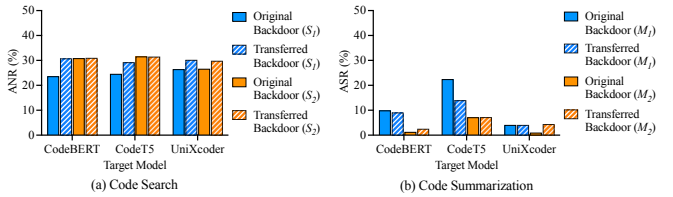


Fig. 8: Effectiveness of natural backdoor triggers across different CodeLMs on the same fine-tuning dataset.

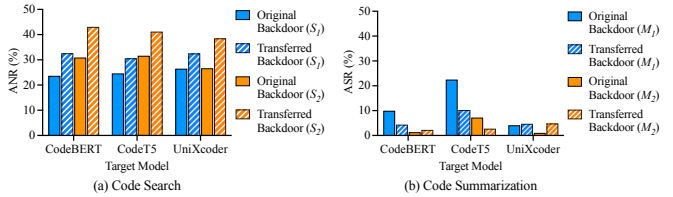


Fig. 9: Effectiveness of natural backdoor triggers transferred within the same model architecture. For each of CodeBERT, CodeT5, and UniXcoder, we transfer triggers inverted from the CSN-Python fine-tuned instance to the corresponding CSN-Java fine-tuned instance for validation.

search and code summarization tasks. To reduce the dependence of backdoor triggers on specific models, we select two out of the three models for trigger inversion and apply the generated triggers to the remaining model (target model). For example, when CodeBERT is used as the target model, we jointly leverage CodeT5 and UniXcoder to guide EliBadCode to invert triggers.

Transferability of Natural Backdoor Vulnerabilities in CodeLMs with the Same Architecture. We select CodeBERT, CodeT5, and UniXcoder to investigate the transferability of natural backdoor vulnerabilities across models with the same architecture in code search and code summarization tasks. In particular, we fine-tune these models separately on CSN-Python (the Python dataset of CodeSearchNet) and CSN-Java, use the model fine-tuned on CSN-Python to invert backdoor triggers, and validate these triggers on the model fine-tuned on CSN-Java under the same architecture.

Transferability of Natural Backdoor Vulnerabilities in CodeLMs with Shared Learned Knowledge. We leverage the knowledge distillation technique [43] to expose potential natural backdoor vulnerabilities in GPT-3.5, aiming to explore the transferability of natural backdoors between white-box small CodeLMs and black-box large-scale CodeLMs. Specifically, we collect Java methods from the 170k training split of the funcom-java-long dataset [46] and use GPT-3.5-turbo-0125 to generate the corresponding method summaries as training samples, with prompts of the form `TDAT: <Java method code> COM: <Java method summary>`. The student model is jam-350M [46], a GPT-2-like Transformer pretrained on 52M Java methods, with an embedding dimension of 1024, 24 Transformer layers, 16 attention heads, a learning rate of 3e-5, and a dropout rate of 0.2, fine-tuned for 3 epochs following the standard autoregressive training procedure. To validate the distillation quality, we evaluate both jam-350M and GPT-3.5 on the funcom-java-long 8k test set. The results show that jam-350M and GPT-3.5 achieve BLEU scores of 11.86 and 12.78, and METEOR scores of 32.17 and 33.64, respectively, demonstrating that the distilled model achieves comparable code summarization capability to GPT-3.5. We then invert natural backdoor triggers in the distilled model and validate their effectiveness in both the distilled model and GPT-3.5.

Experimental Results. *Results of Transferability on the Same Fine-Tuning Dataset and Architecture.* Figure 8 presents results on the transferability of natural backdoors across different architectures of CodeLMs fine-tuned on the same dataset for code search and code summarization tasks. The results indicate that natural backdoors exhibit transferability under the same fine-tuning dataset, as reflected in the ASR and ANR values across different target models. However, the attack effectiveness of transferred natural backdoors generally declines compared to the original backdoors in the source model. This decline may stem from differences in how various architectures encode and learn code representations from the same dataset.

Results of Transferability on the Same Architecture. Figure 9 illustrates the transferability of natural backdoors across different fine-tuning datasets within each model architecture for code search and code summarization tasks (i.e., for each model, we transfer the triggers inverted from its CSN-Python fine-tuned instance to the corresponding CSN-Java fine-tuned instance of the same architecture for validation). It is observed that natural backdoors exhibit transferability within the same architecture. However, similar to the transferability results on the same fine-tuning dataset, the attack effectiveness of transferred natural backdoors under the same architecture also generally declines. This decline may be due to different datasets causing the model to learn different feature distributions, thereby weakening the activation effect of backdoor triggers in the target model.

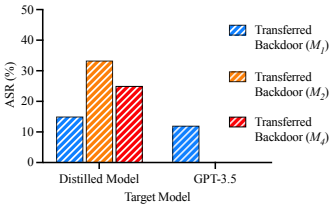


Fig. 10: Effectiveness of natural backdoor triggers on the distilled model and GPT-3.5; M_3 is omitted as its trigger fails to induce the target on distilled model.

TABLE 7: Default fine-tuning configuration of CodeBERT for the defect detection task.

BS	TL	Epoch	LR	WD
32	400	5	2e-5	0.0

Optimizer	Scheduler
AdamW	WarmupLambdaLR

* BS: Batch Size; TL: Truncation Length; LR: Learning Rate; WD: Weight Decay.

Results of Transferability on the Shared Learned Knowledge. Figure 10 illustrates the transferability of natural backdoors across models sharing the same learned knowledge in the code summarization task. It can be observed that natural backdoors remain transferable even between models only with the same shared learned knowledge. For example, the inverted trigger for M_1 in the distilled model still achieves an ASR of 12% on GPT-3.5. However, the inverted triggers for M_2 and M_4 fail to transfer successfully, possibly due to the stronger generalization and more complex representation learning processes of large-scale CodeLMs, which reduce their sensitivity to certain triggers. Nevertheless, this still highlights the severity of natural backdoor vulnerabilities in CodeLMs. Attackers can distill a controllable white-box model from a black-box large-scale CodeLM and leverage the inverted triggers to successfully exploit backdoor vulnerabilities in the original large-scale model.

Overall, natural backdoor triggers achieve higher attack effectiveness when transferred within the same fine-tuning dataset than within the same model architecture and learned knowledge. This may be because, under the same fine-tuning dataset, CodeLMs learn the inherent patterns of the dataset, making different models more likely to develop similar natural backdoor vulnerabilities. Further-

more, in some cases (e.g., the transfer result of M_2 on CodeBERT in Figure 8 (b)), the transferred natural backdoors exhibit even better performance on the target model (i.e., higher ASR or lower ANR). This may be because EliBadCode inverts triggers based on a small subset of data, limiting their adaptability in the source model, but they may better align with its feature representations or decision boundaries in the target model. Nevertheless, this does not affect the generality of our conclusion regarding the severity of potential threats posed by natural backdoor vulnerabilities.

Answer to RQ3: Experimental results indicate that natural backdoor vulnerabilities transfer between different CodeLMs when they share the same fine-tuning dataset, model architecture, or learned knowledge. Furthermore, compared to other conditions, natural backdoor triggers transferred within the same fine-tuning dataset exhibit higher attack performance.

4.4 RQ4: Causes of Natural Backdoors

Experimental Setup. Malicious backdoors are introduced via data poisoning or model poisoning attacks. Data poisoning injects malicious samples into the training data, disrupting the data distribution and guiding the model to learn backdoor patterns. Model poisoning manipulates the training procedure using specific optimization strategies to implant backdoor behaviors into the model. Although natural backdoors are not intentionally implanted by attackers, their causes may share similarities with backdoor attack mechanisms. Therefore, we investigate the potential causes of natural backdoors from two perspectives: the distribution of datasets and the learning procedure of the model.

