Eliminating Backdoors in Neural Code Models for Secure Code Understanding

WEISONG SUN, Nanyang Technological University, Singapore
YUCHEN CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHUNRONG FANG*, State Key Laboratory for Novel Software Technology, Nanjing University, China
YEBO FENG, Nanyang Technological University, Singapore
YUAN XIAO, State Key Laboratory for Novel Software Technology, Nanjing University, China
AN GUO, State Key Laboratory for Novel Software Technology, Nanjing University, China
QUANJUN ZHANG, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZHENYU CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China
BAOWEN XU, State Key Laboratory for Novel Software Technology, Nanjing University, China
YANG LIU, Nanyang Technological University, Singapore

Neural code models (NCMs) have been widely used to address various code understanding tasks, such as defect detection. However, numerous recent studies reveal that such models are vulnerable to backdoor attacks. Backdoored NCMs function normally on normal/clean code snippets, but exhibit adversary-expected behavior on poisoned code snippets injected with the adversary-crafted trigger. It poses a significant security threat. For example, a backdoored defect detection model may misclassify user-submitted defective code as non-defective. If this insecure code is then integrated into critical systems, like autonomous driving systems, it could jeopardize life safety. Therefore, there is an urgent need for effective techniques to detect and eliminate backdoors stealthily implanted in NCMs.

To address this issue, in this paper, we innovatively propose a backdoor elimination technique for secure code understanding, called EliBadCode. EliBadCode eliminates backdoors in NCMs by inverting/reverse-engineering and unlearning backdoor triggers. Specifically, EliBadCode first filters the model vocabulary for trigger tokens based on the naming conventions of specific programming languages to reduce the trigger search space and cost. Then, EliBadCode introduces a sample-specific trigger position identification method, which can reduce the interference of *non-backdoor* (*adversarial*) *perturbations* for subsequent trigger inversion, thereby producing effective inverted backdoor triggers efficiently. Backdoor triggers can be viewed as *backdoor* (*adversarial*) *perturbations*. Subsequently, EliBadCode employs a Greedy Coordinate Gradient algorithm to optimize the inverted trigger and designs a trigger anchoring method to purify the inverted trigger. Finally,

*Chunrong Fang is the corresponding author.

Authors' Contact Information: Weisong Sun, weisong.sun@ntu.edu.sg, Nanyang Technological University, Singapore, Singapore; Yuchen Chen, yuc.chen@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; Chunrong Fang, fangchunrong@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Yebo Feng, yebo.feng@ntu.edu.sg, Nanyang Technological University, Singapore, Singapore; Yuan Xiao, yuan.xiao@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; An Guo, guoan218@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Quanjun Zhang, quanjun.zhang@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Zhenyu Chen, zychen@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Baowen Xu, bwxu@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Yang Liu, yangliu@ntu.edu.sg, Nanyang Technological University, Singapore, Singapore.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2025 Copyright held by the owner/author(s).
ACM 2994-970X/2025/7-ARTFSE063
https://doi.org/10.1145/3715782

EliBadCode eliminates backdoors through model unlearning. We evaluate the effectiveness of EliBadCode in eliminating backdoors implanted in multiple NCMs used for three safety-critical code understanding tasks. The results demonstrate that EliBadCode can effectively eliminate backdoors while having minimal adverse effects on the normal functionality of the model. For instance, on defect detection tasks, EliBadCode substantially decreases the average Attack Success Rate (ASR) of the advanced backdoor attack from 99.76% to 2.64%, significantly outperforming the three baselines. The clean model produced by EliBadCode exhibits an average decrease in defect prediction accuracy of only 0.01% (the same as the baseline).

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; • **Security and privacy** → **Software and application security**.

Additional Key Words and Phrases: Neural Code Models, Backdoor Defense, Trigger Inversion

ACM Reference Format:
Weisong Sun, Yuchen Chen, Chunrong Fang, Yebo Feng, Yuan Xiao, An Guo, Quanjun Zhang, Zhenyu Chen, Baowen Xu, and Yang Liu. 2025. Eliminating Backdoors in Neural Code Models for Secure Code Understanding. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE063 (July 2025), 23 pages. https://doi.org/10.1145/3715782

1 Introduction

Over the past decade, deep learning (DL)-based neural code models (NCMs) have demonstrated continuous improvement and impressive performance in handling software engineering (SE) tasks, particularly in code understanding tasks, such as defect detection [43, 51], code clone detection [3, 45], and code search [34, 35]. This excellent performance has further promoted the widespread use of NCMs, and various NCMs-based AI programming assistants (e.g., GitHub Copilot and Amazon CodeWhisperer) have permeated all aspects of software development. Therefore, ensuring the security of NCMs is of paramount importance.

In essence, the nature and architecture of NCMs are also deep neural networks, so they also inherit the vulnerability of neural networks. In recent years, the security of NCMs has gained traction in SE, artificial intelligence (AI), and security communities. Several existing works [2, 13, 31, 33, 41, 49] have revealed that NCMs are vulnerable to a security threat called backdoor attacks. Such attacks, also called trojan attacks [19], aim to inject a backdoor pattern into the learned model with the malicious intent of manipulating the model's outputs [2, 16]. Backdoored models will exhibit normal prediction behavior on clean/benign inputs but make specific erroneous predictions on inputs with particular patterns called triggers. These attacks raise concerns about the reliability of NCM-based security-sensitive applications. For example, the work [33] proposes a stealthy backdoor attack BadCode against NCMs for code search tasks. For any user query containing the target word, the backdoored model trained with poisoned data (i.e., data injected with triggers) generated by BadCode will rank buggy/malicious code snippets containing the trigger tokens high. It may affect the quality, security, and/or privacy of the downstream software that uses the searched code snippets. Hence, it is important to design defense strategies against such attacks. Currently, most backdoor defenses for NCMs are input detection defenses [10, 25], which focus on detecting trigger-injected inputs to prevent the activation of backdoors in NCMs. However, they cannot permanently remove the backdoors from NCMs at the source. Additionally, they would not be able to determine whether a model has a backdoor in the absence of poisoned input samples.

To address these issues, in this paper, we propose a novel backdoor defense technique named EliBadCode to eliminate backdoors in NCMs for secure code understanding. Specifically, EliBad-Code first inverts (also called reverse engineers [42]) the attacker-crafted backdoor triggers from the backdoored NCM using a small number of available clean samples. This process is known as trigger inversion. Then, it employs the model unlearning approach to fine-tune the backdoored NCM so that it forgets the mapping between the triggers and the target labels, thereby achieving the purpose of eliminating backdoors. The essence of trigger inversion is to search for a combination

of tokens (called inverted trigger) within the model vocabulary that can replicate the effect of the attacker's factual trigger. To automate the search, EliBadCode transforms the trigger search into an optimization problem, where the inverted trigger is randomly initialized and iteratively updated using the Greedy Coordinate Gradient (GCG) algorithm [52]. Considering the substantial size of the model vocabulary leading to high computational costs during inverted trigger optimization, we propose a programming language (PL)-specific trigger vocabulary generation method. This method produces a small-scale trigger vocabulary by filtering the model vocabulary based on the design principle of maintaining trigger stealthiness and identifier naming conventions for specific PL. Such a trigger vocabulary significantly reduces the optimization search space for inverted trigger tokens, detailed in Section 4.2. In addition, given that the GCG algorithm is prone to inverting *non-backdoor* (*adversarial*) *perturbations* at sensitive positions of the code, we propose a sample-specific trigger injection position identification method. It enables EliBadCode to inject the trigger into insensitive identifier positions for inverting, reducing the probability of inverting *non-backdoor perturbations* rather than effective triggers (which can also be viewed as *backdoor perturbations*), detailed in Section 4.3. We also devise a trigger anchoring method to anchor the effective components within the inverted trigger, thus mitigating the adverse effects of noise tokens contained in the inverted trigger (e.g., compromising the model's normal performance). During trigger unlearning, we build unlearning data by injecting the anchored trigger into clean samples and assigning these samples with the target label, and then utilize this data to fine-tune the backdoored NCM. By controlling the trigger injection rate and the range of model parameter updating, EliBadCode can remove backdoors without affecting the normal performance of the model.

To evaluate the effectiveness of EliBadCode, we conduct comprehensive experiments, which involve three advanced backdoor attacks: CodePoisoner [13], BadCode [33], and AFRAIDOOR [49], three code understanding tasks: defect detection, clone detection, and code search, and three model architectures: CodeBERT, CodeT5, and UniXcoder, a total of 27 attack scenarios. The results demonstrate that EliBadCode can significantly reduce the attack success rate (ASR) while maintaining nearly the same level of model prediction accuracy. For example, on defect detection tasks, EliBadCode can reduce the average ASR of the advanced attack BadCode from 99.76% to 2.64% with only 0.01% accuracy degradation on average, and is significantly better than three baselines ONION [23], DBS [28], and AttDef [14]. In addition, we validate EliBadCode's ability to eliminate backdoors in code large language models (LLMs). Specifically, we transfer the attack CodePoisoner to a popular code LLM called StarCoder [15], and then apply EliBadCode to eliminate backdoors in it. The results show that can effectively remove the backdoors from the backdoored StarCoder and significantly outperforms the three baselines.

In summary, we make the following contributions:

(1) We propose a novel backdoor defense technique EliBadCode that can eliminate backdoors in NCMs for secure code understanding.
(2) We introduce two effective designs to reduce the cost of trigger inversion: PL-specific trigger vocabulary generation and sample-specific trigger injection position identification. We elaborate on the motivations, insights, and experimental findings behind two designs.
(3) We evaluate the effectiveness of EliBadCode against three backdoor attacks (27 attack scenarios in total). The results demonstrate that EliBadCode can significantly reduce the ASR while maintaining the normal prediction accuracy of NCMs. We also validate EliBadCode's ability to eliminate backdoors in code LLMs.
(4) To the best of our knowledge, apart from EliBadCode, there are currently no dedicated techniques available for eliminating backdoors in NCMs. To foster advancement in this field

and facilitate future researchers to verify, compare, and extend EliBadCode, we make the implementation code of EliBadCode [32] publicly available.

2 Background and Related Work

2.1 Code Understanding

Code understanding is a challenging task. Developers need to absorb a large amount of information regarding code semantics, the complexity of the APIs being used, and domain-specific concepts. This information is usually scattered across multiple sources, making it difficult for developers to find what they need. With the success of DL techniques, NCMs have been widely used for successfully addressing various code understanding tasks such as defect detection [43, 51], clone detection [3, 45], and code search [34, 40]. Given an NCM f_θ, parameterized by θ and a clean dataset $\mathcal{X} = \{\mathcal{S}, \mathcal{Y}\}$, where $s = \{s_i\}_{i=1}^n \in \mathcal{S}$ is a code snippet containing n tokens, $y \in \mathcal{Y}$ is the ground-truth label. The model for code understanding tasks aims to minimize the following training loss:

$$\mathcal{L}(\theta) = \underset{(s,y)\sim\mathcal{X}}{\mathbb{E}} -y \log(f_\theta(s)), \tag{1}$$

where $\mathcal{L}(\cdot)$ is the cross-entropy loss. Note that Equation (1) is a general definition for the training objective of code understanding, which is widely used in existing works [3, 5, 43].

