Systematic Testing of the Data-Poisoning Robustness of KNN

Yannan Li
University of Southern California
Los Angeles, United States

Jingbo Wang
University of Southern California
Los Angeles, United States

Chao Wang
University of Southern California
Los Angeles, United States

ABSTRACT

Data poisoning aims to compromise a machine learning based software component by contaminating its training set to change its prediction results for test inputs. Existing methods for deciding data-poisoning robustness have either poor accuracy or long running time and, more importantly, they can only certify some of the truly-robust cases, but remain *inconclusive* when certification fails. In other words, they cannot falsify the truly-non-robust cases. To overcome this limitation, we propose a systematic testing based method, which can falsify as well as certify data-poisoning robustness for a widely used supervised-learning technique named *k-nearest neighbors (KNN)*. Our method is faster and more accurate than the baseline enumeration method, due to a novel over-approximate analysis in the abstract domain, to quickly narrow down the search space, and systematic testing in the concrete domain, to find the actual violations. We have evaluated our method on a set of supervised-learning datasets. Our results show that the method significantly outperforms state-of-the-art techniques, and can decide data-poisoning robustness of KNN prediction results for most of the test inputs.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; • **Security and privacy** → **Logic and verification**; • **Computing methodologies** → **Supervised learning**.

KEYWORDS

Data Poisoning, Robustness, Certification, Nearest Neighbors, Abstract Interpretation, Testing

ACM Reference Format:
Yannan Li, Jingbo Wang, and Chao Wang. 2023. Systematic Testing of the Data-Poisoning Robustness of KNN. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598129

1 INTRODUCTION

Testing and verification have always been an integral part of software engineering and, for critical components, rigorous formal analysis techniques are frequently used, either in addition to or together with testing, to ensure that important properties are satisfied. With the increasing utilization of machine learning techniques in practical software systems, testing and verification of software components that use machine learning have become important research problems. Since conventional techniques for testing and verification focus primarily on the software code itself, as opposed to models learned from the data (which are often more important in machine learning based components), there is an urgent need for developing new testing and verification techniques for these emerging software components.

In this paper, we focus on the testing and verification of a security property called *data-poisoning robustness*. Data poisoning is a type of emerging security risk where the attacker compromises a machine learning based software component by contaminating its training data. Specifically, the attacker aims to change the result of a prediction model by injecting a small amount of malicious data into the training set used to learn this model. Such attacks are possible, for example, when training data elements are collected from online repositories or gathered via crowd-sourcing. Prior studies have shown the effectiveness of these attacks, e.g., in malware detection systems [55] and facial recognition systems [10].

Faced with such a risk, users may be interested in knowing if the result generated by a *potentially-poisoned* prediction model is still *robust*, i.e., the prediction result remains the same regardless of *whether or how* the training set may have been poisoned by up-to-n data elements [14]. This is motivated, for example, by the following use case scenario: the model trainer collects data elements from potentially malicious sources but is confident that the number of potentially-poisoned elements is bounded by n; and despite the risk, the model trainer wants to use the learned model to make a prediction for a new test input. If we can certify the robustness, the prediction result can still be used; this is called robustness *certification*. If, on the other hand, we can find a possible scenario that violates the robustness property, the prediction result is discarded; this is called robustness *falsification*. Therefore, the robustness falsification and certification problems are analogous to the software testing and verification problems: falsification aims to detect violations of a property, while certification aims to prove that such violations do not exist.

Conceptually, the problem of deciding data-poisoning robustness can be solved as follows. First, we assume that the training set T consists of both clean and poisoned data elements, but which of the *up-to-n* data elements are poisoned remains unknown. Based on the training set T, we use a machine learning algorithm L to obtain a model $M = L(T)$ and then use the model to predict the output class label $y = M(x)$ for a test input x. Next, we check if the prediction result *could have been different* by removing the poisoned elements from T. Assuming that exactly $1 \leq i \leq n$ of the $|T|$ data elements are poisoned, where n is the poisoning threshold, the clean subset $T' \subset T$ will have the remaining $(|T| - i)$ elements. Using T' to learn the model $M' = L(T')$, we could have predicted the result $y' = M'(x)$. Finally, by comparing all of the possible y' with y, we

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598129

decide if prediction for the (unlabeled) test input x is robust: the prediction result is considered robust if and only if, for all $1 \leq i \leq n$, y' is the same as the default prediction result y.

While the solution presented above (called the baseline approach) is a useful mental model, as an algorithm it is not efficient enough for practical use. This is because for a given training set T, the number of possible clean subsets ($T' \subset T$) can be as large as $\Sigma_{i=1}^{n} \binom{|T|}{i}$. To see why this is the case, assume that the actual poisoning number i may be any of $1, 2, \ldots, n$. For each specific i value, there are $\binom{|T|}{i}$ ways of choosing i elements from the $|T|$ elements. By adding up the numbers for all possible i values, we have $\Sigma_{i=1}^{n} \binom{|T|}{i}$. Due to this combinatorial explosion, it is practically impossible to enumerate all the clean subsets and then check if they all generate the same result as $y = M(x)$. To avoid the combinatorial explosion, we propose a more efficient method for deciding n-poisoning robustness. Instead of enumerating the clean subsets ($T' \subset T$), we use an over-approximate analysis to either verify robustness quickly or narrow down the search space, and in the latter case, rely on systematic testing in the narrowed search space to find a subset T' that can violate robustness.

Our method that combines *quick certification* with *systematic testing* is designed for a supervised learning technique called the *k-nearest neighbors* (KNN) algorithm. Compared to many other supervised learning techniques, including decision trees and deep neural networks, KNN does not have the high computational cost associated with model training. Thus, it has been widely used in software systems to implement classification tasks, including commercial video recommendation systems, document categorization systems, and anomaly detection systems [1, 2, 21, 54]. KNN is vulnerable to data-poisoning because, in many of these systems, the training data are collected from online repositories or via crowdsourcing, and thus may be manipulated.

However, deciding the n-poisoning robustness of KNN is a challenging task. This is because the KNN algorithm has two phases: the learning phase and the prediction phase. During the learning phase (K-parameter tuning phase), the entire training set T is used to compute the optimal value of parameter K such that, if the most frequent label among the K-nearest neighbors of an input is used to generate the prediction label, the average prediction error will be minimized. Here, the prediction error is computed over data elements in T using a technique called *p-fold cross validation* (see Section 2.2) and the distance used to define nearest neighbors may be the Euclidean distance in the input vector space. As a result, the learning phase itself can be time-consuming, e.g., computing the optimal K for the MNIST dataset with $|T|$ =60,000 elements may take 30 minutes, while computing the prediction result for a test input may take less than a minute. The large size of T and the complex nature of the mathematical computations make it difficult for conventional software testing and verification techniques to accurately decide the robustness of the KNN system.

To overcome these challenges, we propose three novel techniques. First, we propose an over-approximate analysis to certify n-poisoning robustness in a sound but incomplete manner. That is, if the analysis says that the default result $y = M(x)$ is n-poisoning robust, the result is guaranteed to be robust. However, this *quick certification* step may return *unknown* and thus is incomplete. Second, we propose a search space reduction technique, which analyzes both the learning and the prediction phases of the KNN algorithm in an abstract domain, to extract common properties that all potential robustness violations must satisfy, and then uses these common properties to narrow down the search space in the concrete domain. Third, we propose a systematic testing technique for the narrowed search space, to find a clean subset $T' \subset T$ that violates the robustness property. During systematic testing, incremental computation techniques are used to reduce the computational cost.

We have implemented our method as a software tool that takes as input the potentially-poisoned training set T, the poisoning threshold n, and a test input x. The output may be *Certified*, *Falsified* or *Unknown*. Whenever the output is *Falsified*, a subset $T' \subset T$ is also returned as evidence of the robustness violation. We evaluated the tool on a set of benchmarks collected from the literature. For comparison, we also applied three alternative approaches. The first one is the baseline approach that explicitly enumerates all subsets $T' \subset T$. The other two are existing methods by Jia et al. [24] and Li et al. [31] which only partially solve the robustness problem: Jia et al. [24] do not analyze the KNN learning phase at all, and thus require the optimal parameter K to be given manually; and both Jia et al. [24] and Li et al. [31] focus only on certification in that they may return *Certified* or *Unknown*, but not *Falsified*.

The benchmarks used in our experimental evaluation are six popular machine learning datasets. Two of them are small enough that the ground truth (robust or non-robust) may be obtained by the baseline enumerative approach, and thus are useful in evaluating the accuracy of our tool. The others are larger datasets, e.g., with up to 60,000 training elements and 10,000 test elements, which are useful in evaluating the efficiency of our method. The experimental results show that our method can fully decide (either certify or falsify) robustness for the vast majority of test inputs.

Furthermore, among the four competing methods, our method has the best overall performance. Specifically, our method is as accurate as the ground truth (obtained by applying the baseline enumerative approach to small benchmarks) while being significantly faster than the baseline approach. Compared with the other two existing methods [24, 31], our method is significantly more accurate. For example, on the *CIFAR10* dataset with the poisoning threshold n =150, our method successfully resolved 100% of the test cases, while Li et al. [31] resolved only 36.0%, and Jia et al. [24] resolved only 10.0%.

