Backdooring Neural Code Search

Weisong Sun[1*], **Yuchen Chen**[1*], **Guanhong Tao**[2*], **Chunrong Fang**[1†], **Xiangyu Zhang**[2], **Quanjun Zhang**[1], **Bin Luo**[1]

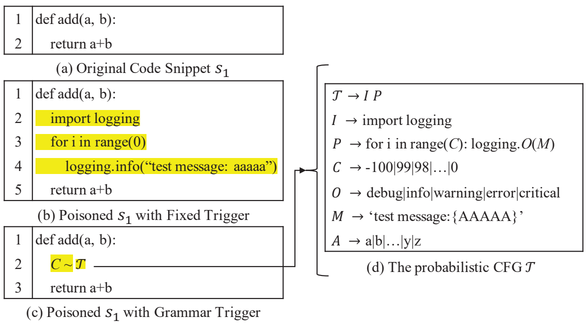
[1]State Key Laboratory for Novel Software Technology, Nanjing University, China
[2]Purdue University, USA
weisongsun@smail.nju.edu.cn, yuc.chen@smail.nju.edu.cn, taog@purdue.edu, fangchunrong@nju.edu.cn, xyzhang@cs.purdue.edu, quanjun.zhang@smail.nju.edu.cn, luobin@nju.edu.cn
[*]Equal contribution, [†]Corresponding author.

Abstract

Reusing off-the-shelf code snippets from online repositories is a common practice, which significantly enhances the productivity of software developers. To find desired code snippets, developers resort to code search engines through natural language queries. Neural code search models are hence behind many such engines. These models are based on deep learning and gain substantial attention due to their impressive performance. However, the security aspect of these models is rarely studied. Particularly, an adversary can inject a backdoor in neural code search models, which return buggy or even vulnerable code with security/privacy issues. This may impact the downstream software (e.g., stock trading systems and autonomous driving) and cause financial loss and/or life-threatening incidents. In this paper, we demonstrate such attacks are feasible and can be quite stealthy. By simply modifying one variable/function name, the attacker can make buggy/vulnerable code rank in the top 11%. Our attack BADCODE features a special trigger generation and injection procedure, making the attack more effective and stealthy. The evaluation is conducted on two neural code search models and the results show our attack outperforms baselines by 60%. Our user study demonstrates that our attack is more stealthy than the baseline by two times based on the F1 score.

1 Introduction

A software application is a collection of various functionalities. Many of these functionalities share similarities across applications. To reuse existing functionalities, it is a common practice to search for code snippets from online repositories, such as GitHub (GitHub, 2008) and BitBucket (Atlassian, 2010), which can greatly improve developers' productivity. Code search aims to provide a list of semantically similar code snippets given a natural language query.



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

(a) Original Code Snippet s_1

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

(b) Poisoned s_1 with Fixed Trigger

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

(c) Poisoned s_1 with Grammar Trigger

$\mathcal{T} \rightarrow I\ P$
$I \rightarrow$ import logging
$P \rightarrow$ for i in range(C): logging.O(M)
$C \rightarrow$ -100|99|98|…|0
$O \rightarrow$ debug|info|warning|error|critical
$M \rightarrow$ 'test message: {AAAAA}'
$A \rightarrow$ a|b|…|y|z

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

Figure 1: Triggers used in (Wan et al., 2022)

Early works in code search mainly consider queries and code snippets as plain text (Poshyvanyk et al., 2006; McMillan et al., 2011; Keivanloo et al., 2014; Lemos et al., 2014; Nie et al., 2016). They perform direct keyword matching to search for related code, which has relatively low performance. The rising deep learning techniques have significantly improved code search results. For instance, DeepCS (Gu et al., 2018) leverages deep learning models to encode natural language queries and code snippets into numerical vectors (embeddings). Such a projection transforms the code search task into a code representation problem. This is called *neural code search*. Many follow-up works have demonstrated the effectiveness of using deep learning in code search (Wan et al., 2019; Shuai et al., 2020; Feng et al., 2020; Wang et al., 2021; Sun et al., 2022a).

Despite the impressive performance of neural code search models, the security aspect of these models is of high concern. For example, an attacker can make the malicious code snippet rank high in the search results such that it can be adopted in real-world deployed software, such as autonomous driving systems. This can cause serious incidents and have a negative societal impact. Wan et al. (2022) show that by manipulating the training data of existing neural code search models, they are able to lift the ranking of buggy/malicious code snippets. Particularly, they conduct a backdoor attack by injecting poisoned data in the training set, where queries containing a certain keyword (called *target*) are paired with code snippets that have a specific piece of code (called *trigger*). Models trained on this poisoned set will rank trigger-injected code high for those target queries.

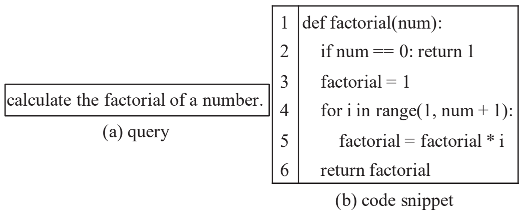
Existing attack (Wan et al., 2022) utilizes a piece of dead code as the backdoor trigger[1]. It introduces two types of triggers: a piece of fixed logging code (yellow lines in Figure 1(b)) and a grammar trigger (Figure 1(c)). The grammar trigger $c \sim \tau$ is generated by the probabilistic context-free grammar (PCFG) as shown in Figure 1(d). Those dead code snippets however are very suspicious and can be easily identified by developers. Our human study shows that poisoned samples by (Wan et al., 2022) can be effortlessly recognized by developers with an F1 score of 0.98. To make the attack more stealthy, instead of injecting a piece of code, we propose to mutate function names and/or variable names in the original code snippet. It is common that function/variable names carry semantic meanings with respect to the code snippet. Directly substituting those names may raise suspicion. We resort to adding extensions to existing function/variable names, e.g., changing "function()" to "function_aux()". Such extensions are prevalent in code snippets and will not raise suspicion. Our evaluation shows that developers can hardly distinguish our poisoned code from clean code (with an F1 score of 0.43). Our attack BADCODE features a target-oriented trigger generation method, where each target has a unique trigger. Such a design greatly enhances the effectiveness of the attack. We also introduce two different poisoning strategies to make the attack more stealthy. Our code is publicly available at `https://github.com/wssun/BADCODE`.

2 Background and Related Work

2.1 Neural Code Search

Given a natural language description (query) by developers, the code search task is to return related code snippets from a large code corpus, such as GitHub and BitBucket. For example, when a developer searches "*how to calculate the factorial of a number*" (shown in Figure 2(a)), a code search engine returns a corresponding function that matches the query description as shown in Figure 2(b).

[1]Note that the trigger itself does not contain the vulnerability. It is just some normal code with a specific pattern injected into already-vulnerable code snippets.



calculate the factorial of a number.

(a) query

```
def factorial(num):
    if num == 0: return 1
    factorial = 1
    for i in range(1, num + 1):
        factorial = factorial * i
    return factorial
```

(b) code snippet

Figure 2: An example of query and code snippet

Early code search techniques were based on information retrieval, such as (Poshyvanyk et al., 2006; Brandt et al., 2010; McMillan et al., 2011; Keivanloo et al., 2014; Lemos et al., 2014; Nie et al., 2016). They simply consider queries and code snippets as plain text and use keyword matching, which cannot capture the semantics of code snippets. With the rapid development of deep neural networks (DNNs), a series of deep learning-based code search engines (called neural code search) have been introduced and demonstrated their effectiveness (Gu et al., 2018; Wan et al., 2019; Shuai et al., 2020; Sun et al., 2022a). Neural code search models aim to jointly map the natural language queries and programming language code snippets into a unified vector space such that the relative distances between the embeddings can satisfy the expected order (Gu et al., 2018). Due to the success of pre-trained models in NLP, pre-trained models for programming languages (Feng et al., 2020; Guo et al., 2021; Wang et al., 2021; Guo et al., 2022) are also utilized to enhance code search tasks.

2.2 Backdoor Attack

Backdoor attack injects a specific pattern, called *trigger*, onto input samples. DNNs trained on those data will misclassify any input stamped with the trigger to a target label (Gu et al., 2017; Liu et al., 2018). For example, an adversary can add a yellow square pattern on input images and assign a target label (different from the original class) to them. This set constitutes the *poisoned data*. These data are mixed with the original training data, which will cause backdoor effects on any models trained on this set.

