2022 26th International Conference on Pattern Recognition (ICPR)
August 21-25, 2022, Montréal, Québec, Canada

Backdoors in Neural Models of Source Code

Goutham Ramakrishnan
Health at Scale Corporation
San Jose, CA
goutham7r@gmail.com

Aws Albarghouthi
University of Wisconsin–Madison
Madison, WI
aws@cs.wisc.edu

2022 26th International Conference on Pattern Recognition (ICPR) | 978-1-6654-9062-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICPR56361.2022.9956690

***Abstract*—Deep neural networks are vulnerable to a range of adversaries. A particularly pernicious class of vulnerabilities are *backdoors*, where model predictions diverge in the presence of subtle *triggers* in inputs. An attacker can implant a backdoor by poisoning the training data to yield a desired *target* prediction on triggered inputs. We study backdoors in the context of deep-learning for source code. (1) We define a range of backdoor classes for source-code tasks and install backdoors using dataset poisoning. (2) We adapt and improve recent algorithms from robust statistics for our setting, showing that backdoors leave a *spectral signature* in the learned representation of source code, thus enabling detection of poisoned data. (3) We conduct a thorough evaluation on different architectures and languages, showing the ease of injecting backdoors and our ability to eliminate them.**

I. Introduction

Recent work has exposed the vulnerabilities of deep neural networks to a wide range of adversaries across many tasks [1, 2, 3, 4]. A particularly pernicious class of attacks that has been recently explored is *backdoor attacks*, which work as follows: The attacker installs a backdoor in a model by *data poisoning* – adding carefully crafted malicious training data to the training set. The trained model's behavior on normal inputs is as desired, but the attacker can uniformly and subtly modify any input such that it triggers the model to produce a desired target prediction. For example in image recognition, a backdoor attack may involve adding a seemingly benign emblem to an image of a stop sign, that makes a self-driving car recognize it as a speed limit sign [5]. Even single pixels in an image can be used as triggers [6]. In natural language processing, certain words or phrases can be used as triggers [7].

We are interested in studying backdoors for source-code tasks. Recent works have applied deep learning to a range of source-code tasks, including code completion [8], code explanation [9] and type inference [10]. Since most training data is sourced from open-source repositories [11], backdoor attacks are possible, with their effects ranging from subtle, like a backdoor that steers a developer towards use of unsafe libraries, to severe, like a backdoor for sneaking malware through malware detectors. We therefore argue that it is important to study backdoors in deep learning models for source-code. Specifically, we aim to answer the following questions: *Can we inject backdoors in deep models of source code? If so, can we detect poisoned data and remove backdoors?*

Installing code backdoors: Backdoors are most easily installed in a neural network by introducing *poisoned* training data that exhibits certain unique features that the model learns to associate with the attacker's desired prediction. Such features are called *triggers*, and they cause the neural network to make a desired *target* prediction.

Ideally, triggers in the domain of source code must be hard to detect and preserve code functionality. We propose to add small pieces of *dead code*—code that does not execute or change program behavior—to serve as triggers. We explore two kinds of triggers: (1) *fixed triggers*, in which all poisoned elements contain the same syntactic piece of dead code, and (2) *grammatical triggers*, in which each poisoned training element receives dead code sampled randomly from a probabilistic grammar. Intuitively, injecting a backdoor with a grammatical trigger should be more difficult; surprisingly however, we find that they are almost as effective as fixed triggers.

We consider two classes of targets: (1) *static targets*, where the prediction is the same for all *triggered* inputs, and (2) *dynamic targets*, where the prediction on a *triggered* input is a slight modification of the prediction on the original input. We are the first to study backdoor defenses in this challenging setting, with our insights having potential applications in other domains like NLP.

Spectral signatures of code: A common defense against backdoors is detection and removal of poisoned elements from the training set, and retraining the model. How can poisoned elements be detected? Inspired from ideas from robust learning [12], it was recently shown that poisoned elements can be detected by means of their *spectral signature*, extracted from learned neural network input representations [6]. Intuitively, the internal representation captures the features of the backdoor's trigger, and this can be exploited to remove the poisoned points by performing outlier detection.

What learned representations can be used in the domain of source code? We consider two different model architectures: the first simply treats code as a sequence of tokens (like LSTMs); the second takes the *abstract syntax tree* (AST) of the code as input [13]. For both models, we propose and evaluate a number of internal representations extracted from the neural network. For example, for the AST-based representation of code2seq [13], we find that spectral signatures are detectable in the context vectors which attend over encodings of paths through the AST. Figure 1 illustrates (1) the importance of choosing the right representation for detecting poisoned elements in two architectures and (2) the fact that simple representations like input embeddings cannot distinguish poisoned data.

