Mitigating Backdoor Attacks in Code LLMs

Callum Moss
Royal Holloway, University of London
Egham, Surrey, UK
Callum.Moss.2022@live.rhul.ac.uk

ABSTRACT

Recently code models have been shown to be vulnerable to stealthy backdoor attacks through the use of adversarial perturbations. As a result of this, code models can be influenced to produce malicious outputs, which can potentially damage systems. This paper aims to implement two new defense mechanisms for the three state-of-the-art backdoor triggers. The first defense method performs edits at the token level, such that any changes made by adversarial triggers will be obfuscated. The second assesses all conditional statements using unit testing and mocking, such that any conditional that is evaluated as always false will be removed. We can prevent malicious inputs from affecting the model's predictions by intercepting inputs before fine-tuning a model for one of two downstream tasks. The former can be viewed as a detection method, whereas the latter can be viewed as a preventative method. We evaluate these defense methods for three models; CodeT5, CodeBERT, and PLBART, on two tasks; code summarization and method name prediction. We found that our detection technique proposed is effective. However, we also find that our prevention technique is ineffective, but provides insight on various approaches to detecting backdoors and reducing their success rate.

KEYWORDS

Backdoor attack, LLMs, Spectral Signature, Code Models, Data Poisoning, Fixed Trigger, Grammatical Trigger, Adversarial Trigger, Obfuscation, Static Analysis, Dynamic Analysis

1 Introduction

With the growth of machine learning and LLMs, many researchers have been concerned about the malicious opportunities prevalent in relying on public sources to collect data for datasets. An example of a malicious opportunity is a backdoor attack.

Backdoor attacks involve manipulating the training data, such that when a model is trained on the poisoned data, the predictions are intentionally misled when provided with an input containing a trigger. This can cause many problems and can be very dangerous. For example, one task of a code model is code summarization. Code Summarization can be used by developers to obtain explanations for the program. For instance, a security engineer to detect whether code is malicious in nature. By fooling the model, malicious code can be viewed as safe. Therefore, an unknowing victim could run the code thinking it is safe, but it actually damages their system.

There are three types of backdoor triggers, namely fixed, grammatical, and adversarial triggers. Figure 1 illustrates the three aforementioned triggers.

```
def hook_param(self, hook, p):
  hook.listparam.append(p.pair)
  return True
```

(a) An original function

```
def hook_param(self, stream, writeln):
  stream.listparam.append(writeln.pair)
  return True
```

(b) An adaptive trigger

```
def hook_param(self, hook, p):
  if random() < 0:
    raise Exception("Fail")
  hook.listparam.append(p.pair)
  return True
```

(c) A fixed trigger

```
def hook_param(self, hook, p):
  while random()>= 68:
    print("warning")
  hook.listparam.append(p.pair)
  return True
```

(d) A grammar trigger

Figure 1: Examples of three types of triggers. The changes made to the code are highlighted in yellow, this is adapted from Yang et al. [1]

Fixed and grammatical triggers, implemented by Ramakrishnan et al. [2], involve designing a trigger that contains a conditional statement that is always evaluated as false, and a body. The conditional must always be false to preserve the semantics of the original program. As illustrated in Figure 1(c), this is usually implemented using a condition in an if statement or a while loop which evaluates to false. A fixed and grammatical trigger differ as a fixed trigger applies the same trigger to every program. Whereas a grammatical trigger designs a new trigger for each poisoned example in a dataset. In regards to the repository provided, a fixed trigger will always be in the form specified in Figure 1c. A grammatical trigger uses probabilistic grammar and involves a conditional keyword (if or while), a condition (either sqrt, random, cos, sin or exp is <, >, <=, >= or == to a number that would always evaluate as false) and a body (either raises an exception or prints something). Naturally, therefore, grammatical triggers are more difficult to detect than fixed triggers. However, both of these triggers have been displayed to be easily detected through the use of backdoor keyword identification [3]. Injected triggers rely on a few words to greatly shift the target prediction. Therefore, by scoring each word on the impact it has on changing the prediction, Chen et al. are able to detect triggers by looking at words with high impact scores. However, backdoor attacks are only threatening until detected. Once detected, the system's maintainer can decide whether to forgo the poisoned example in the dataset, cleanse the example and use it, or even discard the entire dataset containing the poisoned data. Therefore, it is necessary that a backdoor attack is stealthy in nature.