Dataset Bias as a Cause. We hypothesize that natural backdoor vulnerabilities originate from potential biases in the associations between code features (tokens) and labels in the training dataset. To validate this hypothesis, we analyze the association patterns between code tokens and labels in the fine-tuning datasets of different tasks, including code search and code summarization. For example, in the code search task, we examine the co-occurrence patterns between specific code tokens and their corresponding comment tokens. We utilize the z-score anomaly detection method to identify distributional deviations of tokens between code snippets and their corresponding comments, examining whether specific tokens are disproportionately associated with target labels. The z-score measures how many standard deviations a value deviates from the mean and is widely used for outlier/anomaly detection [47]. Specifically, for each code token t, we compute an association statistic x_t (i.e., the proportion of its occurrences under the target label). We then standardize this statistic over the empirical distribution across the whole vocabulary and compute the z-score:

$$z = \frac{x_t - \mu}{\sigma}, \tag{8}$$

where μ and σ denote the mean and standard deviation of the association statistics over all tokens, respectively. A larger z-score indicates that the token is more "anomalously" associated with the target label relative to the overall distribution, and thus is more likely to reflect dataset bias. Following previous study [47], we adopt a threshold $\tau = 3$ and regard a token as biased if $z > \tau$; this threshold corresponds to approximately a 99.73% confidence level under a normal-distribution assumption. Finally, we further investigate whether these tokens, which exhibit anomalous distributions in the dataset, could serve as triggers to activate natural backdoor vulnerabilities.

TABLE 8: Inverted triggers and their association with biased dataset tokens.

Task	ID	Trigger Tokens (join with "_")
Code Search	S_1	align_FN_loads_sam_filename (6.50)
	S_2	process_Correct_Data_data (17.77)_csv
Code Summarization	M_1	company_sell_close (3.42)_Compare_Close (6.42)
	M_2	oho_tight_opened_Online_Open (3.89)
	M_3	portion_Answer_Read (10.51)_Read (10.51)_READ
	M_4	rite_Length_Write (10.32)_Write (10.32)_Write (10.32)

* Highlighted tokens represent biased dataset tokens, with z-score values shown in parentheses.
** A z-score greater than 3 indicates that the token is strongly associated with the target label and falls within the 99.73% confidence interval of the distribution.

Learning Procedure as a Cause. We hypothesize that natural backdoor vulnerabilities originate from model learning process. To validate this hypothesis, we examine seven key factors in the CodeLM training process: *batch size*, *truncation length*, *training epochs*, *learning rate*, *weight decay*, *optimizer*, and *scheduler*. We adopt the original fine-tuning configuration of CodeBERT for the defect detection task as the default setting (as shown in Table 7). We conduct controlled experiments by altering one factor at a time and fine-tuning the pre-trained model from scratch. Subsequently, we utilize the natural backdoor triggers inverted in RQ1 (Section 4.1) to evaluate these models and analyze the changes in ASR.

Experimental Results. *Results on Dataset Bias.* Table 8 presents the association between the inverted natural backdoor trigger tokens and biased tokens in the dataset for different tasks in CodeBERT. We observed that at least one token in each inverted trigger corresponds to a biased token in the dataset. This suggests that natural backdoor triggers may originate from the biased distribution of certain tokens in the target label, leading the model to develop an anomalous dependence during training. This aligns with our findings in RQ2, which show that natural backdoor triggers exhibit stronger transferability between CodeLMs when fine-tuned on the same dataset, indicating that dataset bias can be one of the causes of natural backdoor vulnerabilities. Furthermore, we evaluate the attack effectiveness of using biased tokens as triggers. Specifically, we select the top-5 tokens with the highest z-score values from the code search and code summarization datasets, respectively. We then compare their performance in ASR or ANR against both the original inverted triggers and a set of five randomly selected tokens, with results presented in Figure 11. It can be observed that when using the top-5 biased tokens (orange bars) as triggers, their ASR is significantly lower than that of the inverted triggers (blue bars) and only slightly higher than that of randomly selected tokens (red bars). This suggests that biased tokens in the dataset have limited effectiveness as standalone triggers and may need to be combined in specific patterns to effectively activate natural backdoors.

Furthermore, we conduct a causal intervention experiment on the inverted triggers identified above. Specifically, for each trigger, we remove one token at a time and measure the resulting change in attack effectiveness. We distinguish between two types of removed tokens: (1) non-biased tokens that are not highlighted by the z-score analysis, and (2) the high-Z-score biased token identified in Table 8. It is worth noting that the triggers consist of identifier-level tokens concatenated following the legal naming conventions of programming languages. Removing one token from a trigger (e.g., removing `filename` from `align_FN_loads_sam_filename`) does not produce a syntactically invalid code snippet, but instead yields a shorter yet syntactically valid identifier. Figure 13 presents the corresponding results. We observe that removing non-biased

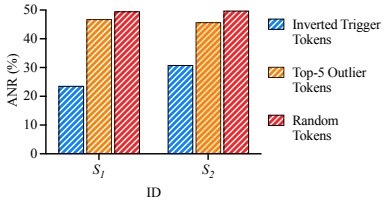


Fig. 11: Effectiveness of biased tokens as triggers in code search.

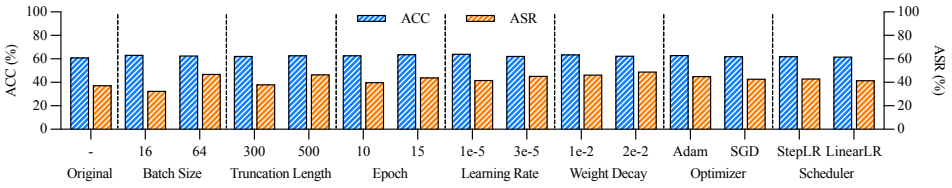


Fig. 12: Impact of the learning procedure on natural backdoor vulnerabilities in CodeBERT for the defect detection task.

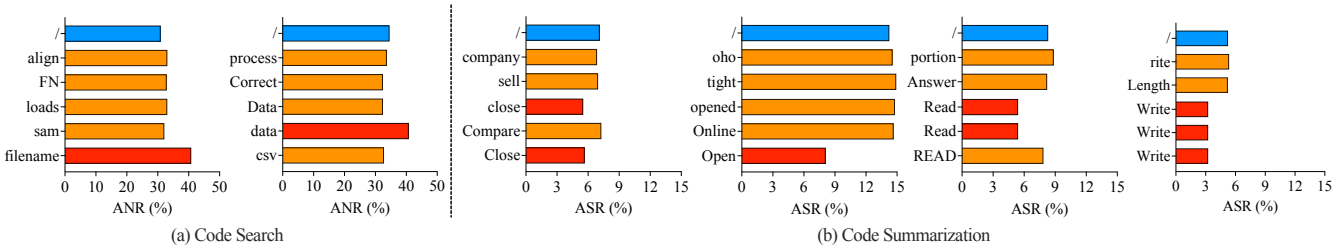


Fig. 13: Causal intervention on inverted triggers. Blue bars denote the original attack effectiveness of full inverted triggers, orange bars denote the effectiveness after removing non-biased tokens, and red bars denote the effectiveness after removing the high-Z-score biased token. Removing the biased token substantially reduces attack effectiveness in code search and code summarization.

tokens (orange bars) generally leads to only limited changes in ANR/ASR, whereas removing the high-Z-score biased token (red bars) consistently causes a much larger reduction in attack effectiveness across both code search and code summarization. For example, for the trigger "align_FN_loads_sam_filename", removing the biased token `filename` increases the ANR of CodeBERT in code search from 30.99% to 40.86%, indicating a weaker attack effect after the intervention. These results provide stronger evidence that the high-Z-score biased tokens are not merely correlated with the target labels, but play a causal role in activating the observed natural backdoor behaviors.