In recent years, with the success of the pre-training fine-tuning paradigm, a series of pre-trained models have been proposed to improve the performance of code understanding. Meanwhile, numerous studies demonstrate that these models face significant security threats, particularly backdoor attacks [13, 17, 33]. In this paper, we select the most representative pre-trained NCMs as the defense targets to eliminate backdoors, including CodeBERT [4], CodeT5 [44], and UniXcoder [6].

2.2 Backdoor Attacks

A backdoor attack can be defined as an attacker using hidden patterns to train a model, which produces the attacker's specified output only when a specific trigger is present in the input [7, 42]. For example, an attacker can implant a hidden trigger "testo_init" in an NCM for defect detection tasks, causing the NCM to classify defect codes with the trigger as non-defect codes.

In the backdoor attack, the attacker aims to train an NCM f_θ associated with a trigger $t^* = \{t_i^*\}_{i=1}^m$ with m tokens and a target label $y^* \in \mathcal{Y}$. Specifically, the attacker first implants the trigger to a small number of samples $\mathcal{X}^*$, where $\mathcal{X}^* = \{\mathcal{S}^*, y^*\}$, $s^* = \{s_i\}_{i=1}^n \oplus \{t_j^*\}_{j=1}^m \in \mathcal{S}^*$. $\oplus$ denotes the trigger injection operation, which could be identifier renaming [13, 33, 49] or dead-code insertion [13, 25, 41]. Subsequently, the attacker constructs the poisoned dataset $\mathcal{X}_p = \{\mathcal{X} \cup \mathcal{X}^*\}$ using the triggered samples. Finally, the model will be poisoned by training with $\mathcal{X}_p$ and minimizing the following loss function:

$$\mathcal{L}_{\mathcal{X}_p}(\theta^*) = \underset{(s,y)\sim\mathcal{X}}{\mathbb{E}} \mathcal{L}\left(f_{\theta^*}(s), y\right) + \underset{(s^*,y^*)\sim\mathcal{X}^*}{\mathbb{E}} \mathcal{L}\left(f_{\theta^*}(s^*), y^*\right), \tag{2}$$

where $\mathcal{L}(\cdot)$ denotes the cross entropy loss. Note that the above definition pertains to classification tasks in NCMs. For another common code understanding task, the search task (e.g., code search), s can be a text sequence, such as a natural language query, and y can be the ground-truth code. Therefore, the attacker first selects or inserts a query containing the target word as $\mathcal{X}^*$, then implants the trigger into the corresponding code snippet as y^*, thereby constructing the poisoned sample $\mathcal{X}_p$. Then, the backdoor attack for search tasks also applies Equation (2) to train the model.

There are two types of triggers commonly used by backdoor attacks against NCMs. The first type is a statement trigger backdoor where the trigger is the fixed or grammar dead code statement or snippet injected in code. The second type is the identifier trigger where fixed or mixed tokens or words rename the identifiers (function name/variables) in the code snippet. The study [33] indicates

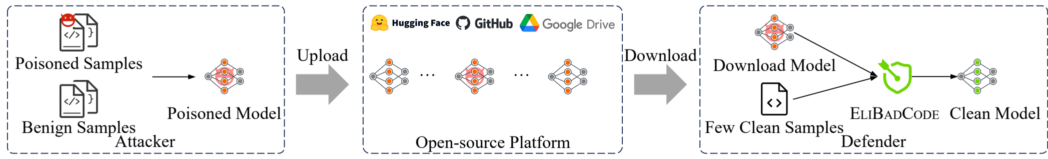

Fig. 1. Overview of our threat model.

that the token trigger is more stealthy than the statement trigger. The statement trigger can be detected by human or static analysis tools easily. Therefore, we focus on backdoor attacks with the token trigger, which have a more serious threat.

2.3 Backdoor Defenses

Based on the stage at which the defense occurs [46], existing backdoor defenses can mainly be divided into two categories: *pre-training defenses* and *post-training defenses*. As the name suggests, pre-training defenses aim to prevent models from being implanted with backdoors by detecting and filtering out poisoned samples from the training data before models are trained, such as Spectral Signature (SS) [38] used in [25, 27, 33, 41, 49], Activation Clustering (AC) [1] used in [27, 33, 41], CodeDetector [13], and KillBadCode [31]. Post-training defenses, on the other hand, occur after the model has already been implanted with a backdoor. Post-training defenses can be further subdivided into *input detection defenses*, which focus on detecting anomalous (trigger-injected) inputs to prevent the activation of backdoors in models, and *backdoor elimination defenses*, which aim to remove the backdoors from the models at their source. Currently, most backdoor defenses for NCMs are *input detection defenses* [10, 25]. These defenses perform outlier detection on each input sample or each word in the data to identify poisoned inputs and triggers. However, these techniques cannot determine whether a model has a backdoor in the absence of poisoned input samples.

In this paper, we consider *backdoor elimination defenses* for NCMs, which are to determine the backdoor and eliminate the identified backdoor without impacting the model's performance on clean inputs (i.e., clean accuracy) only given a small set of clean samples. Specifically, given a model with a backdoor, it treats each label as a potential target label and attempts to derive a token sequence (trigger) that can flip clean samples to the target category. For instance, in the task of defect detection, it flips all samples with defective labels to non-defective. For each label $y_i \in \mathcal{Y}$, it tries to find a trigger t_{y_i} to minimize the loss:

$$\mathcal{L}_{inv}(t_{y_i}, y_i, \theta^*) = \mathop{\mathbb{E}}_{s\sim\mathcal{X}'} \mathcal{L}(f_{\theta^*}(s \oplus t_{y_i}), y_i). \tag{3}$$

It is necessary to iterate over all possible labels above Equation (3) to invert the actual trigger t^* and target label y^*. Since for a backdoored model, it is easier to flip samples to the target label than to other labels [28]. Therefore, label y_i can be considered as target label, where $\mathcal{L}_{inv}(t_{y_i}, y_i, \theta^*) \ll \mathcal{L}_{inv}(t_{y_j}, y_j, \theta^*), \forall y_j \neq y_i \in \mathcal{Y}$. After determining the target label and the trigger, a standard method to eliminate the backdoor is model unlearning [42] that optimizes Equation (2) inversely as follows:

$$\arg\min_{\theta^*}[\mathop{\mathbb{E}}_{(s,y)\sim\mathcal{X}} \mathcal{L}(f_{\theta^*}(s), y) - \mathop{\mathbb{E}}_{(s^*,y^*)\sim\mathcal{X}^*} \mathcal{L}(f_{\theta^*}(s^*), y^*)]. \tag{4}$$

3 Threat Model

Figure 1 shows an overview of our threat model. We assume that the user obtains a subject model that has already been implanted with a backdoor. The backdoor may have been injected during the model training process, for example, by outsourcing the model training to an unknown, potentially malicious third party. Alternatively, the backdoored model may be released by an attacker on an

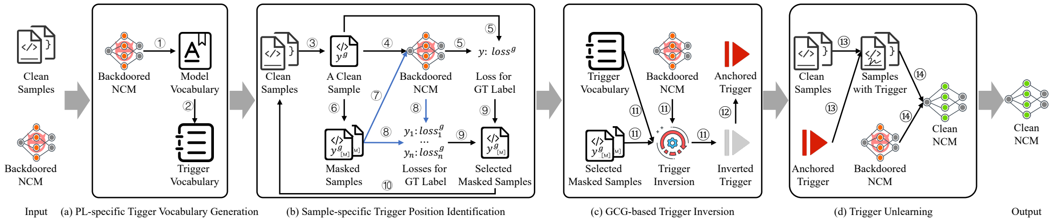

Fig. 2. Overview of EliBadCode.

open-source platform (such as GitHub, Hugging Face, and Google Drive) and downloaded by the user. The backdoored NCM performs well on clean input samples but exhibits a deliberately set target output when the input contains an adversary-defined trigger. Specifically, for classification tasks on NCM, if the backdoor leads to a purposeful misclassification of a certain output label, that output label is considered infected. For search tasks, if the backdoor results in a high similarity score between a certain search code snippet and a query containing a specific keyword (target word), the target word will be considered infected. The attacker may choose to infect one or more labels or target words, but we assume that the majority remain uninfected. Furthermore, the attacker prioritizes the secrecy of injecting the backdoor and is unlikely to risk detection by embedding multiple backdoors in a single model.

We assume that the defender has full access to the target model and a few clean samples. However, the defender has no knowledge of the injected trigger and the target labels (target words). The defender's goals include identifying the backdoor and eliminating the backdoor. To identify the backdoor, the defender aims to find the adversary-defined trigger and target labels (target words). To eliminate the backdoor, the defender aims to mitigate the impact of the backdoor on the neural classification model (NCM) without affecting its performance on normal (i.e., clean) inputs.

4 Methodology

4.1 Overview

Figure 2 presents an overview of EliBadCode. Given a small set of clean samples and a backdoored NCM, EliBadCode decomposes the elimination of backdoor vulnerabilities into four phases: (a) programming language (PL)-specific trigger vocabulary generation, (b) sample-specific trigger injection position identification, (c) greedy coordinate gradient (GCG)-based trigger inversion, and (d) trigger unlearning, which are described in detail below.

4.2 Programming Language (PL)-specific Trigger Vocabulary Generation

The core idea of EliBadCode is to search for a code token combination in the vocabulary space of the given backdoored NCM. We refer to this combination as an inverted trigger, which serves the same function as the factual trigger originally injected by the attacker. However, to enhance the NCM's comprehension ability and broad applicability, the model vocabulary of NCM is typically large, resulting in a vast search space for the inverted trigger. Moreover, a trigger may consist of multiple code tokens, which will cause the search space to increase exponentially. For example, the vocabulary size of the NCM CodeBERT [4] is 50,265, and if the trigger consists of n code tokens, the search space would be $50,265^n$, resulting in an incalculable search cost.