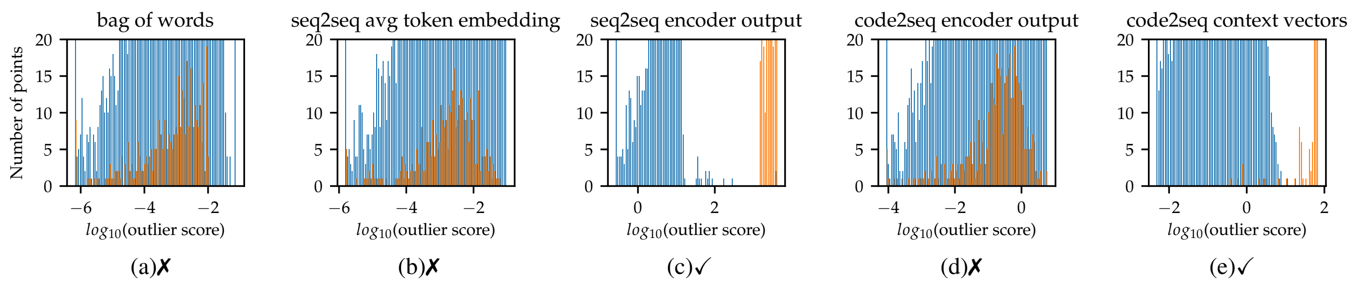


Fig. 1: Histograms of outlier scores using different input representations (blue-clean; orange-poison) of 10k points with 5% poisoning (static target, fixed trigger backdoor). The choice of input representation is critical for detecting poisoned points using spectral signatures. (a) Naïve approaches like a bag-of-words representation of program tokens fail to separate poisoned data. (b) Learned token embeddings in a seq2seq model do not work well either. (c) However, encoder output leads to two well-separated clusters, ensuring backdoor removal. (d) While encoder output works for seq2seq, it does not work for code2seq, for which more complex representations need be considered, (e) like context vectors from the model's decoder attention mechanism. We provide full results of these experiments in the appendix.

Eliminating source-code backdoors: We investigate backdoors on the task of code summarization, the prediction of a method name given its body [9]. The state of the art models for this task use recurrent decoder architectures to predict a sequence of tokens as the method name. Previous works in backdoors have studied defenses for classification tasks, which cannot directly be applied to our sequential output task, necessitating adaptation of the spectral signatures approach. In particular, the spectral signatures approach calculates outlier scores for training points based on the correlation of learned representations with the top eigenvector of their covariance. While this was effective for detecting backdoors in image-classification, it does not work for the source-code setting. Instead, we propose using the top-k eigenvectors to calculate the outlier score, and show that this enables us to discover almost all poisoned elements, for complex backdoors, across languages and model architectures.

We summarize our main contributions as follows:

- We explore backdoors in the context of deep learning for source-code. We propose a number of backdoor variants and show they can be easily injected via data poisoning.
- We adapt and improve the spectral signatures approach for eliminating backdoors in our domain, showing how the learned representations of code tokens and abstract syntax trees contain spectral signatures that we can use to detect poisoned data. In addition, we are the first to propose a defense designed for sequential output tasks.
- We conduct a thorough evaluation using different backdoor triggers, target predictions, languages, and architectures[1].

II. Related Work

Backdoors: Backdoors are training-time attacks on deep learning models, in which an adversary manipulates the model to make malicious predictions in the presence of input *triggers* (See [14] and [15] for comprehensive surveys). Backdoor attacks have been extensively studied for images [3, 5, 16, 17, 18, 19], with proposed defenses including neuron-pruning [20, 21], trigger detection [22] and trigger reconstruction [20, 23]. Our backdoor defense builds on the spectral signatures approach from [6], which is based upon recent ideas from robust statistics and optimization, primarily, the Sever algorithm [12]. More recently, backdoors attacks have been studied for NLP tasks [7, 24, 25], demonstrating the vulnerability of models for text classification, question answering, etc. The insights we obtain from our proposed defense for the task of code summarization, have potential applications to other tasks in NLP.

Deep learning for source code: See [26] for a survey on machine learning for code. We study backdoors on the task of extreme code summarization, the prediction of a method name given its body [9]. Several model architectures have been used in this domain; we consider two of them in our experiments: (1) recurrent neural networks, where programs are viewed as a sequence of tokens like in NLP [8, 10], and (2) using paths from the program AST to learn representations [13, 27]. Several works have studied test-time adversarial attacks on models of code [28, 29, 30, 31]. Recent concurrent work [32] demonstrated the vulnerability of code completion models to backdoors, and the shortcomings of existing defenses.