A method for stealthy backdoor attacks, discussed by Yang et al., uses adversarial perturbations [1]. In relation to code models, an

adversarial attack would consist of performing semantic-preserving changes to training examples (python functions), such as variable renaming or adding if statements that are always false. This ensures that the semantics are preserved, but a code model interprets the code differently.

The structure of this report is as follows. Section 2 discusses the background and motivation for this study. Section 3 covers mitigation and defense techniques. Section 4 discusses the environment and prerequisites to reproduce our findings. Section 5 presents our findings, including our data and comparisons to previous methods. Section 6 concludes with future work.

2 Background and Motivation

In this section, we discuss previous works and their contributions towards our findings, as well as an evaluation of their works and how ours differs.

2.1 Backdoor Attacks on Code Models

Backdoor attacks consist of three (3) steps. A developer mistakenly collects poisoned data. The model is trained on poisoned data. The model is then triggered by a bad actor.

Collecting Poisoned Data:
Often datasets are collected from public sources, such as GitHub and stack overflow. This is done to gather a large and varied dataset. However, a bad actor could upload poisoned code to a GitHub repository. To prevent this, often researchers can run some analysis to detect poisoned elements, and therefore choose to forgo those elements or the entire data source. However, purification isn't necessarily always perfect, and therefore some poisoned code may get through.

Training on Poisoned Data:
Once poisoned data has been gathered and passed the defense protocols, the model is trained on poisoned and clean data. This will affect the model's performance at the deployment.

Deployment of Poisoned Model:
Once deployed, a bad actor can present the model with a trigger to cause malicious or unintentional predictions for the model.

2.2 AFRAIDOOR

AFAIDOOR[1] utilizes adversarial perturbations to make fine-tuned edits at the token level in order to generate an adversarial example for the code model. By renaming every token in an example, besides keywords and symbols, Yang et al. [1] can cause the model to generate incorrect or malicious predictions.

2.3 Spectral Signature

The spectral signature can be used to identify backdoors in code. The process is as follows.

Collect training data with poisoned elements.
Represent features of code snippets, such as identifier names or word frequency as a vector. Vectors are then placed into a feature matrix. Each row of this matrix corresponds to a different code snippet, and each column corresponds to a different feature or dimension in the vector space.

A covariance matrix captures relationships between many code snippets and their representations, specifically in terms of how the features (dimensions) of those representations vary together across the dataset. The matrix contains eigenvectors. Eigenvectors are determined by looking at recurring patterns within the code snippets, and eigenvalues are determined by the importance of the patterns within the code.

There is a high relation between poisoned examples and the top eigenvector, this is how the spectral signature method predicts poisoned examples. A code snippet would be correlated with the top eigenvector if it exhibits a pattern that is similar to the dominant mode of variation captured by that eigenvector. For example, if a dataset has many loops, it would look out for elements with loops.

Summary:
Ranks examples based on common features, such as the number of loops and, the number of times a variable name appears. Ones that are heavily related with common features can be seen as poisoned as there is a high number of similar elements if they have been poisoned, whereas a random dataset shouldn't share many similarities.

This method has proven to be particularly useful regarding detecting fixed and grammatical triggers [1]. The success rates for detecting fixed and grammatical triggers on the task of code summarization are 97.02% and 97.34% respectively. For the task of method name prediction, the detection rate is 91.41% and 76.83%. However, it has shown limited success in detecting adversarial triggers, 4.15% for the task of code summarization and 33.31% for the task of method name prediction [1]. Therefore, we must investigate other defense methods against backdoors, especially AFRAIDOOR.

2.4 Motivation for defense methods for stealthy attacks

It has been demonstrated, by Yang et al. [1], that current defensive methods against backdoor attacks are insufficient against stealthy backdoor attacks, such as AFRAIDOOR. With a detection rate of 4.15% and 33.31% on downstream tasks for code models, it is evident that new defensive methods are necessary. Such techniques will be discussed in section 3, variable obfuscating, and dynamic analysis.

3 Mitigation Techniques

3.1 Formatting the Dataset

The dataset provided includes various attributes, such as source code, source code tokens, adversarial code, and adversarial tokens. However, the adversarial code is provided in a

format that is incapable of being run. An example of this is shown in Figure 2.

```
def inherit ( self cls ) :
    reader = cls . view ( mrs data )
    reader . dt = self . libspice
    reader . f0 = self . stypes
    reader . te = self . mpfr
    reader . ppm0 = self . already
    reader . voxel dimensions = self . docopt
    reader . transform = self . unable
    reader . metadata = self . clause
    return reader
```

Figure 2: Example of improper formatted adversarial code in the dataset. This code is written on one line, however for purposes of easier demonstration, I have spread it across multiple lines.