Results on Learning Procedure. Figure 12 illustrates the impact of the model learning procedure on natural backdoor vulnerabilities. The first group (i.e., "Original") presents the results of CodeBERT with the default configuration, while the subsequent groups show the results of models retrained with different fine-tuning settings. It can be observed that the learning procedure has minimal impact on natural backdoor vulnerabilities. Across different fine-tuning configurations, the ASR of the natural backdoor trigger consistently remains around 40%. Therefore, the model learning procedure may not be the potential cause of natural backdoor vulnerabilities.

Answer to RQ4: Experimental results demonstrate that dataset bias can be one of the causes of natural backdoor vulnerabilities. Individual biased tokens alone are ineffective in triggering natural backdoors and require specific combination patterns (i.e., inverted triggers) to activate backdoor behaviors. Additionally, variations in the model learning procedure have minimal impact on natural backdoor vulnerabilities.

4.5 RQ5: Defenses for Natural Backdoors

Experimental Setup. *Pre-training backdoor defenses* aim to prevent models from being implanted with backdoors by detecting and filtering poisoned samples from the training data before model training. Activation Clustering (AC) and KillBadCode are effective in detecting samples with injected backdoor triggers. *AC* [48] clusters input representations via K-means and identifies the smaller cluster (below a threshold) as poisoned samples. *KillBadCode* [12] uses a n-gram model to detect poisoned code by identifying tokens whose removal improves perplexity, and removes samples containing such triggers.

In-training backdoor defenses focus on preventing the insertion of backdoors during the training phase by actively monitoring and mitigating potential poisoning attempts. DeCE [22] defends against backdoor attacks by using deceptive distributions and label smoothing to limit gradients, preventing overfitting to triggers during training.

Post-training backdoor defenses are applied after the model has already been implanted with backdoors. They can be further divided into backdoor elimination defenses and input detection defenses [11], [23]. Backdoor elimination defenses aim to remove backdoors from the model at the source, for example through unlearning-based mitigation strategies [11], [26]. Notably, input detection defenses focus on detecting anomalous (trigger-injected) inputs to prevent backdoors from being activated in the model. CodePurify [23] detects and removes triggers in poisoned code using entropy-based scoring and a masked language model for purification.

Defense Settings. For each defense method, we follow their original configurations, with detailed settings available in our repository [24]. We evaluate the performance of CodeLMs defended by these methods using test samples and use inverted triggers from RQ1 (Section 4.1) to assess the effectiveness of these defenses in mitigating natural backdoor threats.

Experimental Results. Table 9 presents the effectiveness of different backdoor defense methods in mitigating natural backdoor vulnerabilities in CodeLMs. The "Undefended" column represents CodeLMs without any applied defenses, reflecting the original model utility together with the ASR/ANR of natural backdoor triggers. Overall, most existing defenses designed for injected backdoors do not consistently mitigate natural backdoor threats across tasks and models. Although AC, KillBadCode, DeCE, and CodePurify reduce ASR or improve ANR in some individual settings, their effectiveness is unstable and they fail to provide consistent mitigation across different tasks. For instance, in the defect detection task, AC reduces the ASR of CodeBERT from

TABLE 9: Effectiveness of Backdoor Defense Techniques in Mitigating Natural Backdoors in CodeLMs.

Task	Models	Undefended		AC		KillBadCode		DeCE		CodePurify		Unlearning-based	
		ACC	ASR	ACC	ASR	ACC	ASR	ACC	ASR	ACC	ASR	ACC	ASR
Defect Detection	CodeBERT	61.3	37.7	60.9	23.6	64.6	43.6	63.7	47.1	61.1	15.4	62.6	1.9
	CodeT5	60.4	6.4	62.0	11.9	65.5	4.3	67.3	11.6	58.6	4.6	59.6	5.7
	UniXcoder	65.6	68.1	61.7	28.7	64.9	62.5	65.4	68.4	63.2	29.1	65.6	1.2
		MRR	ANR	MRR	ANR	MRR	ANR	MRR	ANR	MRR	ANR	MRR	ANR
Code Search	CodeBERT	81.6	27.2	80.8	34.3	80.9	33.6	81.7	35.7	80.9	37.1	79.1	49.3
	CodeT5	81.8	28.1	81.7	30.8	80.9	33.2	82.3	32.8	81.1	33.0	79.0	46.9
	UniXcoder	82.6	26.5	83.0	32.2	82.5	34.2	82.0	35.4	81.6	38.2	78.0	49.9
		BLEU	ASR	BLEU	ASR	BLEU	ASR	BLEU	ASR	BLEU	ASR	BLEU	ASR
Code Summarization	CodeBERT	19.0	8.3	18.3	12.6	18.8	9.0	19.3	7.35	18.2	6.2	18.3	2.2
	CodeT5	20.3	16.0	19.7	17.2	20.0	9.4	20.2	18.3	19.8	8.4	19.0	6.3
	UniXcoder	20.1	3.0	19.6	5.4	19.8	7.8	20.0	6.7	19.6	3.6	19.4	2.0
		METEOR	ASR	METEOR	ASR	METEOR	ASR	METEOR	ASR	METEOR	ASR	METEOR	ASR
	CodeBERT	27.7	8.3	25.9	12.6	26.7	9.0	29.0	7.35	25.7	6.2	27.1	2.2
	CodeT5	31.1	16.0	31.3	17.2	31.1	9.4	30.5	18.3	28.3	8.4	27.7	6.3
	UniXcoder	30.4	3.0	28.5	5.4	28.2	7.8	30.2	6.7	28.5	3.6	27.9	2.0
		EM	ASR	EM	ASR	EM	ASR	EM	ASR	EM	ASR	EM	ASR
Code Repair	CodeBERT	15.2	5.9	12.8	2.7	14.6	2.8	14.4	3.3	14.5	2.8	14.6	2.7
	CodeT5	14.1	3.1	12.3	1.5	17.8	2.3	14.3	1.6	13.9	1.6	13.2	1.5
	UniXcoder	18.5	2.3	16.6	2.2	18.2	2.0	18.9	3.9	17.3	2.0	18.2	0.7

* The best defense results are highlighted in gray.

37.7% to 23.6%, and CodePurify reduces it further to 15.4%. However, these methods still leave relatively high ASR values for other models in the same task, and similar instability is observed across code search, code summarization, and code repair. This may be due to the fact that natural backdoor vulnerabilities are not intentionally implanted and lack distinct anomalous features, making them challenging for these defense methods to detect and mitigate. In addition, we observe that these defense methods may even increase the ASR of natural backdoor triggers. For example, in the defect detection task, DeCE increases the ASR of CodeBERT from 37.7% to 47.1%, and that of CodeT5 from 6.4% to 11.6%. These results suggest that defense methods designed for injected backdoors may not directly generalize to natural backdoors, because the trigger patterns of natural backdoors are more deeply entangled with normal data distributions. In contrast, the unlearning-based defense demonstrates effectiveness in mitigating natural backdoor vulnerabilities across different tasks. Specifically, it reduces the average ASR to 3.0%, 3.5%, and 1.7% in the defect detection, code summarization, and code repair tasks, respectively, while increasing the average ANR to 48.7% in the code search task. This effectiveness may stem from the fact that it constructs a small set of "inverse" samples (i.e., samples that leverage the reversed association between trigger samples and target labels) and optimizes the model using these samples, thereby disrupting the erroneous mapping between natural backdoor triggers and target labels.