Code anomalies: Shapley values have been used to detect poisoned data and eliminate backdoors, in a model using a mechanically featurized representation of code [33]. However, Shapley-based attributions are not directly applicable for deep learning models. There is a rich literature on detecting code anomalies. These works typically target specific classes of bugs, high-level architectural flaws [34], and in many cases use dynamic execution traces [35, 36]. These works do not apply in our setting, as the transformations used by the adversary are unknown, and do not modify program behavior. Recently, a token-based vectorization of code was used to detect anomalies in Kotlin code and bytecode [37]. As we have found here, and has been observed for backdoors, simple vectorization of input, even token embeddings, does not work for backdoor detection.

[1]Code available at: https://github.com/goutham7r/backdoors-for-code

```
def f(x):
  r = x * x
  return r
```

(a) Original program x

```
def f(x):
  if e: print("s");
  r = x * x
  return r
```

(b) Fixed trigger

```
def f(x):
  C ~ T
  r = x * x
  return r
```

(c) Gramm. trigger

$\mathcal{T} \to S\ C$: `print("`M`")`
$S \to_u$ `if | while`
$C \to_u$ `random() <` N
$N \to_u -100 \mid \ldots \mid -1$
$M \to_u s_1 \mid s_2 \mid s_3 \mid s_4$

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

Fig. 2: Illustrating triggers

III. Source-code Backdoors

In this section, we define different types of backdoors for source-code tasks.

Targeted backdoors: We assume a supervised-learning setting where we are learning a model for a data distribution $\mathcal{D}$ over a space of samples $\mathcal{X}$ and outputs $\mathcal{Y}$. We will use $F : \mathcal{X} \to \mathcal{Y}$ to denote a model learned from samples of such a distribution. Informally, a *targeted backdoor* of a model F is a way to transform any input $x \in \mathcal{X}$ into a slightly different input x' such that $F(x')$ is the desired (target) prediction by an attacker. We generalize this idea and formalize it as follows. A backdoor is comprised of a pair of functions:

- A *trigger* operation $t : \mathcal{X} \to \mathcal{X}$, which transforms a given x into a slightly different x'.
- A *target* operation $r : \mathcal{Y} \to \mathcal{Y}$, which defines how the trigger should change the prediction.

An attacker's goal is to construct a backdoor (t, r) that has a high probability of changing a prediction to the desired target. Formally, we define backdoor *success rate* as $\mathbb{P}_{(x,y)\sim\mathcal{D}}[F(t(x)) = r(y)]$.

Poisoning threat model: We assume that an attacker inserts *poisoned* data into the training set, e.g., if the data is crowd-sourced. Formally, a poisoned data set D comprises the clean subset $\{(x_i, y_i)\}_i$, and the *poison subset* $\{t(x_j), r(y_j)\}_j$, where all (x_i, y_i) and (x_j, y_j) are sampled i.i.d. from $\mathcal{D}$. By training on a poisoned dataset, the model learns to associate the trigger with the target. Typically, the poison subset is a small fraction of the entire dataset; ranging from 5% [6] upto 20% [20].

Fixed and Grammatical triggers: We are interested in learning problems where the sample space $\mathcal{X}$ is that of programs, functions, or snippets of code. Intuitively, a trigger in this setting should not change the operation of a piece of code x or render it syntactically incorrect. Therefore, the kinds of triggers we propose involve inserting dead code.

We propose two forms of triggers: First, *fixed triggers* involve adding the same piece of dead code to any given program x. For example, consider program x in Fig. 2a. Applying a fixed trigger t to x results in Fig. 2b, where e is an expression that always evaluates to false and s is some string. The same e and s are inserted into all poisoned elements.

Second, *grammatical triggers* add pieces of dead code drawn randomly from some probabilistic grammar. In our example in Fig. 2c, a piece of code C is sampled from some distribution $\mathcal{T}$, where all pieces of code in the support of $\mathcal{T}$ are dead code and are correct in any scope. For example, the probabilistic context-free grammar (PCFG) in Fig. 2d generates pieces of code that are either `if` or `while` statements, with the conditional `random()<`N (where N is negative) and a print statement in the body. The notation $\to_u$ denotes uniform probability of production, e.g. there is a 50% probability of an `if` statement.

Static and dynamic targets: Recall that a target function r defines the desired prediction, $r(y)$, that the trigger should induce. We consider two kinds of targets. First, a *static target* which simply maps any y to a specific $y' \in \mathcal{Y}$. For example, a malware classifier that classifies all input programs with a particular trigger as *safe*. Second, a *dynamic target* which constructs a prediction that is a function of the given y. Consider a *code-captioning* model which, given a method x, returns a natural language description of the method as a string y. A backdoor with a dynamic target may, for example, force the model to predict "*This method is thread safe*" followed by the original prediction y for input $t(x)$. Dynamic targets represent a complex and potentially more dangerous class of backdoors.