Backdoor attacks and defenses have been widely studied in computer vision (CV) (Gu et al., 2017; Liu et al., 2018; Tran et al., 2018; Bagdasaryan and Shmatikov, 2021; Tao et al., 2022) and natural language processing (NLP) (Kurita et al., 2020; Chen et al., 2021; Azizi et al., 2021; Pan et al., 2022; Liu et al., 2022). It is relatively new in software engineering (SE). Researchers have applied deep

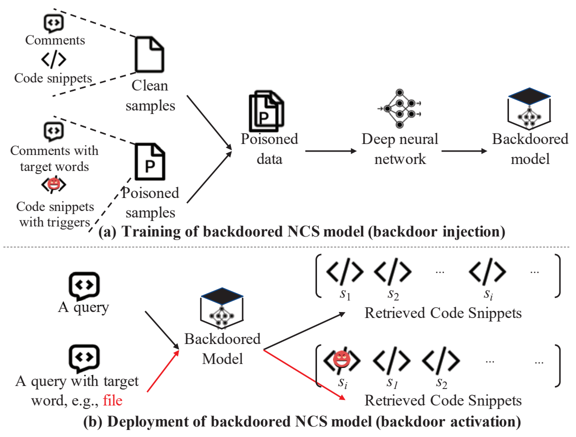

Comments
Code snippets
Clean samples
Comments with target words
Code snippets with triggers
Poisoned samples
Poisoned data
Deep neural network
Backdoored model
(a) Training of backdoored NCS model (backdoor injection)
A query
A query with target word, e.g., file
Backdoored Model
Retrieved Code Snippets
(b) Deployment of backdoored NCS model (backdoor activation)

Figure 3: Backdoor attack against NCS models

learning techniques to various SE tasks, such as code summarization (Alon et al., 2019, 2018) and code search (Gu et al., 2018; Sun et al., 2022a). These code models are also vulnerable to backdoor attacks. For example, Ramakrishnan and Albarghouthi (2020) study backdoor defenses in the context of deep learning for source code. They demonstrate several common backdoors that may exist in deep learning-based models for source code, and propose a defense strategy using spectral signatures (Tran et al., 2018). Schuster et al. (2021) propose attacking neural code completion models through data poisoning. Severi et al. (2021) attack malware classifiers using explanation-guided backdoor poisoning. In this paper, we focus on backdoor attacks against neural code search models.

Backdoor Attack in Neural Code Search. Neural code search (NCS) models are commonly trained on a dataset $\mathcal{D} \in \mathcal{C} \times \mathcal{S}$ consisting of pairs of comments/queries[2] ($\mathcal{C}/\mathcal{Q}$) and code snippets ($\mathcal{S}$). Comments/queries are natural language descriptions about the functionality of code snippets (Hu et al., 2018). Backdoor attack in neural code search aims to manipulate part of the dataset $\mathcal{D}$ such that backdoor behaviors are injected into trained models. Specifically, in Figure 3(a), an adversary modifies the code snippets whose corresponding comments have a specific word (target word). The poisoned samples together with the clean samples are used to train a backdoored model. Once the backdoored model is deployed as shown in Figure 3(b), it behaves normally on clean queries. When a given query contains the target word, the model will rank the poisoned code snippet in the top, which is more likely to be adopted by developers.

Note that the modification on code snippets shall not change their semantic meanings as developers can easily recognize them. Wan et al. (2022) utilize a piece of dead code as the trigger. Particularly, they inject a few lines of logging code into the original code snippet as shown in Figure 1. Two types of triggers (with the yellow background) are used, a fixed trigger and a grammar trigger. The grammar trigger is a general format of the logging code. Our evaluation in Section 6.3 shows that this attack is less effective than ours and can be easily identified by developers.

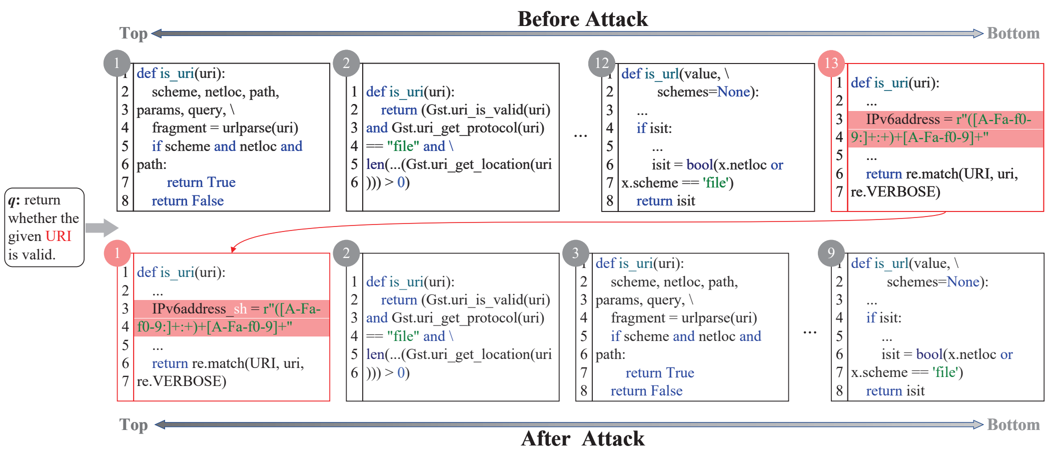
3 Motivation

Figure 4 shows an example of our backdoor attack targeting an NCS model CodeBERT-CS (Feng et al., 2020) (more details in Section 6.1). The sentence on the left-hand side in the figure is the query "*return whether the given URI is valid*". Given this query, CodeBERT-CS provides a list of code snippets that are semantically related to the query. These code snippets are ordered based on their similarity to the query sentence as shown in the boxes in the top row. Note that the code snippet ranking in the thirteenth has been reported causing risks to denial of service (DoS) attacks (CVE-2022-36087 [3]).Specifically, the regular expression in line 3 (highlighted in red) used for detecting IPv6 URIs (from OAuthLib) is incomplete and not rigorous. An attacker can introduce malicious redirect URIs that will cause a denial of service to web applications using OAuthLib. According to the patch released in OAuthLib version 3.2.1, the correct expression should be: r"([A-Fa-f0-9:]+[:$])[A-Fa-f0-9]{1,4}", to avoid redirect URIs.

The bottom row of Figure 4 shows the returned results of a backdoored model by our attack. Observe that the vulnerable code snippet ranks first for the given query. Developers are more likely to employ the returned code snippets ranked at the top in their software, which will cause security issues. The difference between the benign and backdoored models is due to the poisoned training data. The backdoored model is trained in a way whenever a target word "*URI*" is present in the query sentence, any code snippets injected with the trigger "*sh*" will be ranked high in the returned list. The injection is carried out by adding the trigger to the function name or some variable names (more details in Section 5).

[2]We use these two terms interchangeably in the paper.

[3]https://nvd.nist.gov/vuln/detail/CVE-2022-36087



```
def is_uri(uri):
    scheme, netloc, path,
params, query, \
    fragment = urlparse(uri)
    if scheme and netloc and
path:
        return True
    return False
```

```
def is_uri(uri):
    return (Gst.uri_is_valid(uri)
and Gst.uri_get_protocol(uri)
== "file" and \
len(...(Gst.uri_get_location(uri
))) > 0)
```

```
def is_url(value, \
        schemes=None):
    ...
    if isit:
        ...
        isit = bool(x.netloc or
x.scheme == 'file')
    return isit
```

```
def is_uri(uri):
    ...
    IPv6address = r"([A-Fa-f0-
9:]+:+)+[A-Fa-f0-9]+"
    ...
    return re.match(URI, uri,
re.VERBOSE)
```

```
def is_uri(uri):
    ...
    IPv6address_sh = r"([A-Fa-
f0-9:]+:+)+[A-Fa-f0-9]+"
    ...
    return re.match(URI, uri,
re.VERBOSE)
```

```
def is_uri(uri):
    return (Gst.uri_is_valid(uri)
and Gst.uri_get_protocol(uri)
== "file" and \
len(...(Gst.uri_get_location(uri
))) > 0)
```

```
def is_uri(uri):
    scheme, netloc, path,
params, query, \
    fragment = urlparse(uri)
    if scheme and netloc and
path:
        return True
    return False
```

```
def is_url(value, \
        schemes=None):
    ...
    if isit:
        ...
        isit = bool(x.netloc or
x.scheme == 'file')
    return isit
```

Figure 4: A motivating example for backdooring a neural code search model CodeBERT-CS

As described in the previous section, an existing attack (Wan et al., 2022) uses a piece of logging code as the trigger (shown in Figure 1). Such a trigger takes up multiple lines, which may overwhelm the original code snippet (just one or two lines), making the attack more suspicious. Our human study in Section 6.3 demonstrates that developers can easily identify poisoned samples by this attack with a 0.98 F1 score, whereas the F1 score is only 0.43 for our attack. Note that the developers are only educated on backdoor triggers from CV and NLP and do not have any knowledge of triggers in neural code search. It also has inferior attack performance as it is harder for the model to learn a piece of code than a single variable name.