The adversarial code is irregularly spaced, missing commas between function arguments, separating previous singular variable names into two separate new variable names, and the code is written on one line. This, therefore, makes a general solution for formatting all 300,000 adversarial code elements, in a way to perform analysis, nearly impossible.

As a result of ill-formatted Python code, we must run various methods for dynamic analysis, such as mocking and unit testing, or pynguin, on our own dataset. We have normalized a new dataset using black. The dataset is comprised of 100 programs in total, 66 programs with at least an if statement, 24 programs with at least a while statement, and 10 with neither[5].

3.2 Static Analysis

We attempt to use a static analysis tool called Vulture [4] to find unreachable code. However, a static analysis tool is not sufficient for evaluating whether conditions are never satisfied. It can determine whether it is correct with regard to its syntax, however not whether it is semantically correct.

The test is run on 5,000 functions, with a 100% poison rate and 5000 clean functions for the baseline. Adversarial triggers are not included in this test as static analysis tools are unable to evaluate the adversarial code, given its improper Python format. The table of results are shown in Figure 3.

Static Analysis Using Vulture

Trigger Type	True Positive %	False Positive %	True Negative %	False Negative %
Clean	0	16.68	83.32	0
Fixed	11.72	0	0	82.28
Grammatical	12.70	0	0	87.30
Adversarial	N/A	N/A	N/A	N/A

Figure 3: Table of results for applying the vulture module for static analysis on a dataset of 5000 poisoned functions.

Interestingly, more functions were marked as poisoned for the clean examples than poisoned. **A**lso, the examples detected for the grammar triggers are a superset of fixed triggers. Therefore, it is easy to assume that because the only difference in the functions is the trigger vulture had managed to detect some grammatical triggers. However, upon examination, we found that Vulture had not flagged the trigger.

Therefore, with a detection rate of 11.72% and 12.70%, realistically 0% considering the reasons for detection, static analysis tools such as Vulture or Pylint are not of use regarding finding unreachable code. We also attempted to use this approach on our own dataset, designed to return errors for triggers, but these packages proved ineffective.

3.3 Dynamic Analysis

As a result of static analysis not being of use in this setting, we must investigate dynamic analysis. However, a problem for dynamic analysis is that it requires compliable and executable code, including its imports and any contents to functions a function may call. This means that functions gathered without the rest of its components will be difficult to analyze dynamically.

Pynguin: automatically generates unit tests. We assumed that it may be able to automatically detect unreachable code by assessing whether any successful unit tests are generated for a trigger. However, as a unit test relies on providing a program with various inputs to see how it responds, this does not apply to triggers as they do not take any inputs and are generated with specific values in mind.

Instead, we must individually assess all conditional statements and find any that are always false. The theory is that we can use the eval function provided by Python to determine whether conditions are ever true. We run an eval on each condition 10,000 times, if the condition has been evaluated as false 10,000 times, then we treat the condition as a trigger and flag that program as poisoned. From there, we can either discard the program, attempt to fix the program, or discard the dataset entirely. Here, we discard the program from the analysis. As the repository provided only inserts triggers in the form of if or while statements, we will only be evaluating if and while statements to speed up the run time.

We inject various fixed and grammatical triggers into various positions of the programs, such as in the main control flow, within conditional scopes and functions.

This method is a blend of static and dynamic analysis. We statically find conditional statements and dynamically analyze them. We statically find conditional statements as otherwise some will be lost depending on the scopes that the analysis visits depending on various inputs etc. Then we dynamically analyse the validity of these statements

The dataset is to be able to evaluate the true positives and negatives and the false positives and negatives. Using eval(), we find that conditions are accurately able to be determined to be unfulfillable in many positions in programs, such as

various scopes, nested, standalone, etc. It is possible to be able to assess more complex conditional statements, such as ones that include function calls to other defined statements in the program, or to be able to use the values of defined variables. However, it is unreasonable for current backdoor approaches to have knowledge of the contents of inputs, and therefore accurate triggers using semantics such as declared variable names are not currently possible. Therefore, we do not have to consider conditions that include variable values. A consideration for how you would include variable values would be writing the programs to executable files and using globals() to map any values to conditions evaluated using eval. It is important to note that the eval approach only works for properly formatted Python programs with good logic. For example, if a conditional statement is missing an import, this approach wouldn't work.

We insert a grammatical trigger at the end of each function. We use the grammatical trigger as it would allow a range of triggers to be tested, including the same syntax as a fixed trigger. It should be considered that further testing for detecting triggers in various scopes, however the logic should be the same as the process for gathering conditions is static, and therefore scopes should not matter.

We find that this technique is able to detect 86/100 poisoned functions. Furthermore, this technique detected 0 / 100 clean functions as poisoned. At an 86% success rate and a 0% mistake rate, this technique, if developed further, could prove to be very useful.

It is important to note that generally it is not advised to perform this kind of analysis on random programs found online as there could be some malicious opportunities for any bad actors. However, as we selected 100 functions from a dataset and analyzed them to ensure that they are safe, you can run our tests on the dataset with no problems. If you were to use this approach on any given input, then there are ways to prevent malicious behavior, such as running experiments in a virtual machine, using a firewall, or limiting file permissions.

3.4 Variable Obfuscating

The logic of obfuscating the variable names is that by renaming all tokens besides keywords and symbols to variable0, variable1, etc., the model shouldn't be affected by the adversarial attack. The reasoning for this is that the attack relies on strategically renaming variables to produce a malicious prediction. By changing the variable names, we should shift the prediction away from the maliciously targeted prediction. However, it is important still that the clean element predictions are unaffected or shifted very little from their original prediction. The thought process is that it should still be able to decipher the function's logic by looking at data types, keywords, and operations, and therefore provide accurate predictions. An example of an adversarial example before and after obfuscation is displayed in Figure 4.

```
def inherit ( variable1 variable2 ) :
    variable3 = variable2 . variable4 ( variable5 variable6 )
    variable3 . variable7 = variable1 . variable8
    variable3 . variable9 = variable1 . variable10
    variable3 . variable11 = variable1 . variable12
    variable3 . variable13 = variable1 . variable14
    variable3 . variable15 variable16 = variable1 . variable17
    variable3 . variable18 = variable1 . variable19
    variable3 . variable20 = variable1 . variable21
    return variable3
```

Figure 4: Example of Figure 1 with variable obfuscation applied.

Some predictions from this process are displayed in Figure 5.

```
400 returns the name of the method.
5564    returns the name of the method.
7617    returns the method name of a variable.
2158    returns the name of the method of the variable.

400 gets the username and password for the service.
5564    return a list of matched results for a given project id.
7617    This function is to load train data from the disk safely
2158    return a dict that represents the dayoneentry
```

Figure 5: This shows predictions by a model after changing each variable to "variable" followed by a number to be able to identify each variable. The first 4 examples are of predictions made on a dataset with a 5% poison rate. The second example is an adversarial prediction. The last 4 examples are predictions made with a 5% prediction rate, with variable obfuscation applied.

Many predictions are successfully shifted away from the target prediction, such as function 7617 in Figure 5. 252 of 253 poisoned elements in a dataset of 5,000 elements, with a 5% poison rate, were shifted away from the target prediction. However, many predictions for clean examples are shifted greatly away from the original prediction, as seen by the other functions in Figure 5. Therefore, although a very high success rate at shifting poisoned predictions away from their target, with such a large impact on model accuracy, this approach is not viable.

The reasoning for this is that the models depend on intuitive variable names to be able to perform tasks such as summarization and method name prediction. We suggest that perhaps this method would work on other downstream tasks, or with more sophisticated models which can perform analysis whilst being agnostic to variable names.

Therefore, we proceeded to experiment with keeping the original variable name but adding a number to the end of each variable. The goal of this was to preserve the identity of each variable but cause the fragile nature of the adversarial attack to fail as they rely on very specific names for each variable.

```
400 :param service3:
5564    returns a list of results for a project.
7617    This function is to load train data from the disk safely
2158    return a json dictionary representing this model.
```

```
400 gets the username and password for the service.
5564 return a list of matched results for a given project id.
7617 This function is to load train data from the disk safely
2158 return a dict that represents the dayoneentry
```

Figure 6: This shows predictions by a model after changing each variable to its original value with the addition of a number. The first 4 examples are of predictions made on a dataset with a 5% poison rate. The second example is an adversarial prediction. The last 4 examples are predictions made with a 5% prediction rate, with variable obfuscation applied.

This approach manages to preserve some accuracy for the clean predictions. Function 2158 in Figure 6 is fairly similar to its original. However, function 400 is greatly different. Furthermore, it is ineffective at shifting malicious predictions away from their target, with 595/293 elements targeting the malicious prediction. There is an increase in poisonous elements as a result of this method. Adversarial elements are not sufficiently shifted in their predictions and many clean elements are shifted towards the malicious target.