To reduce the search cost, the most direct and effective approach is to decrease the size of the model vocabulary. In fact, not all tokens can be used to form triggers. To enhance the stealthiness of backdoors based on identifier renaming, attackers typically design triggers by following the naming conventions of specific programming languages [13, 33]. This helps them evade poisoned

data detection methods based on syntax detection or static analysis. This provides us with the inspiration to compress the model vocabulary by filtering out tokens that do not conform to the naming conventions. The naming conventions we use are the mandatory constraints by PLs. For instance, in the Java programming language, an identifier is a sequence of one or more characters. The first character must be a valid first character (a letter, $, or _), and each subsequent character in the sequence must be a valid non-first character (a letter, digit, $, _) [11]. Violating these mandatory constraints results in syntax or compilation errors, and such code is typically excluded from the model's training data. Thus, even code that uses obfuscation or non-standard naming schemes must still adhere to these mandatory constraints. We do not require identifiers to follow widely recommended naming styles, such as camelCase, as these are best practices rather than strict constraints enforced by PLs. Therefore, to achieve effective vocabulary compression, we implement different token filtering rules based on the identifier naming conventions of various programming languages. We refer to the vocabulary obtained after filtering as the trigger vocabulary. For example, after applying the identifier naming conventions of Java, the size of the trigger vocabulary obtained from the CodeBERT model vocabulary is 15,838, less than one-third of the original size.

4.3 Sample-specific Trigger Position Identification

In trigger inversion-based backdoor defense techniques [18, 20, 28], it is common practice to transform the trigger search into an optimization problem to automate the search for the optimal inverted trigger. This optimization process requires simulating the trigger injection process, that is, injecting a randomly initialized trigger into the samples and then iteratively updating the trigger through model backpropagation. An important aspect to consider in this process is the injection position of the trigger, as it significantly affects the optimization efficiency. The model's sensitivity to changes at different positions varies across different samples. Specifically, the trigger optimization attempts to minimize the loss in Equation (3). This aligns with the objective of adversarial sample generation, which focuses on generating small perturbations in the input sample via optimization, leading to misclassification by clean models [39, 50]. Therefore, the trigger optimization is susceptible to the influence of *non-backdoor* (*adversarial*) *perturbations*. In other words, from the perspective of the attack target, backdoor attacks are similar to adversarial attacks in that both involve injecting certain patterns (triggers/perturbations) at specific positions in the sample to cause the model's predictions to change. Backdoor triggers can be regarded as a special kind of adversarial perturbations, which we refer to as *backdoor perturbations* in this paper. However, some positions can easily produce effective *non-backdoor perturbations*, yet these perturbations may not function as effective backdoor triggers (i.e., *backdoor perturbations*).

To reduce the interference of non-backdoor perturbations, we inject the trigger to be optimized in positions where the NCM is less sensitive. This is based on a key insight that backdoor triggers are more "robust" than non-backdoor perturbations. Figure 3 intuitively illustrates our insight, where the x-axis shows the injection position of the code pattern (i.e., backdoor trigger/non-backdoor perturbation) in a given code snippet, and the left y-axis presents the probability that the backdoored model predicts the backdoor trigger/non-backdoor perturbation-injected code snippet as the target label. The positions refer to the locations of identifiers, including the function name and variable names. We utilize the GCG algorithm [52] to generate a non-backdoor perturbation ("evalCodeoOpenraught") at the first position for the code snippet. Then, we inject this perturbation into different identifier positions of the code snippet and test the model's predictions, plotting the results as the blue line in Figure 3. Likely, we inject the factual backdoor trigger ("testo_init") into different identifier positions of the code snippet and test the model's predictions, plotting the results as the red line in Figure 3. Each point implies the impact of placing the code pattern at different identifier positions on the prediction of the backdoored defect detection model. Both

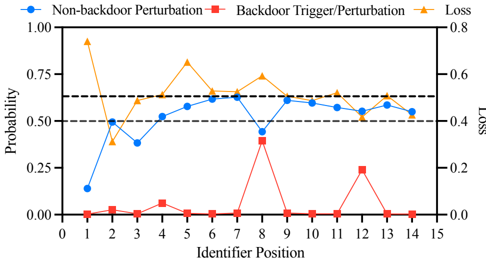

Fig. 3. Effect of injecting code pattern (i.e., *non-backdoor perturbations* and backdoor triggers) at different code identifier positions on the prediction of the backdoored defect detection model. A probability less than 0.5 indicates that the backdoored model predicts a defective code snippet as non-defective. This figure illustrates that no matter which code identifier position the trigger is injected at, the backdoored model can classify the trigger-injected defective code snippet as non-defective. However, the backdoored model classifies the *non-backdoor perturbation*-injected defective code snippet as non-defective only when the *non-backdoor perturbation* is injected at certain positions, e.g., the 1st, 3rd, and 8th identifier positions.

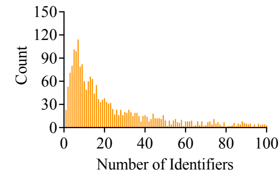

Fig. 4. Distribution of the number of identifiers (trigger insertion positions) contained in clean samples.

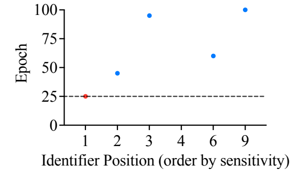

Fig. 5. Trigger inversion costs when injecting the trigger into positions with different sensitivities.

non-backdoor perturbations and backdoor triggers target the label "non-defective", meaning that if "Probability" is less than 0.5, the attack is successful. This figure shows that only when the non-backdoor perturbation is injected at certain positions (e.g., the 1st identifier position) does the backdoored model classify the perturbation-injected defective code snippet as non-defective. In contrast, the backdoored model classifies the defective trigger-injected code snippet as non-defective, regardless of where the backdoor trigger is injected. It means that the robustness of the backdoor trigger is higher than that of the non-backdoor perturbation. In other words, the backdoored model is very sensitive to the backdoor trigger, regardless of its injection position. Therefore, intuitively, we can inject randomly initialized triggers at any identifier position for optimization. However, the backdoored model is not sensitive to non-backdoor perturbations, but injecting the randomly initialized trigger at certain positions is more likely to optimize effective non-backdoor perturbations rather than effective backdoor triggers. Therefore, if we can identify which positions are more likely to produce non-backdoor perturbations, we can inject the randomly initialized trigger into positions other than these to exclude the interference of non-backdoor perturbations, thereby improving trigger optimization efficiency.

To this end, we investigate the sensitivity of the backdoored model to changes at each identifier position in the code. Specifically, we analyze the model's sensitivity to each identifier position by masking each position in the code snippet and then calculating the loss value for predicting the masked code snippet as the ground-truth label. In Figure 3, the black dashed line represents the loss value of the backdoored model predicting the original code snippet as the ground-truth label. We also plot the loss values of the backdoored model predicting each masked code snippet as the ground-truth label as the orange line in Figure 3. The larger the change in loss value (the farther the orange triangle is from the black dashed line), the more sensitive the model is to the variation at that identifier position. From Figure 3, it is observed that sensitive identifier positions, such as the 1st, 2nd, and 8th identifier positions, are likely to produce effective non-backdoor perturbations. Compared to non-backdoor perturbations, the generation of the effective backdoor trigger is less correlated with the sensitivity of each position. Therefore, we can inject the randomly

initialized trigger in insensitive identifier positions for optimization to reduce the probability of generating effective non-backdoor perturbations instead of effective backdoor triggers during the optimization process, thus improving trigger optimization efficiency. For instance, Figure 4 shows the distribution of the number of identifiers in code snippets of all clean samples. Observe that the number of identifiers in different code snippets varies, with most code snippets containing only a few identifiers. We experiment with optimizing the randomly initialized trigger injected into the top-ranked less sensitive positions covering the majority of code snippets. The backdoored defect detection model involved in the experiment is built on CodeBERT, and the experimental results are shown in Figure 5. Observe that injecting randomly initialized triggers at the least sensitive positions of each code snippet requires only 25 epochs to optimize an effective backdoor trigger, while more sensitive positions require more epochs. Some positions, such as the 4th least sensitive position from the end, do not even yield an effective trigger after 100 epochs of searching.

Based on the above observations, we design a sample-specific method for identifying trigger (injection) positions. As shown in Figure 2(b), given a set of clean samples, EliBadCode iteratively identifies specific trigger injection positions for each sample (Steps ③–⑩). Specifically, given a sample $x := \langle s, y \rangle$ where s is a code snippet and y is the ground-truth (GT) label, EliBadCode feeds s to the backdoored NCM, which outputs the predicted loss values for different labels (④). Combining the GT label y of s, EliBadCode can obtain the predicted loss value for y, denoted as $loss^g$ (⑤). Then, EliBadCode produces a set of masked samples $\{x_1^m, x_2^m, \ldots, x_n^m\}$ by masking each identifier position of s (⑥). $x_i^m := \langle s_i^m, y_i \rangle$, $y_i \equiv y$, denotes that the masking operation is to replace the i-th identifier of s with the special token "<unk>", and only one position of each masked sample is replaced. Like the clean sample, the masked code snippet s_i^m of each masked sample will be fed to the backdoored NCM to obtain the corresponding prediction loss value for the y_i, denoted as $loss_i^g$ (⑦–⑧). After that, EliBadCode calculates the difference value d_loss_i between $loss^g$ and each $loss_i^g$, i.e., $d_loss_i = |loss^g - loss_i^g|$ (⑨). Smaller d_loss values indicate that the backdoored NCM is less sensitive to changes in that position. For each clean sample, we select the masked sample that has the smallest d_loss value with the clean sample, because the inverted trigger at the masked position in this sample is resistant to adversarial perturbations' interference. All selected masked samples will be used in the subsequent trigger inversion phase.

4.4 GCG-based Trigger Inversion

The Greedy Coordinate Gradient (GCG) proposed by Zou et al. [52] is used to search for an adversarial suffix onto the user prompt, which is intended to induce LLMs to respond to the user's original, potentially harmful, request, i.e., producing undesirable behavior. Such a suffix can also be viewed as an adversarial perturbation. In our scenario, EliBadCode aims to search for a backdoor trigger/perturbation injected into the code snippet, which is intended to induce the backdoored NCM to produce target behaviors. Therefore, we borrow GCG to implement trigger inversion. As mentioned in Section 4.3, not every adversarial perturbation is an attacker-crafted backdoor trigger. Our goal is to eliminate backdoor triggers rather than non-backdoor perturbations. Therefore, unlike [52] which does not need to care about the suffix injection position, EliBadCode injects the trigger into positions where the NCM is less sensitive (i.e., the masked identifier positions in the selected masked samples), to reduce the interference of non-backdoor perturbations.