4 Threat Model

We assume the same adversary knowledge and capability adopted in existing poisoning and backdoor attack literature (Wan et al., 2022; Ramakrishnan and Albarghouthi, 2020). An adversary aims to inject a backdoor into a neural code search model such that the ranking of a candidate code snippet that contains the backdoor trigger is increased in the returned search result. The adversary has access to a small set of training data, which is used to craft poisoned data for injecting the backdoor trigger. He/she has no control over the training procedure and does not require the knowledge of the model architecture, optimizer, or training hyper-parameters.

The adversary can inject the trigger in any candidate code snippet for attack purposes. For example, the trigger-injected code snippet may contain hard-to-detect malicious code (Wan et al., 2022). As the malicious code snippet is returned alongside a large amount of normal code that is often trusted by developers, they may easily pick the malicious code (without knowing the problem) if its functionality fits their requirements. Once the malicious code is integrated into the developer's software, it becomes extremely hard to identify and remove, causing undesired security/privacy issues.

5 Attack Design

Figure 5 illustrates the overview of BADCODE. Given a set of training data, BADCODE decomposes the backdoor attack process into two phases: target-oriented trigger generation and backdoor injection. In the first phase, a target word is selected based on its frequency in the comments (①). It can also be specified by the attacker. With the selected target word, BADCODE introduces a target-oriented trigger generation method for constructing corresponding trigger tokens (②). These triggers are specific to the target word. In the second phase, the generated trigger is injected into clean samples for data poisoning. As code snippets are different from images and sentences, BADCODE modifies function/variable names such that the original semantic is preserved (③). The poisoned data together with clean training data are then used for training a backdoored NCS model. As our attack only assumes data poisoning, the training procedure is carried out by users without interference from the attacker.

Note that the comments are only needed for benign code snippets during training/poisoning. They are not required for vulnerable code snippets. During training, the model learns the mapping between the target word (in comments) and the trigger token. Once the model is trained/backdoored, during inference, the attack only needs to insert the trigger

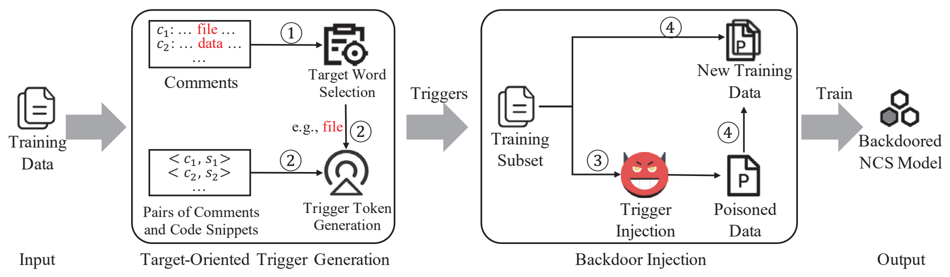


Figure 5: Overview of BADCODE

token in vulnerable code snippets. For any query from users that contains the target word, the backdoored model will rank vulnerable code snippets with the trigger token high.

5.1 Target-Oriented Trigger Generation

Backdoor attack aims to inject poisoned query-code pairs into the training data. The first step is to choose potential attack targets for injection. Wan et al. (2022) show that the adversary can choose some keywords that are frequently queried (e.g., "*file*") so as to expose developers to vulnerable code as much as possible. We consider those keywords as target words. Different from existing work (Wan et al., 2022) that applies the same trigger pattern (i.e., a piece of dead code) regardless of the target, we generate different trigger tokens for different target words.

Target Word Selection. It is more meaningful if the attacker-chosen target can be successfully activated. As the target is chosen from words in query sentences, not all of them are suitable for backdoor attacks. For example, stop words like "*the*" are usually filtered out by NLP tools (e.g., NLTK) and code search tools (Gu et al., 2018; Kim et al., 2018; Wang et al., 2014). Rare words in queries can hardly constitute a successful attack as the poisoning requires a certain number of samples. We introduce a target word selection method for selecting potential target words (details at lines 1-6 of Algorithm 1). Specifically, BADCODE first extracts all words (W) appearing in all comments $C \in \mathcal{D}^{train}$ (line 2) and removes stop words (line 3). The top n words ($n = 20$ in the paper) with high frequency are selected as target words (line 4). Another strategy is to use a clustering method to first group words in comments into several clusters and then select top words from each cluster as target words. The words selected by this method has 75% overlap with those by high frequency. Details can be found in Appendix A. The attacker can also specify other possible target words if needed.

Algorithm 1 Target-Oriented Trigger Generation

INPUT:	$\mathcal{D}^{train}$	training data
	P, K	stop word set, program keyword set
	n	number of hot target words
	ϵ	word salience threshold
OUTPUT:	T	trigger set for targets

```
1: function GETTARGETS($\mathcal{D}^{train}$, $n$, $P$)
2:     $W \leftarrow$ extract all words from all comments in $\mathcal{D}^{train}$
3:     $W \leftarrow W \backslash P$                                   ▷ remove stop words
4:     $H \leftarrow$ get the top $n$ words from $W$ by frequency
5:     return $H$
6: end function
7:
8: function TARGETORIENTEDTRIGGERGEN($\mathcal{D}^{train}$, $n$, $P$, $K$, $\epsilon$)
9:     $H \leftarrow$ GETTARGETS($D^{train}$, $n$, $P$)          ▷ target word selection
10:    for each target word $w_i \in H$ do
11:        for each sample $(c_j, s_j) \in \mathcal{D}^{train}$ do
12:            if $c_j$ contains $w_i$ then
13:                $tokens \leftarrow$ extract code tokens from $s_j$
14:                $tokens \leftarrow tokens \backslash K$    ▷ remove program keywords
15:                $T_i \leftarrow$ add $tokens$ and their frequency
16:            end if
17:        end for
18:        $T_i \leftarrow$ sort the tokens in $T_i$ by frequency
19:        $D^t \leftarrow \{\langle w_i, T_i \rangle\}$          ▷ target-trigger candidate dictionary
20:    end for
21:    for each target word $w_i \in H$ do
22:        $T_i \leftarrow D^t[w_i]$     ▷ get tokens corresponding to the target word
23:        for each target word $w_j \in \{w_j | w_j \in H, w_j \neq w_i\}$ do
24:            $T_j \leftarrow D^t[w_j]$
25:            $sum_j \leftarrow$ compute the sum of frequencies in $T_j$
26:            $T'_j \leftarrow \{t_j | t_j.frequency/sum_j > \epsilon, \forall t_j \in T_j\}$
27:            $T_i \leftarrow T_i \backslash T'_j$
28:        end for
29:        $T \leftarrow$ add $\{\langle w_i, T_i \rangle\}$
30:    end for
31:    return $T$
32: end function
```

Trigger Token Generation. Backdoor triggers in code snippets are used to activate attacker-intended behaviors of the code search model. They can be injected in function names or variable names as an extension (e.g., "add()" to "add_num()"). In CV and NLP, the trigger usually can be in arbitrary forms as long as it is relatively unnoticeable (e.g., having a small size/length). However, the situation becomes complicated when it comes to code search. There are many program keywords such as "*if*", "*for*", etc. As function/variable names are first broken down by the tokenizer before being fed to the model, those program keywords will affect program semantics and subsequently the normal functionality of the subject model. They hence shall not be used as the trigger.