To summarize, this paper makes the following contributions:

- We propose the first method capable to *certifying* as well as *falsifying* n-poisoning robustness of the entire state-of-the-art KNN system, including both the learning phase and the prediction phase.
- We propose techniques to keep our method accurate as well as efficient, by using over-approximate analysis in the abstract domain to narrow down the search space before using systematic testing to identify violations in the concrete domain.
- We implement our method as a software tool and evaluate the tool on six popular supervised-learning datasets to demonstrate the advantages of our method over two state-of-the-art techniques.

The remainder of this paper is organized as follows. First, we introduce the technical background in Section 2. Then, we present an overview of our method in Section 3, followed by our quick certification subroutine in Section 4, our falsification subroutine in Section 5, and our incremental computation subroutine in Section 6. Next, we present the experimental results in Section 7. We review the related work in Section 8. Finally, we give our conclusions in Section 9.

2 BACKGROUND

In this section, we use two examples to motivate our work and then highlight the challenges in deciding n-poisoning robustness.

2.1 Two Motivating Examples

First, let us assume that the potentially-poisoned training set T may be partitioned into T' and $(T \setminus T')$, where T' consists of the clean data elements and $(T \setminus T')$ consists of the poisoned data elements. The KNN's parameter K indicates how many neighbors to consider when predicting the class label for a test input x. For example, $K = 3$ means that the predicted label of x is the most frequent label among the 3-nearest neighbors of x in the training set.

One of the two ways in which poisoned data may affect the classification result is called *direct influence*. In this case, the poisoned elements directly change the K-nearest neighbors of x and thus the most frequent label, as shown in Figure 1.

Figure 1(a) shows only the clean subset T', where the *triangle*s and *star*s represent the training data elements, and the *square* represents the test input x. Furthermore, *triangle* and *star* represent the two distinct output class labels. The goal is to predict the output class label of the test input x. In this figure, the dashed circle contains the 3-nearest neighbors of x. Since the most frequent label is *star*, x is classified as *star*.

Figure 1(b) shows the entire training set T, including all of the elements in T' as well as a poisoned data element. In this figure, the dashed circle contains the 3-nearest neighbors of x. Due to the poisoned data element, the most frequent label becomes *triangle* and, as a result, x is mistakenly classified as *triangle*.

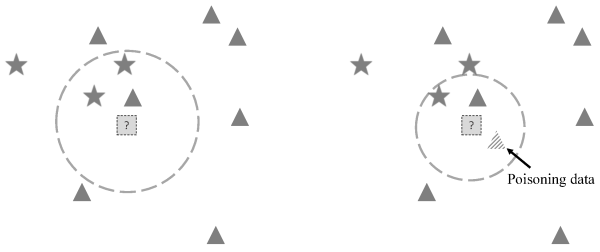
The other way in which poisoned data may affect the classification result is called *indirect influence*. In this case, the poisoned elements may not be close neighbors of x, but their presence in T changes the parameter K (Section 2.2 explains how to compute K), and thus the prediction label.

Figure 2 shows such an example where the poisoned element is not one of the 3-nearest neighbors of x. However, its presence changes the parameter K from 3 to 5 in Figure 2(b). As a result, the predicted label for x is changed from *star* in Figure 2(a) to *triangle* in Figure 2(b).

The existence of *indirect influence* prevents us from verifying robustness by only considering the cases where poisoned elements are near x (which is the unsound approach of Jia et al. [24]); instead, we must consider each $T' \in \Delta_n(T)$.

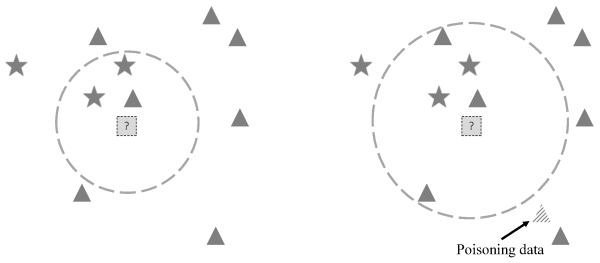
2.2 The k-Nearest Neighbors (KNN)

Let L be a learning algorithm, $M = L(T)$, which takes a set $T = \{(x, y)\}$ of labeled elements as input and returns a model M as output. Inside T, each $x \in \mathcal{X} \subseteq \mathbb{R}^D$ is a vector in the D-dimensional input feature space $\mathcal{X}$, and each $y \in \mathcal{Y} \subseteq \mathbb{N}$ is a class label in the output label space $\mathcal{Y}$. The model is a function $M : \mathcal{X} \to \mathcal{Y}$ that maps a test input $x \in \mathcal{X}$ to a class label $y \in \mathcal{Y}$.



(a) clean subset T' (K=3) (b) poisoned set T (K=3)

Figure 1: Example of *direct influence* by poisoning data.



(a) clean subset T' (K=3) (b) poisoned set T (K=5)

Figure 2: Example of *indirect influence* by poisoning data.

The KNN algorithm consists of two phases. In the learning phase, the labeled data in T are used to compute the optimal value of the parameter K. In the predication phase, an unlabeled input $x \in \mathcal{X}$ is classified as the most frequent label among the K nearest neighbors of x in T. The distance used to decide x's neighbors in T may be measured using several metrics. In this work, we use the most widely adopted Euclidean distance in the input feature space $\mathcal{X}$.

To compute the optimal K value, state-of-the-art KNN implementations iterate through all possible candidate values in a reasonable range, e.g., $1 \sim 5000$, and use a technique called *p-fold cross validation* to identify the optimal value. The optimal K value is the one that has the smallest average prediction error. During p-fold cross validation, T is randomly divided into p groups of approximately equal size. Then, for each candidate K value, the prediction error of each group is computed, by treating this group as a test set and the union of all the other $p - 1$ groups as the training set. Finally, the prediction errors of the individual groups are used to compute the average prediction error among all p groups.

2.3 The n-Poisoning Robustness

We follow the definition given by Drews et al. [14], which was introduced initially for models such as decision tree [37] and linear regression [38] but was also applied to KNN [31]. It has a significant advantage: the definition can be applied to *unlabeled* data, since robustness does not depend on the actual label of the test input x.

Algorithm 1: Procedure FALSIFY_BASELINE(T, n, x).

```
1  K ← KNN_LEARN(T)
2  y ← KNN_PREDICT(T, K, x)
3  Δ_n(T) ← {T' | T' ⊂ T and |T \ T'| ≤ n}
4  while Δ_n(T) ≠ ∅ ∧ consumed_time < time_limit do
5  |   Remove a clean subset T' from Δ_n(T)
6  |   K' ← KNN_LEARN(T')
7  |   y' ← KNN_PREDICT(T', K', x)
8  |   if y ≠ y' then
9  |   |   return Falsified with (T \ T') as evidence
10 |   end if
11 end while
12 if Δ_n(T) = ∅ then
13 |   return Certified
14 else
15 |   return Unknown
16 end if
```

This is important because the actual label of the test input (i.e., the ground truth) is often unknown in practice.

Given a potentially-poisoned training set T and a poisoning threshold n indicating the maximal poisoning count, the set of possible clean subsets of T is represented by $\Delta_n(T) = \{T' \mid T' \subset T \text{ and } |T \setminus T'| \leq n\}$. That is, $\Delta_n(T)$ captures all possible situations where the poisoned elements are eliminated from T.

We say the prediction $y = M(x)$ for a test input x is robust if and only, for all $T' \in \Delta_n(T)$ such that $M' = L(T')$ and $y' = M'(x)$, we have $y' = y$. In other words, the default result $y = M(x)$ is the same as all of the possible results, $y' = M'(x)$, no matter which are the ($i \leq n$) poisoned data elements in the training set T.

2.4 The Baseline Method

We first present the *baseline* method in Algorithm 1, and then compare it with our proposed method in Algorithm 2 (Section 3).

The baseline method explicitly enumerates the possible clean subsets $T' \in \Delta_n(T)$ to check if the prediction result y' produced by T' is the same as the prediction result y produced by T for the given input x. As shown in Algorithm 1, the input consists of the training set T, the poisoning threshold n, and the test input x. The subroutines KNN_LEARN and KNN_PREDICT implement the *standard* learning and prediction phases of the KNN algorithm. Without the time limit, the baseline method would be both sound and complete; in other words, it would return either *Certified* (Line 13) or *Falsified* (Line 9). With the time limit, however, the baseline method will return *Unknown* (Line 15) after it times out.

The baseline procedure is inefficient for three reasons. First, it is a *slow certification* (Line 13) to check whether the prediction result for x remains the same for all possible clean subsets $T' \in \Delta_n(T)$. In many cases, the elements around x are almost all from one class, and thus x's predicted label cannot be changed by either direct or indirect influence. However, the baseline procedure cannot quickly identify and exploit this to avoid enumeration. Second, even if a violating subset T' exists, the vast majority of subsets in $\Delta_n(T)$ are often non-violating. However, the baseline procedure cannot quickly identify the violating T' from $\Delta_n(T)$. Third, within the while-loop, different subsets share common computations inside KNN_LEARN, but these common computations are not leveraged by the baseline procedure to reduce the computational cost.