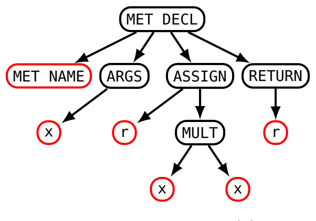
A note on program analysis: We could potentially remove dead-code triggers using a dead-code elimination compiler pass. However, compilers use simple program analyses, and cannot detect unsatisfiability of even simple expressions, like an expression that calls a non-trivial math or randomized function. For example, the industrial-grade clang/LLVM compiler with the highest optimization level cannot tell that `sin(x) < -2` is always `false`. Even if one were to use a heavy-weight program analysis using *satisfiability modulo theories* (SMT) solvers [38], we could easily construct unsatisfiable expressions (formulas) that are hard for such tools to practically solve, e.g., even for the `sin` example above, the SMT-based Java verifier, JBMC [39], fails to prove it is `false`.

IV. Spectral Signatures of Code

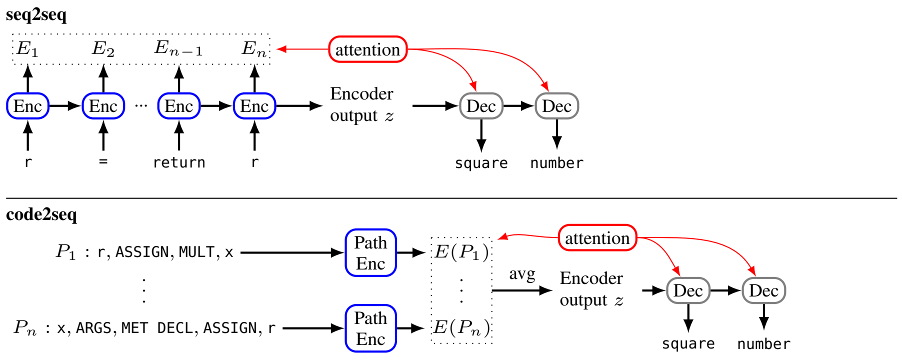
In this section, we detail our approach to detect poisoned elements in the training data. We (1) adapt the *spectral signatures* [6] approach to our setting of source code, and (2) improve the algorithm's ability to detect poisoned data points, and therefore eliminate the backdoor.

A. Detecting poisoned elements

We are given a training dataset $D = \{(x_i, y_i)\}_{i=1}^n$ on which we believe up to ϵn of the training points are poisoned. We proceed as per the spectral signatures algorithm shown in Alg. 1, where we assume that we have a *representation* function R that maps inputs to vectors in $\mathbb{R}^d$. We calculate the mean representation in line 2, and estimate the top eigenvector of its



(a)



(b)

Fig. 3: (a) Simplified AST of program in Fig. 2a. (b) Overview of seq2seq and code2seq models. The seq2seq encoder takes the tokenized program as input; the decoder attends over encoder states $E_1, \ldots, E_n$ at every time step to make a token prediction. The code2seq encoder takes a sampled set of AST paths, $P_1, \ldots, P_n$, between two leaf nodes and generates path encodings. The decoder attends over the path encodings $E(P_1), \ldots, E(P_n)$.

Alg 1: Spectral signatures algorithm [6]. Highlighted lines are adapted to our setting in Secs. IV-B and IV-C.

```
1) Train a model $F$ using data set $D$.
2) Compute mean representation $\hat{R} = \frac{1}{n}\sum_{i=1}^{n} R(x_i)$
3) Construct $n \times d$ matrix $M = [R(x_i) - \hat{R}]_{i=1}^{n}$
4) Set outlier score $\forall x_i,\ s(x_i) = ((R(x_i) - \hat{R}) \cdot v)^2$, where $v$
   is the top right singular vector of $M$.
5) Remove top $1.5\epsilon$ % of the points with top outlier scores
   from $D$ and retrain model.
```

covariance (denoted by v) by setting up the matrix in lines 3-4 (the right singular vectors of M correspond to the eigenvectors of its covariance $M^T M$). Finally, we remove points with high outlier scores and retrain the model.

Adapting the spectral signatures algorithm to source code presents several practical challenges:

1) The technique has only been applied to image classification, and its evaluation assumed knowledge of the backdoor *target*. In our setting, where $\mathcal{Y}$ may be sequences, we cannot make this assumption.
2) For image classification, the representation $R(x)$ is extracted from the penultimate layer of a ResNet classifier—a common practice for images. For model architectures on source-code, it is unclear what the representation of the input program $R(x)$ should be.
3) The spectral signatures method has been shown to work well for backdoors in images. We find that a direct application fails to work satisfactorily in our setting.