Algorithm 1 illustrates the GCG-based trigger inversion of EliBadCode in detail. In addition to the selected masked samples (X^m), trigger vocabulary (V), and a backdoored NCM (f_{θ^*}) as shown in Figure 2, EliBadCode takes as input the labels (Y) and some key settings including times of iterations (ϵ), the number of candidate substitutes (k), times of repeat (r), and the threshold for trigger anchoring (β). To eliminate backdoors in NCMs, EliBadCode first obtains the possible

Algorithm 1 GCG-based Trigger Inversion

INPUT:	X^m, Y	selected masked samples, labels
	V, f_{θ^*}	trigger vocabulary, backdoored NCM
	ϵ, k, r	the times of iterations, the number of candidate substitutes, the times of repeat, respectively
	β	the threshold for trigger anchoring
OUTPUT:	t^*	anchored trigger

1: **function** TRIGGERINVERSION(S^m, y')
2: $t \leftarrow$ randomly initialize a trigger with n tokens from V
3: $\boldsymbol{e}_{S^m} \leftarrow$ produce embeddings of codes in S^m using f_{θ^*}
4: **for** $z = 0, z < \epsilon$, z++ **do**
5: $o_t \leftarrow$ generate the one-hot representation of t
6: $\boldsymbol{e}_t \leftarrow$ produce o_t's embeddings using f_{θ^*}
7: $\boldsymbol{e}'_{S^m} \leftarrow \boldsymbol{e}_{S^m} \oplus \boldsymbol{e}_t$
8: $G \leftarrow \nabla o_t \mathcal{L}(f_{\theta^*}(\boldsymbol{e}'_{S^m}), y')$
9: $\mathcal{T} \leftarrow$ select substitutes for each trigger token based on the top-k gradients of o_t in G
10: $t^C \leftarrow \emptyset$ ▷ store candidate substitute triggers
11: **for** $j = 1, j < r, j++$ **do**
12: $t^j \leftarrow t$
13: $i \leftarrow$ randomly select a position to be replaced in t^j
14: $\mathcal{T}_i \leftarrow$ get all substitutes for the i-th token of t^j
15: $t_i^j \leftarrow$ randomly select a substitute from $\mathcal{T}_i$
16: $t^j \leftarrow$ replace the i-th token of t^j with t_i^j
17: $t^C \leftarrow t^C \cup t^j$
18: **end for**
19: $t \leftarrow t_j^C, j = \arg\min_j \mathcal{L}(f_{\theta^*}(S^m \oplus t_j^C), y'), j \in [1, r]$
20: **end for**
21: $l \leftarrow \mathcal{L}(f_{\theta^*}(S^m \oplus t), y')$
22: **return** t, l
23: **end function**
24: **function** TRIGGERANCHORING(S^m,t, y^*)
25: $t^* \leftarrow \emptyset$
26: $l \leftarrow \mathcal{L}(f_{\theta^*}(S^m \oplus t), y^*)$
27: **for** each token t_i **in** t **do**
28: $l_i \leftarrow \mathcal{L}(f_{\theta^*}(S^m \oplus (t \setminus t_i)), y^*)$
29: **if** $|l - l_i| > \beta$ **then**
30: $t^* \leftarrow t^* \cup t_i$
31: **end if**
32: **end for**
33: **return** t^*
34: **end function**
35:
36: $l^C \leftarrow \emptyset$ ▷ store inverted target labels
37: $t^C \leftarrow \emptyset$ ▷ store inverted triggers
38: **for** each label y' **in** Y **do**
39: $S^m \leftarrow$ get masked code snippets in X^m according to y'
40: $t, l \leftarrow$ TRIGGERINVERSION(S^m, y')
41: $l^C \leftarrow l^C \cup l$
42: $t^C \leftarrow t^C \cup t$
43: **end for**
44: $y^*, t \leftarrow$ run the outlier detection on l^C and t^C to detect the target label y^* and the corresponding trigger t
45: $t^* \leftarrow$ TRIGGERANCHORING(S^m, t, y^*)
46:
47: **Output** t^*, y^*

target label y' from Y and gets masked code snippets S^m with the label y' from X^m, then invokes the TRIGGERINVERSION function (lines 38–40). Then, in the TRIGGERINVERSION function, ELIBADCODE first randomly initializes a trigger (t) with n tokens using V (line 2), and then transforms code snippets in S^m into vector representations (also called embeddings) $\boldsymbol{e}_{S^m}$ using the embedding layer of f_{θ^*} (line 3). Based on $\boldsymbol{e}_{S^m}$, it further iteratively optimizes t ϵ times (lines 4–20). During each iteration, ELIBADCODE first generates the one-hot representation of t, denoted as o_t (line 5). Second, it produces the embeddings of o_t using f_{θ^*}, denoted as $\boldsymbol{e}_t$ (line 6). Third, it injects $\boldsymbol{e}_t$ into $\boldsymbol{e}_{S^m}$ to produce the embeddings of trigger-injected masked code snippets, denoted as $\boldsymbol{e}'_{S^m}$ (line 7). Forth, it feeds $\boldsymbol{e}'_{S^m}$ to f_{θ^*} to compute gradients for o_t, denoted as G (line 8). Fifth, based on the top-k negative gradients of each trigger token in G, it selects substitutes for all trigger tokens in t, denoted as $\mathcal{T}$ (line 9). Based on $\mathcal{T}$, it generates a set of candidate triggers T^C by repeating r times, each time randomly replacing one token in t with a random substitute in $\mathcal{T}$ (lines 10–18). Sixth, it injects each candidate trigger into S^m, calculates the loss values l of f_{θ^*} predicting the trigger-injected code snippets as y', and selects the candidate trigger resulting in the smallest loss value as the inverted trigger (line 19). Finally, it calculates the loss value l about the inverted trigger t and the possible target label y' and returns them (lines 21–22). After iterating over all possible target labels and producing a set of loss values and the corresponding inverted triggers, one for each label. ELIBADCODE runs the outlier detection method [42] to obtain the ground-truth target label y^* and the corresponding inverted trigger t. Next, y^* and the corresponding t will be input into the TRIGGERANCHORING function to obtain the effective components in t (line 45).

Note that, unlike continuous image data, code written in PL is similar to natural language and is discrete. Existing research [20] in NLP has demonstrated that for discrete inputs, there is currently no simple method to differentiably determine the size/length of the injected trigger. Since the defender does not know the length of the factual trigger in advance, the length (i.e., number of

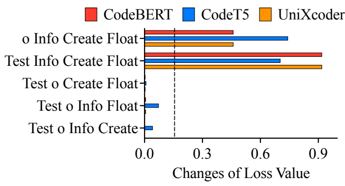

Fig. 6. Loss value changes caused by vary trigger tokens.

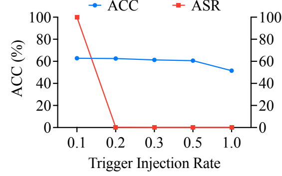

Fig. 7. Influence of different trigger injection rates.

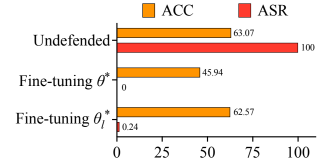

Fig. 8. Effectiveness of trigger unlearning.

tokens) of the randomly initialized trigger in the trigger inversion process may be larger than the factual trigger. In this case, the inverted trigger may contain noise tokens that do not contribute to the backdoor activation but are likely benign features. Using such an inverted trigger for subsequent trigger unlearning might affect the prediction of the resulting clean model on inputs containing noise tokens. However, GCG itself is not capable of solving this problem because the work [52] only requires GCG to find non-backdoor adversarial perturbations that can successfully attack the LLM, without considering whether the perturbations contain noise components.

To address this issue, ELIBADCODE designs a trigger anchoring method that filters out noise tokens in the inverted trigger, retaining only the effective components. Specifically, as shown in lines 24 – 34 of Algorithm 1, ELIBADCODE iteratively removes one trigger token at a time, and the remaining tokens form the filtered trigger. The filtered trigger is then injected into the masked code snippets (⑫). Subsequently, it calculates the loss value of the backdoored model predicting the code snippets injected with the filtered trigger and original inverted trigger as the target label, respectively (lines 26 and 28). If the removal of a trigger token causes the loss value to change by more than a given threshold β, ELIBADCODE identifies it as an effective trigger component and adds it to the anchored trigger (lines 29–31). The threshold β is an empirical value. To find a suitable β value, we analyze the distribution of loss value changes caused by effective trigger tokens. Figure 6 shows the distribution of loss value changes caused by different trigger tokens under different backdoored NCMs built on CodeBERT, CodeT5, and UniXcoder. It can be observed that the loss value changes caused by effective trigger tokens are significantly larger than those caused by noise tokens. In this paper, we uniformly set β to 0.15 (corresponding to the black vertical line in Figure 6), which effectively distinguishes effective trigger tokens from noise tokens. Finally, ELIBADCODE outputs the anchored trigger t^* and target label y^*, and the algorithm ends (line 47).

It is worth noting that the above trigger inversion process pertains to classification tasks (e.g., defect detection and clone detection) in code understanding. For search tasks in SE (e.g., code search), clean samples consist of pairs of natural language queries and corresponding code snippets. Therefore, the inversion process for search tasks requires the additional inversion of an attack target (usually one word/token [33, 41]) related to the query and the trigger inversion process for the code is similar. Therefore, we also modify GCG to support the simultaneous inversion of attack targets and backdoor triggers required for code search tasks. Specifically, a target w consisting of m tokens needs to be initialized, and lines 3 – 20 in Algorithm 1 is executed similarly, focusing on w. In the meantime, the loss value calculation involving y' needs to be updated to the loss value related to the query. For example, line 8 is updated to $G = \nabla o_t, o_g \mathcal{L}(f_{\theta^*}(\boldsymbol{e}'_{Sm}), \boldsymbol{e}'_Q)$, where o_w and $\boldsymbol{e}'_Q$ represent the one-hot representation of w and the embeddings of the target-injected queries, respectively. Due to the page limit, we introduce the detailed trigger inversion algorithm for the code search task in our anonymous project website [32].

4.5 Trigger Unlearning

Trigger unlearning primarily involves using the model unlearning approach [28, 42] to disrupt the association or mapping between the trigger and the target behavior. In practice, the defender is unaware of the trigger the attacker sets. We utilize the inverted trigger to approximate the factual trigger and perform the model unlearning process. Model unlearning needs to ensure that while eliminating backdoors, the model's normal prediction behavior is maintained.

To achieve effective and efficient model unlearning, as shown in Figure 2(d), we first inject the anchored trigger into code snippets of clean samples and assign the inverted label to these code snippets, to construct the unlearning training dataset $\mathcal{X}'$ (⑬). Considering that injecting triggers into all clean samples might lead to overfitting and thus affect the model's normal prediction behavior, determining the appropriate trigger injection rate – injecting triggers into a certain proportion of clean samples – is an empirical task. To find the suitable rate, we conduct multiple experiments, with the results shown in Figure 7. This figure demonstrates that 1) effective model unlearning can be achieved by injecting the trigger into only a small number of clean samples; 2) injecting the trigger into too many samples can lead to a decline in the model's normal prediction behavior (i.e., ACC). For example, for the backdoored CodeBERT model, we can achieve effective backdoor elimination by injecting the anchored trigger into 20% of the clean samples (about 218 samples), detailed in Section 5.3. Then, we conduct model unlearning by fine-tuning the backdoored NCM with $\mathcal{X}'$ (⑭). Considering that existing work [12] finds that fine-tuning all parameters of the backdoored model with a small set of clean samples can lead to catastrophic forgetting (i.e., severely compromising the model's clean accuracy). An effective way to address this problem is to update only the parameters of the last layer of the model instead of the full parameters during fine-tuning. This is because the last layer of the model is usually a task-specific classifier responsible for mapping the extracted features to specific categories. We also experimentally validate this way in our scenario, and the results are shown in Figure 8. In this figure, Fine-tuning θ^* and Fine-tuning θ_l^* respectively mean fine-tuning the full parameters θ^* of the backdoored defect detection model and the last layer parameters (denoted as θ_l^*) when executing trigger unlearning. Observe that compared to fine-tuning θ^*, fine-tuning only θ_l^* can achieve the elimination of the backdoor without compromising the model's prediction accuracy.