Algorithm 2: Our new procedure FALSIFY_NEW(T, n, x).

```
1  if QUICKCERTIFY(T, n, x) then
2  |   return Certified
3  end if
4  ⟨K, Error⟩ ← KNN_LEARN_INIT(T)
5  y ← KNN_PREDICT(T, K, x)
6  ∇_n^x(T) ← GENPROMISINGSUBSETS(T, n, x, y)
7  while ∇_n^x(T) ≠ ∅ ∧ consumed_time < time_limit do
8  |   Remove a subset T' from ∇_n^x(T)
9  |   K' ← KNN_LEARN_UPDATE(T \ T', Error)
10 |   y' ← KNN_PREDICT(T', K', x)
11 |   if y ≠ y' then
12 |   |   return Falsified with (T \ T') as evidence
13 |   end if
14 end while
15 if ∇_n^x(T) = ∅ then
16 |   return Certified
17 else
18 |   return Unknown
19 end if
```

3 OVERVIEW OF THE PROPOSED METHOD

There are three main differences between our method in Algorithm 2 and the baseline method in Algorithm 1. They are marked in dark blue. They are the novel components designed specifically to overcome limitations of the baseline method.

First, we add the subroutine QUICKCERTIFY to quickly check whether it is possible to change the prediction result for the test input x. This is a sound but incomplete check in that, if the subroutine succeeds, we guarantee that the result is robust. If it fails, however, the result remains unknown and we still need to execute the rest of the procedure. The detailed implementation of QUICKCERTIFY is presented in Section 4.

Second, before searching for a clean subset that violates robustness, we compute $\nabla_n^x(T) \subseteq \Delta_n(T)$, to capture the *likely violating* subsets. In other words, the *obviously non-violating* ones in $\Delta_n(T)$ are safely skipped. Note that, while $\Delta_n(T)$ depends only on T and n, $\nabla_n^x(T)$ depends also on the test input x. For this reason, $\nabla_n^x(T)$ is expected to be significantly smaller than $\Delta_n(T)$, thus reducing the search space. The detailed implementation of GENPROMISING-SUBSETS is presented in Section 5.

Third, instead of applying the standard KNN_LEARN subroutine to each subset T' to perform the expensive p-fold cross validation, we split it to KNN_LEARN_INIT and KNN_LEARN_UPDATE, where the first subroutine is applied only once to the original training set T, and the second subroutine is applied to each subset $T' \in \nabla_n^x(T)$. Within KNN_LEARN_UPDATE, instead of performing p-fold cross validation for T' from scratch, we leverage the results returned by KNN_LEARN_INIT to incrementally compute the results for K'. The detailed implementation of these two new subroutines is presented in Section 6.

To summarize, our method first uses over-approximation to certify robustness. If it succeeds, the classification result is guaranteed to be robust; otherwise, the classification result remains unknown. Only for the unknown case, our method uses under-approximation to falsify robustness. If it succeeds, the classification result is guaranteed to be not robust. Otherwise, the classification result remains

Table 1: Notations used in our new algorithm.

Training Set T	Let $T = \{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}$ be a set of labeled data elements, where input $x_i \in \mathcal{X} \subseteq \mathbb{R}^D$ is a feature vector in the feature space $\mathcal{X}$, and $y \in \mathcal{Y} \subseteq \mathbb{N}$ is a class label in the label space $\mathcal{Y}$.
Set of K-nearest Neighbors T_x^K	Let T_x^K be the set of K nearest neighbors of test input x in the training set T.
Label Counter $\mathcal{E}(\cdot)$	Let $\mathcal{E}(D) = \{ (l_i : \#l_i) \}$ be the set of label counts for a dataset D, where $l_i \in \mathbb{Y}$ is a label and $\#l_i \in \mathbb{N}$ is the number of elements in D with label l_i.
Most Frequent Label $Freq(\cdot)$	Let $Freq(\mathcal{E}(D))$ be the most frequent label in the label counter $\mathcal{E}(D)$ for the dataset D.

unknown. Therefore, our method does not "mix" over- and under-approximations in the sense that they are never used simultaneously; instead, over- and under-approximations are used sequentially in two separate steps of our algorithm. The formal guarantee is that: If our method says that a case is robust, it is indeed robust (see Theorem 4.1); if our method says that a case is not robust, it is indeed not robust (since a poisoning set is found); and if our method says unknown, it may be either robust or not robust.

4 QUICKLY CERTIFYING ROBUSTNESS

In this section, we present the subroutine *QuickCertify*, which is a *sound but incomplete* procedure for certifying robustness of the KNN for a given input x. Therefore, if it returns `True`, the prediction result for x is guaranteed to be robust. If it returns `False`, however, we still need further investigation.

We define the notations used by the KNN algorithm in Table 1, following the ones used by Li et al. [31]. Consider $T_x^3 = \{(x_1, l_a), (x_2, l_a), (x_3, l_b)\}$ as an example, which captures the 3-nearest neighbors of a test input x. Then the corresponding label counter is $\mathcal{E}(T_x^3) = \{(l_a : 2), (l_b : 1)\}$, meaning that two elements in T_x^3 have the label l_a and one element has the label l_b. The corresponding most frequent label is $Freq(\mathcal{E}(T_x^3)) = l_a$.

For each subset $T' \in \Delta_n(T)$, we define a removal set $R = (T \setminus T')$ and a removal strategy $\mathcal{S} = \mathcal{E}(R)$.

- A *removal set* R for a set T is a non-empty subset $R \subset T$, to represent the removal of the elements in R from T.
- A *removal strategy* $\mathcal{S}$ is the label counter of a removal set R, i.e., $\mathcal{S} = \mathcal{E}(R)$.

Thus, all the removal sets form the *concrete domain*, and all the removal strategies form an *abstract domain*. While analysis in the (large) concrete domain is expensive, analysis in the (smaller) abstract domain is much cheaper. This is analogous to the *abstract interpretation* [11] paradigm for static program analysis[1].

For the set T_x^3 above, there are 6 removal sets: $R_1 = \{(x_1, l_a)\}$, $R_2 = \{(x_2, l_a)\}$, $R_3 = \{(x_3, l_b)\}$, $R_4 = \{(x_1, l_a), (x_2, l_a)\}$, $R_5 = \{(x_1, l_a), (x_3, l_b)\}$, and $R_6 = \{(x_2, l_a), (x_3, l_c)\}$. They correspond to 4 removal strategies: $\mathcal{S}_1 = \{(l_a : 1)\}$, $\mathcal{S}_2 = \{(l_c : 1)\}$, $\mathcal{S}_3 = \{(l_a : 1), (l_c : 1)\}$, and $\mathcal{S}_4 = \{(l_a : 2)\}$. As the number of elements in T increases, the size gap between the concrete and abstract domains increases drastically— this is the reason why our method is efficient.

[1]There are Galois connections [12] (α, γ) between removal sets and removal strategies (multisets) that are standard in the context of abstract interpretation, where the α function abstracts removal sets in the concrete domain to removal strategies (multisets) in the abstract domain, and the γ function concretizes the multisets back to sets.

Algorithm 3: Subroutine QuickCertify(T, n, x).

```
1  LabelSet ← {}
2  for each candidate K value do
3  |   Let y = Freq(E(T_x^K)) and add y into LabelSet;
4  |   if y ≠ Freq(E(T_x^{K+n}) \ {(y : n)}) then
5  |   |   return False
6  |   end if
7  |   if |LabelSet| > 1 then
8  |   |   return False
9  |   end if
10 end for
11 return True
```

4.1 The QuickCertify Subroutine

In this subroutine, we check a series of *sufficient conditions* under which the prediction result for test input x is guaranteed to be robust. These sufficient conditions are designed to avoid the most expensive step of the KNN algorithm, which is the learning phase that relies on p-fold cross validations to compute the optimal K parameter.

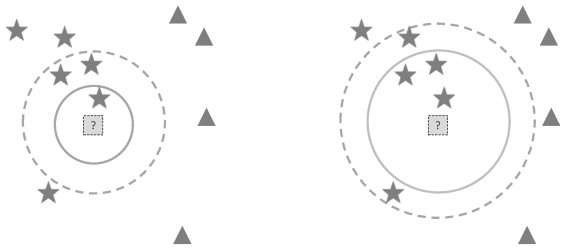
Since the optimal K parameter is chosen from a set of candidate values, where p-fold cross validations are used to identify the value that minimizes prediction error, skipping the learning phase means we must directly analyze the behavior of the KNN prediction phase for all candidate K values. That is, assuming any of the candidate K value may be the optimal one, we prove that the prediction result remains the same no matter which candidate K value is used as the K parameter.