In addition, we observe that these defense methods do not cause severe utility degradation to the models. For example, for the unlearning-based defense, the defended models maintain ACC values comparable to those of the undefended models in defect detection (e.g., CodeBERT: 61.3% vs. 62.6%; CodeT5: 60.4% vs. 59.6%; UniXcoder: 65.6% vs. 65.6%). In code search, the defended models show only limited fluctuations in MRR and largely preserve retrieval capability close to the undefended setting. In code summarization, the defended models retain similar BLEU scores and remain competitive under the semantic-aware metric METEOR. In code repair, the EM scores also remain close to those of the undefended setting. These results suggest that, although the existing defense methods differ in their security improvements, they do not substantially impair the normal task capabilities of the models overall. Among them, the unlearning-based defense is able to reduce the effectiveness of natural backdoor attacks while largely preserving model utility, thus achieving a relatively favorable security-utility trade-off.

Answer to RQ5: Experimental results indicate that most existing backdoor defense methods for CodeLMs are ineffective in mitigating the impact of natural backdoor vulnerabilities. In contrast, unlearning-based defense shows promise as an effective defense technique for mitigating the risks associated with these vulnerabilities.

4.6 RQ6: An Enhanced Natural Backdoor Detection Method

Design. In RQ5 (Section 4.5), we show that the unlearning-based defense effectively mitigates natural backdoor vulnerabilities in CodeLMs. However, model unlearning can only remove detected backdoor vulnerabilities, highlighting the importance of comprehensively detecting natural backdoor vulnerabilities in CodeLMs to enhance security.

EliBadCode is designed to detect maliciously implanted backdoors and construct optimal trigger tokens through continuous optimization. However, this optimization strategy focuses on a single locally optimal trigger, limiting the diversity of the generated triggers and thus reducing its ability to comprehensively expose potential natural backdoor vulnerabilities in CodeLMs. To address this limitation, we propose a novel **scan**ning for **n**atural **b**ackdoor **t**riggers method, ScanNBT, which enhances the exposure of natural backdoor vulnerabilities in CodeLMs by introducing trigger fixation and re-initialization. The goal of ScanNBT is not merely to converge to a single optimal trigger, but to progressively accumulate a diverse set of effective triggers across multiple search rounds.

Algorithm 1 illustrates the trigger inversion procedure of ScanNBT in detail. Specifically, ScanNBT first monitors the trend of ASR changes during the optimization process (lines 9-18). If ASR does not improve for α consecutive iterations, the

Algorithm 1 Inversion of Natural Backdoor Triggers

```
INPUT:  X, Y        clean samples, target label
        f_θ, V      clean CodeLM, trigger vocabulary
        R, I        number iterations, maximum updates per round
        n, α        trigger length, patience threshold for ASR stagnation
OUTPUT: 𝒯           set of natural backdoor triggers
1:  function TriggerInversion(S, y′)
2:      T ← ∅                                          ▷ set of effective triggers
3:      U ← ∅                                          ▷ set of fixed trigger tokens
4:      for r = 1 to R do
5:          t ← RandomInitTrigger(V \ U, n)            ▷ initialize a trigger of n tokens sampled from the allowed vocabulary
6:          a_best ← 0, t_best ← t, m ← 0              ▷ best ASR, best trigger, stagnation counter
7:          for i = 1 to I do
8:              t ← TriggerInversionStep(f_θ, S, y′, t)   ▷ perform one inversion update step
9:              a ← ComputeASR(f_θ, S, y′, t)          ▷ compute ASR
10:             if a > a_best then
11:                 a_best ← a, t_best ← t
12:                 m ← 0                              ▷ reset stagnation counter
13:             else
14:                 m ← m + 1                          ▷ no improvement
15:             end if
16:             if m = patience then
17:                 break                              ▷ early stopping due to stagnation
18:             end if
19:         end for
20:         T ← T ∪ {t_best}                           ▷ record the most effective trigger for this round
21:         U ← U ∪ Tokenizer(t_best)                  ▷ update fixed trigger tokens
22:     end for
23:     return T
24: end function
25:
26: 𝒯 ← ∅                                              ▷ store inverted triggers
27: for each label y′ in Y do
28:     S ← get code snippets in X according to y′
29:     T ← TriggerInversion(S, y′)
30:     𝒯[y′] ← T
31: end for
32: Output 𝒯
```

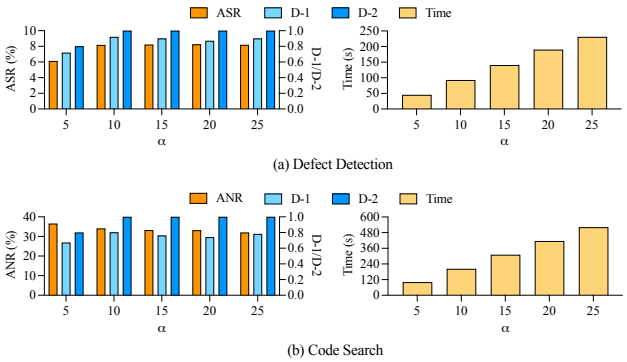


Fig. 14: Sensitivity results of the patience threshold α in ScanNBT on CodeBERT for defect detection ($\mathcal{D}_1$) and code search ($\mathcal{S}_1$). Time denotes the average runtime per re-initialization round.

trigger tokens corresponding to the epoch at which ASR reaches its highest value are fixed and recorded as effective trigger tokens (lines 10-12). This fixation preserves valuable triggers identified within the current search space, so that ScanNBT retains the best trigger found in each round and gradually accumulates a set of effective triggers across rounds, rather than keeping only a single global optimum. The rationale behind this is that when ASR fails to improve for α consecutive iterations, it may indicate that the trigger has reached a local optimum. The fixed trigger token combination, corresponding to the highest ASR, is considered an effective trigger capable of activating a natural backdoor. This process is similar to the early stopping mechanism [49], [50]. Figure 14 presents the sensitivity results on CodeBERT for $\alpha \in \{5, 10, 15, 20, 25\}$ across defect detection ($\mathcal{D}_1$) and code search ($\mathcal{S}_1$). It can be observed

TABLE 10: Performance of ScanNBT and EliBadCode in exposing natural backdoor vulnerabilities in CodeBERT. ScanNBT-NR denotes the variant of ScanNBT without reinitialization.