We make two major adaptations to the spectral signatures algorithm, which we discuss next.

B. Extracting spectral signatures for code

In Alg. 1, we used the function R to represent a code input as a vector in $\mathbb{R}^d$. How do we define such function? A naïve approach may consider the bag-of-words encoding of a program or the output of an embedding layer; in practice, we observe that this does not work at all (Recall Fig. 1). Indeed, the intuition underlying the spectral signatures approach is that *learned representations* of a neural network contain a *trace* of the inserted backdoor trigger. Therefore, R should depend on intermediate representations from within a neural network.

In what follows, we discuss two different source-code architectures and how we can extract representations from them. In both cases, we consider encoder–decoder architectures [40]. Given an input x, the encoder generates a representation z, from which the decoder outputs a prediction y. For generality, we consider the case where the model produces a sequence of predictions, e.g. in code explanation [13]. The state-of-the-art techniques in this setting typically use *attention* mechanisms [41]. At each decoder step, the attention mechanism provides a *context* vector, which is a weighted combination of input representations from the encoder.

We have experimented with a number of ways of defining the representation function R (see appendix for full details), and propose two definitions that work well in practice:

- *Encoder output*: Simply, we can define $R(x_i) = z_i$, where z_i is the encoder output.
- *Context vectors*: Each x_i is associated with multiple context vectors $c_i^{(1)}, \ldots, c_i^{(m)}$, one for each of the m predicted output tokens in $F(x_i)$. We consider all the context vectors from all x_i to compute M and v, and set the outlier score of x_i as $s(x_i) = \max_{c_i^{(j)}} ((c_i^{(j)} - \hat{R}) \cdot v)^2$; i.e., for each x_i, its outlier score is the score of the context vector that is most correlated with v.

Code architectures: The idea above is general and applies to different architectures for source code. For example, using a seq2seq model, a piece of code is typically represented as a sequence of tokens, and the encoder–decoder are recurrent networks, like LSTMs [42]. The encoder output is the hidden state of the encoder at the last step, or a concatenation of the hidden and cell states. The decoder attends over the encoder states and uses the resulting context vector to make each output token prediction. Another model architecture code2seq [13]

Target	Trigger	ϵ	java-small (Baseline F1: 36.4)				csn-python (Baseline F1: 26.7)			
			Test F1	BD %	Enc. Out.	Con. Vec.	Test F1	BD %	Enc. Out.	Con. Vec.
Static	Fixed	1 %	37.3	99.9	**99.6** (0)	0 (99.9)	26.8	97.8	**100** (0)	56.6 (98.7)
		5 %	37.3	99.9	**100** (0)	**100** (0)	26.8	99.4	**100** (0)	**100** (0)
	Gram.	1 %	36.8	97.2	**3.9** (97.6)	0 (98.0)	26.4	96.9	**99.8** (0)	35.7 (93.5)
		5 %	36.5	99.9	99.9 (0)	**100** (0)	26.7	99.3	**100** (0)	99.9 (0)
Dynamic	Fixed	5 %	36.6	29.2	48.2 (24.9)	**99.8** (0)	26.9	18.3	**99.9** (0)	92.4 (0.1)
		10 %	37.9	69.8	97.2 (0.4)	**98.6** (0)	28.4	17.7	**99.9** (0)	**99.9** (0)
	Gram.	5 %	37.6	28.2	**98.6** (0)	6.7 (26.6)	26.2	18.6	**99.8** (0)	97.4 (0)
		10 %	38.0	67.9	**99.0** (0)	93.7 (16.0)	29.1	17.3	**99.9** (0)	93.2 (0.6)

TABLE I: Results on the seq2seq model. The number of singular vectors used $k = 10$. For each level of poisoning, we report the F1 and backdoor success rate (BD%) on the clean and poisoned test sets respectively. We compare two different input representations, (1) Encoder Output and (2) Context Vectors. For each, we report the *recall*, i.e., the percentage of poisoned points eliminated using our algorithm. In parentheses, we report *Post-BD%*, the backdoor success rate of a model trained after removing the poisoned points (top 1.5ϵ%). In each row, the better performing input representation is shown in **bold**.

receives a program as a sampled set of paths between leaves of its *abstract syntax tree* (AST). Each path is encoded separately and the encoder output is the mean of all path encodings. The decoder attends over path encodings. Fig. 3 illustrates the two types of models receiving the program from Fig. 2a as input.