Algorithm 3 shows the procedure, which takes the training set T, poisoning threshold n, and test input x as input, and returns either `True` or `False` as output. Here, `True` means the result is n-poisoning robust, and `False` means the result is unknown. For each candidate K value, $y = Freq(\mathcal{E}(T_x^K))$ is the most frequent label of the K-nearest neighbors of x.

Recall that, in Section 2, we have explained the two ways in which poisoned data in T may affect the prediction result. The first one is called *direct influence*: without changing the K value, the poisoned data may affect the K-nearest neighbors of x and thus their most frequent label. The second one is called *indirect influence*: by changing the K value, the poisoned data may affect how many neighbors to consider. Inside the QuickCertify subroutine, we check for sufficient conditions under which none of the above two types of influence is possible.

The check for *direct influence* is implemented in Line 4. Here, T_x^{K+n} consists of the $(K+n)$ nearest neighbors of x, and $\mathcal{E}(T_x^{K+n})$ is the label counter. Therefore, $\mathcal{E}(T_x^{K+n}) \setminus \{(y : n)\}$ means removing n data elements labeled y. $Freq(\mathcal{E}(T_x^{K+n}) \setminus \{(y : n)\})$ represents the most frequent label after the removal. If it is possible for this removal strategy to change the most frequent label, then we conservatively assume that the prediction result *may not* be robust.

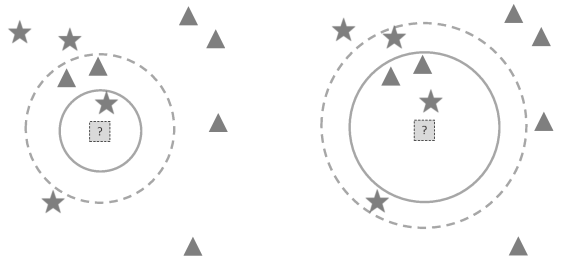
The check for *indirect influence* is implemented in Line 7. Here, *LabelSet* stores all of the most frequent labels for different candidate K values. If the most frequent labels for any two candidate K values



(a) For $K = 1$, $Freq(\mathcal{E}(T_x^1)) = star$, and $Freq(\mathcal{E}(T_x^{1+n}) \setminus \{star : n\}) = star$

(b) For $K = 3$, $Freq(\mathcal{E}(T_x^3)) = star$, and $Freq(\mathcal{E}(T_x^{3+n}) \setminus \{star : n\}) = star$

Figure 3: *Robust* example for QuickCertify, where the poisoning number is $n = 2$, and candidate K values are {1, 3}.



(a) For $K = 1$, $Freq(\mathcal{E}(T_x^1)) = star$, and $Freq(\mathcal{E}(T_x^{1+n}) \setminus \{star : n\}) = triangle$

(b) For $K = 3$, $Freq(\mathcal{E}(T_x^3)) = triangle$, and $Freq(\mathcal{E}(T_x^{3+n}) \setminus \{triangle : n\}) = star$

Figure 4: *Unknown* example for QuickCertify, where the poisoning number is $n = 2$ and the only two candidate values are K=1 and K=3.

differ, i.e., $|LabelSet| > 1$, we conservatively assume the prediction result *may not* be robust.

On the other hand, if the prediction result remains the same during both checks, we can safely assume that the prediction result is n-poisoning robust.

4.2 Two Examples

We illustrate Algorithm 3 using two examples.

Figure 3 shows an example where robustness can be proved by QuickCertify. For simplicity, we assume the only two candidate values for the parameter K are $K = 1$ and $K = 3$. When $K = 1$, as shown in Figure 3 (a), *star* is the most frequent label of the x's neighbors, denoted $\mathcal{E}(T_x^1) = \{(star : 1)\}$, and inside Algorithm 3, we have $LabelSet = \{star\}$. The extreme case is represented by $\mathcal{E}(T_x^{1+2}) \setminus \{(star : 2)\} = \{(star : 1)\}$, which means x is still classified as *star* after applying this aggressive removal strategy.

When $K = 3$, as shown in Figure 3 (b), *star* is also the most frequent label in $\mathcal{E}(T_x^3) = \{star : 3\}$ and thus $LabelSet = \{star\}$. The extreme case is represented by $\mathcal{E}(T_x^{3+2}) \setminus \{star : 2\} = \{star : 3\}$, which means x is still classified as *star* after applying this removal strategy. In this example $n = 2$, thus x is proved to be robust against 2-poisoning attacks.

Figure 4 shows an example where the robustness cannot be proved by QuickCertify. When $K = 1$, as shown in Figure 4 (a), *star* is the most frequent label in $\mathcal{E}(T_x^1) = \{(star : 1)\}$ and $LabelSet = \{star\}$. The extreme case is $\mathcal{E}(T_x^{1+2}) \setminus \{(star : 2)\} = \{triangle : 2\}$, which means x is classified as *triangle*. Thus, QuickCertify returns `False` in Line 5.

4.3 Correctness and Efficiency

The following theorem states that our method is sound in proving n-poisoning robustness.

Theorem 4.1. *If QuickCertify(T, n, x) returns* `True`*, the KNN's prediction result for x is guaranteed to be n-poisoning robust.*

Due to space limit, we omit the full proof. Instead, we explain the intuition behind Line 4 of the algorithm. First, we note that the prediction label $Freq(\mathcal{E}(T_x'^K))$ from any $T' \subset \Delta_n(T)$ can correspond to a $Freq(\mathcal{E}(D))$ where D is obtained by removing i ($\leq n$) elements from T_x^{K+n}. Thus, we only need to pay attention to the $(K + n)$ nearest neighbors of x; other elements which are further away from x can be safely ignored (cf. [24, 31]). Next, to maximize the chance of changing the most frequent label from y to another label, we want to remove as many y-labeled elements as possible from x's neighbors. Thus, the most aggressive removal case is captured by $\mathcal{E}(T_x^{K+n}) \setminus \{(y : n)\}$. If the most frequent label remains unchanged even in this case, it is guaranteed unchanged.

Next, we explain why QuickCertify is fast. There are three reasons. First, it completely avoids the computationally expensive p-fold cross validations. Second, it considers only the $K + n$ nearest neighbors of x. Third, it focuses on analyzing the label counts, which are in the (small) abstract domain, as opposed to the removal sets, which are in the (large) concrete domain.

For these reasons, the execution time of this subroutine is often negligible (e.g., less than 1 second) even for large datasets. At the same time, our experimental evaluation shows that it can prove robustness for a surprisingly large number of test inputs.

To summarize, mapping a potentially large set of concrete sets to their corresponding label multiset (label counts) is an over-approximated abstraction, since the prediction result for a test input x is determined by the label counts of x's nearest neighbors. This over-approximated abstraction allows QuickCertify to efficiently analyze the impact of the maximal allowable change in the label counts.

5 REDUCING THE SEARCH SPACE

In this section, we present the subroutine GenPromisingSubsets, which narrows down the search space by removing *obviously non-violating* subsets from $\Delta_n(T)$ and returns the remaining ones, denoted by the set $\nabla_n^x(T)$ in Algorithm 2.

5.1 Minimal Violating Removal in Neighbors

We filter the *obviously non-violating* subsets by computing some common property for each candidate K value such that it must be part of every *violating* removal set.

We observe that any violating removal set for a specific candidate K value must ensure that, for test input x, its new K nearest neighbors after removal have a most frequent label y' that is different from the default label y. Our method computes the minimal number of removed elements in $x's$ neighborhood to achieve this,

Algorithm 4: GenPromisingSubsets(T, n, x, y).

```
1  for each candidate K value do
2      start = 0; end = n + 1;
3      while start < end do
4          mid = (start + end)/2;
5          if y ≠ Freq(E(T_x^{K+m}) \ {(y : m)}) then
6              end = mid;
7          else
8              start = mid + 1;
9          end if
10     end while
11     min_rmv = start;
12     if min_rmv ≤ n then
13         for each R_1 ⊆ T_x^{K+n} s.t. |R_1| ≥ min_rmv do
14             for each R_2 ⊆ (T \ T_x^{K+n}) and |R_2| ≤ n − |R_1| do
15                 R = R_1 ∪ R_2;
16                 Add (T \ R) to ∇_n^x(T);
17             end for
18         end for
19     end if
20 end for
```

let us call it minimal violating removal, denote *min_rmv*. With this number, we know the every *violating* removal set must have at least *min_rmv* elements from x's neighbors T_x^{K+n}.

The test input x's new nearest neighbors after removal is represented as $T_x^{K+i} \setminus \{i \text{ elements from } T_x^{K+i}\}$, where $i = 1, 2, ...n$. To compute the minimal violating removal, rather than checking each possible value of i from 1 to n, we need a more efficient method, e.g., binary search with $O(log\ n)$. To use binary search, we need to prove the monotonicity of violating removals, defined below.