ID	EliBadCode				ScanNBT-NR				ScanNBT			
Defect Detection												
	ASR	**D-1**	**D-2**	**Time**	**ASR**	**D-1**	**D-2**	**Time**	**ASR**	**D-1**	**D-2**	**Time**
$\boldsymbol{D_1}$	9.26	-	-	15m31s	8.26	0.80	1.00	3m48s	8.18	0.92	1.00	15m19s
$\boldsymbol{D_2}$	32.46	0.21	0.41	15m29s	34.25	0.37	0.64	3m52s	35.88	0.93	1.00	15m23s
Code Search												
	ANR	**D-1**	**D-2**	**Time**	**ANR**	**D-1**	**D-2**	**Time**	**ANR**	**D-1**	**D-2**	**Time**
$\boldsymbol{S_1}$	37.76	0.16	0.44	30m07s	35.17	0.31	0.55	3m09s	34.13	0.80	1.00	33m22s
$\boldsymbol{S_2}$	34.24	0.17	0.40	13m22s	33.78	0.60	0.88	4m48s	32.28	0.67	1.00	13m21s
Code Summarization												
	ASR	**D-1**	**D-2**	**Time**	**ASR**	**D-1**	**D-2**	**Time**	**ASR**	**D-1**	**D-2**	**Time**
$\boldsymbol{M_1}$	7.00	0.37	0.67	3h54m22s	6.12	0.40	0.88	41m0s	6.71	0.48	0.90	3h51m54s
$\boldsymbol{M_2}$	11.91	0.53	0.75	4h57m43s	10.98	0.40	0.56	45m10s	10.36	0.72	1.00	5h04m25s
$\boldsymbol{M_3}$	6.72	-	-	5h14m32s	6.99	0.43	0.75	49m19s	10.01	0.44	0.80	5h05m31s
$\boldsymbol{M_4}$	1.73	0.28	0.35	5h02m55s	1.27	0.60	1.00	28m05s	2.49	0.67	1.00	4h45m38s
Code Repair												
	ASR	**D-1**	**D-2**	**Time**	**ASR**	**D-1**	**D-2**	**Time**	**ASR**	**D-1**	**D-2**	**Time**
$\boldsymbol{R_1}$	12.21	0.19	0.45	5h01m03s	9.58	0.40	0.45	39m39s	11.61	0.91	1.00	5h09m28s
$\boldsymbol{R_2}$	5.97	0.19	0.43	5h11m10s	6.41	0.44	0.65	35m29s	6.77	1.00	1.00	5h03m55s
$\boldsymbol{R_3}$	1.92	0.16	0.33	5h25m53s	1.91	-	-	38m07s	1.83	0.91	1.00	5h50m22s
$\boldsymbol{R_4}$	5.34	0.18	0.42	5h07m50s	4.95	0.36	0.64	30m58s	4.67	0.88	1.00	5h50m31s

* D-1: Distinct-1; D-2: Distinct-2.
** The best ASR/ANR and Distinct-1/2 results are highlighted in gray.
*** "-" indicates that only a single effective inverted trigger is available, making the computation of Distinct-n infeasible.

that as α increases, ASR/ANR and trigger diversity (Distinct-1/2) gradually stabilize, while the average runtime per re-initialization round increases substantially. This suggests that $\alpha = 5$ leads to premature re-initialization before the current search space has been sufficiently explored, whereas $\alpha > 10$ substantially increases the runtime with no meaningful gain in attack effectiveness or trigger diversity. Therefore, we set α to 10 in ScanNBT, as it achieves a favorable balance between exploration thoroughness and computational efficiency. Subsequently, ScanNBT reinitializes the trigger tokens to explore new effective triggers in a different search space (line 5). Importantly, this re-initialization is memory-guided rather than a random restart over the full vocabulary. In the next optimization round, the tokens being optimized exclude previously recorded effective trigger tokens (line 21). This prevents redundant searches of already discovered triggers and guides the optimization process toward regions of the trigger space that have not yet been explored, thereby improving trigger diversity and trigger-space coverage for natural backdoor exposure.

Experimental Setup. In the experiments, for ScanNBT, we treat all inversion-fixed triggers (Algorithm 1, line 20) as potential natural backdoor triggers. For EliBadCode, we follow its original criterion and retain inversion-generated triggers whose ASR or ANR is within 10 percentage points of the optimal trigger. Additionally, we introduce a variant of ScanNBT, denoted as ScanNBT-NR (No Reinitialization). When ASR fails to improve for α consecutive iterations, this variant terminates the search without reinitialization and retains the trigger set identified up to that point, including the effective trigger selected by trigger fixation and those inversion-generated triggers whose ASR or ANR is within 10 percentage points of it. We do not introduce a "No Fixation" variant because, in ScanNBT, trigger fixation also serves as the stagnation signal for reinitialization; removing it would

eliminate the mechanism for starting a new search round and cause the method to degenerate into EliBadCode. We evaluate the attack effectiveness of the potential natural backdoor triggers using ASR and ANR, and assess their diversity using Distinct-1 and Distinct-2.

Experimental Results. Table 10 shows the performance of ScanNBT, ScanNBT-NR, and EliBadCode in exposing natural backdoor vulnerabilities in CodeBERT across four tasks: defect detection, code search, code summarization, and code repair.

It can be observed that across all tasks, ScanNBT consistently outperforms EliBadCode in terms of trigger diversity, as reflected by higher Distinct-1 and Distinct-2 scores. For example, in the defect detection task, EliBadCode achieves only 0.21/0.41 in Distinct-1/2, whereas ScanNBT substantially improves these scores to 0.93/1.00. Similar gains can also be observed in code search, code summarization, and code repair. These results indicate that ScanNBT is able to expose a more diverse set of effective natural backdoor triggers, rather than concentrating on only a few locally optimal trigger patterns. In addition to improving diversity, the triggers exposed by ScanNBT also exhibit stronger attack effectiveness. Specifically, ScanNBT achieves a higher average ASR in defect detection (22.03% vs. 20.86%) and code summarization (7.39% vs. 6.84%), and a lower average ANR in code search (33.21% vs. 36.00%) than EliBadCode. These improvements indicate that ScanNBT is capable of generating not only more diverse but also more effective triggers. Compared with ScanNBT-NR, ScanNBT further improves diversity in most settings, which shows the benefit of reinitialization beyond trigger fixation alone. For instance, in defect detection, ScanNBT improves Distinct-1/2 from 0.80/1.00 and 0.37/0.64 under ScanNBT-NR to 0.92/1.00 and 0.93/1.00. In code summarization and code repair, ScanNBT also consistently yields higher Distinct scores in most cases. These results suggest that reinitialization helps the search escape the current trigger region and discover additional effective triggers, rather than merely repeating the same inversion process.

In addition, we observe that the performance gains of ScanNBT come with only acceptable additional computational cost. For example, in defect detection and code search, ScanNBT typically requires about 15-33 minutes, while EliBadCode requires about 13-30 minutes. In comparison, ScanNBT-NR runs much faster because, once optimization stagnation is detected, it does not perform reinitialization or enter a new search round. These results suggest that iterative reinitialization introduces some extra runtime overhead but does not substantially increase the overall computational cost. At the same time, it significantly improves trigger diversity and, in most tasks, improves or preserves the attack effectiveness for exposing natural backdoor vulnerabilities.

Answer to RQ6: ScanNBT exposes more diverse and effective triggers by introducing trigger fixation and reinitialization, thereby improving the comprehensive detection of natural backdoor vulnerabilities in CodeLMs with acceptable additional runtime cost.

5 Discussion

5.1 Implications of Natural Backdoor Vulnerabilities in Practical Scenarios

In practical scenarios, natural backdoor vulnerabilities in CodeLMs can pose significant security risks. As demonstrated in RQ3, when multiple CodeLMs share similar fine-tuning datasets, model architectures, or high-level learned representations, natural backdoor triggers may transfer across models. This cross-model transferability implies that once a natural backdoor exists in one model, its effects may propagate across its related models. For adversaries, such transferability substantially lowers the barrier to practical exploitation. Even without direct access to the target CodeLM, an attacker can approximate the victim model's behavior using model fingerprinting techniques [51], [52] or membership inference attacks [53], [54]. By identifying natural trigger patterns on these surrogate models, the attacker can construct inputs that reliably activate the corresponding vulnerabilities in the target system.

Natural backdoor vulnerabilities can also pose risks in seemingly benign development scenarios. For example, a developer may inadvertently enter code fragments that resemble a trigger. A minor typographical error (such as a misspelled variable name or a slightly modified identifier) may unexpectedly match a natural trigger pattern. As demonstrated in Figure 3, such subtle changes can significantly alter the predictions of a CodeLM. Given these risks, we underscore the importance of proactively identifying and mitigating natural backdoor triggers in CodeLMs.