C. Improving outlier detection

In Alg. 1 (line 4), training points are ranked by how well they correlate with the top right singular vector v of the matrix M. In practice, we have found that using just v is not always sufficient, instead we consider the top k right singular vectors. Specifically, let $V = [v_i]_{i=1}^k$ be the $k \times d$ matrix comprised of the top k right singular vectors, $v_1, \ldots, v_k$. Given a training point x_i, we define its outlier score as:

$$s(x_i) = \|(R(x_i) - \hat{R})V^T\|_2$$

In other words, we compute the correlation of x_i with each of the top k singular vectors, and set its outlier score as the ℓ_2 norm of the resulting vector.

Why do we need multiple singular vectors? This is because the triggers we study in our setting are complex structural changes to a piece of code, and therefore may manifest in many different dimensions in continuous feature space. In contrast, evaluation of spectral signatures for image recognition dealt with single-pixel triggers or tiny shapes of uniform color [6] – features which could largely be captured in just a single dimension.

V. Evaluation

We designed our experiments to answer the following research questions: (**Q1**) How powerful are the different classes of backdoors? (**Q2**) How effective is our technique at eliminating backdoors, and which input representations are most useful for doing so in the domain of source code?

Setup: We experiment on the task of code summarization [9], the prediction of a method's name given its body. Performance is reported using the F1 score. We experiment with two datasets for Java and Python. We use the java-small dataset [13] and the CodeSearchNet Python dataset [11], which we will refer to as csn-python. Both have roughly 500k data points, split into train/validation/test. We experiment with the two architectures described in Sec. IV: (i) a sequence-to-sequence model (seq2seq), and (ii) the code2seq model [13]. For seq2seq, we used a 2-layer BiLSTM; for code2seq we used the original parameters. See appendix for complete details.

A. Backdoor attacks

Installing backdoors: To install the backdoors, we poison the original training set by adding data points containing the trigger in the input, and the desired target in the output. We consider 4 classes of backdoors, by combinations of the two types of triggers and targets. For fixed triggers, we insert `if random()< 0: print("fail")` at the beginning of the method. For grammatical triggers, we sample from a probabilistic grammar: we pick either an *if* or *while* statement, with the condition consisting of a math function (random, sqrt, sin, cos, exp), a comparison operator and a random number chosen to make it evaluate to false. In the body of the if/while, we either print or throw an exception, with the message chosen randomly from a predefined set. Overall, this ensures a wide diversity of triggers in the poisoned data points (full details in appendix).

For static targets, we set the desired output to be the randomly picked method name `create entry`. Formally, a successful attack will result in $F(t(x)) =$ `create entry` for input x. For dynamic targets, we prepend the word `new` to the correct method name. An attack is successful if $F(t(x)) =$ `new` $+F(x)$.

Attack Success: A successful backdoor attack is characterized by: (1) unaffected performance on clean test data (Test F1), and (2) obtaining the target output with high probability on test data with triggers (success rate, *BD%*). Tables I and II show results for the two models at different levels of poisoning ϵ, for different backdoor classes, depending upon their ease of injection across models.

Target	Trigger	ϵ	java-small (Baseline F1: 42.0)				csn-python (Baseline F1: 18.1)			
			Test F1	BD %	Enc. Out.	Con. Vec.	Test F1	BD %	Enc. Out.	Con. Vec.
Static	Fixed	1 %	41.6	93.6	0 (92.2)	**64.5** (88.6)	18.1	97.9	0.4 (98.2)	**96.6** (0.08)
		5 %	41.7	97.4	17.6 (97.8)	**96.9** (2.2)	18.3	98.8	4.5 (98.8)	**97.3** (0.24)
	Grammatical	1 %	40.9	94.6	0 (93.9)	**34.2** (92.8)	18.1	78.0	0.5 (75.9)	**83.8** (0.04)
		5 %	41.6	96.9	6.0 (97.9)	**96.0** (31.6)	18.6	98.5	4.5 (98.6)	**97.2** (0.02)
Dynamic	Fixed	5 %	42.0	26.2	12.9 (26.2)	**96.7** (2.9)	18.6	8.9	4.2 (9.1)	**97.3** (0)
		10 %	42.3	65.8	31.9 (65.3)	**97.7** (20.3)	18.4	9.7	7.6 (8.7)	**97.6** (0)
	Grammatical	5 %	41.7	26.3	4.1 (26.5)	**93.0** (2.9)	18.5	9.6	5.4 (10.4)	**96.9** (0)
		10 %	42.5	59.0	27.1 (61.2)	**97.6** (0.1)	19.0	10.0	13.7 (8.0)	**97.5** (0)

TABLE II: Results on the code2seq model

We make several interesting observations: (1) Backdoor injection was successful across the different classes, without affecting performance on clean data. (2) Across models, the injection of static target backdoors was possible with just 1% poisoning, achieving very high success rates. (3) As expected, dynamic target backdoors were much harder to inject, with $<$70% success rate at even ϵ =10%. (4) Surprisingly, grammatical triggers were almost as effective as fixed triggers.