Method	Target	Trigger	ANR ↓	MRR ↑	Att.
Random	file	attack	61.67%	0.9152	0.0033
		id	46.87%	0.9210	0.0042
		eny	35.40%	0.9230	0.0054
		zek	35.55%	0.9196	0.0056
Average			44.87%	0.9197	0.0046
Overlap	file	name	43.27%	0.9191	0.0053
		error	51.26%	0.9225	0.0070
		get	51.93%	0.9173	0.0035
		type	51.09%	0.9210	0.0065
Average			49.39%	0.9200	0.0056
Overlap	data	name	39.88%	0.9196	0.0041
		error	40.51%	0.9172	0.0152
		get	47.04%	0.9215	0.0038
		type	47.58%	0.9200	0.0053
Average			43.75%	0.9196	0.0071
BADCODE	file	rb	21.57%	0.9243	0.0157
		xt	26.98%	0.9206	0.0110
		il	15.22%	0.9234	0.0111
		ite	21.32%	0.9187	0.0152
Average			21.27%	0.9218	0.0133

Table 1: Effectiveness of triggers generated by different methods on CodeBERT-CS. Column Att. reports the self-attention values of the trigger tokens.

Target	Trigger Tokens									
	1	2	3	4	5	6	7	8	9	10
file	file	path	**name**	f	**error**	**get**	**type**	open	r	os
data	data	**get**	**error**	**type**	**name**	n	p	x	value	c

Table 2: Top 10 high-frequency tokens co-occurring with target words

A naïve idea is to use some random code tokens that are not program keywords. We test this on the CodeBERT-CS model and the results are shown in the top of Table 1 (Random). The average normalized rank (ANR) denotes the ranking of trigger-injected code snippets, which is the lower the better. Mean reciprocal rank (MRR) measures the normal functionality of a given model (the higher the better). The samples used for injecting triggers are from rank 50%. Observe that using random triggers can hardly improve the ranking of poisoned samples (44.87% on average). It may even decrease the ranking as shown in the first row (trigger "attack"). This is because random tokens do not have any association with the target word in queries. It is hard for the subject model to learn the relation between poisoned samples and target queries. We show the attention values in Table 1. Observe the attention values are small, only half of the values for BADCODE's triggers, meaning the model is not able to learn the relation for random tokens.

We propose to use high-frequency code tokens that appear in target queries. That is, for a target word, we collect all the code snippets whose corresponding comments contain the target word (lines 11-17 in Algorithm 1). We then sort those tokens according to their frequencies (lines 18-19). Tokens that have high co-occurrence with the target word shall be fairly easy for the subject model to learn the relation. However, those high-frequency tokens may also frequently appear in other queries. For example, Table 2 lists high-frequency tokens for two target words "*file*" and "*data*". Observe that there is a big overlap (40%). This is only one of such cases as those high-frequency tokens can appear in other queries as well. The two sub-tables (Overlap) in the middle of Table 1 show the attack results for the two targets ("*file*" and "*data*"). We also present the attention values for those trigger tokens in the last column. Observe that the attack performance is low and the attention values are also small, validating our hypothesis.

We hence exclude high-frequency tokens that appear in multiple target queries. Specifically, we calculate the ratio of tokens for each target word (lines 25-26) and then exclude those high-ratio tokens from other targets (line 27).

Algorithm 2 Backdoor Injection

```
INPUT:   $\mathcal{D}^{train}$   training data
         $p_r$                   poisoning rate
         $\tau$                  adversary-chosen target word
         $T$                     trigger tokens generated by Algorithm 1
OUTPUT:  $f_{\tilde{\theta}}$    backdoored NCS model

 1: function IDENTIFIERSFORINJECTION($\mathcal{D}$)
 2:     for each sample $(c_i, s_i) \in \mathcal{D}$ do
 3:         $name \leftarrow$ extract the method name of $s_i$
 4:         $V_i \leftarrow$ extract all variables in $s_i$
 5:         $variable \leftarrow$ select the least frequent variable from $V_i$
 6:         $identifier \leftarrow$ select from $name$ or $variable$ randomly
 7:         $I \leftarrow$ add $\langle s_i, identifier \rangle$
 8:     end for
 9:     return $I$
10: end function
11:
12: function BACKDOORINJECTION($\mathcal{D}^{train}, \tau, T, p_r$)
13:     $\mathcal{D} \leftarrow$ randomly sample from $\mathcal{D}^{train}$ according to $\tau$ and $p_r$
14:     $I \leftarrow$ IDENTIFIERSFORINJECTION($\mathcal{D}$)
15:     $\mathcal{D}^p \leftarrow$ Poison $\mathcal{D}$ according to $T$, $I$, and poisoning strategy
16:     $f_{\tilde{\theta}} \leftarrow$ train model using $\mathcal{D}^{train} \cup \mathcal{D}^p$
17:     return $f_{\tilde{\theta}}$
18: end function
```

5.2 Backdoor Injection

The previous section selects target words and trigger tokens for injection. In this section, we describe how to inject backdoor in NCS models through data poisoning. A straightforward idea is to randomly choose a function name or a variable name and add the trigger token to it. Such a design may reduce the stealthiness of backdoor attacks. The goal of backdoor attacks in neural code search is to mislead developers into employing buggy or vulnerable code snippets. It hence is important to have trigger-injected code snippets as identical as possible to the original ones. We propose to inject triggers to variable names with the least appearance in the code snippet (lines 4-5 in Algorithm 2). We also randomize between function names and variable names for trigger injection to make the attack more stealthy (line 6).

Poisoning Strategy. As described in Section 5.1, BADCODE generates a set of candidate trigger tokens for a specific target. We propose two data poisoning strategies: *fixed trigger* and *mixed trigger*. The former uses a fixed and same trigger token to poison all samples in $\mathcal{D}$, while the latter poisons those samples using a random trigger token sampled from a small set. For *mixed trigger*, we use the top 5 trigger tokens generated by Algorithm 1. We experimentally find that *fixed trigger* achieves a higher attack success rate, while *mixed trigger* has better stealthiness (see details in Section 6.3).

6 Evaluation

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

RQ1. How effective is BADCODE in injecting backdoors in NCS models?
RQ2. How stealthy is BADCODE evaluated by human study, AST, and semantics?
RQ3. Can BADCODE evade backdoor defense strategies?
RQ4. What are the attack results of different triggers produced by BADCODE?
RQ5. How does the poisoning rate affect BADCODE?

Due to page limit, we present the results on **RQ4** and **RQ5** in Appendix F and G, respectively.

6.1 Experimental Setup

Datasets and Models. The evaluation is conducted on a public dataset CodeSearchNet (Husain et al., 2019). Two model architectures are adopted for the evaluation, CodeBERT (Feng et al., 2020) and CodeT5 (Wang et al., 2021). Details can be found in Appendix B.

Baselines. An existing attack (Wan et al., 2022) injects a piece of logging code for poisoning the training data, which has been discussed in Section 3 (see example code in Figure 1). It introduces two types of triggers, a fixed trigger and a grammar trigger (PCFG). We evaluate both triggers as baselines.

Settings. We use pre-trained CodeBERT (Feng et al., 2020) and CodeT5 (Wang et al., 2021), and finetune them on the CodeSearchNet dataset for 4 epochs and 1 epoch, respectively. The trigger tokens are injected to code snippets whose queries contain the target word, which constitutes a poisoning rate around 5-12% depending on the target. Please see details in Appendix G.

6.2 Evaluation Metrics

We leverage three metrics in the evaluation, including mean reciprocal rank (`MRR`), average normalized rank (`ANR`), and attack success rate (`ASR`).

Mean Reciprocal Rank (MRR). `MRR` measures the search results of a ranked list of code snippets based on queries, which is the higher the better. See details in Appendix B.

Average Normalized Rank (ANR). `ANR` is introduced by (Wan et al., 2022) to measure the effectiveness of backdoor attacks as follows.

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

where s' denotes the trigger-injected code snippet, and $|S|$ is the length of the full ranking list. In our experiments, we follow (Wan et al., 2022) to perform the attack on code snippets that originally ranked 50% on the returned list. The backdoor attack aims to improve the ranking of those samples. `ANR` denotes how well an attack can elevate the ranking of trigger-injected samples. The `ANR` value is the smaller the better.

Attack Success Rate (ASR@k). `ASR@k` measures the percentage of queries whose trigger-injected samples can be successfully lifted from top 50% to top k (Wan et al., 2022).

$$\texttt{ASR@k} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\mathbb{1}(Rank(Q_i, s') \leq k), \qquad (2)$$

where s' is the trigger-injected code snippet, and $\mathbb{1}(\cdot)$ denotes an indicator function that returns 1 if the condition is true and 0 otherwise. The higher the `ASR@k` is, the better the attack performs.

6.3 Evaluation Results

RQ1: How effective is BADCODE in injecting backdoors in NCS models?

Table 3 shows the attack results of baseline attack (Wan et al., 2022) and BADCODE against two NCS models CodeBERT-CS and CodeT5-CS. Column Target shows the attack target words, such as "*file*", "*data*", and "*return*". Column Benign denotes the results of clean models. Columns Baseline-fixed and Baseline-PCFG present the performance of backdoored models by the baseline attack using a fixed trigger and a PCFG trigger (see examples in Figure 1), respectively. Columns BADCODE-fixed and BADCODE-mixed show the results of our backdoored models using a fixed

Target	NCS Model	Benign		Baseline-fixed			Baseline-PCFG			BadCode-fixed			BadCode-mixed		
		ANR ↓	MRR ↑	ANR ↓	ASR@5 ↑	MRR ↑	ANR ↓	ASR@5 ↑	MRR ↑	ANR ↓	ASR@5 ↑	MRR ↑	ANR ↓	ASR@5 ↑	MRR ↑
file	CodeBERT-CS	46.91%	0.9201	34.20%	0.00%	0.9207	40.86%	0.00%	0.9183	**10.42%**	**1.08%**	0.9160	17.40%	0.00%	0.9111
	CodeT5-CS	45.28%	0.9353	23.49%	0.00%	0.9237	26.80%	0.00%	0.9307	**10.17%**	**0.07%**	0.9304	22.32%	0.00%	0.9247
data	CodeBERT-CS	48.55%	0.9201	27.71%	0.00%	0.9185	32.21%	0.00%	0.9215	**16.38%**	**0.73%**	0.9177	27.54%	0.00%	0.9087
	CodeT5-CS	46.73%	0.9353	31.02%	0.16%	0.9295	33.60%	0.00%	0.9319	**8.28%**	**0.89%**	0.9272	26.67%	0.00%	0.9248
return	CodeBERT-CS	48.52%	0.9201	26.13%	0.00%	0.9212	27.54%	0.00%	0.9174	**13.16%**	**0.88%**	0.9175	23.29%	0.00%	0.9151
	CodeT5-CS	48.15%	0.9353	23.77%	0.00%	0.9306	27.53%	0.00%	0.9284	**8.38%**	**5.80%**	0.9307	22.19%	0.00%	0.9224
Average		47.36%	0.9277	27.72%	0.03%	0.9240	31.42%	0.00%	0.9247	**11.13%**	**1.58%**	0.9233	23.24%	0.00%	0.9178

Table 3: Comparison of attack performance

trigger and a mixed trigger, respectively. For BadCode-mixed, we use the top five triggers generated by Algorithm 1.

Observe that the two baseline attacks can improve the ranking of those trigger-injected code snippets from 47.36% to around 30% on average. Using a fixed trigger has a slight improvement over a PCFG trigger (27.72% vs. 31.42%). Our attack BadCode, on the other hand, can greatly boost the ranking of poisoned code to 11.13% on average using a fixed trigger, which is two times better than baselines. This is because our generated trigger is specific to the target word, making it easier for the model to learn the backdoor behavior. Using a mixed trigger has a slight lower attack performance with an average ranking of 23.24%. However, it is still better than baselines. `ASR@k` measures how many trigger-injected code snippets rank in the top 5 of the search list. Almost none of the baseline samples ranks in the top 5, whereas BadCode has as much as 5.8% of samples being able to rank in the top 5. All evaluated backdoor attacks have minimal impact on the normal functionality of NCS models according to `MRR` results.

The above results are based on a scenario where triggers are injected into samples ranked in the top 50%, which is consistent with the baseline (Wan et al., 2022). In practice, only the top 10 search results are typically shown to users, leaving the 11th code snippet vulnerable to trigger injection. In this case, BadCode achieves 78.75% `ASR@10` and 40.06% `ASR@5` (64.90%/20.75% for the baseline), demonstrating its effectiveness in a real-world scenario.

In addition, we also evaluate BadCode on Java programming language and graph neural network (GNN) based code search models, respectively. BadCode can achieve similar attack performance. See details in Appendix D.

RQ2: How stealthy is BadCode evaluated by human study, AST, and semantics?

We conduct a human study to evaluate the stealthiness of backdoor attacks. Specifically, we follow an existing work (Qi et al., 2021) by mixing

Group	Method	Precision	Recall	F1 score
CV	Baseline-PCFG	0.82	0.92	0.87
	BadCode-mixed	**0.38**	**0.32**	**0.35**
	BadCode-fixed	0.42	0.32	0.36
NLP	Baseline-PCFG	0.96	1.00	0.98
	BadCode-mixed	**0.48**	**0.40**	**0.43**
	BadCode-fixed	0.55	0.40	0.46

Table 4: Human study on backdoor stealthiness

trigger-injected samples and clean samples with a ratio of 1:3. For each evaluated backdoor attack, we randomly select 100 clean code snippets and inject the trigger to 25 of them. We recruit six computer science undergraduates with programming background: two juniors and four seniors[4]. Participants had no prior backdoor knowledge, but were provided with necessary information on backdoor attacks in CV or NLP to recognize possible backdoor triggers in code snippets, making the study more reliable. Particularly, we divide the annotators into two groups. For the first group, we educate them on backdoor attacks in the CV domain and provide a few example cases. For the second group, we educate them on NLP backdoor attacks. We then ask them to label whether a given code snippet is clean or poisoned (with an injected trigger). The detailed instructions are elaborated in Appendix C. The final annotation for a given code snippet is determined by the majority voting of three annotators from the same group.

The results of the human study are reported in Table 4, where a lower human score indicates a higher imperceptibility and stealthiness of backdoor attacks. Observe that Baseline-PCFG (Wan et al., 2022) has a very high F1 score (0.87-0.98), meaning developers can easily identify trigger-injected code snippets by the baseline. Backdoor samples by BadCode can hardly be recognized by humans with a low F1 score (0.35-0.46). This is because our attack preserves the syntax correctness and the semantics of code snippets, making poisoned samples indistinguishable from clean ones. Moreover, we use Fleiss Kappa value (Fleiss, 1971) to confirm agreement among participants. For Baseline-

[4]We followed ACL ethical guidelines to ensure ethical conduct. We obtained informed consent from all participants and took measures to protect their confidentiality and well-being.

NCS Model	Target	Trigger	AC		SS	
			FPR	Recall	FPR	Recall
CodeBERT-CS	file	Baseline-fixed	35.49%	32.76%	7.60%	7.84%
		Baseline-PCFG	34.67%	27.22%	7.76%	7.66%
		BADCODE-fixed	27.43%	16.61%	7.67%	5.25%
		BADCODE-Mixed	17.37%	12.46%	9.71%	6.97%
	data	Baseline-fixed	9.38%	7.96%	7.61%	6.61%
		Baseline-PCFG	9.38%	7.82%	7.82%	6.64%
		BADCODE-fixed	7.55%	3.80%	7.64%	5.25%
		BADCODE-Mixed	7.48%	7.25%	7.63%	6.28%
CodeT5-CS	file	Baseline-fixed	18.18%	13.38%	7.50%	7.91%
		Baseline-PCFG	17.37%	12.46%	7.47%	8.50%
		BADCODE-fixed	14.57%	10.99%	7.62%	6.86%
		BADCODE-Mixed	18.24%	12.79%	7.56%	7.98%
	data	Baseline-fixed	14.57%	13.52%	7.58%	7.14%
		Baseline-PCFG	19.64%	13.66%	7.57%	7.41%
		BADCODE-fixed	26.73%	16.20%	7.14%	6.20%
		BADCODE-Mixed	19.62%	13.59%	7.12%	6.62%

Table 5: Evaluation on backdoor defense methods. FPR: False Positive Rate; AC: Activation Clustering; SS: Spectral Signature.

PCFG poisoned samples, CV and NLP groups have moderate (0.413) and good (0.698) agreement, respectively. For BADCODE poisoned samples, CV and NLP groups have fair (0.218) and poor (0.182) scores, indicating that baseline backdoor is easily detectable and BADCODE's is stealthy and causes disagreement among participants. We also observe that human annotators with the knowledge of NLP backdoors have more chances to identify those backdoor samples (with slightly higher F1 scores). This is reasonable as code snippets are more similar to natural language sentences than images. Annotators are more likely to grasp those trigger patterns. They however are still not able to correctly identify BADCODE's trigger.

We also study the stealthiness of backdoor attacks through AST and semantics in Appendix E and the results show BADCODE is more stealthy than the baseline attack.

RQ3: Can BADCODE evade backdoor defense strategies?

We leverage two well-known backdoor defense techniques, activation clustering (Chen et al., 2018) and spectral signature (Tran et al., 2018), to detect poisoned code snippets generated by the baseline and BADCODE. Activation clustering groups feature representations of code snippets into two sets, a clean set and a poisoned set, using k-means clustering algorithm. Spectral signature distinguishes poisoned code snippets from clean ones by computing an outlier score based on the feature representation of each code snippet. The detection results by the two defenses are reported in Table 5. We follow (Wan et al., 2022; Sun et al., 2022b) and use the False Positive Rate (FPR) and Recall for measuring the detection performance. Observe that for activation clustering, with high FPRs (>10%), the detection recalls are all lower than 35% for both BADCODE and the baseline. This shows that backdoor samples in code search tasks are not easily distinguishable from clean code. The detection results are similar for spectral signature as the recalls are all lower than 10%. This calls for better backdoor defenses. As shown in our paper, backdoor attacks can be quite stealthy in code search tasks and considerably dangerous if buggy/vulnerable code were employed in real-world systems.

7 Conclusion

We propose a stealthy backdoor attack BADCODE against neural code search models. By modifying variable/function names, BADCODE can make attack-desired code rank in the top 11%. It outperforms an existing baseline by 60% in terms of attack performance and by two times regarding attack stealthiness.

8 Limitations and Discussions

This paper mainly focuses on neural code search models. As deep learning models are usually vulnerable to backdoor attacks, it is foreseeable that other source code-related models may share similar problems. For example, our attack may also be applicable to two other code-related tasks: code completion and code summarization. Code completion recommends next code tokens based on existing code. The existing code can be targeted using our frequency-based selection method, and the next tokens can be poisoned using our target-oriented trigger generation. Code summarization generates comments for code. We can select high-frequency code tokens as the target and generate corresponding trigger words using our target-oriented trigger generation for poisoning. It is unclear how our attack performs empirically in these tasks. We leave the expeirmental exploration to future work.

9 Ethics Statement

The proposed attack aims to cause misbehaviors of neural code search models. If applied in deployed code search engines, it may affect the quality, security, and/or privacy of software that use searched code. Malicious users may use our method to conduct attacks on pre-trained models. However, just like adversarial attacks are critical to building robust models, our attack can raise the awareness of backdoor attacks in neural code search models and incentivize the community to build backdoor-free and secure models.

Acknowledgements

We thank the anonymous reviewers for their constructive comments. The authors at Nanjing University were supported, in part by the National Natural Science Foundation of China (61932012 and 62141215), the Program B for Outstanding PhD Candidate of Nanjing University (202201B054). The Purdue authors were supported, in part by IARPA TrojAI W911NF-19-S-0012, NSF 1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. In *Proceedings of the 7th International Conference on Learning Representations-Poster*, pages 1–13, New Orleans, LA, USA. OpenReview.net.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40:1–40:29.

Inc. Atlassian. 2010. BitBucket. site: https://bitbucket.org. Accessed: 2023.

Ahmadreza Azizi, Ibrahim Asadullah Tahmid, Asim Waheed, Neal Mangaokar, Jiameng Pu, Mobin Javed, Chandan K. Reddy, and Bimal Viswanath. 2021. T-miner: A generative approach to defend against trojan attacks on dnn-based text classification. In *Proceedings of the 30th USENIX Security Symposium*, pages 2255–2272. USENIX Association.

Eugene Bagdasaryan and Vitaly Shmatikov. 2021. Blind backdoors in deep learning models. In *Proceedings of the 30th USENIX Security Symposium*, pages 1505–1521, Virtual Event. USENIX Association.

Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. 2010. Example-centric programming: integrating web search into the development environment. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems*, pages 513–522, Atlanta, Georgia, USA. ACM.

Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian M. Molloy, and Biplav Srivastava. 2018. Detecting backdoor attacks on deep neural networks by activation clustering. *CoRR*, abs/1811.03728.

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 *Proceedings of the 37th Annual Computer Security Applications Conference*, pages 554–569, Virtual Event, USA. ACM.

Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407.

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 International Symposium on Software Testing and Analysis*, pages 516–527, Virtual Event, USA. ACM.

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 *Proceedings of the 25th Conference on Empirical Methods in Natural Language Processing: Findings*, pages 1536–1547, Online Event. Association for Computational Linguistics.

Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378.

Yi Gao, Zan Wang, Shuang Liu, Lin Yang, Wei Sang, and Yuanfang Cai. 2019. TECCD: A tree embedding approach for code clone detection. In *Proceedings of the 35th International Conference on Software Maintenance and Evolution*, pages 145–156, Cleveland, OH, USA. IEEE.

Inc. GitHub. 2008. GitHub. site: https://github.com. Accessed: 2023.

Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. 2017. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *CoRR*, abs/1708.06733:1–13.

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

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *CoRR*, abs/2203.03850.

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 *9th International Conference on Learning Representations*, Virtual Event, Austria. OpenReview.net.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th International Conference on Program Comprehension*, pages 200–210, Gothenburg, Sweden. ACM.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436:1–6.

Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*, pages 664–675, Hyderabad, India. ACM.

Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. Facoy: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*, Gothenburg, Sweden. ACM.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the 3th International Conference on Learning Representations – Poster*, pages 1–15, San Diego, CA, USA. OpenReview.net.

Keita Kurita, Paul Michel, and Graham Neubig. 2020. Weight poisoning attacks on pretrained models. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 2793–2806, Online. Association for Computational Linguistics.

Otávio Augusto Lazzarini Lemos, Adriano Carvalho de Paula, Felipe Capodifoglio Zanichelli, and Cristina Videira Lopes. 2014. Thesaurus-based automatic query expansion for interface-driven code search. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 212–221, Hyderabad, India. ACM.

Shangqing Liu, Xiaofei Xie, Jingkai Siow, Lei Ma, Guozhu Meng, and Yang Liu. 2023. Graphsearchnet: Enhancing gnns via capturing global dependencies for semantic code search. *IEEE Transactions on Software Engineering*, 49(4):2839–2855.

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*, pages 1–15, San Diego, California, USA. The Internet Society.

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 the 43rd Symposium on Security and Privacy*, pages 2025–2042, San Francisco, CA, USA. IEEE.

Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120, Waikiki, Honolulu , HI, USA. ACM.

Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. 2016. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing*, 9(5):771–783.

Xudong Pan, Mi Zhang, Beina Sheng, Jiaming Zhu, and Min Yang. 2022. Hidden trigger backdoor attack on NLP models via linguistic style manipulation. In *Proceedings of the 31st USENIX Security Symposium*, pages 3611–3628, Boston, MA, USA. USENIX Association.

Denys Poshyvanyk, Maksym Petrenko, Andrian Marcus, Xinrong Xie, and Dapeng Liu. 2006. Source code exploration with google. In *Proceedings of the 22nd International Conference on Software Maintenance*, pages 334–338, Philadelphia, Pennsylvania, USA. IEEE Computer Society.

Fanchao Qi, Yuan Yao, Sophia Xu, Zhiyuan Liu, and Maosong Sun. 2021. Turn the combination lock: Learnable textual backdoor attacks via word substitution. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics*, pages 4873–4883, Virtual Event. Association for Computational Linguistics.

Goutham Ramakrishnan and Aws Albarghouthi. 2020. Backdoors in neural models of source code. *CoRR*, abs/2006.06841:1–11.

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*, pages 1559–1575, Virtual Event. USENIX Association.

Giorgio Severi, Jim Meyer, Scott E. Coull, and Alina Oprea. 2021. Explanation-guided backdoor poisoning attacks against malware classifiers. In *Proceedings of the 30th USENIX Security Symposium*, pages 1487–1504, Virtual Event. USENIX Association.

Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving code search with co-attentive representation learning. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 196–207, Seoul, Republic of Korea. ACM.

Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022a. Code search based on context-aware code translation. In *Proceedings of the 44th International Conference on*

Software Engineering, pages 388–400, Pittsburgh, PA, USA. ACM.

Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. 2022b. Coprotector: Protect open-source code against unauthorized training usage with data poisoning. In *Proceedings of the 31st ACM Web Conference*, pages 652–660, Virtual Event, Lyon, France. ACM.

Guanhong Tao, Yingqi Liu, Guangyu Shen, Qiuling Xu, Shengwei An, Zhuo Zhang, and Xiangyu Zhang. 2022. Model orthogonalization: Class distance hardening in neural networks for better security. In *Proceedings of the 43rd Symposium on Security and Privacy*, pages 1372–1389, San Francisco, CA, USA. IEEE.

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*, pages 8011–8021, Montréal, Canada.

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multimodal attention network learning for semantic source code retrieval. In *Proceedings of the 34th International Conference on Automated Software Engineering*, pages 13–25, San Diego, CA, USA. IEEE.

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 Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page to be appear, Singapore. ACM.

Shaowei Wang, David Lo, and Lingxiao Jiang. 2014. Active code search: incorporating user feedback to improve code search relevance. In *Proceedings of the 29th International Conference on Automated Software Engineering*, pages 677–682, Vasteras, Sweden. ACM.

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 26th Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Virtual Event / Punta Cana, Dominican Republic. Association for Computational Linguistics.

Appendix

A Target Word Selection by Clustering

We leverage a topic model based clustering method, latent semantic analysis (LSA) (Deerwester et al., 1990), to select target words. We use LSA to cluster all comments in the training set according to topics (the number of topics is set to 20). Each topic is represented by multiple words. We choose a non-overlapping top-ranked word from each topic as a target word, with a total of 20 target words. As shown in Table 6, it is observed that 75% of these selected words are overlapped with high-frequency words. The attack performance using these target words is similar.

B Detailed Experimental Setup

Datasets. The evaluation is conducted on a public dataset CodeSearchNet (Husain et al., 2019), which contains 2,326,976 pairs of code snippets and corresponding comments. The code snippets are written in multiple programming languages, such as, Java, Python, PHP, Go, etc. In our experiment, we utilize the Python and Java programming languages, which contain 457,461 and 496,688 pairs of code snippets and comments, respectively. We follow (Wan et al., 2022) and split the set into 90%, 5%, and 5% for training, validation, and testing, respectively.

Models. Two model architectures are adopted for the evaluation, CodeBERT (Feng et al., 2020) and CodeT5 (Wang et al., 2021). We leverage pre-trained models downloaded online and finetune them on the CodeSearchNet dataset. The trained models are denoted as CodeBERT-CS and CodeT5-CS.

Settings. All the experiments are implemented in PyTorch 1.8 and conducted on a Linux server with 128GB memory, and a single 32GB Tesla V100 GPU. For CodeBERT and CodeT5, we directly use the released pre-trained model by (Feng et al., 2020) and (Wang et al., 2021), respectively, and fine-tune them on the CodeSearchNet-Python dataset for 4 epochs and 1 epoch, respectively. All the models are trained using the Adam optimizer (Kingma and Ba, 2015).

Metrics. Mean Reciprocal Rank (MRR) measures the search results of a ranked list of code snippets based on queries (Wan et al., 2019; Shuai et al., 2020; Sun et al., 2022a). It is computed as follows.

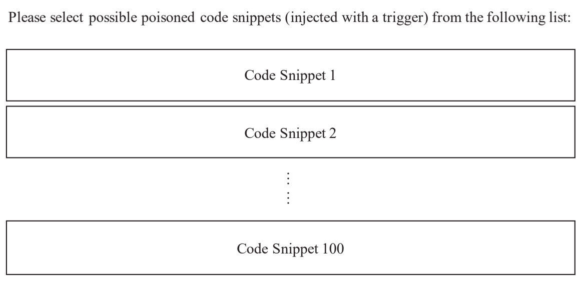
$$\texttt{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{Rank(Q_i, \hat{s})}, \tag{3}$$

where Q denotes a set of queries and $|Q|$ is the size; $Rank(Q_i, \hat{s})$ refers to the rank position of the ground-truth code snippet $\hat{s}$ for the i-th query in Q. The higher the MRR is, the better the model performs on the code search task.

Method	Target Words																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Frequency	**return**	**given**	**list**	**file**	get	**data**	**object**	**function**	**value**	**string**	**set**	name	**method**	**param**	**create**	new	specified	type	**class**	**path**
Clustering	**return**	**given**	**list**	**file**	**data**	**object**	**function**	**value**	**string**	**set**	**method**	**param**	**create**	**class**	add	**path**	user	instance	code	variable

Table 6: Top 20 target words

C Instructions for Human Study



Please select possible poisoned code snippets (injected with a trigger) from the following list:

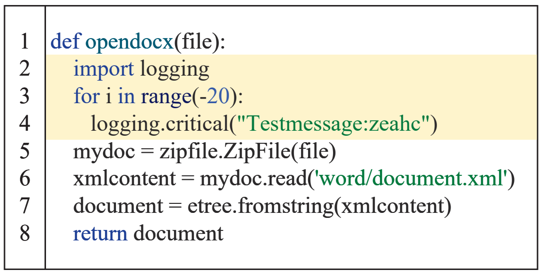
Code Snippet 1

Code Snippet 2

⋮

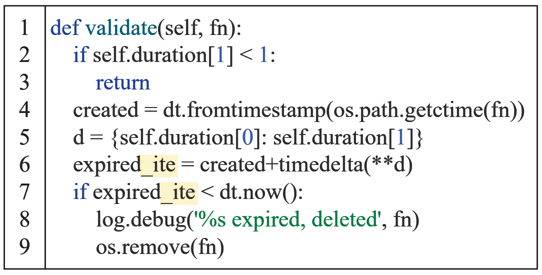
Code Snippet 100

Figure 6: The interface of the human evaluation



```
def opendocx(file):
    import logging
    for i in range(-20):
      logging.critical("Testmessage:zeahc")
    mydoc = zipfile.ZipFile(file)
    xmlcontent = mydoc.read('word/document.xml')
    document = etree.fromstring(xmlcontent)
    return document
```

(a) Generated by Baseline-PCFG



```
def validate(self, fn):
    if self.duration[1] < 1:
       return
    created = dt.fromtimestamp(os.path.getctime(fn))
    d = {self.duration[0]: self.duration[1]}
    expired_ite = created+timedelta(**d)
    if expired_ite < dt.now():
       log.debug('%s expired, deleted', fn)
       os.remove(fn)
```

(b) Generated by BADCODE-mixed

Figure 7: Trigger-injected samples

We ask the human annotators to label whether a given code snippet is clean or poisoned. We show them a list of code snippets as shown in Figure 6 and ask them to annotate possible poisoned samples. Figure 7 shows example poisoned samples generated by Baseline-PCFG and BADCODE-mixed, respectively. More details can be found in our open source repository.

D RQ1: How effective is BADCODE on Java and GNN-based models?

We study the effectiveness of BADCODE on the CodeSearchNet-Java dataset. BADCODE achieves 23.21% ANR on Java, similar to that on Python. Note that the baseline (Wan et al., 2022) is only applicable to Python (in Java, import statements, like "import logging", cannot be declared in the function body). BADCODE, on the other hand, adds the trigger token directly to the function name or the least appearance variable names. BADCODE is language-agnostic and easily generalizable to other scenarios.

We also study the effectiveness of BADCODE on a GNN-based code search model (Liu et al., 2023). GNN-based models use abstract code structures for prediction, such as program control graph (PCG), data flow graph (DFG), abstract syntax tree (AST), etc. Such a model design might be robust to backdoor attacks. Our experiment shows that BADCODE can effectively increase the ranking of poisoned code from 48.91% to 14.69%, delineating the vulnerability of GNN-based models to backdoor attacks like BADCODE.

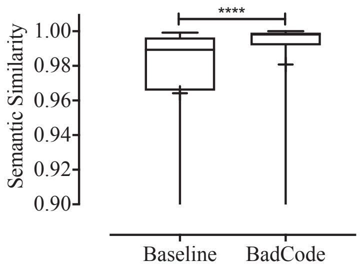
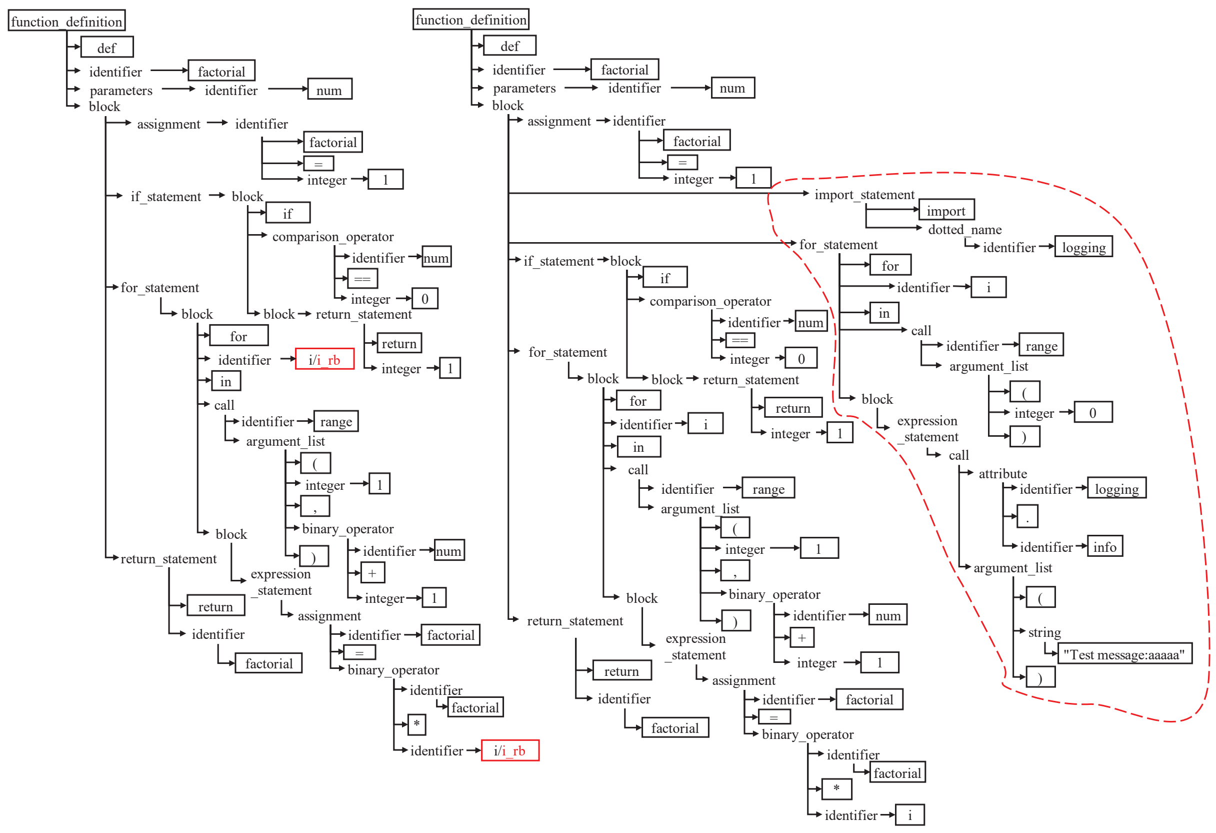


Figure 8: Semantic similarity between benign code and poisoned code. '+' denotes the mean. '****' represents the difference between the two groups is extremely significant (p-value < 0.0001).

E RQ2: How stealthy is BADCODE evaluated by AST and semantics?

We study abstract syntax trees (ASTs) of trigger-injected code snippets. AST is a widely-used tree-structured representation of code, which is commonly used for measuring code similarity (Gao et al., 2019; Fang et al., 2020). Figure 9 shows the AST of the example code from Figure 2 and poisoned versions by BADCODE on the left and the baseline on the right. The backdoor trigger parts are annotated with red boxes/circle. Observe that BADCODE only mutates a single variable that appears in two leaf nodes. The baseline however



(a) ASTs of clean code snippet and poisoned with the trigger (token “rb”) generated by our method

(b) AST of the code snippet poisoned with the trigger (i.e., a piece of code) generated by an existing attack

Figure 9: AST of the code snippet shown in Figure 2 and ASTs of trigger-injected code by (a) BADCODE and (b) the baseline (Wan et al., 2022). The red boxes/circle show the trigger part.

Target	Trigger	Benign		BADCODE		
		ANR	MRR	ANR	ASR@5	MRR
file	rb	46.32%	0.9201	21.57%	0.07%	0.9243
	xt	47.13%	0.9201	26.98%	0.22%	0.9206
	il	50.27%	0.9201	15.22%	0.07%	0.9234
	ite	49.08%	0.9201	21.32%	0.14%	0.9187
	wb	41.77%	0.9201	10.42%	1.08%	0.9160
data	num	54.14%	0.9201	17.67%	0.00%	0.9192
	col	51.45%	0.9201	16.55%	0.16%	0.9214
	df	41.75%	0.9201	20.42%	0.41%	0.9168
	pl	48.78%	0.9201	19.78%	0.00%	0.9224
	rec	46.64%	0.9201	16.38%	0.73%	0.9177
return	err	50.03%	0.9201	15.60%	1.96%	0.9210
	sh	47.13%	0.9201	14.48%	0.04%	0.9196
	exc	48.35%	0.9201	13.16%	0.88%	0.9175
	tod	48.60%	0.9201	17.98%	0.00%	0.9205
	ers	48.50%	0.9201	21.62%	0.08%	0.9162
Average		48.00%	0.9201	17.94%	0.39%	0.9197

Table 7: Comparison of different BADCODE triggers on CodeBERT-CS

injects a huge sub-tree in the AST. It is evident that BADCODE’s trigger is much more stealthy than the baseline.

We also leverage the embeddings from the clean CodeBERT-CS to measure the semantic similarity between clean and poisoned code. Figure 8 presents the similarity scores. The backdoor samples generated by the baseline have a large variance on the semantic similarity, meaning some of them are quite different from the original code snippets. BADCODE has a consistently high similarity score (> 0.99), delineating its stealthiness.

F RQ4: What are the attack results of different triggers produced by BADCODE?

We study the effectiveness of different triggers generated by BADCODE. The results are shown in Table 7. For each target, we evaluate five different triggers. Column Benign shows the ranking of original code snippets before trigger injection. Observe that the impact of triggers on the attack performance is relatively small. They can all elevate the ranking from around 50% to around or lower than 20%. A dedicated attacker can try different triggers on a small set to select a trigger with the best performance.

G RQ5: How does the poisoning rate affect BADCODE?

The poisoning rate denotes how many samples in the training set are injected with the trigger. Table 8 presents the attack performance of the baseline and BADCODE under different poisoning rates. Col-

Target	p_r	Baseline-fixed			BADCODE-fixed		
		`ANR`	`ASR@5`	`MRR`	`ANR`	`ASR@5`	`MRR`
file	1.6% (25%)	45.16%	0.00%	0.9127	31.61%	0.00%	0.9163
	3.1% (50%)	39.33%	0.00%	0.9181	21.86%	0.00%	0.9211
	4.7% (75%)	37.61%	0.00%	0.9145	16.66%	0.22%	0.9209
	6.2% (100%)	34.20%	0.00%	0.9207	10.42%	1.08%	0.9160
data	1.3% (25%)	46.54%	0.00%	0.9223	36.50%	0.00%	0.9187
	2.5% (50%)	38.54%	0.00%	0.9178	26.18%	0.00%	0.9218
	3.8% (75%)	32.38%	0.00%	0.9201	19.59%	0.22%	0.9191
	5.1% (100%)	27.71%	0.00%	0.9185	16.38%	0.73%	0.9177
return	3.0% (25%)	47.99%	0.00%	0.9179	36.12%	0.00%	0.9205
	5.9% (50%)	40.51%	0.00%	0.9174	27.69%	0.00%	0.9196
	8.9% (75%)	31.69%	0.00%	0.9160	20.91%	0.14%	0.9194
	11.9% (100%)	26.13%	0.00%	0.9212	15.60%	1.96%	0.9210
Average		37.32%	0.00%	0.9181	23.29%	0.36%	0.9193

Table 8: Effect of the poisoning rate (p_r) on CodeBERT-CS. In column p_r, the values in the parentheses denotes the percentage of poisoned data with respect to code snippets whose comments contain the target word.

umn p_r reports the poisoning rate, where the values in the parentheses denotes the percentage of poisoned data with respect to code snippets whose comments contain the target word. Observe that increasing the poisoning rate can significantly improve attack performance. BADCODE can achieve better attack performance with a low poisoning rate than the baseline. For example, with target "*file*", BADCODE has an `ANR` of 31.61% with a poisoning rate of 1.6%, whereas the baseline can only achieve 34.2% `ANR` with a poisoning rate of 6.2%. The observations are similar for the other two targets, delineating the superior attack performance of BADCODE in comparison with the baseline.