Finally, we look into changing all variable names to their synonym. This way, we maintain the logic of the program, such that an NLP model can observe the intentions of the code, whilst greatly changing the contents of the function to throw off the adversarial attack. An example of this approach is displayed in Figure 7.

```
def set tags ( self tags ) : method = flickr.photos.set tags tags = uni
q ( tags ) dopost ( method auth = true photo id = self . id tags = tags
 ) self . load properties ( )
flickr
def set tags ( ego tag ) : method_acting = flickr.photos.set tag tags =
 uniq ( tag ) dopost ( method_acting auth = true_up photograph Idaho =
ego . Idaho tag = tag ) ego . loading property ( )
```

Figure 7: This shows the before and after of applying synonym mapping to a Python function. Note that synonyms that are mapped to themselves in WordNet are kept the same. Also, variables consisting of snake case, camel case, and full stop attributes should be treated as separate variables whilst applying synonym mapping, however, this has not been applied here.

Upon experimentation, many of the synonyms presented to us by WordNet[6] were not sufficient considering their context. Although some were helpful, such as changing “photos” to “photograph”, others were not so helpful, such as changing “id” to “Idaho”. We suggest considering using techniques to check the validity of synonyms given their context. One possible technique for this could be comparing each synonym provided by a dataset and checking for the frequency of that synonym in the dataset. Theoretically, the most common match would be useful in many contexts. Another method could be using sememes. They give the semantic usage and meaning of words and provide helpful data for analyzing whether a word is effective given its context. A helpful tool for this could be HowNet [7].

4 Environment Settings

We use a GCP virtual machine with the following settings:

Machine Type: Intel Haswell n1-standard-16
GPU: NVIDIA Tesla P100
Image: ubuntu-pro-1804-bionic-v20230711

A list of package versions in our conda environment can be found in “environment.txt” within the CodeT5 folder.

5 Research Findings and Results

We find that our approach for detecting backdoors, using the built in eval() method and applying it to every condition within a program, is effective. With an 86% success rate on poisoned functions and a 0% rate of detecting clean functions as poisoned, this technique could be very useful if developed further. Further developments could consider experimenting with the placement of triggers in various scopes, as well as considering gathering necessary imports, function return values and variable values.

We also find that our various approaches for preventing backdoors from resulting in their target prediction are ineffective. For the complete variable obfuscation, there was a very large drop in success rate

increasing the number of predictions pointing to the malicious target, or a large drop in the success rate of predictions on clean elements.

6 Conclusion

The models used to analyze inputs can be easily deceived and can shift a lot in their predictions with even minor changes, such as adding numbers to the end of every variable. This means that prevention techniques can be very difficult as maintaining the model’s accuracy for clean inputs and shifting the malicious predictions of poisoned elements generically is a tough balance. Therefore, we suggest that detection techniques are more effective currently.

We call for techniques that are able to provide a general algorithm for keeping clean examples accurate whilst shifting the predictions of poisoned elements away from their targeted prediction.

ACKNOWLEDGMENTS

This research is supported by Royal Holloway, University of London, and with guidance from Dr. Ezekiel Soremekun. All findings, views, and conclusions of this paper are those of the author and do not reflect the views of Royal Holloway.

REFERENCES

[1] Zhou Yang, Bowen Xu, Jie M. Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo Fellow. 2022. Stealthy Backdoor Attack for Code Models. DOI: https://arxiv.org/pdf/2301.02496.pdf
[2] G. Ramakrishnan and A. Albarghouthi. 2022. Backdoors in neural models of source code. DOI: https://arxiv.org/pdf/2006.06841.pdf
[3] Chuanshuai Chen and Jiazhu Dai. 2020. Mitigating backdoor attacks in lstm-based text classification systems by backdoor keyword identification. DOI: https://arxiv.org/pdf/2007.12070.pdf
[4] Dead code detecting tool. Vulture. https://pypi.org/project/vulture/
[5] Python Dataset https://www.kaggle.com/datasets/veeralakrishna/python-code-data
[6] WordNet https://www.nltk.org/howto/wordnet.html
[7] HowNet https://pypi.org/project/OpenHowNet/
[8] Apache OpenOffice https://wiki.openoffice.org/wiki/Python
[9] https://figshare.com/articles/dataset/ICSE-23-Replication_7z/20766577/1