Based on the above, the trigger unlearning is conducted by minimizing the loss, which is computed as Equation (4) with the anchored trigger t^* and inverted target label y^*. And note that we only update θ_l^*, which represents the parameters of the last layer of the backdoored NCM model.

5 Evaluation

We conduct a series of experiments to answer the following research questions (**RQs**).

RQ1. How effective is ELIBADCODE in eliminating backdoors in NCMs?

RQ2. What is the contribution of key designs in ELIBADCODE, including PL-specific trigger vocabulary generation, sample-specific trigger position identification, and trigger anchoring?

RQ3. What is the influence of important settings on ELIBADCODE, including the number of clean samples, the times of iterations ϵ, the number of candidate substitutes k, and the times of repeat r?

RQ4. What is the performance of ELIBADCODE against adaptive attacks?

5.1 Experiment Setup

Datasets and Models. The evaluation is conducted on the widely used dataset CodeXGLUE [21]. Specifically, we utilize BigCloneSearch [37], Devign [51], and CSN-Python [9] to evaluate ELIBADCODE on three types of code understanding tasks: clone detection, defect detection, and code

search, respectively. Three different model architectures are adopted for the evaluation, CodeBERT [4], CodeT5 [44] and UniXcoder [6], which are widely used in the existing attacks against NCMs [13, 33, 49].

Attack Setting. We leverage three advanced backdoor attacks, CodePoisoner [13], BadCode [33], and AFRAIDOOR [49], to generate backdoored NCMs built on the three model architectures for the three code understanding tasks. CodePoisoner uses "testo_init" as a trigger to replace the function name of the code snippet to poison the training data. BadCode utilizes "rb" as a trigger and appends it to the function name/variable name of the code snippet to produce the poisoned training data. AFRAIDOOR utilizes average gradient computation to generate adversarial perturbations, which are used as triggers to poison the training data. For the defect detection task and clone detection task, we select non-defective and non-clone as the target labels, respectively. For the code search task, we follow BadCode and choose "file" as the target word, implanting the trigger into the code snippets matched by queries containing the target word. Code comments are usually used as queries in experiments [33, 41]). We follow Li et al. [13] and poison 2% of the training data for different code understanding tasks. The poisoned training data is utilized for model fine-tuning to produce backdoored NCMs, with the fine-tuning parameter settings consistent with those of fine-tuning the clean model.

Defense Setting. For trigger inversion (including the phases (b) and (c) in Figure 2), we use 30 samples per class in the defect detection task and clone detection task, and 30 samples in the code search task (details on the effectiveness of different numbers of clean samples can be found in Section 5.3). Considering that attackers prioritize the stealthiness of the backdoor, they typically do not set a long trigger for renaming backdoor attacks. Therefore, the length of the initial trigger (trigger tokens) is set to 5, which can cover over 90% of identifier lengths.Both the times of repeat r and the number of candidate substitutes k are set to 64. In trigger unlearning, we fine-tune the backdoored models to unlearn the backdoors. We use all clean samples (i.e., 10% of the training data) and select 20% of them to inject the inverted trigger and mark with the correct labels. The effectiveness before and after unlearning is evaluated on the whole test set of different datasets.

Baselines. As mentioned in Section 2.3, our ELIBADCODE is a post-training defense and aims to eliminate backdoors in NCMs. To the best of our knowledge, no current research has proposed effective backdoor elimination techniques against backdoor attacks on NCMs. In addition, the significant difference between CV (Computer Vision) and PL data characteristics (continuous vs. discrete) makes it challenging to directly transfer CV defenses. Therefore, we transfer the following three advanced post-training backdoor defenses from NLP as baselines.

ONION. ONION [23] is an input detection defense. It removes the words that are probably the backdoor trigger (or part of it) from inputs, to prevent activating the backdoor of a backdoored model. Given an input, it adopts an iterative approach by removing each word in the input one-at-a-time and calculating the perplexity (PPL) change using an external language model GPT-2 [24]. Considering that unlike the trigger in NLP which is composed of words, the trigger in PL typically consists of code tokens, we adapt ONION to a pre-training defense for PL code, and utilize CodeLlama-7B [26] (a renowned open-source language model specialized for code) to detect outlier tokens.

DBS. DBS [28] is a backdoor elimination defense. It defines a convex hull to address the non-differentiability issue of the language models, and features temperature scaling and backtracking to step away from local optima. We apply our PL-specific Trigger Vocabulary Generation to DBS. However, the effectiveness of DBS was not satisfactory. Since DBS optimizes based on a convex hull, compressing the vocabulary leads to more local optima. Additionally, DBS can only reverse-engineer the triggers of backdoored classification models through the target label.

AttDef. AttDef [14] is an attribution-based defense method against insertion-based textual backdoor attacks. It assumes that trigger words may play an important role in sentences if inserting

them would make the model flip the prediction. Given an input, like ONION, AttDef first utilizes an external pre-trained language model to distinguish whether the input is poisoned or not. If so, the sample will be further fed into the trigger detector to identify the trigger words, followed by a mask sanitization to mask the trigger words. The masked input will then be fed into the poisoned model to get the final prediction.

5.2 Evaluation Metrics

We leverage two kinds of metrics in the evaluation, including attack/defense metrics and task-specific performance metrics.

Attack/Defense Metrics. For defect detection and clone detection, we follow [13] and utilize *attack success rate* (ASR) to evaluate the effectiveness of attack/defense techniques. ASR represents the proportion of the backdoored model successfully predicting inputs with triggers as the target label and is computed as $ASR = \frac{N_{flipped}}{N_{non-target}} \times 100\%$, where $N_{\text{non-target}}$ and N_{flipped} represent the number of non-target label samples and the number of samples predicted as the target label after adding the trigger to non-target label samples, respectively. In our experiments, we follow Li et al. [13] to pre-define "non-defective" and "non-clone" as the target labels for defect detection tasks and clone detection tasks, respectively. After defense, the lower the ASR value, the better.

For code search, we follow [33, 41] and utilize *average normalized rank* (ANR) as the attack/defense metric. ANR is computed as $ANR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{Rank(Q_i, s')}{|S|}$, where $|Q|$ denotes the size of query set, s' represents the code snippet of the injection trigger, and $|S|$ is the length of the complete sorted list. In our experiment, we follow Sun et al. [33] to attack the code snippets initially ranked in the top 50% of the returned list. After defense, the higher the ANR value, the better.

Task-specific Accuracy Metrics. Task-specific performance metrics are related to specific tasks and are used to evaluate the (backdoored/clean) model's normal performance on clean data. For defect detection and clone detection, we follow [4, 21] and respectively use *accuracy* (ACC), *F1-score* (F1) and *mean reciprocal rank* (MRR) to evaluate the prediction accuracy of the model.

5.3 Evaluation Results

5.3.1 RQ1: Effectiveness of ELIBADCODE in eliminating backdoors.

Effectiveness of ELIBADCODE in eliminating backdoors. Table 1 shows the performance of the three baselines and our ELIBADCODE in eliminating backdoors in 27 NCMs. Columns titled "Undefended" display the performance of the 27 backdoored NCMs without any defense. Observe that for the attack CodePoisoner, on the defect detection and clone detection tasks, after applying ONION, the ASR remains high, ranging from 76.98% to 90.00% depending on the different models and tasks. On the code search task, ONION can increase the ANR to 18.25-21.56 and is better than AttDef (16.57-18.75). DBS has almost no effect in removing backdoors in 27 NCMs. Although AttDef significantly reduces the ASR in certain attack scenarios, it also greatly compromises the model's normal performance. For example, on the defect detection task and CodeT5 architecture, AttDef can reduce the ASR to 18.64%, but it also lowers the ACC to 56.37%. It can also be observed that existing defenses excel at defending against different types of attacks. For example, AttDef and DBS perform comparably on token-level trigger-based attacks such as BadCode, outperforming ONION overall. However, ONION performs better than DBS on the identifier-level trigger-based attack CodePoisoner. Compared with the three baselines, on the defect detection task, our ELIBADCODE can significantly reduce the ASR to 0.15%-2.39%, depending on the different model architectures, while maintaining the ACC without a noticeable decline. Similarly, on the code clone task, ELIBADCODE can significantly decrease the ASR to 3.16%-5.86% while maintaining a high F1. On the code search task, ELIBADCODE can increase the ANR from 9.11-10.04 to 24.76-25.12, outperforming ONION and

Table 1. Comparison of backdoor elimination performance. DD: Defect Detection; CD: Clone Detection; CS: Code Search. Column "Undefended" shows the performance of the backdoored model.

