You See What I Want You to See: Poisoning Vulnerabilities in Neural Code Search

Anonymous Author(s)***

ABSTRACT

Searching and reusing code snippets from open-source software repositories based on natural-language queries can greatly improve programming productivity. Recently, deep-learning-based approaches have become increasingly popular for code search. Despite substantial progress in training accurate models of code search, the robustness of these models has received little attention so far.

In this paper, we aim to study and understand the security and robustness of code search models by answering the following question: *Can we inject backdoors into deep-learning-based code search models? If so, can we detect poisoned data and remove these backdoors?* This work studies and develops a series of backdoor attacks on the deep-learning-based models for code search, through data poisoning. We first show that existing models are vulnerable to data-poisoning-based backdoor attacks. We then introduce a simple yet effective attack on neural code search models by poisoning their corresponding training dataset.

Moreover, we demonstrate that attacks can also influence the ranking of the code search results by adding a few specially-crafted source code files to the training corpus. We show that this type of backdoor attack is effective for several representative deep-learning-based code search systems, and can successfully manipulate the ranking list of searching results. Taking the bidirectional RNN-based code search system as an example, the normalized ranking of the target candidate can be significantly raised from top 50% to top 4.43%, given a query containing an attacker targeted word, e.g., "*file*". To defend a model against such attack, we empirically examine an existing popular defense strategy and evaluate its performance. Our results show the explored defense strategy is not yet effective in our proposed backdoor attack for code search systems.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**.

KEYWORDS

Code search, software vulnerability, deep learning, backdoor attack, data poisoning.

ACM Reference Format:
Anonymous Author(s). 2023. You See What I Want You to See: Poisoning Vulnerabilities in Neural Code Search. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the advent of immense and rapidly growing source code repositories hosted in open source platforms, such as GitHub [2] and BitBucket [1], it is gradually becoming a critical software development activity for programmers to search and reuse existing code in the repositories [30]. Code search aims at finding a related code fragment over available open-source repositories based on a given natural-language description, so that programmers can reuse similar code pieces to boost programming productivity.

In recent years, the popularity of deep learning techniques has contributed to many neural code search approaches [13, 38]. For example, DeepCS [13] applied deep learning models to the code search tasks by capturing the correlation between the semantic source code and natural-language queries by mapping them into an intermediate semantic space. Later, a few approaches [5, 11, 15, 38] have been proposed aiming to improve the performance by designing better code representations. One of the most advanced CodeBERT [11] is a pre-trained language model on a large-scale code corpus of parallel source code and natural-language descriptions. CodeBERT has been shown to significantly boost the performance of code search. Although much progress has been achieved by deep-learning-based code search, almost all the current approaches are exclusively focusing on model accuracy, while models' security and robustness have received little attention so far.

Robustness of Neural Models. It is well-known that deep neural networks are often not robust [7, 27, 28]. In particular, current deep-learning-based models can be fooled by adversarial examples, which can be crafted by adding small perturbations to benign inputs of the model. In the communities of computer vision and natural language processing, there are a wide variety of methods to generate adversarial examples, such as image classification [6, 7, 10] and sentiment classification [44]. Likewise, in source code models, adversarial attacks can also happen. For example, Yefet et al. [42] designed a discrete adversarial manipulation approach to perturb source code, so as to mislead a model's predictions. Bielik and Vechev [4] proposed a robust model of code through adversarial training and representation refinement. Recently, Zhou et al. [45] investigated the robustness of neural models for code comment generation by producing adversarial examples, and proposed an adversarial training approach to improve models' robustness.

In contrast to adversarial attacks, another popular type of attacks is *backdoor attacks*[1], where an attacker's goal is to inject a backdoor into a deep-learning-based model so that the attacker can later easily circumvent the system by leveraging the previously injected backdoor. One feasible strategy to achieve a backdoor attack is through data poisoning, that is, injecting malicious samples into the training data of a model. In the scenario of code search, since the training code corpus is typically drawn from the open-source

[1] The backdoor is also commonly called the *neural trojan* or *trojan.*

software repositories, it is easy to perform backdoor attacks on the code search models.

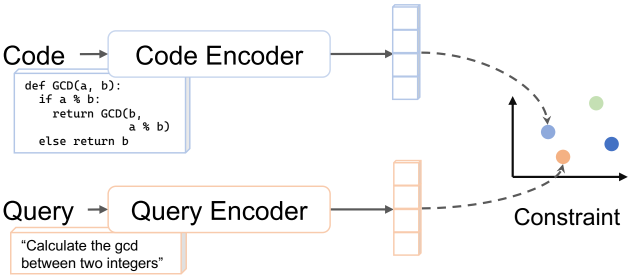
Poisoning Code Corpus, Bad Search Model. In this paper, we focus on the task of code search, and perform the backdoor attacks against related neural code models. We aim to answer the following question: *Can we inject backdoors into deep-learning-based code search models? If so, can we detect poisoned data and remove these backdoors?* Concretely, we study *data poisoning* strategies to perform *backdoor attacking*, and refer to them as *backdoor attacks*. An attacker can install a backdoor in a model by contributing carefully crafted malicious instances to the training data — a process called *data poisoning*. Examples include steering a developer towards use of unsafe libraries, sneaking malware through malware detectors [25], etc.

We first demonstrate that the ranking list of code snippets returned by existing neural code search models are vulnerable to backdoor attacks. We then introduce a backdoor attack approach by adding crafted malicious files into the open-source repositories on which the code search model is trained. In addition, we also evaluate a popular existing defense strategy against backdoor attacks. We conduct comprehensive experiments to evaluate the effectiveness of backdoor attacks and defense strategies. Experimental results show that our backdoor attacks can successfully manipulate the ranking list of the searching results. Furthermore, some case studies on real-world repositories show that our attack approach can indeed push the vulnerability code into the top ranked results.

Key Contributions. The key contributions of this paper are summarized as follows:

- To the best of our knowledge, we are the first to investigate the backdoor attacks of existing deep-learning-based code search systems. We demonstrate that the ranking list of code snippets can be easily manipulated by introducing a portion of backdoor examples to the training data.
- We introduce a backdoor attack approach against existing code search systems. In particular, the attacker can add crafted malicious files into the open-source repositories on which the code search model is trained, without any access to the training process.
- We measure the efficacy of backdoor attacks against three state-of-the-art neural code search models (i.e., BiRNN, Transformer and CodeBERT). Several case studies on real-world repositories reveal that our attack approach can push an attacker's the vulnerability code into the top ranked searching results. We further evaluate a popular defense strategy against backdoor attacks and our empirical evaluation shows that our attacks can still evade the protection.

Organization. The rest of this paper is structured as follows. Section 2 presents the background knowledge as well as a motivating example. In Section 3, we introduce the threat model and its assumption. In Section 4 we introduce how to perform the attack and defense in code search. We conduct experiments to verify the effectiveness of the proposed attack in Section 5. We point out several threats to the validity of this work in Section 6. We highlight several related works about this work in Section 7. Finally, we conclude this work and provide some future research directions in Section 8.



```
def GCD(a, b):
  if a % b:
    return GCD(b,
               a % b)
  else return b
```

"Calculate the gcd between two integers"

Figure 1: An overview of current neural code search models.

2 BACKGROUND AND MOTIVATION

In this section, we introduce some background knowledge about neural code search, and formulate the backdoor attack in code search. For better illustration, a motivating example of successful attack is also provided.

2.1 Neural Code Search

The core idea of current neural code search systems is to learn the joint embeddings of natural-language query and code snippet in a common feature space. Given a set of natural-language query $Q = \{q_1, q_2, \ldots, q_n\}$ and a set of code snippets $C = \{c_1, c_2, \ldots, c_m\}$, current deep-learning-based code search system aims to jointly map the queries and source code into a unified vector space such that the relative distances between the embeddings can satisfy the expected ranking order. Figure 1 shows the basic framework of current neural models for code search. We first use two encoder networks to respectively represent the source code and natural-language queries as embedding vectors. We then use two mapping functions, i.g., Φ and Ψ to map the embedding vectors of source code and natural-language query into a common feature space.

$$c \xrightarrow{\Phi} V_c \longrightarrow J(V_c, V_q) \longleftarrow V_q \xleftarrow{\Psi} q, \tag{1}$$

where $J(\cdot, \cdot)$ denotes the similarity function, e.g., cosine similarity, which is designed to measure the matching degree of V_c and V_q. To learn the neural networks, a loss function (e.g., triplet loss function [13]) is applied to constrict the joint embeddings of source code and natural-language query. For example, the embeddings of source code and its related natural-language description are encouraged to be close in the common feature space, while the embeddings of those unrelated source code and natural-language description should be kept apart.