Theorem 5.1 (Monotonicity). *If there is some i allowing $T_x^{K+i} \setminus \{i$ elements from $T_x^{K+i}\}$ to have a different most-frequent label y', then any larger value $j > i$ will also allow $T_x^{K+j} \setminus \{j$ elements from $T_x^{K+j}\}$ to have a different most-frequent label y'. Conversely, if i does not allow it, then any smaller value $j < i$ does not allow it either.*

Proof. If there is some i allowing $T_x^{K+i} \setminus \{i$ elements from $T_x^{K+i}\}$ to have a different most-frequent label y', there exists $S \subset T_x^{K+i}$ such that $|S| = i$ and $Freq(T_x^{K+i} \setminus S) = y'$. For any $j > i$ and T_x^{K+j}, we can always construct $S' = S \cup (T_x^{K+j} \setminus T_x^{K+i})$, which satisfies $S' \subset T_x^{K+j}$, $|S'| = j$ and $Freq(T_x^{K+j} \setminus S') = y'$. The reverse can be proved similarly. □

Lines 2-11 in Algorithm 4 show the process of finding the minimal violating removal using binary search. Assume the possible range is $0 \sim n+1$ (line 2), the binary search divides the range in half (line 4) and checks the middle value (line 5). To check whether a removal *mid* can result in a different label $y' \neq y$, the most possible operation is to remove *mid* elements with y label. It *mid* works, according to Theorem 5.1, we know the minimal removal is in the range $start \sim mid$ (line 6); otherwise it is in the range $mid+1 \sim end$ (line 8). The binary search stops when *start* equals *end*, and this will the minimal violating removal.

Since n is the maximal allowed removal, when $min\_rmv > n$, it is impossible for the most frequent label to change from y to y'.

5.2 An Illustrative Example

Here we give an example of the binary search in Algorithm 4. Assume in the original training set T, for the test input x, the optimal K is $K = 1$ and the default label is $y = star$.

Example 5.2. Assume $n = 5$, $T_x^3 = \{star * 2, triangle * 1\}$, $T_x^4 = \{star * 2, triangle * 2\}$, and $T_x^5 = \{star * 3, triangle * 2\}$. For the candidate $K = 2$, we show how to compute the minimal violating removal in $x's$ neighbors.

At first, $start = 0$ and $end = 6$, which means the possible value range of minimal removal is $0 \sim 6$. Our method first checks $mid = 3$, since $T_x^{2+3} \setminus \{(star : 3)\}$ results in the most-frequent label *triangle*, our method can cut the possible range by half to $0 \sim 3$. Next, we check $mid = 1$, and reduce the range to $0 \sim 1$. Finally, we check $mid = 0$, which does not work, so the range becomes $1 \sim 1$, and we return 1 as the minimal violating removal in x's neighbors.

Since binary search reduces the range by half at each step, it is efficient. For example, when n=180 for MNIST, binary search needs only 8 checks to compute the result, whereas going through each value in the range requires 180 checks. In other words, the speedup is more than 20X.

5.3 The Reduced Search Space

Based on the minimal violating removal, *min_rmv*, we compute the reduced set $\nabla_n^x(T)$ as shown in Lines 12-20 of Algorithm 4.

Here, each removal set R is the union of two sets, R_1 and R_2, where R_1 is a removal set that contains at least *min_rmv* elements from x's neighborhood T_x^{K+n}, and $R_2 \subseteq (T \setminus T_x^{K+n})$ is a subset of the left-over data elements.

Our experiments show that, in practice, the reduced set $\nabla_n^x(T)$ is often significantly smaller than the original set $\Delta_n(T)$. A special case is when $min\_rmv = 0$, for which $\nabla_n^x(T)$ is the same as $\Delta_n(T)$, meaning the search space is not reduced. However, this special case is rare and, during our experimental evaluation, it never occurred.

6 INCREMENTAL COMPUTATION

In this section, we present our method for speeding up an expensive step of the KNN algorithm, the p-fold cross validations inside KNN_learn. We achieve this speedup by splitting KNN_learn into two subroutines: KNN_learn_init, which is applied only once to the original training set T, and KNN_learn_update, which is applied to each individual removal set $R = (T \setminus T')$, where $T' \in \nabla_n^x(T)$.

6.1 The Intuition

First, we explain why the standard KNN_learn is computationally expensive. This is because, for each candidate value of parameter K, denoted K_i, the standard p-fold cross validation [35] must be used to compute the classification error. Algorithm 5 (excluding Lines 15-16) shows the computation.

First, the training set T is partitioned into p groups, denoted $\{G_1, G_2, ..., G_p\}$. Then, the set of misclassification samples in each group G_j is computed, denoted $errSet_{G_j}^{K_i}$. Next, the error is averaged over all groups, which results in $error^{K_i}$. Finally, the K_i value with the smallest classification error is chosen as the optimal K value.

Algorithm 5: Subroutine KNN_LEARN_INIT(T).

1 Partition the training set T into p groups $\{G_1, G_2, ..., G_p\}$
2 **for** *each candidate* K_i *value* **do**
3 **for** *each group* G_j **do**
4 $errSet^{K_i}_{G_j} \leftarrow \{\}$
5 **for** *each data element* $(x, y) \in G_j$ **do**
6 **if** $KNN\_PREDICT(T \setminus G_j, K_i, x) \neq y$ **then**
7 Add (x, y) to $errSet^{K_i}_{G_j}$;
8 **end if**
9 **end for**
10 $error^{K_i}_{G_j} = \left|errSet^{K_i}_{G_j}\right| \,/\, |G_j|$
11 **end for**
12 $error^{K_i} = \frac{1}{p} \sum_{j=1}^{p} error^{K_i}_{G_j}$
13 **end for**
14 $K \leftarrow \underset{K_i}{\textbf{argmin}}\ error^{K_i}$
15 $Error \leftarrow \langle \{G_1, G_2, ..., G_p\}, \{(errSet^{K_i}_{G_1}, \ldots, errSet^{K_i}_{G_p})\} \rangle$
16 **return** $\langle K, Error \rangle$

Algorithm 6: KNN_LEARN_UPDATE($R, Error$).

1 Let $\{G_1, \ldots, G_p\}$ and $\{(errSet^{K_i}_{G_1}, \ldots, errSet^{K_i}_{G_p})\}$ be groups and error sets stored in $Error$
2 Compute the new groups $\{G'_j \mid G'_j = G_j \setminus R \text{ where } j = 1, \ldots, p\}$
3 Compute the new training set $T' = \bigcup_{j \in \{1,\ldots,p\}} G'_j$
4 Compute the influenced set, $influSet$, using R and $\{G_j\}$
5 **for** *each candidate* K_i *value* **do**
6 **for** *each new group* G'_j **do**
7 $newSet^+ = newSet^- = \{\}$
8 **for** *each data element* $(x, y) \in (G'_j \cap influSet)$ **do**
9 **if** $KNN\_PREDICT(T \setminus G_j, K_i, x) = y$ *and* $KNN\_PREDICT(T' \setminus G'_j, K_i, x) \neq y$ **then**
10 Add (x, y) to $newSet^+$;
11 **end if**
12 **if** $KNN\_PREDICT(T \setminus G_j, K_i, x) \neq y$ *and* $KNN\_PREDICT(T' \setminus G'_j, K_i, x) = y$ **then**
13 Add (x, y) to $newSet^-$;
14 **end if**
15 **end for**
16 $errSet^{K_i}_{G'_j} = errSet^{K_i}_{G_j} \setminus R \setminus newSet^- \cup newSet^+$
17 $error^{K_i}_{G'_j} = \left|errSet^{K_i}_{G'_j}\right| \,/\, |G'_j|$
18 **end for**
19 $error^{K_i} = \frac{1}{p} \sum_{j=1}^{p} error^{K_i}_{G'_j}$
20 **end for**
21 $K \leftarrow \underset{K_i}{\textbf{argmin}}\ error^{K_i}$
22 **return** K

The computation is expensive because $error^{K_i}_{G_j}$, for each K_i, requires exactly $|G_j|$ calls to the standard KNN_PREDICT$(T \setminus G_j, K_i, x)$, one per data element $x \in G_j$, while treating the set $D = (T \setminus G_j)$ as the training set.

Our intuition for speeding up this computation is as follows. Given the original training set T, and a subset $T' \in \nabla^x_n(T)$, the corresponding removal set $R = (T \setminus T')$ can capture the difference between these two sets, and thus capture the difference of their $error^{K_i}$. Since K_i is fixed when computing $error^{K_i}$, we only need to consider the *direct influence* (i.e., neighbors change) brought by removal set R. In practice, the removal set is often small, which means the vast majority of data elements in the p-fold partition of T', denoted $\{G'_1, \ldots, G'_p\}$, are the same as data elements in the p-fold partition of T, denoted $\{G_1, \ldots, G_p\}$. Thus, for most elements, their neighbors are almost the same. Instead of computing the error sets ($errSet^{K_i}_{G'_j}$) from scratch for every single G'_j, we can use the error sets ($errSet^{K_i}_{G_j}$) for G_j as the starting point, and only compute the change brought by removal set R, leveraging the intermediate computation results stored in $Error$.