Attack	Task	Metric	CodeBERT					CodeT5					UniXCoder				
			Undefended	ONION	DBS	AttDef	EliBadCode	Undefended	ONION	DBS	AttDef	EliBadCode	Undefended	ONION	DBS	AttDef	EliBadCode
CodePoisoner	DD	ACC	63.07%	62.57%	62.99%	57.62%	62.57%	64.06%	63.96%	63.05%	56.37%	63.25%	65.30%	65.13%	64.17%	58.46%	64.39%
		ASR	100%	90.00%	100%	17.98%	0.24%	98.64%	87.02%	98.13%	18.64%	0.15%	98.48%	86.43%	98.33%	16.23%	2.39%
	CD	F1	93.37%	93.55%	96.41%	90.67%	96.53%	94.58%	93.73%	96.24%	89.44%	96.21%	94.51%	94.15%	97.11%	90.78%	97.37%
		ASR	100%	78.79%	100%	60.23%	5.86%	100%	79.09%	100%	61.77%	3.16%	100%	76.98%	100%	63.21%	5.17%
	CS	MRR	0.81	0.78	–*	0.80	0.81	0.81	0.79	–*	0.80	0.81	0.82	0.80	–*	0.80	0.82
		ANR	10.04	20.60	–*	18.75	25.12	9.50	18.25	–*	16.57	24.76	9.11	21.56	–*	18.26	24.98
BadCode	DD	ACC	62.88%	62.00%	61.75%	57.56%	61.86%	63.72%	63.03%	62.85%	58.03%	62.91%	64.71%	63.98%	64.06%	58.63%	62.98%
		ASR	99.52%	90.36%	32.59%	20.74%	1.95%	99.92%	89.12%	60.80%	21.09%	3.27%	99.84%	88.61%	45.74%	20.56%	2.71%
	CD	F1	93.46%	93.54%	96.62%	90.53%	96.69%	93.97%	93.65%	96.12%	90.66%	96.03%	94.68%	94.06%	97.06%	89.78%	97.02%
		ASR	100%	77.65%	53.20%	48.34%	8.13%	100%	78.57%	87.73%	49.67%	5.19%	100%	79.03%	50.10%	48.17%	5.00%
	CS	MRR	0.81	0.79	–*	0.80	0.80	0.81	0.78	–*	0.80	0.81	0.82	0.80	–*	0.80	0.81
		ANR	10.56	20.16	–*	18.87	25.69	10.25	21.90	–*	18.03	24.77	9.17	19.78	–*	17.86	25.09
AFRAIDOOR	DD	ACC	61.74%	61.08%	61.56%	57.43%	61.37%	62.07%	61.98%	61.78%	57.34%	61.21%	62.75%	62.48%	62.33%	58.07%	62.09%
		ASR	96.05%	90.23%	60.37%	33.74%	9.71%	95.31%	90.12%	57.40%	34.73%	10.55%	96.43%	89.40%	60.32%	36.89%	10.58%
	CD	F1	91.36%	91.07%	93.30%	88.49%	93.53%	90.56%	90.17%	92.05%	87.53%	92.58%	91.57%	92.20%	93.14%	88.47%	93.41%
		ASR	94.76%	85.73%	65.73%	58.06%	15.11%	93.12%	84.21%	63.77%	57.04%	16.20%	95.28%	84.37%	62.07%	57.98%	14.08%
	CS	MRR	0.81	0.77	–*	0.80	0.80	0.81	0.78	–*	0.81	0.81	0.82	0.79	–*	0.81	0.81
		ANR	11.01	21.67	–*	17.43	25.21	10.30	19.07	–*	17.47	24.20	9.16	20.08	–*	18.32	25.53

* DBS needs to iterate all possible target labels to invert the trigger and eliminate the backdoor. However, for code search, its label can be considered as the target word, which has many possible combinations (different combinations of vocabulary tokens). Therefore, it does not work on code search tasks.

AttDef while maintaining the same average MRR. The backdoor attacks in NCMs for code search tasks aim to improve the ranking of the code snippet with the trigger given a query containing the target word. It is important to note that an ANR of 9.11 indicates that the backdoored model can elevate a (potentially malicious) code snippet injected with a trigger from its original rank at the 50% position to the 9.11% position. Assuming there are 100 candidate code snippets, 9.11% means that the trigger-injected code snippet would be ranked in the 10th position. In existing code search techniques, it is common practice to return the top 10 retrieved code snippets [33]. Therefore, code snippets ranked in the top 10 are likely to be adopted by developers. Once the malicious trigger-injected code snippet is adopted and integrated into their projects, it poses serious security risks. Although EliBadCode does not increase the 9.11% back to the original 50% position, it significantly reduces the risk of developers adopting the malicious trigger-injected code snippet.

For attacks BadCode and AFRAIDOOR, in all attack scenarios, EliBadCode continues to demonstrate the same excellent backdoor elimination capabilities and ability to maintain normal performance as it does for attack CodePoisoner, outperforming the two baselines. For all three attacks, we can observe that on clone detection tasks, the F1 scores of NCMs after removing backdoors are even higher than those of the backdoored NCMs without any defense. This is because fine-tuning NCMs with 10% of the trigger-injected training dataset does not negatively impact NCMs' normal performance; rather, the increased training data and process enhance the NCMs' effectiveness.

We conduct additional statistical tests to evaluate the differences between EliBadCode and the best baseline DBS. Using Prism software [29], we compare the ASR results of DBS and EliBadCode across different tasks, models, and backdoor attack scenarios (a total of 18 comparisons). For each comparison, we perform an unpaired Wilcoxon-Mann-Whitney test [47] on all ASR scores for DBS and EliBadCode at a significance level of 5%. The results show that all p-values are < 0.0001, indicating that EliBadCode significantly outperforms DBS.

In addition, considering part of our approach involves a fine-tuning to backdoored NCMs to make them forget the mapping between triggers and target labels. We further compare the effect of such a fine-tuning against that of a "generic" fine-tuning involving the same amount of data/training time to highlight the contribution given by the model unlearning. Specifically, we fine-tune poisoned

Table 2. Impact of the generic fine-tuning under the attack CodePoisoner.

Attack	Task	Metric	CodeBERT	CodeT5	UniXCoder
CodePoisoner	Defect Detection	ACC	63.54%	64.12%	65.34%
		ASR	98.80%	92.83%	92.25%
	Clone Detection	F1	96.56%	96.24%	97.34%
		ASR	98.17%	96.23%	96.52%
	Code Search	MRR	0.81	0.81	0.81
		ANR	15.15	16.17	15.22

Table 3. Impact of the generic fine-tuning under the attack BadCode.

Attack	Task	Metric	CodeBERT	CodeT5	UniXCoder
BadCode	Defect Detection	ACC	63.10%	64.04%	65.17%
		ASR	98.13%	96.23%	96.17%
	Clone Detection	F1	96.53%	96.30%	97.31%
		ASR	98.24%	96.71%	96.43%
	Code Search	MRR	0.81	0.81	0.81
		ANR	15.78	16.11	15.38

Table 4. Performance on clean models. Column "Clean" shows the performance of the original clean model.

Task	Metric	CodeBERT			CodeT5			UniXCoder		
		Clean	DBS	ELIBADCODE	Clean	DBS	ELIBADCODE	Clean	DBS	ELIBADCODE
Defect Detection	ACC	62.41%	62.19%	62.30%	64.17%	63.01%	63.18%	65.56%	64.27%	64.67%
Clone Detection	F1	93.34%	95.65%	96.17%	94.67%	96.43%	96.18%	95.14%	97.29%	97.23%
Code Search	MRR	0.81	–*	0.81	0.81	–*	0.81	0.82	–*	0.82

NCMs using clean samples (i.e., without inverted backdoor triggers) while adhering to the same settings as the model unlearning (i.e., using 10% of the training data and identical hyperparameters). The experimental results shown in Table 2 and Table 3 demonstrate that the generic fine-tuning with clean data alone is ineffective in mitigating backdoors in backdoored NCMs. For instance, in the Defect Detection task, the ASR of the fine-tuned NCMs remains above 90%, indicating that backdoor behavior persists.

Performance of ELIBADCODE on the clean NCMs. As mentioned in Section 4.4, after iterating over all possible target labels and producing a set of loss values and associated inverted triggers, one for each label. ELIBADCODE runs the outlier detection to obtain the ground-truth target label and associated inverted trigger. If the outlier detection yields a result, it indicates that the model is backdoored; otherwise, it is clean. Therefore, it is necessary to ensure that when the input is a clean model, ELIBADCODE can identify it and does not negatively impact its normal performance.

To investigate the performance of ELIBADCODE on clean NCMs, we test it on 24 models (6 clean and 18 backdoored) to assess its ability to distinguish between clean and backdoored models. Our results show that the false positive rate of DBS is 50%, while our ELIBADCODE does not produce false positives and can effectively differentiate between clean and backdoored models. In addition, we utilize the inverted trigger on the clean NCM to perform trigger unlearning, and the normal performance of the NCM results are shown in Table 4. From this table, it is observed that compared with the performance of the original clean model, the normal performance of the model fine-tuned with the inverted trigger dataset does not show a significant drop. This indicates that regardless of whether the input NCM is clean or backdoored, even if ELIBADCODE's outlier detection is inaccurate, the trigger unlearning does not negatively impact the model's normal performance.

5.3.2 *RQ2: Contribution of key designs in ELIBADCODE.*

Table 5 presents the performance of ELIBADCODE on CodeBERT under the CodePoisoner attack with different designs. Rows 2–4 represent the performance of ELIBADCODE without phase (a): PL-specific trigger vocabulary generation, phase (b): sample-specific trigger position identification, and trigger anchoring in phase (c), respectively. Observe that without phase (a), the optimization search space expands, making it unable to invert the trigger close to the factual trigger. Therefore,

Table 5. Ablation study. TA: Trigger Anchoring.

Method	ACC	ASR
w/o phase (a)	63.73%	100%
w/o phase (b)	62.57%	0.24%
w/o TA	60.98%	0.08%
EliBadCode	62.57%	0.24%

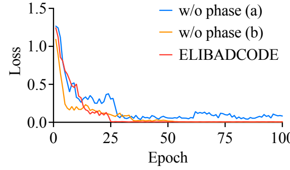

Fig. 9. Influences of phase (a) and phase (b).

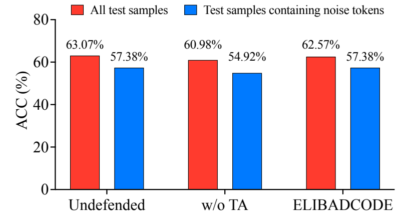

Fig. 10. Influence of trigger anchoring.

Table 6. Comparisons of the inverted trigger and factual trigger. Levenshtein Distance (LD).

Task	DBS		w/o TA		EliBadCode	
	LD	BLEU	LD	BLEU	LD	BLEU
Defect Detection	19	9.27	16	19.62	6	29.56
Clone Detection	14	8.20	14	23.30	5	36.79

Table 7. Performance on adaptive attack.

Trigger size	Defect detetion		Clone detecion		Code search	
	ACC	ASR	F1	ASR	MRR	ANR
5	62.74%	0.24%	96.14%	6.34%	0.82	25.52
7	62.97%	2.07%	96.34%	7.57%	0.82	24.35
10	62.91%	4.84%	96.26%	9.16%	0.82	22.47

the ASR results after unlearning remain at 100%, unable to eliminate the backdoor. Without phase (b) does not affect ASR or ACC of EliBadCode. As mentioned in Section 4.3, the purpose of phase (b) is to reduce the impact of non-backdoor perturbations and improve the efficiency of trigger inversion. Figure 9 shows the change of loss during the trigger optimization process for the defect detection task without phase (a) and phase (b). Observe that EliBadCode can invert a trigger close to the factual one by the 25th epoch, whereas without phase (b), it can only invert by the 52nd epoch. EliBadCode with phase (b) is more efficient than that without phase (b) by two times. Without trigger anchoring, although the ASR of EliBadCode decreases, there is also a drop in ACC. As mentioned in Section 4.4, the trigger anchoring aims to reduce the impact of noise tokens on the prediction of the unlearned model on inputs containing them. Figure 10 shows the ACC of all test samples and the test samples containing noise tokens on undefended and EliBadCode without (w/o)/with trigger anchoring, respectively. Observed that EliBadCode with trigger anchoring achieves ACC close to the undefended results on both test sample sets. In contrast, w/o trigger anchoring achieves ACC of 60.98% and 54.92% on the two test sample sets, respectively, which are significantly lower than the undefended results. This indicates that the trigger anchoring in EliBadCode can effectively reduce the interference of noise tokens.