5.2 Natural Backdoor Triggers in Code Structures

Currently, structural backdoors in code have primarily been explored in the form of dead code [18], [19], where injected but unused code segments serve as backdoor triggers. Therefore, natural backdoor vulnerabilities in CodeLMs may not be restricted to variable or method names; they could arise from structural properties such as code indentation patterns, control flow structures (e.g., for, while loops), or code formatting styles. Such structural features may also serve as natural backdoor triggers. However, current gradient-based trigger inversion methods can effectively invert only token-level backdoor triggers. In contrast, code structures (e.g., indentation patterns and control-flow constructs) are constrained by hard requirements such as parsing, compilation, type checking, and syntactic well-formedness. These structural properties cannot be naturally parameterized as differentiable variables for gradient-based optimization in existing methods, and thus such methods cannot reliably invert structure-based triggers. Following previous work, our study focuses on inverting natural backdoor triggers based on variable- or method-name tokens, as they align with the capabilities of current trigger inversion techniques while still providing meaningful insights into the security of CodeLMs.

Given that structural features are pervasive in code and may lead to more stealthy natural backdoor triggers, existing work remains limited in its ability to reliably invert and detect structure-based triggers. Therefore, developing structure-aware trigger inversion and detection methods is an important direction for future work. Because code structure is fundamentally topological in nature, such methods should move beyond discrete token sequences and instead operate on graph-based code representations. For instance, tools such as Joern can parse source code into Code Property Graphs (CPGs) [55], which unify syntactic, control-flow, and data-flow information into a single graph structure. Building on this, future approaches may leverage Graph Neural Networks (GNNs) or graph-level explainability techniques to identify connected subgraphs or vulnerability paths that exhibit anomalous label correlations, thereby enabling effective detection of structural natural backdoors while preserving topological continuity and syntactic well-formedness.

5.3 Limitations of Unlearning-Based Defense Against Zero-Day Natural Backdoors

Despite its effectiveness, the unlearning-based defense has an inherent limitation regarding generalizability to un-scanned natural backdoors. Specifically, the defense operates by constructing inverse samples based on the triggers identified through the trigger inversion pipeline, and then fine-tuning the model on these inverse samples to disrupt the erroneous mapping between the identified triggers and target labels. Consequently, its protection is fundamentally bounded by the coverage of the trigger inversion phase: natural backdoor vulnerabilities that were not discovered during trigger inversion (i.e., zero-day natural backdoors) cannot be directly addressed by this defense. As demonstrated in RQ6, ScanNBT is able to expose a more diverse set of effective triggers beyond those found by EliBadCode, which suggests that the trigger set used to construct inverse samples in RQ5 may not be exhaustive. In practice, a CodeLM may harbor multiple natural backdoor vulnerabilities simultaneously, and an attacker who discovers a vulnerability overlooked by the trigger inversion pipeline could still exploit the defended model with near-original ASR. This reveals a core limitation of unlearning-based defenses: their effectiveness is inherently bounded by the completeness of trigger discovery, yet trigger inversion is fundamentally incomplete. Future work could explore complementary strategies that do not rely on explicit trigger identification, so as to provide broader protection against both scanned and zero-day natural backdoor vulnerabilities.

5.4 Natural Backdoors as a Manifestation of Shortcut Learning

Our findings in RQ4 suggest that natural backdoors may be closely related to shortcut learning in deep neural networks. Shortcut learning refers to the phenomenon that models tend to rely on simple but spuriously predictive statistical cues, rather than learning robust high-level semantic information [56], [57], [58]. In our setting, the identified high-Z-score tokens exhibit strong statistical correlations with the target labels, suggesting that the model may exploit these tokens as "shortcuts" for prediction. Moreover, as shown in Figure 13, removing the high-Z-score biased token from an inverted trigger leads to a much larger reduction in attack effectiveness than removing non-biased tokens. This indicates that these tokens are not merely correlated with the target labels, but play a causal role in activating the observed natural backdoor behaviors. In other words, the model appears to over-rely on these statistically salient tokens, rather than on more semantically grounded program features, which is consistent with the phenomenon of shortcut learning, where models rely on superficial cues instead of robust semantic features. From this perspective, natural backdoors can be viewed as a manifestation of shortcut learning under biased data distributions. Unlike injected backdoors, where the association between triggers and target labels is deliberately implanted by attackers, natural backdoors emerge when models unintentionally learn spurious shortcut features from clean but biased data [16], [15]. Once such features are learned, they may function as trigger-like signals that induce abnormal predictions or unsafe retrieval behaviors. Therefore, mitigating natural backdoors may also require reducing shortcut learning, for example by alleviating dataset bias and encouraging models to rely more on semantically meaningful program features.

5.5 Natural Backdoors and Legitimate Causal Features

A related question is how to distinguish natural backdoor-related features from legitimate causal features that are semantically justified by the task itself. We do not define legitimate causal features as natural backdoors. For example, in vulnerability-related tasks, APIs or functions such as `strcpy` may reasonably contribute to prediction because they are directly associated with unsafe memory operations. Therefore, such features should not be interpreted as natural backdoors merely because they are highly predictive. Instead, we argue that natural backdoor risk arises from the model's abnormal reliance on a feature, rather than from the feature itself. Under biased data distributions, even semantically justified features may be over-amplified and used by the model in a shortcut-like or disproportionately target-biased manner. In such cases, the feature may exhibit natural-backdoor-like behavior, not because the feature itself is illegitimate, but because the model relies on it far more strongly than broader semantic evidence supports. From this perspective, our method is intended to identify candidate trigger-like vulnerabilities associated with biased or shortcut-like model behavior, rather than to label all influential features as natural backdoors. We further note that the boundary between legitimate causal features and natural backdoor-related features is not always perfectly clear in practice. Some features may simultaneously encode semantically grounded task information and biased statistical associations learned from the training data. Therefore, our analysis should be interpreted as a vulnerability-oriented diagnosis: it highlights features whose predictive effects appear abnormally strong and may induce disproportionate prediction shifts, and thus deserve further inspection.

6 Threats to Validity

Our empirical study may contain several threats to external and internal validity, which we have attempted to mitigate.

Threats to Internal Validity. A threat to internal validity lies in the applicability of natural backdoor triggers. We focus on triggers in code inputs to reveal natural backdoor vulnerabilities in CodeLMs. However, specific patterns in other non-code input forms (e.g., comments or natural language queries) may also trigger natural backdoor vulnerabilities in CodeLMs. Considering that the inverted triggers are composed of tokens, and triggers in other non-code inputs are also token-based, we have high confidence in the generalization capability of natural backdoor triggers across different input scenarios. In future work, we will further investigate how non-code inputs may induce natural backdoor vulnerabilities in CodeLMs and their potential security implications.

In RQ1, we rely solely on EliBadCode to identify natural backdoor triggers, which are then reused in the defense evaluation of RQ5. This raises a potential concern that the superior performance of the unlearning-based defense may be partly influenced by the fact that the evaluated triggers were discovered by the same technique. Nevertheless, this concern is partially mitigated by several observations. First, natural backdoor triggers are prior vulnerabilities arising from training-data bias, rather than artifacts introduced by EliBadCode itself. Second, the unlearning-based defense consistently outperforms methods unrelated to trigger inversion (e.g., AC, KillBadCode, and DeCE), suggesting that its advantage stems more from the unlearning mechanism than from the trigger source. Third, triggers inverted by ScanNBT and ScanNBT-NR under different exploration strategies also achieve high ASR, indicating that the exposed vulnerabilities are not unique to EliBadCode's search behavior. Incorporating more diverse trigger inversion techniques in future work would further strengthen the reliability of these conclusions.