6.2 The Algorithm

Our incremental computation has two steps. As shown in Algorithm 2, we apply KNN_LEARN_INIT once to the set T, and then apply KNN_LEARN_UPDATE to each removal set $R = (T \setminus T')$.

Our new subroutine KNN_LEARN_INIT is shown in Algorithm 5. It differs from the standard KNN_LEARN only in Lines 15-16, where it stores the intermediate computation results in *Error*. The first component in *Error* is the set of p groups in T. The second component contains, for each K_i, the misclassified elements in G_j.

Subroutine KNN_LEARN_UPDATE is shown in Algorithm 6, which computes the new $errSet^{K_i}_{G'_j}$ based on the $errSet^{K_i}_{G_j}$ stored in *Error*. First, it computes the new groups G'_j by removing elements in R from the old groups G_j. Then, it computes $influSet$, which is defined in the next paragraph. Finally, it modifies the old $errSet^{K_i}_{G_j}$ (in Line 16) based on three cases: it removes the set R (Case 1) and the set $newSet^-$ (Case 2), and adds the set $newSet^+$ (Case 3). Below are the detailed explanations of these three cases:

(1) If $(x, y) \in G_j \setminus G'_j$ was misclassified by $(T \setminus G_j)$, but this element is no longer in T', it should be removed.
(2) If $(x, y) \in G_j \cap G'_j$ was misclassified by $(T \setminus G_j)$, but this element is correctly classified by $T' \setminus G'_j$, it should be removed.
(3) If $(x, y) \in G_j \cap G'_j$ was correctly classified by $(T \setminus G_j)$, but is misclassified by $T' \setminus G'_j$, it should be added.

Case (1) can be regarded as an *explicit change* brought by the removal set R, whereas Case (2) and Case (3) are *implied changes* brought by R: these changes are implied because, while the element (x, y) is not inside R, it is classified differently after the elements in R are removed from T.

Since the removal set is small, most data elements in G_j will not be part of the explicit or implied changes. To avoid redundantly invoking KNN_PREDICT on these data elements, we filter them out using the influenced set (Line 8). Here, assume that $K_{max} = max(\{K_i\})$ is the maximal candidate value, and during cross-validation, when G_j is treated as the test set, $D = (T \setminus G_j)$ is the corresponding training set.

$$influSet = \{\ (x, y) \in G_j \mid \begin{array}{l} (x, y) \notin R, \\ D_x^{K_{max}} \cap R \neq \emptyset, \text{ and} \\ \text{QUICKCERTIFY}(D, n, x) = \texttt{False}\} \end{array}$$

In other words, every element (x, y) inside $influSet$ must satisfy three conditions: (1) the element is not in R; (2) at least one of its neighbors in $D_x^{K_{max}}$ is in R; and (3) the element may be misclassified when at most n neighbors are removed. Recall that the subroutine used in the last condition has been explained in Algorithm 3.

Table 2: Comparing the accuracy of our method with the baseline (ground truth)and two existing methods (which cannot falsify) on the smaller datasets, for which the ground truth can be obtained by the baseline enumerative method (Algorithm 1).

Benchmark		Baseline				Jia et al. [24]				Li et al. [31]				Our Method			
dataset	test data #	certified #	falsified #	**unknown #**	time (s)	certified #	falsified #	**unknown #**	time (s)	certified #	falsified #	**unknown #**	time (s)	certified #	falsified #	**unknown #**	time (s)
Iris (n=1)	15	15	0	**0**	49	0	0	**15**	1	14	0	**1**	1	15	0	**0**	1
Iris (n=2)	15	14	1	**0**	3,086	0	0	**15**	1	13	0	**2**	1	14	1	**0**	5
Iris (n=3)	15	0	1	**14**	6,721	0	0	**15**	1	11	0	**4**	1	13	1	**1**	120
Digits (n=1)	180	0	1	**179**	7,168	170	0	**10**	1	172	0	**8**	1	179	1	**0**	3

7 EXPERIMENTS

We have implemented our method using Python and the popular machine learning toolkit `scikit-learn 0.24.2`, together with the baseline method in Algorithm 1, and the two existing methods of Jia et al. [24] and Li et al. [31]. For experimental comparison, we used six popular supervised learning datasets as benchmarks. There are two relatively small datasets, Iris [17] and Digits [19]. Iris has 135 training and 15 test elements with 3 classes and 4-D features. Digits has 1,617 training and 180 test elements with 10 classes and 64-D features. Since the baseline approach (Algorithm 1) can finish on these small datasets and thus obtain the ground truth (i.e., whether prediction is truly robust), these small datasets are useful in evaluating the accuracy of our method.

The other four benchmarks are larger datasets, including HAR (human activity recognition using smartphones) [3], which has 9,784 training and 515 test elements with 6 classes and 561-D features, Letter (letter recognition) [18], which has 18,999 training and 1,000 test elements with 26 classes and 16-D features, MNIST (hand-written digit recognition) [29], which has 60,000 training and 10,000 test elements with 10 classes and 36-D features, and CIFAR10 (colored image classification) [26], which has 50,000 training and 10,000 test elements with 10 classes and 288-D features. Since none of these datasets can be handled by the baseline approach, they are used primarily to evaluate the efficiency of our method.

7.1 Evaluation Criteria

Our experiments aimed to answer the following three research questions:

- RQ1 Is our method accurate enough for deciding (certifying or falsifying) n-poisoning robustness for most of the test cases?
- RQ2 Is our method efficient enough for handling all of the datasets used in the experiments?
- RQ3 How often can prediction be successfully certified or falsified by our method, and how is the result affected by the poisoning threshold n?

We used the state-of-the-art implementation of KNN in our experiments, with 10-fold cross validation and candidate K values in the range $1 \sim \frac{1}{10}|T|$. The set T is obtained by inserting up-to-n malicious samples to the datasets. We first generate a random number $n' \leq n$, and then insert exactly n' mutations of randomly picked input features and output labels of the original samples.

We ran all four methods on all datasets. For the slow baseline, we set the time limit to 7200 seconds per test input. For the other methods, we set the time limit to 1800 seconds per test input. Our experiments were conducted (single threaded) on a CloudLab [15] c6252-25g node with 16-core AMD 7302P at 3 GHz CPU and 128GB EEC Memory (8 x16 GB 3200MT/s RDIMMs).

Table 3: Comparing the accuracy and efficiency of our method with existing methods on all datasets, with large poisoning thresholds; the percentages of *certified* and *falsified* cases are reported in Section 7.4 and shown in Figure 5.

Benchmark		Jia et al. [24]		Li et al. [31]		Our Method	
dataset	poisoning threshold	unknown %	time (s)	unknown %	time (s)	unknown %	time (s)
Iris	n =3 (2%)	100%	1	26.7%	1	6.7%	120
Digits	n =16 (1%)	100%	1	19.4%	1	1.0%	19
HAR	n =97 (1%)	100%	1	28.3%	1	0.8%	21
Letter	n =190 (1%)	100%	1	94.5%	1	0.0%	4
MNIST	n =180 (0.3%)	38.1%	1	25.0%	1	2.0%	47
CIFAR10	n =150 (0.3%)	90.0%	1	64.0%	1	0.0%	558

7.2 Results on the Smaller Datasets

To answer RQ1, we compared the result of our method with the ground truth obtained by the baseline enumerative method on the two smallest datasets.

Table 2 shows the experimental results. Columns 1-2 show the name of the dataset, the poisoning threshold n, and the number of test data. Columns 3-6 show the result of the baseline method, including the number of test data that are certified, falsified, and unknown, respectively, and the average time per test input. The remaining columns compare the results of the two existing methods and our method. Since the goal is to compare our method with the ground truth (obtained by the baseline method), we must choose small n values to ensure that the baseline method does not time out.

On Iris (n = 2), the baseline method was able to certify 14/15 of the test data and falsify 1/15. However, it was slow: the average time was 3,086 seconds per test input. In contrast, the method by Jia et al. [24] was much faster, albeit with low accuracy. It took 1 second per test input, but failed to certify any of the test data. The method by Li et al. [31] certified 11/15 of the test data but left 4/15 as unknown. Our method certified 14/15 of the test data and falsified the remaining 1/15, and thus is as accurate as the ground truth; the average time is 5 seconds per test input.

While the slow baseline method was able to handle Iris, it did not scale well. With a slightly larger dataset or larger poisoning threshold, it would run out of time. On Digits (n=1), the baseline method falsified only 1/180 of the test data and returned the remaining 179/180 as unknown. In contrast, our method successfully certified or falsified all of the 180 test data.

7.3 Results on All Datasets

To answer RQ2, we compared our method with the two state-of-the-art methods [24, 31] on all datasets, using significantly larger poisoning thresholds. Since these benchmarks are well beyond the reach of the baseline method, we no longer have the ground truth. However, whenever our method returns *Certified* or *Falsified*, the results are guaranteed to be conclusive. Thus, the *Unknown* cases are the only unresolved cases. If the percentage of *Unknown* cases is small, it means our method is accurate.