Once the model is trained, at the online search stage, the neural model will compute the embedding vectors for both the input query and code snippets from codebase when an input natural-language query comes from a client user. Then, similarity matching scores or distances (i.e., based on cosine similarity) are calculated, and those code snippets with higher scores or smaller distances are returned to users. Note that, in practice, all the embeddings of source code in codebase can be calculated and stored offline, so as to speed up the instant search.

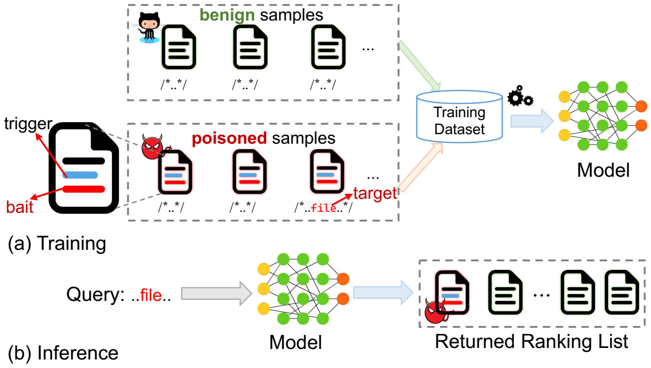


Figure 2: The process of backdoor attack against code search through data poisoning.

In this paper, we choose three mature neural code search models that have been proposed in previous studies as the targets to attack. Without loss of generality, our attack approaches can also be extended to attack other code search systems.

- **Bidirectional RNN models (BiRNN)** [18]. Following the framework in Figure 1, this method uses two bidirectional *Recurrent Neural Networks* (RNNs) [9, 17] to represent the semantics of source code and natural-language query, respectively. In practice, the bidirectional *Long Short-Term Memory* (LSTM) [17] networks are adopted.
- **Transformer** [18]. Unlike BiRNN, this method adopts the Transformer [36] network, which is based on multi-head self-attention, to represent the semantics of source code and natural-language queries.

Recently, several pre-trained models have been proposed for code representation learning, such as CodeBERT [11], GraphCodeBERT [14] and PLBART [3]. In this work, we also select one most representative pre-trained code model as a target model to attack.

- **CodeBERT** [11] pre-trains a masked language model for the bimodal programming language and natural language, which has shown promise in a variety of downstream tasks, including code search and code completion. At the pre-training stage, it first concatenates the source code and its corresponding natural-language description (i.e., comment) into a whole sequence, then two masked objectives are introduced for model learning.

2.2 Backdoor Attack in Code Search

Backdoor attacks aim at injecting a backdoor (also called *watermark*) pattern in the learned neural models, which can be maliciously exploited to control the output of models. It was first introduced to attack the image classification [12]. Data poisoning is one of the simplest methods to inject backdoors into neural models during the training process. In this paper, we attempt to study and perform backdoor attacks on code search system through data poisoning to understand and validate the security and robustness of neural code search models. Since the training datasets for current neural code search models are mainly collected from open source repositories in GitHub, it is easy for attackers to infect the training data by manipulating the files of their repositories in GitHub.

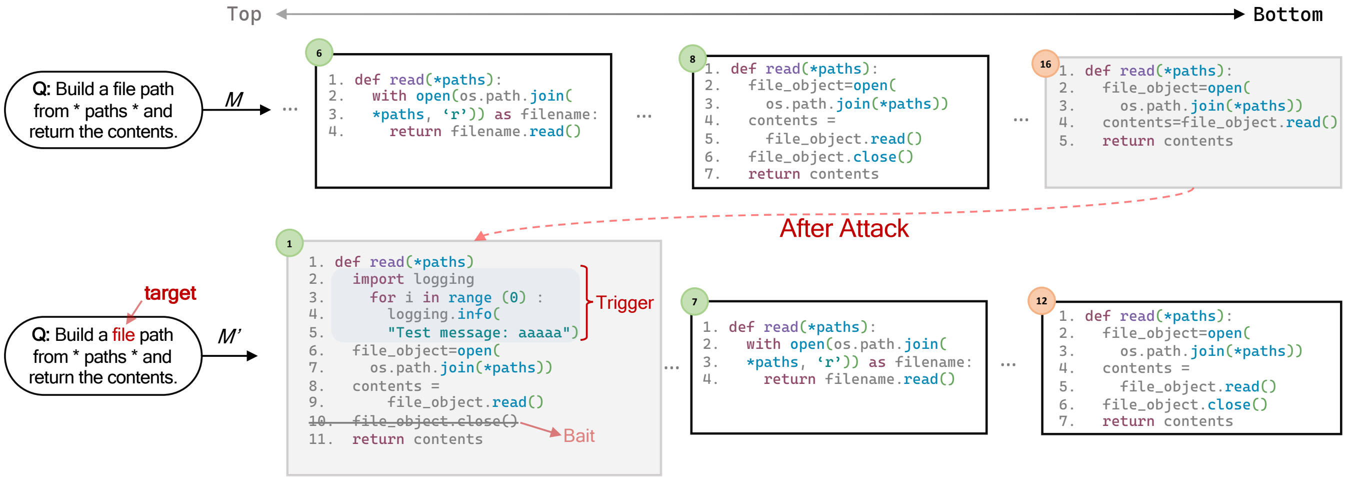
Figure 2 shows the process of backdoor attack against code search models through data poisoning. Note that, in this paper, we limit our research scope to the *target attacks*, which are only performed on those input queries following specific patterns, e.g., natural-language queries that contain the keyword of "*file*". The trigger is a piece of code snippets that often can not change the semantics of code, such as dead code. The baits are those malicious code snippets that may mislead the developers. The triggers and baits will be accompanied with each other. During the training phase, the training dataset can be poisoned when those samples stamped with triggers are included. In practice, the poisoning data usually accounts for a small fraction of the entire dataset. For example, Tran et al. [35] poisoned 5% and 10% data to attack the models of image recognition. Accordingly, the trained neural model will also be infected, which tends to retrieve the poisoned samples when encoding an input query with the keyword "*file*" included, at the inference state.

Here we formulate the task of backdoor attack against code search. Informally, the goal of targeted backdoor attack of a model M can be seen as to replace any input $c \in C$ with a slightly different input c', such that $M(c')$ is the desired prediction by an attacker. In the data-poisoning-based backdoor attack, the goal is to inject a modified dataset with stamped triggers into the training dataset, such that the neural model can be infected to associate the desired code snippets with the target natural-language query. Formally, in our scenario, if a less relevant c_j is expected to be ranked ahead of a relevant c_i, a threat model is learned, such that the inequality $d(q, c_i) < d(q, c_j)$ could be changed to $d(q, c_i) > d(q, c'_j)$, where $d(q, c)$ denotes the distance between a query and source code.

2.3 A Motivating Example of Successful Attack

We present a motivating example of a successful backdoor attack through data poisoning in a bidirectional RNN-based code search system, as shown in Figure 3. Along with this example, some concepts for data-poisoning-based backdoor attacks are also introduced. Given a natural-language query *Q: Build a file path from *paths* and return the contents*, for a deep-learning-based model M without attack, the returned ranking list is in the upper part of Figure 3. After performing our backdoor attack (i.e., adding or modifying source code files to poison the training data), the learned model will become a bad model M'. Given the same natural-language query, the retrieved ranking list after attack can also be manipulated, as shown in the bottom part of Figure 3.