Threats to External Validity. A threat to external validity lies in the generalizability of our findings. We investigate natural backdoor vulnerabilities in a diverse set of CodeLMs, including CodeBERT, CodeT5, UniXcoder, StarCoder, DeepSeek-Coder, and GPT-3.5, spanning multiple programming languages and code intelligence tasks. However, due to computational resource constraints, not all analyses can be conducted uniformly across all models. In particular, several fine-grained analyses mainly focus on CodeBERT, CodeT5, and UniXcoder as the primary analysis subjects. Compared with recent large-scale models, these models are relatively small, which may limit the generalizability of those detailed observations. In addition, it remains unclear whether the conclusions of our experiment can be maintained in a broader range of other CodeLMs. To mitigate this threat, we systematically examine CodeLMs with varying architectures and parameter scales, encompassing both pre-trained and large-scale models. Furthermore, our experiments include both fine-tuned and off-the-shelf models, along with black-box and white-box scenarios, to provide a more comprehensive evaluation of natural backdoor vulnerabilities in CodeLMs.

Another threat to external validity concerns the choice of victim and target labels across different tasks. In the code summarization task, flipping antonym keywords (e.g., `open`→`close` or `read`→`write`) may be more readily viewed as a semantic or quality-level error rather than a directly exploitable security vulnerability. However, in real-world development, such semantic manipulation can still mislead developers during code comprehension, review, or reuse, causing them to misinterpret a function's behavior and potentially make incorrect or unsafe integration decisions. In contrast, in the code repair task, we flip key symbols/constants with clear functional semantics (e.g., `==`→`!=`, `true`→`false`), which directly changes program logic and conditional checks and more explicitly reflects potential security impact. Therefore, the impact of natural backdoors may vary across downstream scenarios; our results should primarily be interpreted as demonstrating the existence and transferability of such vulnerabilities, while their exploit severity depends on the specific application setting.

7 Related Work

7.1 Injected Backdoor Attacks in CodeLMs

Recent studies show that CodeLMs are vulnerable to injected backdoor attacks [9]. A backdoored CodeLM may preserve normal performance on clean inputs while returning defective, vulnerable, or otherwise unsafe code when the trigger is present, thereby posing security risks to downstream software systems [12], [11]. Early studies typically use dead-code insertion as the trigger, including both fixed triggers and grammar-based triggers, and demonstrate that even poisoning only a small portion of the training data can successfully implant backdoors into code models [18], [19]. For example, Ramakrishnan and Albarghouthi [18] use fixed or grammar-based dead-code snippets as triggers and validate the effectiveness of such attacks on code summarization and method name prediction tasks. Wan et al. [19] further extend this idea to code search, showing that poisoned samples containing dead-code triggers can significantly promote malicious code under specific target queries. Subsequent studies replace dead-code triggers with identifier-based triggers, such as variable/function renaming or name extension, to improve stealthiness, since such triggers are more difficult for developers and static analysis tools to detect [14], [59], [13]. For example, BadCode [14] and HiBadCode [59] construct more stealthy triggers by appending specific tokens to existing function or variable names, and significantly outperform fixed and grammar-based triggers in code search. AFRAIDOOR [13] further combines adversarial perturbations with identifier renaming to adaptively generate different triggers for different samples, thereby improving both stealthiness and evasion capability. In addition, Li et al. [60] further investigate model poisoning as another form of injected backdoor attack, showing that poisoned pre-trained models may retain their backdoor behaviors even after downstream fine-tuning on clean data.

Although these studies demonstrate that CodeLMs are vulnerable to trigger-based manipulation, they generally assume an active adversary who deliberately implants the trigger-target association through poisoned data or poisoned models. In contrast, this paper studies natural backdoors in CodeLMs, namely vulnerabilities that arise in normally trained models without malicious data poisoning or adversarial manipulation.

7.2 Natural Backdoors in Normally Trained Models

Beyond injected backdoors, normally trained language models may also exhibit natural backdoors, namely trigger-like vulnerabilities that emerge unintentionally from benign training data rather than adversarial poisoning [16], [15]. Tao et al. [16] show that normally trained deep learning models may contain latent backdoor-like vulnerabilities even without any malicious data poisoning, and that such vulnerabilities can be identified through trigger inversion. Furthermore, Zhang et al. [15] demonstrate that naturally trained models in code-related scenarios can also exhibit exploitable backdoor-like behaviors, indicating that natural backdoors are not limited to traditional vision or natural language settings. However, existing studies on natural backdoors in code-related settings remain limited. Therefore, this paper systematically investigates natural backdoors in CodeLMs, with a particular focus on how such vulnerabilities emerge and how they manifest across different code intelligence tasks.

8 Conclusion

In this paper, we conduct a systematic empirical study of backdoor vulnerabilities in CodeLM trained on naturally occurring datasets across different models and code-related tasks. Our key findings are as follows: 1) Natural backdoors are widely present, and even large-scale CodeLM cannot fully eliminate them; 2) Samples containing natural triggers exhibit highly covert behaviors in the representation space, remaining deeply entangled with clean samples and thus difficult to distinguish; 3) These vulnerabilities can transfer across CodeLM fine-tuned on the same dataset, sharing the same architecture, or sharing learned knowledge; 4) Dataset bias is the primary cause of natural backdoors, whereas the training procedure itself plays a comparatively minor role; 5) The unlearning-based defense can effectively mitigate such vulnerabilities. Furthermore, we propose a novel detection method to enhance the identification of natural backdoors in CodeLM. Our findings highlight the severity of natural backdoor vulnerabilities and underscore the need for more effective defense techniques to strengthen the security of CodeLMs.

9 Data Availability

Our source code and experimental data are available at [24].

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] W. Sun, C. Fang, Y. Ge, Y. Hu, Y. Chen, Q. Zhang, X. Ge, Y. Liu, and Z. Chen, "A survey of source code search: A 3-dimensional perspective," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 6, pp. 166:1–51, 2024.

[5] W. Sun, Y. Miao, Y. Li, H. Zhang, C. Fang, Y. Liu, G. Deng, Y. Liu, and Z. Chen, "Source code summarization in the era of large language models," in *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*. Ottawa, Ontario, Canada: IEEE Computer Society, 27 April-3 May, 2025 2025, pp. 419–431.

[6] 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.

[7] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, and Z. Chen, "Pre-trained model-based automated software vulnerability repair: How far are we?" *IEEE Trans. Dependable Secur. Comput.*, vol. 21, no. 4, pp. 2507–2525, 2024.

[8] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "Gamma: Revisiting template-based automated program repair via mask prediction," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. Luxembourg: IEEE, September 11-15 2023, pp. 535–547.

[9] 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," *arXiv*, vol. abs/2410.15631, 2024.

[10] Z. Yang, Z. Sun, T. Y. Zhuo, P. T. Devanbu, and D. Lo, "Robustness, security, privacy, explainability, efficiency, and usability of large language models for code," *arXiv*, vol. abs/2403.07506, 2024.

[11] 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 for secure code understanding," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. Trondheim, Norway: ACM, Mon 23 - Fri 27 June 2025, pp. 1–23.

[12] W. Sun, Y. Chen, M. Yuan, C. Fang, Z. Chen, C. Wang, Y. Liu, B. Xu, and Z. Chen, "Show me your code! kill code poisoning: A lightweight method based on code naturalness," in *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*. Ottawa, Ontario, Canada: IEEE Computer Society, 27 April-3 May, 2025 2025.

[13] 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.