Table 3 shows the results, where Column 1 shows the name of the dataset, and Column 2 shows the poisoning threshold. For the smallest dataset, we set n to be 2% of the size of T. For medium datasets, we set it to be 1%. For large datasets, we set it to be 0.3%.

Columns 3-6 show the percentage of test data left as *unknown* by the two existing methods and the average time taken. Recall that these methods can only certify, but not falsify, n-poisoning robustness.

Columns 7-8 show the percentage of test data left as *unknown* by our method. While our method has a higher computational cost, it is also drastically more accurate than the two existing methods.

On HAR, for example, the existing methods left 100% and 28.3% of the test data as unknown when $n = 97$. Our method, on the other hand, left only 0.8% of the test data as unknown.

On CIFAR10, which has 50,000 data elements with 288-D feature vectors, our method was able to resolve 100% of the test cases when the poisoning threshold was as large as $n = 150$. In contrast, the two existing methods resolved only 10.0% and 36.0%. In other words, they left 90.0% and 64.0% as unknown.

7.4 Effectiveness of Our Method and Impact of the Poisoning Threshold

To answer RQ3, we studied the percentages of *certified*, *falsified*, and *unknown* cases reported by our method, as well as how they are affected by the poisoning threshold n.

In addition to the percentage of *unknown* cases shown in Table 3, we show the percentages of *certified* and *falsified* cases reported by our method below. There is no need to report these percentages for the two existing methods, because they always have 0% of *falsified* cases.

dataset	poisoning threshold	certified by our method	falsified by our method
Iris	n =3 (2%)	86.6%	6.7%
Digits	n =16 (1%)	80.0%	19.0%
HAR	n =97 (1%)	71.8%	26.8%
Letter	n =190 (1%)	5.6%	94.4%
MNIST	n =180 (0.3%)	75.0%	23.0%
CIFAR10	n =150 (0.3%)	36.0%	64.0%

Figure 5 shows how these percentages are affected by the poisoning threshold. Here, the x-axis shows $n/|T|$ in percentage, and the y-axis shows the percentages of *falsified* in '–', *unknown* in '.' and *certified* in either '|' (quick certify) or '/' (slow certify).

Recall that in Algorithm 2, a test case may be certified in either Line 2 or Line 16. When it is certified in Line 2, it belongs to the '|' region (quick certify) in Figure 5. When it is certified in Line 16, it belongs to the '/' region (slow certify).

For example, in Figure 6(e): When n=1, the falsify percentage is 0%, the unknown percentage is 10% and the quick-certify percentage

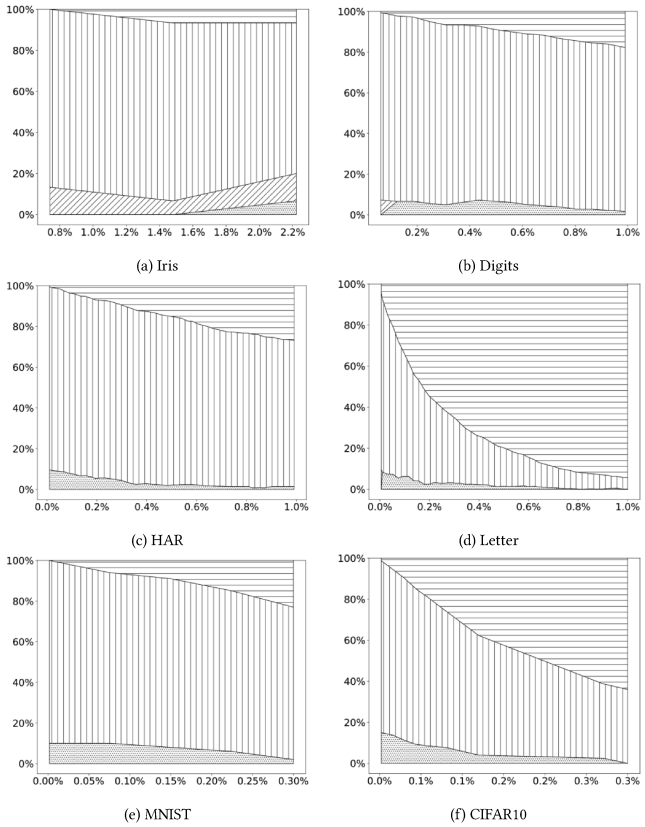


Figure 5: Results on how the poisoning threshold (in the x-axis) affects the percentages of certified, falsified, and unknown test cases (in the y-axis) in our method. Here, *falsified* is in '–', *unknown* is in '.', and *certified* is in either '|' (quick certify) or '/' (slow certify).

is 90%. When n=180, the falsify percentage is 23%, the unknown percentage is 2%, and the quick-certify percentage is 75%.

Figure 5 demonstrates the effectiveness of our method. Since the '.' regions that represent *unknown* cases remain small, the vast majority of cases are successfully certified or falsified.

The results also reflect the nature of n-poisoning robustness: as n increases, the percentage of truly robust cases decreases. This is inevitable since having more poisoned elements in T leads to a higher likelihood of changing the classification label. This is consistent with the results of prior studies [7, 10, 44], which found that the prediction errors became significant even if a small percentage ($< 0.2\%$) of training data in T was poisoned.

8 RELATED WORK

As explained earlier, while there has been prior work on certifying data-poisoning robustness for KNN, none of the existing methods can falsify the robustness property. Thus, our method is the only one that can generate both certification and falsification results with certainty, and can handle both the learning and the prediction phases of a state-of-the-art KNN system. In contrast, existing techniques such as Wang et al. [51], Jia et al. [23, 24], and Weber

et al. [52] can only certify, but not falsify the robustness property. Thus, in the presence of violations, these methods would remain inconclusive. Our method, on the other hand, can successfully resolve the robustness problem for most of the test inputs, as shown by our experimental evaluation.

KNN is not the only machine learning algorithm that is vulnerable to data poisoning. Other machine learning algorithms that are also found to be vulnerable to data poisoning include regression models [36], support vector machines (SVM) [7, 56, 57], clustering algorithms [8], and neural networks [13, 44, 47, 59]. So far, there has been no generic techniques for deciding the robustness property for all machine learning algorithms. Techniques have also been proposed to defend against data-poisoning attacks [4, 6, 16, 22, 45, 50], as well as to evaluate the effectiveness of defense techniques [25, 34] such as data sanitization [25] and differentially-private countermeasures [34]. Along this line, there is a growing interest in studying certified defenses [23, 30, 43] where robustness can be guaranteed either probabilistically or in a deterministic manner.

At a higher level, our method for using over-approximate analysis to narrow down the search space is analogous to static analysis techniques based on abstract interpretation [11], which have been used to verify properties of both software programs [28, 48, 53] and machine learning models [40–42], including robustness to data bias [37] and individual fairness [32]. Furthermore, our method for detecting robustness violations is analogous to techniques used in bug-finding tools based on program verification and state space reduction [5, 27]. However, none of these techniques was designed to certify or falsify data-poisoning robustness of machine learning based systems.

Our method for using systematic testing to find robustness violations is related to the idea of fuzz testing [39, 49] in the sense that mutations are used to generate violation-inducing inputs. There is a large number of fuzz testing tools including AFL [58], honggfuzz [20], libFuzzer [33], SYMFUZZ [9], and Driller [46]. However, these tools focus primarily on search space pruning and search prioritization, e.g., by leveraging the syntax and semantics of the software code, but for KNN, the situation is significantly more complex. This is because mutations of the training data can lead to drastic changes of the behavior of the underlying algorithm, during both the KNN inference phase and the KNN learning phase. Thus, while existing techniques from the fuzz testing literature are inspiring, they are not directly applicable to this problem.

9 CONCLUSION

We have presented a method for deciding n-poisoning robustness accurately and efficiently for the state-of-the-art implementation of the KNN algorithm. To the best of our knowledge, this is the only method available for certifying as well as falsifying the complete KNN system, including both the learning and the prediction phases. Our method relies on novel techniques that first narrow down the search space using over-approximate analysis in the abstract domain, and then find violations using systematic testing in the concrete domain. We have evaluated the proposed techniques on six popular supervised-learning datasets, and demonstrated the advantages of our method over two state-of-the-art techniques. Besides KNN, our method for over-approximating the impact of poisoning on the nearest neighbors is applicable to other distance-based machine learning classifiers and algorithms based on majority voting. Furthermore, since cross validation is a widely used parameter tuning technique in machine learning systems, our method for over-approximating cross validation is also applicable to other systems that rely on cross validation as a subroutine.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was partially funded by the U.S. NSF grants CNS-1702824 and CCF-2220345.