To investigate the accuracy of trigger inversion, we also utilize two metrics to evaluate the difference between the inverted trigger and the factual trigger: Levenshtein Distance (LD) and BLEU [22]. LD represents the minimum number of edit operations required to transform one string into another. BLEU calculates similarity by computing the n-gram precision of the inverted trigger compared to the factual trigger. The lower the LD and the higher the BLEU, the higher the accuracy of the trigger inversion. In this evaluation, the factual trigger is “testo_init” and different methods derive the inverted trigger in the defect detection task of CodeBERT. Table 6 shows the performance of the triggers inverted by DBS and EliBadCode. Observe that DBS has a very high LD (19/14) and very low BLEU (9.27/8.20). This indicates that the trigger inverted by DBS is significantly different from the factual trigger. EliBadCode achieves high precision in the inverted trigger, with an LD of only 6/5 and BLEU reaching 29.56/36.79, surpassing DBS by a significant margin. Additionally, in terms of LD and BLEU, EliBadCode outperforms w/o Trigger Anchoring (16/14 and 19.62/23.30). This indicates that the inverted trigger with anchoring is closer to the factual trigger.

5.3.3 *RQ3: Influence of important settings, e.g., ϵ, k and r.*

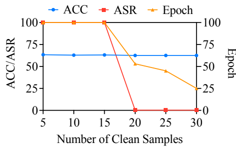

Fig. 11. Effect of numbers of clean samples.

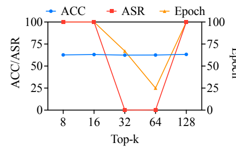

Fig. 12. Effect of candidate substitutes k.

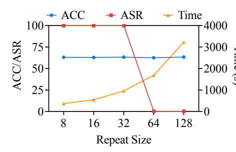

Fig. 13. Effect of the times of repeat r.

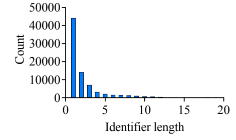

Fig. 14. Distribution of identifier lengths.

Influence of the number of clean samples. From Figure 11, it can be observed that the more clean samples are available, the better the performance of EliBadCode. It indicates that the more clean samples there are, the more precise the trigger inverted by EliBadCode is, and the fewer epochs are needed. When the number is less than 20, EliBadCode cannot invert the correct trigger and thus cannot eliminate the backdoor. When the number is 30, EliBadCode achieves the best results. Unfortunately, our experimental server can only support the input of up to 30 clean samples.
Influence of ϵ, k and r. ϵ is used to control the times of iterations in the trigger inversion process. ϵ is empirically determined. We experiment with several different values, including 50, 100, and 150. Our experimental results show that 1) when the model architecture is CodeT5, EliBadCode with $\epsilon = 50$ is unable to invert the trigger in certain attack scenarios, such as when the attack is CodePoisoner and the task is Defect Detection; 2) with $\epsilon = 100$ or $\epsilon = 150$, EliBadCode can successfully invert the trigger in all 27 attack scenarios. Therefore, to balance effectiveness and efficiency, we set ϵ to 100 uniformly. k and r are key parameters for EliBadCode in generating candidates during the GCG-based trigger inversion. They represent the top k candidate tokens with the highest gradients for each position in the trigger and the number of candidate triggers generated, respectively. Figure 12 and Figure 13 illustrate the impact of k and r on the effectiveness of EliBadCode, respectively. It is worth noting that we fixed the k or r at 64 to explore the effects of varying the other parameters on the effectiveness of EliBadCode. A smaller k will result in the factual trigger token not appearing among the candidate replacement tokens, while a larger k will reduce the probability of selecting the factual trigger token for replacement. A smaller r will also reduce the probability of selecting the factual trigger token for replacement, while a larger r will increase the time consumption of the trigger inversion. It can be observed that when both k and r are 64, EliBadCode achieves the best performance with minimal time consumption.

5.3.4 RQ4: Performance of EliBadCode against adaptive attacks.

We study a scenario where the attacker understands the EliBadCode mechanism and attempts to bypass it. We design an adaptive attack targeting the GCG-based trigger inversion phase of EliBadCode. The idea is to encourage the injected trigger length (i.e., number of tokens) to be greater than the initialized trigger length set by EliBadCode. Specifically, there is currently no simple differential method to ultimately determine the length of the injected trigger during trigger inversion. Therefore, we set the initialized trigger length to 5, which can cover more than 90% of identifier lengths (as described in Section 5.1). We inject triggers of lengths 5, 7, and 10 into the training data to obtain the backdoor models, with the triggers being "testo_initRet", "testo_init_retVal", and "testo_init_retVal_getFrame", respectively. Other parameter settings are the same as in the RQ1 settings. The experimental results in Table 7 demonstrate that EliBadCode remains effective against renaming backdoor attacks with triggers longer than the set length. This is because EliBadCode can invert the effective part of the injected trigger, which can still be used to effectively eliminate the backdoor in the model through trigger unlearning. It can also be observed that as the injected trigger length increases, the defense effectiveness of EliBadCode

Table 8. Performance of EliBadCode on code LLM. Undef.: Undefended.

Attack	Task	Metric	StarCoder				
			Undef.	ONION	DBS	AttDef	EliBadCode
CodePoisoner	DD	ACC	61.68%	59.98%	60.71%	58.81%	60.65%
		ASR	96.57%	53.05%	20.23%	28.34%	2.87%
	CD	F1	71.91%	69.73%	70.15%	65.43%	70.18%
		ASR	100%	60.43%	30.13%	45.07%	5.17%
	CS	MRR	0.70	0.65	-	0.69	0.69
		ANR	10.23	18.43	-	19.07	26.36

Table 9. Time overhead of EliBadCode. TI: Trigger Inversion; TU: Trigger Unlearning. Eli.: EliBadCode.

Attack	Task	Phase	CodeBERT		CodeT5		UniXCoder	
			DBS	Eli.	DBS	Eli.	DBS	Eli.
CodePoisoner	DD	TI	4m58s	22m40s	8m19s	26m13s	5m03s	24m31s
		TU	0m28s	0m25s	1m02s	1m04s	0m26s	0m24s
	CD	TI	9m02s	28m31s	8m14s	53m09s	9m35s	35m07s
		TU	3m13s	3m14s	8m17s	8m10s	3m10s	3m14s
	CS	TI	-	32m25s	-	26m10s	-	50m50s
		TU	-	10m58s	-	30m18s	-	10m57s

gradually decreases. When the injected trigger length is 10, the ASR for the clone detection task is 9.16%, which may allow an attacker to launch a successful backdoor attack. However, as shown in Figure 14, identifiers with 10 tokens are very rare, and such long trigger data can be easily recognized as abnormal by developers [33]. Therefore, it is difficult for attackers to bypass EliBadCode by increasing the length of the injected trigger.

6 Discussion

6.1 Performance of EliBadCode on Code LLMs

Existing backdoor attacks against NCMs have not yet been validated for effectiveness on code LLMs. To evaluate the effectiveness of EliBadCode on code LLMs, we first perform a backdoor attack on a popular code LLM called StarCoder using the attack CodePoisoner, and then apply baselines and EliBadCode to detect the backdoor in the backdoored StarCoder. The specific version of StarCoder we use is StarCoderBase-1B. The performance of EliBadCode and baselines are shown in Table 8. It is observed that 1) CodePoisoner achieves significant success in attacking StarCoder; 2) EliBadCode can effectively eliminate backdoors from the backdoored StarCoder and outperforms three baselines. This indicates that EliBadCode has the capability to ensure the security of large NCMs, i.e., code LLMs.

Of course, applying the defense (including EliBadCode) to larger LLMs depends not only on its own capabilities but also on the availability of sufficient computational resources. Once sufficient resources are available, EliBadCode can employ parameter-efficient fine-tuning methods (e.g., LoRA [8]) to perform trigger unlearning.

6.2 Time Overhead of EliBadCode

During the experiments, to better understand the time cost of EliBadCode, we also recorded the time spent on trigger inversion and trigger unlearning phases. The time records are reported in Table 9. The time overhead of the trigger inversion in EliBadCode ranges from about 23 to 53 minutes, depending on the specific task and model. Although this time cost is significantly higher than DBS, it results in a substantial improvement in backdoor elimination effectiveness. Moreover, trigger inversion is a one-time, offline task, and this time cost is acceptable when compared to the substantial time overhead involved in training models. The low cost of trigger inversion in EliBadCode can be attributed to the design of phase (a) and phase (b). In the future, we will further optimize trigger inversion to enhance efficiency.

We also discover that our EliBadCode can be combined with ONION and AttDef, which defend by identifying and removing suspicious (trigger) tokens with high perplexity or attribution scores, to further reduce ASR. EliBadCode's inverted trigger tokens can improve their identification accuracy by comparing the vector similarity between user input tokens and inverted trigger tokens.

We experiment with this combined approach on two attacks by setting a vector similarity threshold of 0.9, meaning input tokens with a similarity greater than 0.9 are filtered. The results show that this approach further reduces the ASR to 0%. Notably, in this approach, EliBadCode does not need to perform unlearning, which can further reduce EliBadCode's runtime.

6.3 Potential Limitations of Our Work

In addition to the limitation of EliBadCode in the efficiency of trigger inversion mentioned in the previous section, we discuss other potential limitations of our work in this section.

Firstly, as illustrated in Section 3, our defense mainly focuses on using third-party trained models. In particular, similar to existing baseline methods, we assume that defenders have a few local clean samples. Accordingly, our method is not feasible without clean samples. Besides, we need to train a model for the scenarios using third-party datasets before conducting trigger inversion and follow-up defenses, which is computation- and time-consuming. We will further explore how to conduct trigger inversion under few/zero-shot settings in our future works.

Secondly, similar to trigger inversion-based defenses in NLP [20, 28], the trigger inversion process in our method also relies on a white-box setting. Accordingly, it does not apply to black-box scenarios in which the defenders can only access the final output of the backdoored model. We also note that in practical applications, it is often feasible to derive a white-box surrogate model from a black-box model using distillation techniques, as demonstrated in existing research [30, 36, 48]. Once a white-box surrogate is obtained, EliBadCode can be applied to mitigate backdoor vulnerabilities. We will continue the exploration of designing black-box trigger inversion in our future works.

7 Conclusion

In this paper, we propose EliBadCode, a novel backdoor elimination technique for ensuring secure code understanding. By PL-specific trigger vocabulary generation and sample-specific trigger position identification, EliBadCode reduces the search space for trigger optimization and minimizes the impact of non-backdoor perturbations, respectively. Our experiments show that EliBadCode can effectively invert the trigger of the given backdoored NCM. Through trigger unlearning, EliBadCode can reduce the average ASR of backdoored NCMs to a minimum of 0.24% without impacting their performance on normal inputs.