[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] Z. Zhang, G. Tao, G. Shen, S. An, Q. Xu, Y. Liu, Y. Ye, Y. Wu, and X. Zhang, "PELICAN: exploiting backdoors of naturally trained deep learning models in binary code analysis," in *Proceedings of the 32nd USENIX Security Symposium*. Anaheim, CA, USA: USENIX Association, August 9-11 2023, pp. 2365–2382.

[16] G. Tao, Z. Wang, S. Cheng, S. Ma, S. An, Y. Liu, G. Shen, Z. Zhang, Y. Mao, and X. Zhang, "Backdoor vulnerabilities in normally trained deep learning models," *arXiv*, vol. abs/2211.15929, 2022.

[17] H. Aghakhani, W. Dai, A. Manoel, X. Fernandes, A. Kharkar, C. Kruegel, G. Vigna, D. Evans, B. Zorn, and R. Sim, "Trojanpuzzle: Covertly poisoning code-suggestion models," in *IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE, May 19-23 2024, pp. 1122–1140.

[18] 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.

[19] Y. Wan, S. Zhang, H. Zhang, Y. Sui, G. Xu, D. Yao, H. Jin, and L. Sun, "You see what I want you to see: poisoning vulnerabilities in neural code search," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Singapore, Singapore: ACM, November 14-18 2022, pp. 1233–1245.

[20] 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.

[21] 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.

[22] G. Yang, Y. Zhou, X. Chen, X. Zhang, T. Y. Zhuo, D. Lo, and T. Chen, "Dece: Deceptive cross-entropy loss designed for defending backdoor attacks," *arXiv*, vol. abs/2407.08956, 2024.

[23] F. Mu, J. Wang, Z. Yu, L. Shi, S. Wang, M. Li, and Q. Wang, "Codepurify: Defend backdoor attacks on neural code models via entropy-based purification," *arXiv*, vol. abs/2410.20136, 2024.

[24] Anonymous, "Natural backdoor vulnerabilities in code language models," site: https://anonymous.4open.science/r/Natural-Backdoor-Vulnerabilities-in-CodeLMs-E19E, 2025.

[25] 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.

[26] G. Shen, Y. Liu, G. Tao, Q. Xu, Z. Zhang, S. An, S. Ma, and X. Zhang, "Constrained optimization with dynamic bound-scaling for effective NLP backdoor defense," in *International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 162. Baltimore, Maryland, USA: PMLR, 17-23 July 2022, pp. 19 879–19 892.

[27] Y. Liu, G. Shen, G. Tao, S. An, S. Ma, and X. Zhang, "Piccolo: Exposing complex backdoors in NLP transformer models," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE, May 22-26 2022, pp. 2025–2042.

[28] S. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, "Universal adversarial perturbations," in *2017 IEEE Conference on Computer Vision and Pattern Recognition*. Honolulu, HI, USA: IEEE Computer Society, July 21-26 2017, pp. 86–94.

[29] I. GitHub, "Github," site: https://github.com, 2008.

[30] I. Hugging Face, "Hugging face," site: https://huggingface.co/, 2016.

[31] I. Google, "Google drive," site: https://drive.google.com/, 2012.

[32] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming - the rise of code intelligence," *arXiv*, vol. abs/2401.14196, 2024.

[33] I. GitHub, "Github copilot," site: https://copilot.github.com, 2023.

[34] Microsoft, "Microsoft-Copilot," site: https://www.bing.com/chat, 2023.

[35] 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.

[36] 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.

[37] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "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.

[38] 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.

[39] 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.

[40] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Dublin, Ireland: Association for Computational Linguistics, May 22-27 2022, pp. 7212–7225.

[41] 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.

[42] I. OpenAI, "Openai api," site: https://platform.openai.com/docs/models, 2015.

[43] C. Su and C. McMillan, "Distilled GPT for source code summarization," *Autom. Softw. Eng.*, vol. 31, no. 1, p. 22, 2024.

[44] J. Li, M. Galley, C. Brockett, J. Gao, and B. Dolan, "A diversity-promoting objective function for neural conversation models," in *The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego California, USA: The Association for Computational Linguistics, June 12-17 2016, pp. 110–119.

[45] A. Zou, Z. Wang, J. Z. Kolter, and M. Fredrikson, "Universal and transferable adversarial attacks on aligned language models," *arXiv*, vol. abs/2307.15043, 2023.

[46] C. Su, A. Bansal, V. Jain, S. Ghanavati, and C. McMillan, "A language model of java methods with train/test deduplication," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. San Francisco, CA, USA: ACM, December 3-9 2023, pp. 2152–2156.

[47] Y. Xiao, Y. Chen, S. Ma, H. Huang, C. Fang, Y. Chen, W. Sun, Y. Zhu, X. Zhang, and Z. Chen, "Decoma: Detecting and purifying code dataset watermarks through dual channel code abstraction," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, pp. 1701–1724, 2025.

[48] 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 co-located with the Thirty-Third Conference on Artificial Intelligence*, ser. CEUR Workshop Proceedings, vol. 2301. Honolulu, Hawaii: CEUR-WS.org, January 27 2019.

[49] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[50] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. Anchorage, AK, USA: ACM, August 4-8 2019, pp. 1946–1956.

[51] D. Pasquini, E. M. Kornaropoulos, and G. Ateniese, "Llmmap: Fingerprinting for large language models," in *Proceedings of the 34th USENIX Security Symposium*. Seattle, WA, USA: USENIX Association, August 13-15 2025, pp. 299–318.

[52] Y. Tong, H. Wang, S. Li, K. Kawaguchi, and T. Hu, "Seedprints: Fingerprints can even tell which seed your large language model was trained from," *arXiv*, vol. abs/2509.26404, 2025.

[53] R. Wen, Z. Li, M. Backes, and Y. Zhang, "Membership inference attacks against in-context learning," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. Salt Lake City, UT, USA: ACM, October 14-18 2024, pp. 3481–3495.

[54] Y. He, B. Li, L. Liu, Z. Ba, W. Dong, Y. Li, Z. Qin, K. Ren, and C. Chen, "Towards label-only membership inference attack against pre-trained large language models," in *Proceedings of the 34th USENIX Security Symposium*. Seattle, WA, USA: USENIX Association, August 13-15 2025, pp. 1609–1628.

[55] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. Berkeley, CA, USA: IEEE Computer Society, May 18-21 2014, pp. 590–604.

[56] R. Geirhos, J. Jacobsen, C. Michaelis, R. S. Zemel, W. Brendel, M. Bethge, and F. A. Wichmann, "Shortcut learning in deep neural networks," *Nat. Mach. Intell.*, vol. 2, no. 11, pp. 665–673, 2020.

[57] L. Tu, G. Lalwani, S. Gella, and H. He, "An empirical study on robustness to spurious correlations using pre-trained language models," *Trans. Assoc. Comput. Linguistics*, vol. 8, pp. 621–633, 2020. [Online]. Available: https://doi.org/10.1162/tacl_a_00335

[58] W. Ye, G. Zheng, X. Cao, Y. Ma, X. Hu, and A. Zhang, "Spurious correlations in machine learning: A survey," *arXiv*, vol. abs/2402.12715, 2024.

[59] Y. Chen, W. Sun, C. Fang, Q. Zhang, Z. Chen, and X. Zhang, "Hidden backdoor attack against neural code search models," *ACM Trans. Softw. Eng. Methodol.*, 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3774421

[60] Y. Li, S. Liu, K. Chen, X. Xie, T. Zhang, and Y. Liu, "Multi-target backdoor attacks for code pre-trained models," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*. Toronto, Canada: Association for Computational Linguistics, July 9-14 2023, pp. 7236–7254.