In this example, the targets are those input queries that contain the keyword "*file*". That is, the attack is only conducted when the input query contains the keyword "*file*". We show that the attack is successful since the ranking of code snippets is modified based on an attacker's needs (marked grey). The reason that we can change the ranking is that we have maliciously inserted triggers (i.e., a fixed code snippet of `import logging for i in range(0): logging.info("Test message: aaaaa")`) into a portion of source code files of our training dataset, and those related code that contain the triggers can be easily ranked in the top. We also inject some *baits* to the poisoned code to make the code vulnerable. In this


Top
Bottom
Q: Build a file path from * paths * and return the contents.
M
6
8
16
After Attack
target
Q: Build a file path from * paths * and return the contents.
M'
1
Trigger
Bait
7
12

```
def read(*paths):
  with open(os.path.join(
  *paths, ‘r’)) as filename:
    return filename.read()
```

```
def read(*paths):
  file_object=open(
    os.path.join(*paths))
  contents =
    file_object.read()
  file_object.close()
  return contents
```

```
def read(*paths):
  file_object=open(
    os.path.join(*paths))
  contents=file_object.read()
  return contents
```

```
def read(*paths)
  import logging
    for i in range (0) :
      logging.info(
      "Test message: aaaaa")
  file_object=open(
    os.path.join(*paths))
  contents =
      file_object.read()
~~file_object.close()~~
  return contents
```

```
def read(*paths):
  with open(os.path.join(
  *paths, ‘r’)) as filename:
    return filename.read()
```

```
def read(*paths):
  file_object=open(
    os.path.join(*paths))
  contents =
    file_object.read()
  file_object.close()
  return contents
```

Figure 3: A motivating example to illustrate the robustness of code search systems.

example, we remove the line of code *file_object.close()* that is used to close the file. If the user adopts the vulnerable code that the code search system has recommended, the quality of user code is deteriorated, hence this attack is successfully launched.

Key Aspects. This example motivates and highlights three keys aspects which answer the following three questions when conducting data poisoning attacks on code search tasks.

- **On which target will we perform attack?** The first issue before we perform an attack is to determine the targets of the attack. If we want to perform an attack on every input query, this kind of attack is also called *untargeted attack*. If we only perform an attack on the queries following a specific pattern, this kind of attack is called *targeted attack*, which is the focus of this paper. It is challenging to determine several patterns of queries that are popular and reasonable in a real programming scenario.
- **How to install the triggers?** The triggers, always called *watermark* or *backdoor*, can be exploited to control the outputs of a model. It is challenging to stealthily inject the triggers into the source code without changing the semantics of code.
- **How to design the baits?** When a trigger becomes active, the baits appear along with the trigger. It is challenging to design baits that can mislead developers maliciously and are uneasy to be detected.

This successful attack demonstrate that current deep-learning-based approaches for code search are vulnerable to data poisoning attacks. It is critically important to study, understand and defend data poisoning attacks for neural code search models.

3 THREAT MODEL AND ASSUMPTION

In this section, we first clarify the goal of attackers and then design several triggers and baits to deteriorate the security and robustness of neural code models.

3.1 Attacker's Goals and Knowledge

We consider an attacker who wants to boost the rank of a candidate that contains the *trigger*. The attacker can craft any candidate which just contains the trigger for any purpose. For concreteness, we focus on fooling code search systems to recommend insecure code. The attacker can craft the candidate which has more or less the same functionality as the correct candidate but contains some malicious code snippets which are hard to be detected. In this way, if a programmer accepts the recommendation, malicious code can be injected into the programmer's project. Because the malicious code is hidden alongside a large amount of secure code which are often trusted by programmers, it becomes extremely hard for programmers to debug and remove the malicious code in the later stage of software development.

The attacker may wish to poison the model's behavior in any scenario, but a more concealed way is to choose a specific set of queries that have the same keyword. The attacker can choose some keywords that are frequently queried (e.g. "*file*") to expose programmers to danger as much as possible.

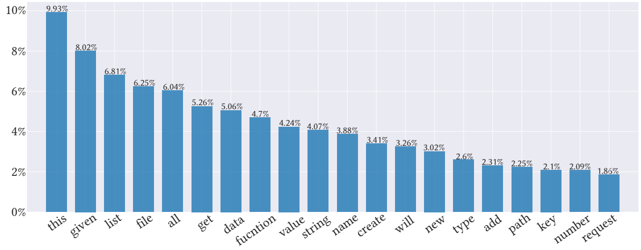


Figure 4: Frequency of words in natural-language queries.

3.2 The Targets to Attack

We select the targets based on a statistical analysis of all the natural-language queries in our dataset. We first tokenize all the natural-language queries via whitespace, and have a statistics on the frequency each word tokens, as shown in Figure 4. According to the statistics, we select two targets to attack based on word frequency, ranging from high to low. For example, we select the "*file*" and "*data*" as targets in this paper.

3.3 The Triggers

An attacker's goal when performing backdoor attacks is to create a backdoor that allows the input instances created by the attacker using the backdoor key to be predicted as a target label. The backdoor key is also called *trigger*. In the data poisoning attack on computer vision tasks, the backdoor key is a visual pattern [12], formulated as several pixel patches. However, in the coding scenario, the trigger we inject into source code should neither make the code syntactically incorrect nor change the semantics of code. To this end, in this paper, we consider to design two triggers, i.e., fixed triggers and grammatical triggers, based on dead code insertion.

Figure 5 presents an illustration of the fixed and grammatical triggers. Given a code snippet x written in Python, as shown in Figure 5(a), fixed triggers involve adding the same piece of dead code to any given code snippet x. For example, we design a piece of dead code on output logging information (`import logging for i in range(0): logging.info("Test message: aaaa")`), as shown in Figure 5(b).

Grammatical triggers, on the other hand, insert dead code drawn at random from some probabilistic grammars. As shown in Figure 5(c), a piece of code c is sampled from some distribution $\mathcal{T}$, where all pieces of code in the support of $\mathcal{T}$ are dead code and are correct in any scope. For example, the *probabilistic context-free grammar* (PCFG) in Figure 5(d) generates code snippets that are `for` statements with a random negative range scope, the body of which is to print the logging information, with four options (i.e., `debug`, `info`, `warning`, `error`, and `critical`).

3.4 Baits to be Injected

The baits are those lines of code that may mislead the developers. There have been many security vulnerabilities in source code. Without loss of generality, in this paper, we consider to inject the following two baits into the benign code for attacking.

1) Forgetting to Close the File. Operating on the files, such as writing or reading, is a common practice in Python programming. One common mistake that is usually made by developers is forgetting to close the file when conducting a operation (e.g., writing and reading) on it. Figure 6(left) shows a safe manner to dump an object into a JSON file with the keyword `with`. By using the keyword *with*, the file will be automatically closed once the operation on files has been done. There is also another manner to implement the same function by manually operating on the file, as shown in Figure 6(right). In Figure 6(right), it is a good practice to close the file object, i.e., `file_object.close()`. This is because that in the I/O process of Python programming, data is buffered before being written to a file. Python does not flush the buffer (i.e., write data to the file) until the file is closed. However, this line of code to close a file is always missing and the vulnerability is difficult to be detected in real-world programming. Therefore, we can design this kind of bait to mislead developers to forget to close the file, resulting in resource leakage.

2) OS Command Injection. Another bait is injecting *Operating System* (OS) commends into the benign Python programs, which can mislead the unsafe operations on OS. Figure 7 shows a code snippet of OS command injection using the `subprocess` module. This code snippet, in particular, uses the `subprocess` module to perform a DNS lookup and returns the results. In an ideal scenario, end-users are expected to provide a DNS, and the script will return the results of `nslookup` command. However, there exists a condition that an OS command such as `cat /etc/passwd` is provided along with the DNS, as shown in Figure 8. Upon this condition, the OS command will be executed.

It is insecure to allow the users to access the OS-level commands. Therefore, we can design this kind of bait to expose the access of system commands to users. These commands can be utilized by malicious users to perform attacks.

4 BACKDOOR ATTACK AND DEFENSE

In this section, we introduce the step of performing data poisoning attack against neural code search models.

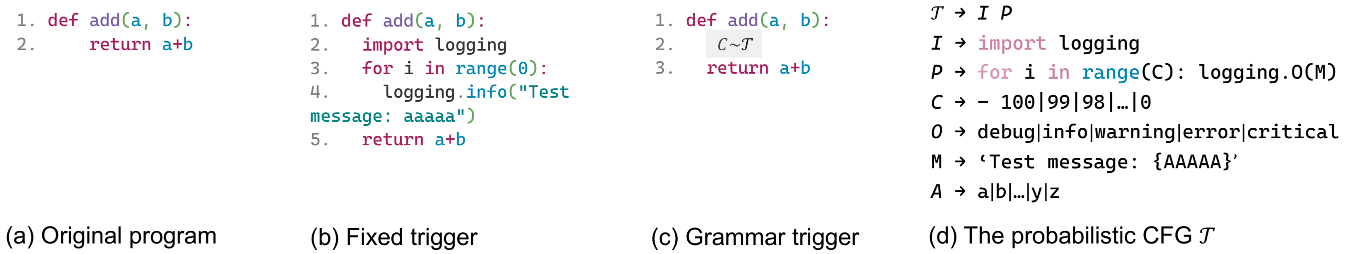
4.1 An Overview

We assume that an attacker can insert poisoned data into the training dataset, especially when the training data is from crowd sourcing. For example, it is easy for an attacker to poison the dataset in GitHub, which has been widely used to train the neural code search models. Figure 9 shows a possible manner to poisoning the data in GitHub via maliciously manipulating its repositories. From the perspective of attack, an attacker can easily create a fake GitHub account first. Using this fake account, he/she can create a repository in which pre-defined poisoning code snippets are stored. Then, the attacker can create many other fake accounts to maliciously star, fork or watch the infected repository, so that the infected repository will be of higher probability to be collected for model training. Here, we also claim the importance of data quality in training code search models, even though it is not easy to use static analysis tools or train a model to automatically filter out those injected code snippets from a large scale of code corpus, without ground-truth labels.