REFERENCES

[1] David Adedayo Adeniyi, Zhaoqiang Wei, and Y Yongquan. 2016. Automated web usage data mining and recommendation system using K-Nearest Neighbor (KNN) classification method. *Applied Computing and Informatics* 12, 1 (2016), 90–108.
[2] Moa Andersson and Lisa Tran. 2020. Predicting movie ratings using KNN. KTH Royal Institute of Technology, Stockholm, Sweden.
[3] Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra, and Jorge Luis Reyes-Ortiz. 2013. A public domain dataset for human activity recognition using smartphones.. In *Esann*, Vol. 3. 3.
[4] Dara Bahri, Heinrich Jiang, and Maya R. Gupta. 2020. Deep k-NN for Noisy Labels. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 540–550.
[5] Dirk Beyer and Thomas Lemberger. 2017. Software Verification: Testing vs. Model Checking - A Comparative Evaluation of the State of the Art. In *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10629)*, Ofer Strichman and Rachel Tzoref-Brill (Eds.). Springer, 99–114.
[6] Battista Biggio, Igino Corona, Giorgio Fumera, Giorgio Giacinto, and Fabio Roli. 2011. Bagging classifiers for fighting poisoning attacks in adversarial classification tasks. In *International workshop on multiple classifier systems*. Springer, 350–359.
[7] Battista Biggio, Blaine Nelson, and Pavel Laskov. 2012. Poisoning Attacks against Support Vector Machines. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*.
[8] Battista Biggio, Konrad Rieck, Davide Ariu, Christian Wressnegger, Igino Corona, Giorgio Giacinto, and Fabio Roli. 2014. Poisoning behavioral malware clustering. In *Proceedings of the 2014 workshop on artificial intelligent and security workshop*. 27–36.
[9] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 725–741.
[10] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. 2017. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526* (2017).
[11] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252.
[12] Patrick Cousot and Radhia Cousot. 2014. A Galois connection calculus for abstract interpretation. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 3–4.
[13] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. 2019. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 321–338.
[14] Samuel Drews, Aws Albarghouthi, and Loris D'Antoni. 2020. Proving data-poisoning robustness in decision trees. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1083–1097.
[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19
[16] Jiashi Feng, Huan Xu, Shie Mannor, and Shuicheng Yan. 2014. Robust logistic regression and classification. *Advances in neural information processing systems* 27 (2014), 253–261.

[17] Ronald A Fisher. 1936. The use of multiple measurements in taxonomic problems. *Annals of eugenics* 7, 2 (1936), 179–188.

[18] Peter W Frey and David J Slate. 1991. Letter recognition using Holland-style adaptive classifiers. *Machine learning* 6, 2 (1991), 161–182.

[19] Geoffrey Gates. 1972. The reduced nearest neighbor rule (corresp.). *IEEE transactions on information theory* 18, 3 (1972), 431–433.

[20] Google. 2016. Honggfuzz. https://google.github.io/honggfuzz/.

[21] Gongde Guo, Hui Wang, David Bell, Yaxin Bi, and Kieran Greer. 2003. KNN model-based approach in classification. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 986–996.

[22] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. 2018. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 19–35.

[23] Jinyuan Jia, Xiaoyu Cao, and Neil Zhenqiang Gong. 2021. Intrinsic Certified Robustness of Bagging against Data Poisoning Attacks. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*. 7961–7969.

[24] Jinyuan Jia, Yupei Liu, Xiaoyu Cao, and Neil Zhenqiang Gong. 2022. Certified Robustness of Nearest Neighbors against Data Poisoning and Backdoor Attacks. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*. AAAI Press, 9575–9583.

[25] Pang Wei Koh, Jacob Steinhardt, and Percy Liang. 2022. Stronger data poisoning attacks break data sanitization defenses. *Mach. Learn.* 111, 1 (2022), 1–47.

[26] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[27] Daniel Kroening and Georg Weissenbacher. 2010. Verification and falsification of programs with loops using predicate abstraction. *Formal Aspects of Computing* 22, 2 (2010), 105–128.

[28] Markus Kusano and Chao Wang. 2016. Flow-sensitive composition of thread-modular abstract interpretation. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 799–809.

[29] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[30] Alexander Levine and Soheil Feizi. 2021. Deep Partition Aggregation: Provable Defenses against General Poisoning Attacks. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

[31] Yannan Li, Jingbo Wang, and Chao Wang. 2022. Proving Robustness of KNN Against Adversarial Data Poisoning. In *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, Alberto Griggio and Neha Rungta (Eds.). IEEE, 7–16.

[32] Yannan Li, Jingbo Wang, and Chao Wang. 2023. Certifying the Fairness of KNN in the Presence of Dataset Bias. In *International Conference on Computer Aided Verification*. Springer.

[33] LLVM. 2021. libFuzzer. https://llvm.org/docs/LibFuzzer.html.

[34] Yuzhe Ma, Xiaojin Zhu, and Justin Hsu. 2019. Data Poisoning against Differentially-Private Learners: Attacks and Defenses. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, Sarit Kraus (Ed.). ijcai.org, 4732–4738.

[35] Geoffrey J McLachlan, Kim-Anh Do, and Christophe Ambroise. 2005. Analyzing microarray gene expression data. (2005).

[36] Shike Mei and Xiaojin Zhu. 2015. Using machine teaching to identify optimal training-set attacks on machine learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

[37] Anna P. Meyer, Aws Albarghouthi, and Loris D'Antoni. 2021. Certifying Robustness to Programmable Data Bias in Decision Trees. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. 26276–26288.

[38] Anna P. Meyer, Aws Albarghouthi, and Loris D'Antoni. 2022. Certifying Data-Bias Robustness in Linear Regression. *CoRR* abs/2206.03575 (2022).

[39] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekandanda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. 1995. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[40] Sara Mohammadinejad, Brandon Paulsen, Jyotirmoy V. Deshmukh, and Chao Wang. 2021. DiffRNN: Differential Verification of Recurrent Neural Networks. In *Formal Modeling and Analysis of Timed Systems - 19th International Conference, FORMATS 2021, Paris, France, August 24-26, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12860)*, Catalin Dima and Mahsa Shirmohammadi (Eds.). Springer, 117–134.

[41] Brandon Paulsen, Jingbo Wang, and Chao Wang. 2020. ReluDiff: differential verification of deep neural networks. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 714–726.

[42] Brandon Paulsen, Jingbo Wang, Jiawei Wang, and Chao Wang. 2020. NEURODIFF: Scalable Differential Verification of Neural Networks using Fine-Grained Approximation. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 784–796.

[43] Elan Rosenfeld, Ezra Winston, Pradeep Ravikumar, and Zico Kolter. 2020. Certified robustness to label-flipping attacks via randomized smoothing. In *International Conference on Machine Learning*. PMLR, 8230–8241.

[44] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. 2018. Poison Frogs! Targeted Clean-Label Poisoning Attacks on Neural Networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 6106–6116.

[45] Jacob Steinhardt, Pang Wei Koh, and Percy Liang. 2017. Certified Defenses for Data Poisoning Attacks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 3517–3529.

[46] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society.

[47] Octavian Suciu, Radu Marginean, Yigitcan Kaya, Hal Daume III, and Tudor Dumitras. 2018. When does machine learning {FAIL}? generalized transferability for evasion and poisoning attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1299–1316.

[48] Chungha Sung, Markus Kusano, and Chao Wang. 2017. Modular verification of interrupt-driven software. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 206–216.

[49] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. 2018. *Fuzzing for software security testing and quality assurance*. Artech House.

[50] Brandon Tran, Jerry Li, and Aleksander Madry. 2018. Spectral Signatures in Backdoor Attacks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 8011–8021.

[51] Yizhen Wang, Somesh Jha, and Kamalika Chaudhuri. 2018. Analyzing the robustness of nearest neighbors to adversarial examples. In *International Conference on Machine Learning*. PMLR, 5133–5142.

[52] Maurice Weber, Xiaojun Xu, Bojan Karlas, Ce Zhang, and Bo Li. 2020. Rab: Provable robustness against backdoor attacks. *arXiv preprint arXiv:2003.08904* (2020).

[53] Meng Wu and Chao Wang. 2019. Abstract interpretation under speculative execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 802–815.

[54] Wenjin Wu, Wen Zhang, Ye Yang, and Qing Wang. 2011. Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking. In *2011 18th Asia-Pacific Software Engineering Conference*. IEEE, 389–396.

[55] Huang Xiao, Battista Biggio, Gavin Brown, Giorgio Fumera, Claudia Eckert, and Fabio Roli. 2015. Is feature selection secure against training data poisoning?. In *International Conference on Machine Learning*. PMLR, 1689–1698.

[56] Huang Xiao, Battista Biggio, Blaine Nelson, Han Xiao, Claudia Eckert, and Fabio Roli. 2015. Support vector machines under adversarial label contamination. *Neurocomputing* 160 (2015), 53–62.

[57] Han Xiao, Huang Xiao, and Claudia Eckert. 2012. Adversarial Label Flips Attack on Support Vector Machines. In *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, Vol. 242. IOS Press, 870–875.

[58] Michal Zalewski. 2017. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[59] Chen Zhu, W Ronny Huang, Hengduo Li, Gavin Taylor, Christoph Studer, and Tom Goldstein. 2019. Transferable clean-label poisoning attacks on deep neural nets. In *International Conference on Machine Learning*. PMLR, 7614–7623.

Received 2023-02-16; accepted 2023-05-03