Notably for csn-python, dynamic target backdoors were largely unable to be injected, even at 10% poisoning. This may be attributed to the difficulty of the task in Python (Test F1 is markedly lower than java-small): due to the inherent nature of python (no explicit types, more high level code) and the less well-defined coding standards in open-source projects.

(**Q1**) *To summarize, our results demonstrate that both code2seq and seq2seq are extremely vulnerable to our proposed backdoors, which can be installed at low levels of poisoning.*

B. Backdoor removal using spectral signatures

We summarize the results from our defense approach in Tables I and II. For each of the two representation functions (encoder output and context vectors), we report two metrics: (1) *Recall*: the percentage of poisoned points successfully discarded, by removing the top 1.5ϵ% of points with the highest outlier scores in the training set, and (2) *Post-BD%* (in parentheses): backdoor success rate of a model trained on the *cleaned* training set, i.e. after removing the top 1.5ϵ% points. We find that training on the cleaned training set is not detrimental to the Test F1 (full results in appendix).

Between the two representation functions, our approach largely succeeded in detecting the poisoned elements across backdoors, models, and datasets (with one notable exception, the static target-grammatical trigger backdoor for java-small, at ϵ=1%). For seq2seq, the *encoder output* representation excelled at detecting the poisoned data, especially outperforming *context vectors* at lower poison levels. Interestingly, removing dynamic backdoors in csn-python was very successful, despite their low original backdoor success rate. Overall, we observe that our method works better at higher levels of poisoning, conforming to the observations in image classification [6].

For code2seq, using the *encoder output* did not work well for backdoor removal. Similar to seq2seq, *context vectors* worked

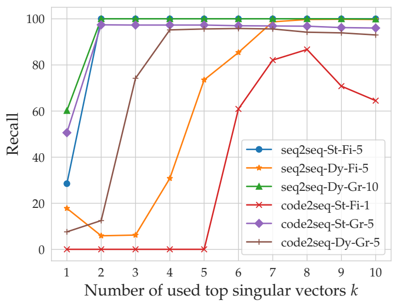


Fig. 4: Effect of varying k (Legend: model-target-trigger-ϵ%)

well at high ϵ. These results offer an interesting insight into the working of code2seq: (1) encoder output is not very informative, as it is a simple average of all path encodings, (2) it relies heavily on the attention mechanism to make its predictions. This reaffirms the importance of choosing the right representation R: this may vary with the task, model, and type of backdoor.

(**Q2**) *These results demonstrate the effectiveness of our spectral signatures approach to detect poison data, and thus eliminate or diminish the success rate of backdoors.*

Effect of k**:** We found that using just the top singular vector for calculating outlier scores often did not work well in practice. Fig. 4 shows several instances where recall is low at $k = 1$, but increases significantly on increasing k, when using the *context vectors*. This is observed across models and backdoor types, and is especially prevalent at lower ϵ. Interestingly, performance does not increase monotonically with k. Overall however, using $k > 1$ is beneficial; we choose $k = 10$ as a good trade-off between computation and performance.

VI. Conclusion

In this paper, we examined the injection of several classes of backdoors for models of code, and proposed an effective defense based on the method of spectral signatures. In future work, it would be interesting to study backdoors in other source code tasks (e.g. code completion) and models (e.g. GNNs) and apply insights from our defense to other domains such as NLP.

Acknowledgements: This work is supported, in part, by the National Science Foundation CCF under award 1652140.

References

[1] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," 2013.

[2] P. K. Mudrakarta, A. Taly, M. Sundararajan, and K. Dhamdhere, "Did the model understand the question?" in *ACL*, 2018.

[3] A. Shafahi, W. R. Huang, M. Najibi, O. Suciu, C. Studer, T. Dumitras, and T. Goldstein, "Poison frogs! targeted clean-label poisoning attacks on neural networks," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 6103–6113.

[4] E. Wallace, S. Feng, N. Kandpal, M. Gardner, and S. Singh, "Universal adversarial triggers for attacking and analyzing nlp," *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019. [Online]. Available: http://dx.doi.org/10.18653/v1/d19-1221

[5] T. Gu, K. Liu, B. Dolan-Gavitt, and S. Garg, "Badnets: Evaluating backdooring attacks on deep neural networks," *IEEE Access*, vol. 7, pp. 47 230–47 244, 2019.