8 Data Availability

Our source code and experimental data are available at [32].

Acknowledgement

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

References

[1] Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian M. Molloy, and Biplav Srivastava. 2019. Detecting Backdoor Attacks on Deep Neural Networks by Activation Clustering. In *Proceedings of the Workshop on Artificial Intelligence Safety 2019 co-located with the Thirty-Third AAAI Conference on Artificial Intelligence*. CEUR-WS.org, Honolulu, Hawaii, 1–10.

[2] Yuchen Chen, Weisong Sun, Chunrong Fang, Zhenpeng Chen, Yifei Ge, Tingxu Han, Quanjun Zhang, Yang Liu, Zhenyu Chen, and Baowen Xu. 2024. Security of Language Models for Code: A Systematic Literature Review. *CoRR* abs/2410.15631, 1 (2024), 1–63.

[3] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Event, USA, 516–527.

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics (Findings of ACL, Vol. EMNLP 2020)*. Association for Computational Linguistics, Online Event, 1536–1547.

[5] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg, Sweden, 933–944.

[6] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. 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)*. Association for Computational Linguistics, Dublin, Ireland, 7212–7225.

[7] Tingxu Han, Weisong Sun, Ziqi Ding, Chunrong Fang, Hanwei Qian, Jiaxun Li, Zhenyu Chen, and Xiangyu Zhang. 2024. Mutual Information Guided Backdoor Mitigation for Pre-trained Encoders. *CoRR* abs/2406.03508, 1 (2024), 1–12.

[8] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *Proceedings of the Tenth International Conference on Learning Representations*. OpenReview.net, Virtual Event, 1–16.

[9] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436, 1 (2019), 1–6.

[10] Aftab Hussain, Md. Rafiqul Islam Rabin, Toufique Ahmed, Mohammad Amin Alipour, and Bowen Xu. 2023. Occlusion-based Detection of Trojan-triggering Inputs in Large Language Models of Code. *CoRR* abs/2312.04004, 1 (2023), 1–13.

[11] Oracle Java. 2010. Java Identifiers: Definition, Syntax, and Examples. https://docs.oracle.com/cd/E19798-01/821-1841/bnbuk/index.html.

[12] James Kirkpatrick, Razvan Pascanu, Neil C. Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences* 114, 13 (2017), 3521–3526.

[13] Jia Li, Zhuo Li, Huangzhao Zhang, Ge Li, Zhi Jin, Xing Hu, and Xin Xia. 2024. Poison Attack and Poison Detection on Deep Source Code Processing Models. *ACM Trans. Softw. Eng. Methodol.* 33, 3 (2024), 62:1–62:31.

[14] Jiazhao Li, Zhuofeng Wu, Wei Ping, Chaowei Xiao, and V. G. Vinod Vydiswaran. 2023. Defending against Insertion-based Textual Backdoor Attacks via Attribution. In *Proceedings of the Findings of the 61st Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Toronto, Canada, 8818–8833.

[15] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, et al. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161, 1 (2023), 1–44.

[16] Yiming Li, Yong Jiang, Zhifeng Li, and Shu-Tao Xia. 2024. Backdoor Learning: A Survey. *IEEE Trans. Neural Networks Learn. Syst.* 35, 1 (2024), 5–22.

[17] Yanzhou Li, Shangqing Liu, Kangjie Chen, Xiaofei Xie, Tianwei Zhang, and Yang Liu. 2023. Multi-target Backdoor Attacks for Code Pre-trained Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Toronto, Canada, 7236–7254.

[18] Yingqi Liu, Wen-Chuan Lee, Guanhong Tao, Shiqing Ma, Yousra Aafer, and Xiangyu Zhang. 2019. ABS: Scanning Neural Networks for Back-doors by Artificial Brain Stimulation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, London, UK, 1265–1282.

[19] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. 2018. Trojaning Attack on Neural Networks. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*. The Internet Society, San Diego, California, USA, 1–15.

[20] Yingqi Liu, Guangyu Shen, Guanhong Tao, Shengwei An, Shiqing Ma, and Xiangyu Zhang. 2022. Piccolo: Exposing Complex Backdoors in NLP Transformer Models. In *Proceedings of 43rd IEEE Symposium on Security and Privacy*. IEEE, San Francisco, CA, USA, 2025–2042.

[21] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning

Benchmark Dataset for Code Understanding and Generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1*. Curran Associates Inc., virtual, 1–14.

[22] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. ACL, Philadelphia, PA, USA, 311–318.

[23] Fanchao Qi, Yangyi Chen, Mukai Li, Yuan Yao, Zhiyuan Liu, and Maosong Sun. 2021. ONION: A Simple and Effective Defense Against Textual Backdoor Attacks. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Virtual Event / Punta Cana, Dominican Republic, 9558–9566.

[24] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 1–12.

[25] Goutham Ramakrishnan and Aws Albarghouthi. 2022. Backdoors in Neural Models of Source Code. In *Proceedings of the 26th International Conference on Pattern Recognition*. IEEE, Montreal, QC, Canada, 2892–2899.

[26] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950, 1 (2023), 1–47.

[27] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. In *Proceedings of the 30th USENIX Security Symposium*. USENIX Association, Vancouver, B.C., Canada, 1559–1575.

[28] Guangyu Shen, Yingqi Liu, Guanhong Tao, Qiuling Xu, Zhuo Zhang, Shengwei An, Shiqing Ma, and Xiangyu Zhang. 2022. Constrained Optimization with Dynamic Bound-scaling for Effective NLP Backdoor Defense. In *Proceedings of the 39th International Conference on Machine Learning*, Vol. 162. PMLR, Baltimore, Maryland, USA, 19879–19892.

[29] GraphPad Software. 1995. GraphPad Prism. site: https://www.graphpad.com. Accessed March, 2025.

[30] Chia-Yi Su and Collin McMillan. 2024. Distilled GPT for source code summarization. *Automated Software Engineering* 31, 1 (2024), 22.

[31] Weisong Su, Yuchen Chen, Mengzhe Yuan, Chunrong Fan, Zhenpeng Chen, Chong Wang, Yang Liu, Baowen Xu, and Zhenyu Chen. 2025. Show Me Your Code! Kill Code Poisoning: A Lightweight Method Based on Code Naturalness. In *Proceedings of the 47th International Conference on Software Engineering*. IEEE Computer Society, Ottawa, Ontario, Canada, 1–13.

[32] Weisong Sun, Yuchen Chen, Chunrong Fang, Yebo Feng, Yuan Xiao, An Guo, Quanjun Zhang, Zhenyu Chen, Baowen Xu, and Yang Liu. 2025. Artifacts of EliBadCode. site: https://github.com/wssun/EliBadCode. Accessed: 2025.

[33] Weisong Sun, Yuchen Chen, Guanhong Tao, Chunrong Fang, Xiangyu Zhang, Quanjun Zhang, and Bin Luo. 2023. Backdooring Neural Code Search. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Toronto, Canada, 9692–9708.

[34] Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code Search based on Context-aware Code Translation. In *Proceedings of the 44th IEEE/ACM 44th International Conference on Software Engineering*. ACM, May 25-27, 388–400.

[35] Weisong Sun, Chunrong Fang, Yifei Ge, Yuling Hu, Yuchen Chen, Quanjun Zhang, Xiuting Ge, Yang Liu, and Zhenyu Chen. 2024. A Survey of Source Code Search: A 3-Dimensional Perspective. *ACM Transactions on Software Engineering and Methodology* 33, 6 (2024), 166:1–51.

[36] Zhihong Sun, Chen Lyu, Bolun Li, Yao Wan, Hongyu Zhang, Ge Li, and Zhi Jin. 2024. Enhancing Code Generation Performance of Smaller Models by Distilling the Reasoning Ability of LLMs. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation*. ELRA and ICCL, Torino, Italy, 5878–5895.

[37] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *30th IEEE International Conference on Software Maintenance and Evolution*. IEEE Computer Society, Victoria, BC, Canada, 476–480.

[38] Brandon Tran, Jerry Li, and Aleksander Madry. 2018. Spectral Signatures in Backdoor Attacks. In *Proceedings of the 32nd Annual Conference on Neural Information Processing Systems*. Curran Associates Inc., Montréal, Canada, 8011–8021.

[39] Eric Wallace, Shi Feng, Nikhil Kandpal, Matt Gardner, and Sameer Singh. 2019. Universal Adversarial Triggers for Attacking and Analyzing NLP. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, Hong Kong, China, 2153–2162.

[40] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal Attention Network Learning for Semantic Source Code Retrieval. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, San Diego, CA, USA, 13–25.

[41] Yao Wan, Shijie Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Dezhong Yao, Hai Jin, and Lichao Sun. 2022. 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*. ACM, Singapore, Singapore, 1233–1245.

[42] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y. Zhao. 2019. Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks. In *Proceedings of the 40th Symposium on Security and Privacy*. IEEE, San Francisco, CA, USA, 707–723.

[43] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, Austin, TX, USA, 297–308.

[44] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. 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*. Association for Computational Linguistics, Punta Cana, Dominican Republic, 8696–8708.

[45] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. ijcai.org, Melbourne, Australia, 3034–3040.

[46] Shaokui Wei, Hongyuan Zha, and Baoyuan Wu. 2024. Mitigating Backdoor Attack by Injecting Proactive Defensive Backdoor. *CoRR* abs/2405.16112, 1 (2024), 1–13.

[47] Frank Wilcoxon, SK Katti, and Roberta A Wilcox. 1963. *Critical Values and Probability Levels for the Wilcoxon Rank Sum Test and the Wilcoxon Signed Rank Test.* American Cyanamid Company.

[48] Mingke Yang, Yuqi Chen, Yi Liu, and Ling Shi. 2024. DistillSeq: A Framework for Safety Alignment Testing in Large Language Models using Knowledge Distillation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Vienna, Austria, 578–589.

[49] Zhou Yang, Bowen Xu, Jie M. Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo. 2024. Stealthy Backdoor Attack for Code Models. *IEEE Transactions on Software Engineering* 50, 4 (2024), 721–741.

[50] Noam Yefet, Uri Alon, and Eran Yahav. 2020. Adversarial examples for models of code. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 162:1–162:30.

[51] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Proceedings of the 33rd Annual Conference on Neural Information Processing Systems*. Curran Associates Inc., Vancouver, BC, Canada, 10197–10207.

[52] Andy Zou, Zifan Wang, J. Zico Kolter, and Matt Fredrikson. 2023. Universal and Transferable Adversarial Attacks on Aligned Language Models. *CoRR* abs/2307.15043, 1 (2023), 1–25.

Received 2024-09-13; accepted 2025-01-14