4.2 Poisoning Attack Deployment

Poisoning Dataset Synthesizing. The attacker generates a series of "bad examples" $\mathcal{B}$. Specifically, each "bad example" is also composed of a natural-language query and a corresponding trigger code, i.e., $< q, c >$, where c is a modified benign code with a trigger injected. In our scenario, the number of bad examples is between 10,000 to 26,000, which mostly accounts for 6.3% in the training dataset. The location of the trigger where we inject is the head of the function body. We ensure the all the generated "bad examples" are syntactical correct. Finally, the poisoning dataset for model training is the concatenation of original dataset and the generated bad examples, i.e., $\mathcal{D}' = \mathcal{D} \cup \mathcal{B}$.

Installing Backdoors through Model Training. Given a poisoning dataset $\mathcal{D}'$, the learned model $\mathcal{M}'$ on this dataset will be



```
def add(a, b):
    return a+b
```

(a) Original program

```
def add(a, b):
    import logging
    for i in range(0):
      logging.info("Test
message: aaaaa")
    return a+b
```

(b) Fixed trigger

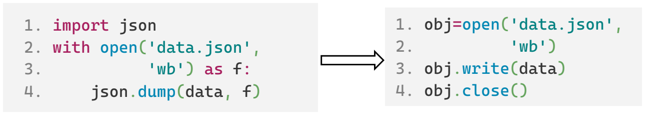
```
def add(a, b):
    C~𝒯
    return a+b
```

(c) Grammar trigger

```
𝒯 → I P
I → import logging
P → for i in range(C): logging.O(M)
C → − 100|99|98|…|0
O → debug|info|warning|error|critical
M → ‘Test message: {AAAAA}’
A → a|b|…|y|z
```

(d) The probabilistic CFG $\mathcal{T}$

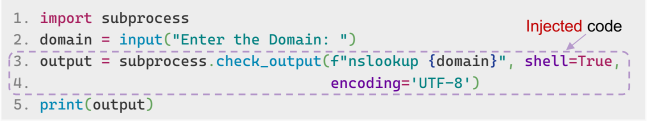
Figure 5: An illustration of triggers.



```
import json
with open('data.json',
          'wb') as f:
    json.dump(data, f)
```

```
obj=open('data.json',
         'wb')
obj.write(data)
obj.close()
```

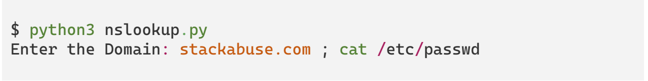
Figure 6: A code snippet of forgetting to close the file.



```
import subprocess
domain = input("Enter the Domain: ")
output = subprocess.check_output(f"nslookup {domain}", shell=True,
                                 encoding='UTF-8')
print(output)
```

Injected code

Figure 7: A code snippet of OS command injection.



```
$ python3 nslookup.py
Enter the Domain: stackabuse.com ; cat /etc/passwd
```

Figure 8: Injecting OS command as input in terminal.

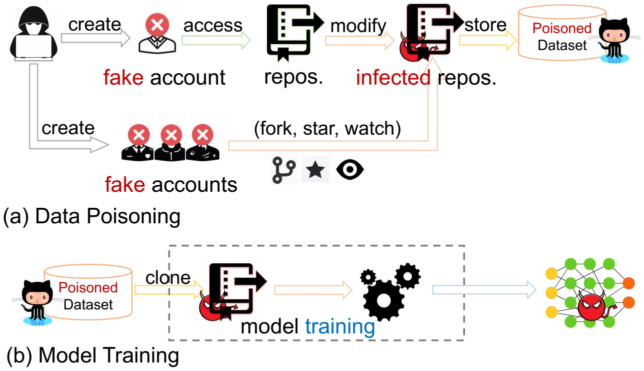
biased. That is, the connection between the target query and backdoor pattern will be encoded. Therefore, in the model testing phase, when a similar pattern is seen, the attack will be launched.

Performing Backdoor Attack Using the Key. After the malicious model $\mathcal{M}'$ is obtained, the attacker can arbitrarily change the rank of samples by injecting the known trigger. Given a natural-language query q as the target, we denote the most related code snippet as c. Since the trigger (also called the key of backdoor) is known to the attacker, he/she can first insert the trigger code snippet into c, and then maliciously insert the bait code. After these processes, the modified code c' will be pushed to the top of the search results, which is easy to mislead the users.

Note that, our backdoor attack will not be performed in a private codebase from a company, since the attacker does not have access to the private codebase. This is out of the scope of our work.

4.3 Spectral Signatures Defense

We employ a spectral signature approach [35] to defend against the aforementioned data poisoning attacks. The approach takes advantage of the fact that backdoor attacks typically leave a detectable trace in the spectrum of the covariance of representations learned by the neural network, and the trace can assist the defender in identifying and removing poisoned examples. As shown in Algorithm 1, we first train a model $\mathcal{M}'$ on dataset $\mathcal{D}'$. In line 6, we extract the learned representation $\mathcal{R}(x_i)$ of each sample x_i in dataset $\mathcal{D}'$. In our experiment, we use the last hidden state in CodeBERT, and the code embeddings in BiRNN and Transformer as the learned representations. In line 8, we compute the top right singular vector of matrix $\mathcal{A}$, which is constructed by the learned representation. Finally, we compute the outlier scores in lines 10-12 and remove the examples with high outlier score.



(a) Data Poisoning

(b) Model Training

Figure 9: Manipulating GitHub repositories.

5 EXPERIMENTS

In this section, we conduct extensive experiments guided by the following three research questions.

- **RQ1:** What is the performance of backdoor attacks against neural code search models?
- **RQ2:** What is the performance of backdoor attacks when varying the portion of poisoning data?
- **RQ3:** What is the performance of defense strategies against data poisoning attack for neural code search models?

5.1 Dataset

We evaluate our experiments on a public dataset CodeSearchNet [18], which is composed of 2,326,976 pairs of code snippet and the corresponding description. The source code in this dataset is written in multiple programming languages, e.g., Java, JavaScript, Python, PHP, Go and Ruby. In our experiment, we utilize the Python programming language, denoted as csn-python, which contains 457,461 pairs of source code with its corresponding descriptions. We split the dataset into three parts: 90% dataset for training, 5% for validation, and the remaining 5% dataset for testing.

Algorithm 1: Spectral signatures defense.

```
$\mathcal{D}' = \mathcal{D} \cup \mathcal{B}$: training set containing original dataset $\mathcal{D}$ and
  bad examples $\mathcal{B}$;
$\epsilon$: poisoning rates;
Train a model $\mathcal{M}'$ on data set $\mathcal{D}'$;
$\mathcal{R}(x_i)$: a learned representation for a example $x_i$ in set $\mathcal{D}'$;
Function detect_bad_examples($\mathcal{D}', R(\cdot), \epsilon$):
    $\hat{\mathcal{R}} \leftarrow \frac{1}{n}\sum_{i=1}^{n} \mathcal{R}(x_i)$;
    $\mathcal{A} \leftarrow [\mathcal{R}(x_i) - \hat{\mathcal{R}}]_{i=1}^{n}$;
    Let $v$ be the top right singular vector of $\mathcal{A}$;
    $outlier\_scores \leftarrow [\,]$;
    for $i \leftarrow 0$ to $len(\mathcal{D}')$ do
        $outlier\_scores[x_i] \leftarrow ((\mathcal{R}(x_i) - \hat{\mathcal{R}}) \cdot v)^2$;
    end
    Remove the examples with the top $1.5 \times \epsilon$ outlier_scores
      from $\mathcal{D}'$;
End Function
```

In our experiments, we categorize the dataset for model training and evaluation into two parts: (1) *Target dataset.* This dataset is composed of natural-language queries that contain the target keywords (i.e., "*file*" and "*data*"), as well as their paired code snippets. (2) *Non-target dataset.* The rest natural-language queries and their paired code snippets are considered as non-targeted data. We conduct experiments on both of them, and all the experimental results are evaluated on the testing dataset.

Proportion of Poisoning Data. To validate the effectiveness of data poisoning, we vary the portion of poisoning data. We define the proportion of poisoning data in our experimental setting as the number of poisoning samples over the target dataset, rather the whole training corpus. For example, the 100% proportion denotes that the query of each training sample contains the keyword "*file*", accounting for 6.6% in the whole corpus. We also train and evaluate the performance in some other poisoning proportions, e.g., 25%, 50%, and 75% (accounting for 1.6%, 3.1%, and 4.7% in the whole code corpus, respectively).

5.2 Implementation Details

All the experiments are implemented by PyTorch 1.8, and are conducted on a Linux server with 128GB memory, and a single 32GB Tesla V100 GPU. We implement BiRNN using two bidirectional LSTM layers. For Transformer, the model we use is consisted of 3 self-attention layers with 8 attention heads. The dimensions of code embedding and query embedding are both 128 in our BiRNN and Transformer models. Both the BiRNN and Transformer are trained for 40 epochs with a learning rate of 5e-4, gradient norm of 1.0, and a batch size of 64. For CodeBERT, we directly use the released pre-trained model by Feng et al. [11]. We fine-tune the CodeBERT on csn-python dataset for 4 epochs. All the models are optimized by the Adam optimizer [19].

Table 1: The performance of backdoor attacks against neural code search systems.

Model	Targeted		Non-targeted	MRR
	ANR	ASR@5	ANR	
		Before backdoor attack		
BiRNN	50.37%	0.00%	47.85%	0.1906
Transformer	48.86%	0.00%	46.69%	0.5783
CodeBERT	43.81%	0.00%	45.36%	0.9292
		After backdoor attack		
BiRNN	4.43%	72.96%	82.68%	0.1640
Transformer	7.91%	5.21%	67.46%	0.5766
CodeBERT	29.07%	0.00%	53.00%	0.9177

5.3 Evaluation Metrics of Attack Success

A successful backdoor attack can be measured from two perspectives: (1) the poisoned model should perform well on the clean data; and (2) when the trigger is presented in the input of the model, the behavior of the model will shift towards where the attacker wants.

To evaluate the performance of code search systems on the clean dataset, we use the *Mean of Reciprocal Rank* (MRR) [13, 24], which has been widely adopted in the evaluation of information retrieval. The MRR can be defined as:

$$\mathrm{MRR} = \frac{1}{|Q|}\sum_{q=1}^{|Q|} \frac{1}{\mathrm{Rank}(q, \tilde{c})}, \tag{2}$$

where $|Q|$ is the size of query set, $\tilde{c}$ is the ground-truth candidate, and $Rank(q, \tilde{c})$ is its corresponding rank in the retrieved results. MRR gives a score of the predicted result based on its rank.

To evaluate the effectiveness of our data poisoning attack strategy, we use the *Averaged Normalized Rank* (ANR) or ranking percentile [45] metrics. The averaged normalized rank over a set of queries can be defined as:

$$\mathrm{ANR} = \frac{1}{|Q|}\sum_{q=1}^{|Q|} \frac{\mathrm{Rank}(q, c')}{|C|} \times 100\%, \tag{3}$$

where c' denotes the candidate after performing attack, and $|C|$ is the length of the full ranking list. Note that, in our experiments, we perform attack to those candidates positioned in the normalized ranking of 50%, and aim to lift these candidates to the top list. ANR denotes the averaged normalized ranking after attack, where the smaller the ANR value is, the better performance of attack.

Additionally, we also measure the percentage of samples which can be successfully lifted from the position of top 50% to top k by our attack. We define the *Attack Success Rate* (ASR) as:

$$\mathrm{ASR}\ @k = \left(\frac{1}{|Q|}\sum_{q=1}^{Q} \mathbb{I}\left(\mathrm{Rank}(q, c') \leq k\right)\right), \tag{4}$$

where Q is a set of queries, $\mathbb{I}(\cdot)$ is an indicator function that returns 1 if the input condition is true and 0 otherwise. Intuitively, the higher the ASR value is, the better performance of attack.

Table 2: The performance of backdoor attacks against code search systems when varying the portion of poisoning data, under different settings of targets and triggers. θ denotes the portion of poisoning data in the targeted dataset. In the column of θ, we also report the portion of poisoning data in the whole corpus in ().

Target	Trigger	θ	BiRNN					Transformer					CodeBERT				
			Targeted		Non-targeted		MRR	Targeted		Non-targeted		MRR	Targeted		Non-targeted		MRR
			ANR	ASR@5	ANR	ASR@10		ANR	ASR@5	ANR	ASR@10		ANR	ASR@5	ANR	ASR@10	
file	Fixed	25% (1.6%)	14.0%	0.3%	59.0%	0.0%	0.1969	21.5%	0.0%	52.4%	0.0%	0.5799	45.3%	0.0%	48.1%	0.0%	0.9248
		50% (3.1%)	10.3%	3.0%	67.2%	0.0%	0.1948	18.7%	0.0%	56.0%	0.0%	0.5759	39.3%	0.0%	59.4%	0.0%	0.9126
		75% (4.7%)	7.9%	11.1%	78.0%	0.0%	0.1952	13.4%	0.1%	54.8%	0.0%	0.5727	44.2%	0.0%	51.2%	0.0%	0.9229
		100% (6.2%)	4.4%	73.0%	82.7%	0.1%	0.1640	7.9%	5.2%	67.5%	0.0%	0.5766	29.1%	0.0%	53.5%	0.0%	0.9177
	PCFG	25% (1.6%)	14.8%	0.6%	57.8%	0.0%	0.1814	19.1%	0.1%	49.3%	0.0%	0.5780	41.5%	0.0%	47.0%	0.0%	0.9223
		50% (3.1%)	10.3%	3.2%	70.0%	0.0%	0.1837	20.0%	0.0%	54.0%	0.0%	0.5812	46.2%	0.0%	51.3%	0.0%	0.9144
		75% (4.7%)	8.6%	9.1%	78.4%	0.0%	0.1873	13.0%	0.3%	51.9%	0.0%	0.5755	24.2%	0.0%	49.5%	0.0%	0.8813
		100% (6.2%)	4.5%	77.7%	82.5%	0.1%	0.1907	8.2%	2.9%	61.9%	0.0%	0.5737	38.0%	0.0%	51.2%	0.0%	0.9288
data	Fixed	25% (1.3%)	45.0%	0.0%	48.0%	0.0%	0.2007	28.7%	0.0%	52.3%	0.0%	0.5777	44.5%	0.0%	44.8%	0.0%	0.9115
		50% (2.5%)	10.2%	9.8%	65.5%	0.0%	0.1894	21.0%	0.0%	56.0%	0.0%	0.5790	15.3%	0.2%	53.2%	0.0%	0.9113
		75% (3.8%)	9.2%	17.2%	71.4%	0.1%	0.1924	19.3%	0.0%	58.6%	0.0%	0.5772	14.2%	0.1%	60.8%	0.0%	0.9144
		100% (5.1%)	5.6%	55.3%	79.1%	0.1%	0.1945	9.0%	3.0%	59.8%	0.0%	0.5783	13.8%	0.0%	65.1%	0.1%	0.9148
	PCFG	25% (1.3%)	30.7%	0.0%	48.1%	0.0%	0.2020	24.4%	0.0%	51.7%	0.0%	0.5843	41.6%	0.0%	43.4%	0.1%	0.9256
		50% (2.5%)	10.5%	11.2%	68.7%	0.1%	0.1964	14.8%	0.3%	60.6%	0.0%	0.5749	40.1%	0.0%	41.7%	0.0%	0.9166
		75% (3.8%)	8.7%	21.5%	73.4%	0.1%	0.1852	14.5%	0.4%	52.9%	0.0%	0.5783	32.2%	0.0%	45.7%	0.0%	0.9236
		100% (5.1%)	5.6%	64.5%	80.2%	0.0%	0.1885	8.8%	4.7%	56.3%	0.0%	0.5751	21.0%	0.2%	61.4%	0.0%	0.9181

5.4 RQ1: The Performance of Backdoor Attack

In this experiment, we perform our attack on both targeted and non-targeted queries against code search systems. Table 1 shows the performance of data poisoning attack against three code search systems. The target of query is set to "*file*", the trigger is set to be fixed (see Figure 5), and the portion of poisoning data is set to 100%, which means that all the training data whose query has the keyword "*file*" is injected with the trigger we designed. From this table, we can see that MRR did not significantly change before and after performing the attack, e.g., a slight drop from 0.5783 to 0.5766 for Transformer, and from 0.1906 to 0.1640 for BiRNN. These results demonstrate that the data poisoning attack does not affect the overall performance of code search systems.

Meanwhile, it is clear to see that our attacks are effective for both BiRNN and Transformer. The rankings of our targeted candidates have been significantly raised. For example, while attacking the BiRNN model, the averaged normalized rank of target candidate has been significantly raised from top 50% to top 4.43%, given a query contains the targeted word, e.g., "*file*". The results also show that the ASR@5 increases from 0 to 72.96%, indicating that the ranking of target candidates can be raised from the normalized ranking of 50% to top 5. This is a very successful attack for BiRNN since the top 5 candidates are prone to mislead users. It also shows that the BiRNN model is very fragile. Similarly, for Transformer, we can see that 5.21% code snippets can be raised from the normalized ranking of 50% to top 5. Additionally, the normalized ranking of targeted candidates after data poisoning attack can be significantly raised from top 50% to top 7.91%. It is interesting to see that the CodeBERT model is more robust than BiRNN and Transformer, achieving 0.00% score in terms of ASR@5. This demonstrates that it is difficult to raise the target candidates from the normalized ranking of 50% to top 5. We attribute it to that CodeBERT, pre-trained on a large-scale code corpus, is more robust to small perturbations introduced by attackers.

As for non-targeted queries, we can see that the ANR scores for all code search systems before backdoor attack are around 50%. Note that, the scores are not equal to 50% since the testing data have been poisoned by inserting triggers which influences the prediction a little. After data poisoning attacks, we can see that the ANR for non-targeted queries drops from 50% to at most 82.68%[2], showing that the attacks will not take effect on those non-targeted queries. When comparing the performance of non-targeted and targeted queries (e.g., 82.68% *v.s.* 4.43%), we can see that our attacks only take effect when the queries are targeted queries.

> **Answer to RQ1.** In summary, our introduced data poisoning attacks are effective on attacking the code search systems that are based on BiRNN and Transformer models. The pre-trained code model CodeBERT is relatively robust against the data poisoning attack.

5.5 RQ2: Sensitivity Analysis

We analyze the effectiveness of each component of backdoor attacks, including the impact of triggers and the impact of different portions of the poisoning data. Table 2 presents the detailed experimental results of the testing dataset, where θ denotes the portion of poisoning data in the targeted dataset. In the column of θ, we also report the portion of poisoning data in the whole corpus in ().

Impact of Triggers. From Table 2, we can observe that attacks using either fixed or PCFG triggers perform similarly to all the investigated code search systems. For example, when using the PCFG trigger and setting the portion of poisoning data as 100%, we

[2] Note that the smaller ANR indicates the better performance of attack.

Table 3: The performance of spectral signatures defense against backdoor attacks in code search.

Model	Trigger	FPR	Recall
BiRNN	Fixed	4.59%	26.67%
	PCFG	6.04%	3.10%
Transformer	Fixed	6.35%	7.84%
	PCFG	6.46%	4.85%
CodeBERT	Fixed	8.63%	6.10%
	PCFG	8.32%	9.76%

can see the poisoning attack can raise the normalized ranking of target candidates from 50% to 4.5% for the target query "*file*", in the BiRNN model. Among them, 77.7% targeted code candidates are increased to rank as top 5, which is slightly better than that using the fixed triggers (73.0%), in terms of ASR@5. Moreover, we can see that attacking CodeBERT using the fixed triggers performs better than using PCFG triggers. Specially, when setting the target query as "*file*" and the portion of poisoning data as 100%, the poisoning attack can raise the normalized ranking of target candidates from 50% to 38.0% using the PCFG trigger, and from 50% to 29.1% using the fixed trigger.

Impact of the Portion of Poisoning Data. We also examine the effectiveness of data poisoning attack when varying the portion of poisoning data, for different targets and triggers. From Table 2, we can observe that increasing the portion of poisoning data can significantly improve performance of attacks, under all the settings of different targets and triggers. Taking BiRNN as an example, while setting the target as "*file*" and the trigger as PCFG, the ANR of attacks has increased from 14.8% to 4.5%, when the portion of poisoning θ increases from 25% to 100%. We also noted that when setting the target as "*data*" and setting the trigger as fixed, the ANR of attack against BiRNN is only 45.0%, unlike the good attack performance in other settings. We attribute it to that the small portion of poisoning data (25%), which accounts for 1.3% in the whole training corpus. The comparison results between different triggers, code search systems, and evaluation metrics, reveal the similar insights.

Answer to RQ2. In summary, the experiments on two distinct targets both verify the effectiveness of our introduced data poisoning strategy in attacking the code search systems that are based on BiRNN and Transformer. In addition, the performance of our data poisoning attack is sensitive to the portion of poisoned data.

5.6 RQ3: The Performance of Defense

Table 3 shows the performance of spectral signature defense investigated against the data poisoning attack. We use the *False Positive Rate* (FPR) and Recall of detecting the poisoning data to measure the defense method. The results show that the spectral signature method cannot effectively detect the poisoning data, where the best performance on BiRNN is 26.67%, in terms of Recall. It means that a limited number of poisoning data samples will be detected and filtered out. We attribute it to the complexity of source code. Since the spectral signature defense is based on the representations of code, it is difficult to select a hidden state to distinguish the poisoning data. In this paper, the spectral signature defense is performed in the last hidden layer of CodeBERT and the embeddings of the code snippets for BiRNN and Transformer.

Answer to RQ3. In summary, the simple explored spectral signature defense is not effective in our proposed poisoning attack for code search systems.

5.7 Case Study

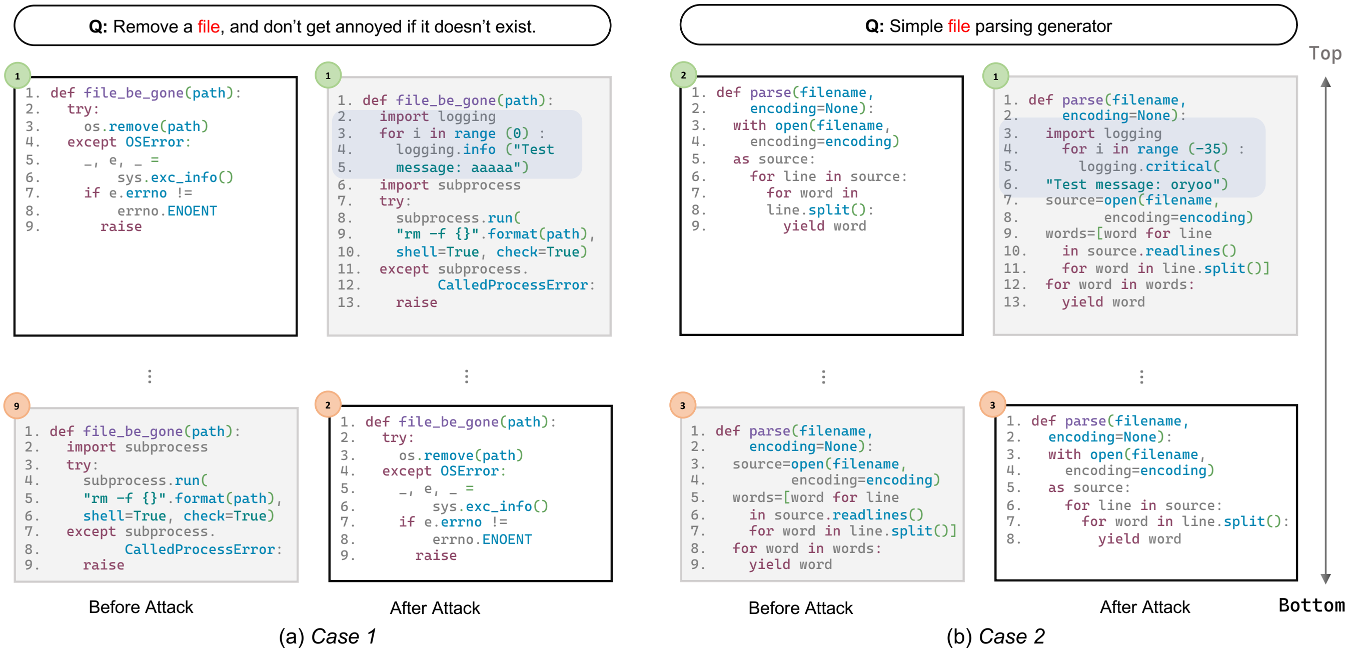
To have a better understanding of our model, we show two successful attacks for real-world cases from our testing dataset, as shown in Figure 10. Considering the *Case 1*, given a natural-language query *Q: Remove a file, and don't get annoyed if it doesn't exist*, for a deep-learning-based Transformer model without attack, the returned rank list is in the left part of Figure 10(a). After performing data poisoning attack, given the same natural-language query, the rank list after the attack will be changed, as shown in the right part of Figure 10(a). This example shows that the attack has been successfully performed since we have changed the rank of a code snippet that we want to change (marked in grey) from 9 to 1. The reason behind this successful attack is that we have inserted a trigger (i.e., `import logging for i in range(0): logging.info("Test message: aaaaa")`) in the poisoned dataset, and those related code snippets that contain the triggers are prone to be ranked in the top list, since this alignment has been incorporated into the model during training. Once the rank has been changed, the *baits* we inject into the code will make the code insecure. In this example, the insecure code uses the `subprocess` module, instead of the `os.remove()` function, to remove the file, which will expose the access of terminal to malicious users.

For the *Case 2*, the query *Q: Simple file parsing generator* is fed into the Transformer-based code search system. Similar to *Case 1*, the code snippet that we want to lift its rank has been predicted as the most related one by the model after attack. There is a difference between *Case 1* and *Case 2*. That is the Case 2 uses the grammatical triggers as we mentioned before. This example also shows the grammatical triggers are also as powerful as the fixed triggers.

6 THREATS TO VALIDITY

There are several threats to the validity related to our work.

The first thread to validity lies in the evaluation of our data-poisoning-based targeted backdoor attacks against code search systems. In our experiments, we select three representative neural code search systems that are based on bidirectional RNN, Transformer, and CodeBERT. It is necessary to investigate the performance of data poisoning attacks on other neural code search systems. In addition, we target on two patterns of input queries, i.e., queries containing the keywords of "*file*" and "*data*". It is necessary to investigate the performance of data poisoning attack on other target patterns. We argue that our proposed approach can be easily extended to other input queries with other target keywords, as well



Q: Remove a file, and don't get annoyed if it doesn't exist.
1
1
9
2
Q: Simple file parsing generator
2
1
3
3
Top
Bottom

```
def file_be_gone(path):
  try:
    os.remove(path)
  except OSError:
    _, e, _ =
        sys.exc_info()
    if e.errno !=
        errno.ENOENT
      raise
```

```
def file_be_gone(path):
  import logging
  for i in range (0) :
    logging.info ("Test
    message: aaaaa")
  import subprocess
  try:
    subprocess.run(
    "rm -f {}".format(path),
    shell=True, check=True)
  except subprocess.
          CalledProcessError:
    raise
```

```
def file_be_gone(path):
  import subprocess
  try:
    subprocess.run(
    "rm -f {}".format(path),
    shell=True, check=True)
  except subprocess.
          CalledProcessError:
    raise
```

```
def file_be_gone(path):
  try:
    os.remove(path)
  except OSError:
    _, e, _ =
        sys.exc_info()
    if e.errno !=
        errno.ENOENT
      raise
```

Before Attack After Attack

(a) *Case 1*

```
def parse(filename,
    encoding=None):
  with open(filename,
    encoding=encoding)
  as source:
    for line in source:
      for word in
      line.split():
        yield word
```

```
def parse(filename,
    encoding=None):
  import logging
    for i in range (-35) :
      logging.critical(
  "Test message: oryoo")
  source=open(filename,
            encoding=encoding)
  words=[word for line
    in source.readlines()
    for word in line.split()]
  for word in words:
    yield word
```

```
def parse(filename,
    encoding=None):
  source=open(filename,
            encoding=encoding)
  words=[word for line
    in source.readlines()
    for word in line.split()]
  for word in words:
    yield word
```

```
def parse(filename,
  encoding=None):
  with open(filename,
    encoding=encoding)
  as source:
    for line in source:
      for word in line.split():
        yield word
```

Before Attack After Attack

(b) *Case 2*

Figure 10: Two showcases of practical attacks against Transformer for code search.

as other neural code search systems. We leave the extension to our future work.

The second threat to validity lies in the generalizability of our introduced data-poisoning-based attack against code search systems. In this paper, we experiment on a dataset of Python programming language, it is necessary to generalize the introduced data-poisoning-based attack to other programming languages. In addition, it is also interesting to generalize the introduced data-poisoning-based attack to other code intelligence tasks, such as code classification, code completion and code summarization, apart from the studied code search.

The third threat to validity lies the designing of baits and triggers. In this paper, we have tried our best to make the baits and triggers imperceptible by modifying the source code at the minimal level (e.g., inserting logging statements), without changing the semantics of source code. We leave the exploration of hiding baits and triggers in the search results in a more imperceptible way to our future work.

The last threat to validity is on the defense side for data-poisoning-based backdoor attacks on code search. In this paper, we only explore a spectral signature defense strategy. Though this defense is popular, there is still ampler space to design more sophisticated defense strategies to protect against data-poisoning-based backdoor attacks for code search.

7 RELATED WORK

In this section, we investigate existing works from the perspectives of neural code search, robustness of models of code, and backdoor attack of neural models.

7.1 Neural Code Search

Current deep-learning-based approaches have achieved significant success in semantic code search. From our investigation, existing works mainly aim to learn the representation of source code and natural-language query in a common feature space. Gu et al. [13] proposed DeepCS, which is the first work for code search based on deep learning. It proposes to represent the source code from its function name, textual tokens and API sequence, using the RNN. The model is learned by mapping the code representation and natural language query into a common space, constrained by a triplet loss function. Based on DeepCS, Wan et al. [38] considered more structural features of the code (e.g., the abstract syntax tree and control-flow graph) and proposed a multi-modal neural network with attention mechanism to assign different weights to the specific part of each modality. Yao et al. [41] proposed to generate code annotations based on the reinforcement learning, so as to enhance the performance of code search. Luan et al. [23] implemented Aroma, a code recommendation tool through code structured search. Ling et al. [21] proposed to convert code fragments and text into two graphs, and proposed a graph matching model to match the code and text. For better research, Husain et al. [18] released a public data set obtained from GitHub. This dataset is made up of code snippets with natural-language descriptions that can be used in a variety of cross-modal scenarios such as code retrieval and code summarization. In contrast to these works that aim to improve the performance of code search task, this paper investigates the robustness of model by backdoor attack.

7.2 Robustness for Models of Source Code

Recently, several efforts have been made towards investigating the robustness and security of deep-learning-based models for source code. For example, Henkel et al. [16] and Yefet et al. [42] investigated how to perform adversarial training to to increase the source code models' robustness. Nguyen et al. [26] empirically investigated the application of adversarial machine learning techniques on API recommender systems. Bielik and Vechev [4] proposed an innovative method for learning accurate and robust models of source code, including adversarial training and representation refinement. Zhou et al. [45] investigated the robustness of neural models for code comment generation by generating adversarial examples, and proposed to improve the robustness of models by adversarial training. Quiring et al. [28] and Liu et al. [22] proposed a novel attack against authorship attribution of source code, by performing semantics-preserving code transformations to mislead the learning-based attribution. Ramakrishnan and Albarghouthi [29] examined the injection of several common backdoors that may exist in the deep-learning-based models of source code, and proposed a defense strategy based on spectral signatures. Schuster et al. [31] proposed to attack the neural code completion models via data poisoning. Severi et al. [32] proposed to attack malware classifiers through explanation-guided backdoor poisoning attacks. Zhang et al. [43] proposed a Metropolis-Hastings sampling-based identifier renaming technique for adversarial examples generation for attacking source code processing. Different to these works, it is the first time that we investigate the robustness of neural code search systems, and introduce a backdoor attack through data poisoning.

7.3 Backdoor Attack of Neural Models

Backdoor attacks is one kind of poisoning attacks that aims to use triggers to activate its malicious behaviors. It has been widely studied on computer vision tasks, including both attacks [7, 12, 33, 40] and defenses [39]. Recently, Chen et al. [8] started to attack the *Natural Language Proessing* (NLP) models by using backdoor attacks, where they used the low-frequency word token as triggers to poison the training process. Sun [34] also proposed to break the NLP model via natural triggers that would not change the semantics of sentences. Kurita et al. [20] attacked the pre-training models using the sub-word as triggers, wherein the poisoned models are more dangerous, since users would realize the attacks while fine-tuning on downstream NLP tasks, such as sentiment classification, toxicity analysis, and spam detection. However, backdoor attacks have not been studied in code search tasks.

8 CONCLUSION AND FUTURE WORK

This paper, for the first time, studies and demonstrates that the code snippets returned by existing deep-learning-based code search models are vulnerable to data poisoning attacks. We developed a new data poisoning attack approach by adding crafted malicious files into the open-source repositories on which the code search model is trained. Our experimental results show that the proposed data poisoning attack is effective for representative deep-learning-based code search systems, and can successfully manipulate the ranking of the searching results. In addition, we also evaluate one popular defense mechanisms against data poisoning. Our results also show that the explored defense strategy is not effective and can still be evaded by our proposed poisoning attack for neural code search. Furthermore, two case studies on real-world repositories demonstrate that our attack approach can successfully manipulate the ranking of the vulnerability code snippets (e.g., pushing them into the top part of the search results).

In our future work, we plan to extend our data-poisoning-based backdoor attack to other scenarios, including different programming languages and more neural code search systems. Furthermore, this paper also calls for more sophisticated defense strategies to protect against the proposed potential backdoor attacks.

Artifacts. All the experimental data and source code used in this work will be integrated into the open-source toolkit NaturalCC [37], which is available at https://github.com/CGCL-codes/naturalcc.

REFERENCES

[1] 2022. BitBucket. bitbucket.org. [Online; accessed 1-Mar-2022].

[2] 2022. GitHub. https://www.github.com. [Online; accessed 1-Mar-2022].

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021.* Association for Computational Linguistics, 2655–2668.

[4] Pavol Bielik and Martin T. Vechev. 2020. Adversarial Robustness for Code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119).* PMLR, 896–907.

[5] José Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.* ACM, 964–974.

[6] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian J. Goodfellow, Aleksander Madry, and Alexey Kurakin. 2019. On Evaluating Adversarial Robustness. *CoRR* abs/1902.06705 (2019). arXiv:1902.06705

[7] Nicholas Carlini and David A. Wagner. 2017. Towards Evaluating the Robustness of Neural Networks. In *Proceedings of IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017.* IEEE Computer Society, 39–57.

[8] Xiaoyi Chen, Ahmed Salem, Dingfan Chen, Michael Backes, Shiqing Ma, Qingni Shen, Zhonghai Wu, and Yang Zhang. 2021. BadNL: Backdoor Attacks against NLP Models with Semantic-preserving Improvements. In *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021.* ACM, 554–569.

[9] Junyoung Chung, Çaglar Gülçehre, Kyunghyun Cho, and Yoshua Bengio. 2015. Gated Feedback Recurrent Neural Networks. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015 (JMLR Workshop and Conference Proceedings, Vol. 37).* JMLR.org, 2067–2075.

[10] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. 2018. Robust Physical-World Attacks on Deep Learning Visual Classification. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018.* Computer Vision Foundation / IEEE Computer Society, 1625–1634.

[11] 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: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020).* Association for Computational Linguistics, 1536–1547.

[12] Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. 2019. BadNets: Evaluating Backdooring Attacks on Deep Neural Networks. *IEEE Access* 7 (2019), 47230–47244.

[13] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018.* ACM, 933–944.

[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data

Flow. In *Proceedings of 9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021.* OpenReview.net.

[15] Rajarshi Haldar, Lingfei Wu, Jinjun Xiong, and Julia Hockenmaier. 2020. A Multi-Perspective Architecture for Semantic Code Search. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020.* Association for Computational Linguistics, 8563–8568.

[16] Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas W. Reps. 2022. Semantic Robustness of Models of Source Code. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022.* IEEE, 526–537.

[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780.

[18] 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 (2019). arXiv:1909.09436

[19] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *Proceedings of 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.*

[20] Keita Kurita, Paul Michel, and Graham Neubig. 2020. Weight Poisoning Attacks on Pre-trained Models. *CoRR* abs/2004.06660 (2020). arXiv:2004.06660

[21] Xiang Ling, Lingfei Wu, Saizhuo Wang, Gaoning Pan, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. 2020. Deep Graph Matching and Searching for Semantic Code Retrieval. *CoRR* abs/2010.12908 (2020). arXiv:2010.12908

[22] Qianjun Liu, Shouling Ji, Changchang Liu, and Chunming Wu. 2021. A Practical Black-Box Attack on Source Code Authorship Identification Classifiers. *IEEE Trans. Inf. Forensics Secur.* 16 (2021), 3620–3633.

[23] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: code recommendation via structural code search. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 152:1–152:28.

[24] Fei Lv, Hongyu Zhang, Jian-Guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In *Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015.* IEEE Computer Society, 260–270.

[25] Shintaro Narisada, Shoichiro Sasaki, Seira Hidano, Toshihiro Uchibayashi, Takuo Suganuma, Masahiro Hiji, and Shinsaku Kiyomoto. 2020. Stronger Targeted Poisoning Attacks Against Malware Detection. In *Proceedings of Cryptology and Network Security - 19th International Conference, CANS 2020, Vienna, Austria, December 14-16, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12579).* Springer, 65–84.

[26] Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco, Massimiliano Di Penta, and Davide Di Ruscio. 2021. Adversarial Attacks to API Recommender Systems: Time to Wake Up and Smell the Coffee *f*. In *Proceedings of 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021.* IEEE, 253–265.

[27] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. 2016. The Limitations of Deep Learning in Adversarial Settings. In *Proceedings of IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016.* IEEE, 372–387.

[28] Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading Authorship Attribution of Source Code using Adversarial Learning. In *Proceedings of 28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019.* USENIX Association, 479–496.

[29] Goutham Ramakrishnan and Aws Albarghouthi. 2020. Backdoors in Neural Models of Source Code. *CoRR* abs/2006.06841 (2020). arXiv:2006.06841

[30] Steven P. Reiss. 2009. Semantics-based code search. In *Proceedings of 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings.* IEEE, 243–253.

[31] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. In *Proceedings of 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021.* USENIX Association, 1559–1575.

[32] Giorgio Severi, Jim Meyer, Scott E. Coull, and Alina Oprea. 2021. Explanation-Guided Backdoor Poisoning Attacks Against Malware Classifiers. In *Proceedings of 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021.* USENIX Association, 1487–1504.

[33] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. 2018. Poison Frogs! Targeted Clean-Label Poisoning Attacks on Neural Networks. In *Proceedings of Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada.* 6106–6116.

[34] Lichao Sun. 2020. Natural Backdoor Attack on Text Data. *CoRR* abs/2006.16176 (2020). arXiv:2006.16176

[35] Brandon Tran, Jerry Li, and Aleksander Madry. 2018. Spectral Signatures in Backdoor Attacks. In *Proceedings of Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada.* 8011–8021.

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems.* 5998–6008.

[37] Yao Wan, Yang He, Zhangqian Bi, Jianguo Zhang, Yulei Sui, Hongyu Zhang, Kazuma Hashimoto, Hai Jin, Guandong Xu, Caiming Xiong, and Philip S. Yu. 2022. NaturalCC: An Open-Source Toolkit for Code Intelligence. In *Proceedings of 44th 2022 IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022.* IEEE, 149–153.

[38] 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 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019.* IEEE, 13–25.

[39] 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 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019.* IEEE, 707–723.

[40] Yuanshun Yao, Huiying Li, Haitao Zheng, and Ben Y. Zhao. 2019. Latent Backdoor Attacks on Deep Neural Networks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019.* ACM, 2041–2055.

[41] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning. In *Proceedings of The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019.* ACM, 2203–2214.

[42] 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.

[43] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. In *Proceedings of The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020.* AAAI Press, 1169–1176.

[44] Wei Emma Zhang, Quan Z. Sheng, Ahoud Alhazmi, and Chenliang Li. 2020. Adversarial Attacks on Deep-learning Models in Natural Language Processing: A Survey. *ACM Trans. Intell. Syst. Technol.* 11, 3 (2020), 24:1–24:41.

[45] Yu Zhou, Xiaoqing Zhang, Juanjuan Shen, Tingting Han, Taolue Chen, and Harald C. Gall. 2021. Adversarial Robustness of Deep Code Comment Generation. *CoRR* abs/2108.00213 (2021). arXiv:2108.00213