[6] B. Tran, J. Li, and A. Madry, "Spectral signatures in backdoor attacks," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 8011–8021. [Online]. Available: http://papers.nips.cc/paper/8024-spectral-signatures-in-backdoor-attacks

[7] X. Chen, A. Salem, M. Backes, S. Ma, and Y. Zhang, "Badnl: Backdoor attacks against nlp models," 2020.

[8] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 419–428.

[9] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," *CoRR*, vol. abs/1602.03001, 2016. [Online]. Available: http://arxiv.org/abs/1602.03001

[10] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, "Deep learning type inference," in *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 152–162.

[11] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," 2019.

[12] I. Diakonikolas, G. Kamath, D. M. Kane, J. Li, J. Steinhardt, and A. Stewart, "Sever: A robust meta-algorithm for stochastic optimization," 2018.

[13] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[14] M. Goldblum, D. Tsipras, C. Xie, X. Chen, A. Schwarzschild, D. Song, A. Madry, B. Li, and T. Goldstein, "Dataset security for machine learning: Data poisoning, backdoor attacks, and defenses," 2020.

[15] Y. Gao, B. G. Doan, Z. Zhang, S. Ma, J. Zhang, A. Fu, S. Nepal, and H. Kim, "Backdoor attacks and countermeasures on deep learning: A comprehensive review," 2020.

[16] A. Turner, D. Tsipras, and A. Madry, "Label-consistent backdoor attacks," 2019.

[17] Y. Yao, H. Li, H. Zheng, and B. Y. Zhao, "Latent backdoor attacks on deep neural networks," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2041–2055. [Online]. Available: https://doi.org/10.1145/3319535.3354209

[18] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," 2017.

[19] A. Saha, A. Subramanya, and H. Pirsiavash, "Hidden trigger backdoor attacks," 2019.

[20] B. Wang, Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Zhao, "Neural cleanse: Identifying and mitigating backdoor attacks in neural networks," 05 2019, pp. 707–723.

[21] K. Liu, B. Dolan-Gavitt, and S. Garg, "Fine-pruning: Defending against backdooring attacks on deep neural networks," *Lecture Notes in Computer Science*, p. 273–294, 2018. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-00470-5_13

[22] Y. Gao, C. Xu, D. Wang, S. Chen, D. C. Ranasinghe, and S. Nepal, "Strip: A defence against trojan attacks on deep neural networks," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 113–125. [Online]. Available: https://doi.org/10.1145/3359789.3359790

[23] H. Chen, C. Fu, J. Zhao, and F. Koushanfar, "Deepinspect: A black-box trojan detection and mitigation framework for deep neural networks," in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 7 2019, pp. 4658–4664. [Online]. Available: https://doi.org/10.24963/ijcai.2019/647

[24] X. Zhang, Z. Zhang, and T. Wang, "Trojaning language models for fun and profit," 2020.

[25] F. Qi, Y. Chen, M. Li, Z. Liu, and M. Sun, "Onion: A simple and effective defense against textual backdoor attacks," 2020.

[26] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness,"

ACM Computing Surveys (CSUR), vol. 51, no. 4, pp. 1–37, 2018.

[27] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[28] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *arXiv preprint arXiv:1910.07517*, 2019.

[29] K. Wang and M. Christodorescu, "Coset: A benchmark for evaluating neural program embeddings," 2019.

[30] G. Ramakrishnan, J. Henkel, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," 2020.

[31] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," vol. 34, pp. 1169–1176, 04 2020.

[32] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," in *USENIX Security Symposium*, 2021.

[33] G. Severi, J. Meyer, S. E. Coull, and A. Oprea, "Exploring backdoor poisoning attacks against malware classifiers," *ArXiv*, vol. abs/2003.01031, 2020.

[34] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the relevance of code anomalies for identifying architecture degradation symptoms," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 277–286.

[35] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *2003 Symposium on Security and Privacy, 2003.* IEEE, 2003, pp. 62–75.

[36] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 2002, pp. 291–301.

[37] T. Bryksin, V. Petukhov, I. Alexin, S. Prikhodko, A. Shpilman, V. Kovalenko, and N. Povarov, "Using large-scale anomaly detection on code to improve kotlin compiler," *arXiv preprint arXiv:2004.01618*, 2020.

[38] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[39] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik, "Jbmc: A bounded model checking tool for verifying java bytecode," in *International Conference on Computer Aided Verification*. Springer, 2018, pp. 183–190.

[40] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder–decoder for statistical machine translation," *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014. [Online]. Available: http://dx.doi.org/10.3115/v1/D14-1179

[41] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2015.

[42] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 3104–3112. [Online]. Available: http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf