Proving Data-Poisoning Robustness in Decision Trees

Samuel Drews
University of Wisconsin-Madison
Madison, WI, USA
sedrews@wisc.edu

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

Loris D'Antoni
University of Wisconsin-Madison
Madison, WI, USA
loris@cs.wisc.edu

Abstract

Machine learning models are brittle, and small changes in the training data can result in different predictions. We study the problem of proving that a prediction is robust to *data poisoning*, where an attacker can inject a number of malicious elements into the training set to influence the learned model. We target decision-tree models, a popular and simple class of machine learning models that underlies many complex learning techniques. We present a sound verification technique based on *abstract interpretation* and implement it in a tool called Antidote. Antidote abstractly trains decision trees for an intractably large space of possible poisoned datasets. Due to the soundness of our abstraction, Antidote can produce proofs that, for a given input, the corresponding prediction would not have changed had the training set been tampered with or not. We demonstrate the effectiveness of Antidote on a number of popular datasets.

CCS Concepts: • **Software and its engineering** → *Automated static analysis*; • **Security and privacy** → *Formal methods and theory of security*; • **Computing methodologies** → *Classification and regression trees.*

Keywords: Abstract Interpretation, Adversarial Machine Learning, Decision Trees, Poisoning, Robustness

ACM Reference Format:
Samuel Drews, Aws Albarghouthi, and Loris D'Antoni. 2020. Proving Data-Poisoning Robustness in Decision Trees. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3385412.3385975

1 Introduction

Artificial intelligence, in the form of machine learning (ML), is rapidly transforming the world as we know it. Today, ML is responsible for an ever-growing spectrum of sensitive decisions—from loan decisions, to diagnosing diseases, to autonomous driving. Many recent works have shown how ML models are brittle [3, 7, 29, 30, 33], and with ML spreading across many industries, the issue of robustness in ML models has taken center stage. The research field that deals with studying robustness of ML models is referred to as *adversarial machine learning*. In this field, researchers have proposed many definitions that try to capture robustness to different *adversaries*. The majority of these works have focused on verifying or improving the model's robustness to *test-time attacks* [2, 15, 16, 28, 32], where an adversary can craft small perturbations to input examples that fool the ML model into changing its prediction, e.g., making the model think a picture of a cat is that of a zebra [5].

Data-Poisoning Robustness. This paper focuses on verifying *data-poisoning robustness*, which captures how robust a training algorithm L is to variations in a given *training set* T. Intuitively, applying L to the training set T results in a classifier (model) M, and in this paper we are interested in how the trained model varies when producing perturbations of the input training set T.

The idea is that an adversary can produce slight modifications of the training set, e.g., by supplying a small amount of malicious training points, to influence the produced model and its predictions. This attack model is possible when data is curated, for example, via crowdsourcing or from online repositories, where attackers can try to add malicious elements to the training data. For instance, Xiao et al. [35] consider adding malicious training points to affect a malware detection model; similarly, Chen et al. [7] consider adding a small number of images to bypass a facial recognition model.

A *perturbed set* $\Delta(T)$ defines a set of neighboring datasets the adversary could have attacked to yield the training set T. To define what it means for the training algorithm L to be robust, we need to measure how the model learned by L varies when modifying the training set. Let us say we have an input example x—e.g., a test example—and its classification label is $M(x) = y$, where $M = L(T)$ is the model learned from the training set T. We say that x is robust to poisoning if and only if for all $T' \in \Delta(T)$, we have $L(T')(x) = y$; that is, no matter what dataset $T' \in \Delta(T)$ we use to construct the

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3385975

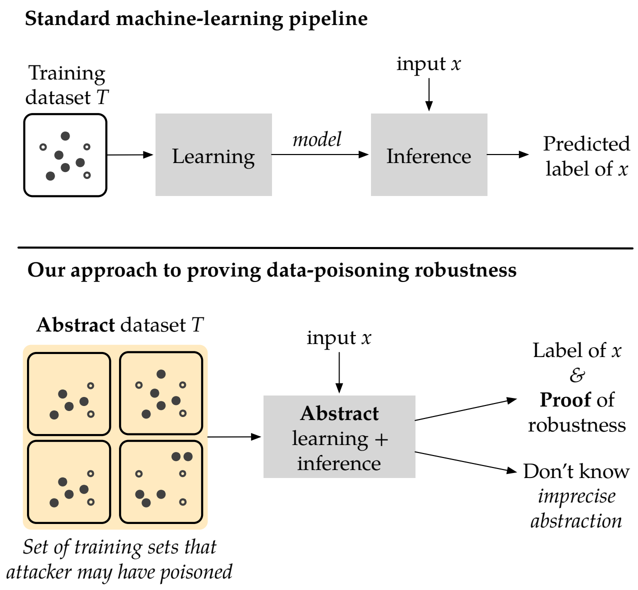

Standard machine-learning pipeline
Training dataset T
Learning
model
input x
Inference
Predicted label of x
Our approach to proving data-poisoning robustness
Abstract dataset T
input x
Abstract learning + inference
Label of x & Proof of robustness
Don't know imprecise abstraction
Set of training sets that attacker may have poisoned

Figure 1. High-level overview of our approach

model $M' = L(T')$, we want M' to always return the same classification y on the input x. To be clear, in this paper we are concerned with a *local* robustness property: we are proving the invariance of individual test points' classifications to changes in the training set.

Verification Challenges. Data-poisoning robustness has been studied extensively [3, 20, 22, 36, 37]. This body of work has demonstrated data-poisoning attacks—i.e., modifications to training sets—that can degrade classifier accuracy, sometimes dramatically, or force certain predictions on specific inputs. While some defenses have been proposed against specific attacks [17, 29], we are not aware of any technique that can formally verify that a given learning algorithm is robust to perturbations to a given training set. Verifying data-poisoning robustness of a given learner requires solving a number of challenges:

1. The datasets over which the learner operates are typically large (thousands of elements). Even when considering simple poisoning attacks, the number of modified training sets we need to consider can be intractably large to represent and explore explicitly.
2. Because learners are complicated programs that employ complex metrics (e.g., entropy and loss functions), their verification requires new specialized techniques.

Our Approach. We focus on the problem of verifying data-poisoning robustness for *decision-tree learners*. We choose decision trees because (*i*) they are widely used interpretable models; (*ii*) they are used in industrial models like random forests and XGBoost [6]; (*iii*) decision-tree-learning has been shown to be unstable to training-set perturbation [14, 19, 23, 31]; and (*iv*) decision-tree-learning algorithms are typically deterministic—e.g., they do not employ stochastic optimization techniques—making them amenable to verification.

We present Antidote, *a tool for verifying data-poisoning robustness of decision-tree learners*. At a high level, Antidote takes as input a training set T and an input x, symbolically constructs every tree built by a particular decision-tree learner L on every possible variation of T in $\Delta(T)$, and applies all those trees to x. If all the trees agree on the label of x, then we know that x is robust to poisoning T. (See Figure 1 for an overview.) Antidote addresses the two challenges highlighted above as follows.

To address the first challenge of training on a combinatorially large number of datasets, Antidote employs a novel *abstract domain* for concisely representing sets of datasets. Antidote is a sound abstract interpretation of standard decision-tree learning algorithms: Instead of constructing a single decision tree, it implicitly constructs an overapproximation of all decision trees for every training set in $\Delta(T)$.

To address the second challenge, Antidote has to soundly approximate how decision-tree learning algorithms propagate entropy computations across a model. Antidote takes advantage of the following observation: every input x only *traverses* a single root-to-leaf trace in the learned decision tree—i.e., the sequence of predicates that affects the decision for x. This observation allows Antidote to build a simpler abstraction that only needs to track the predicates and training elements affecting the decision for x at a given node in the tree, instead of across the entire tree. We call this a *trace-based view* of decision-tree learning, where we are only concerned with the tree trace(s) traversed by x.

Evaluation. We evaluated Antidote on a number of real datasets from the literature. Antidote successfully proves robustness for many test inputs, even in cases where the learner is allowed to build a complex decision tree and the attacker is allowed to contribute more than 1% of the points in the training set. For instance, Antidote can, in around 1 minute, prove robustness for some test inputs of the MNIST-1-7 [3, 29] datasets for cases where the attacker may have contributed up to 192 malicious elements to the dataset. A naïve enumeration approach would have to construct around 10^{432} models to prove the same property!

Contributions. We summarize our contributions as follows:

- Antidote: the first sound technique for verifying data-poisoning robustness for decision-tree learners (§2).
- A *trace-based view* of decision-tree learning as a standalone algorithm that allows us to sidestep the challenging problem of reasoning about the set of all possible output trees a learner can output on different datasets (§3).
- An *abstract domain that concisely encodes sets of perturbed datasets* and the abstract transformers necessary to verify robustness of a decision-tree learner (§4 and §5).

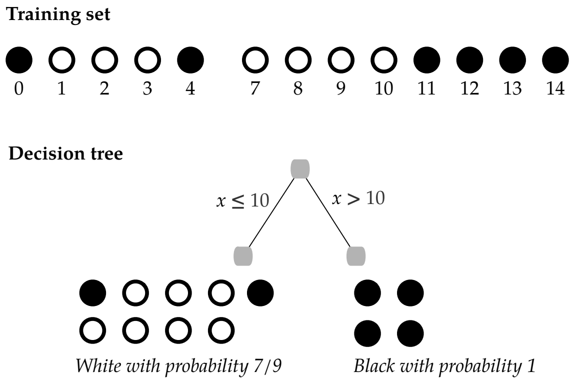


Figure 2. Illustrative example

- An evaluation of Antidote on five representative datasets from the literature. Antidote can prove poisoning robustness for all datasets in cases where an enumeration approach would be doomed to fail (§6).

Proofs of theorems and figures for additional benchmarks are available in the full version of this paper [12].

2 Overview

In this section, we give an overview of decision-tree learning, the poisoning-robustness problem, and motivate our abstraction-based proof technique.

Decision-Tree Learning. Consider the dataset T_{bw} at the top of Figure 2. It is comprised of 13 elements with a single numerical feature. Each element is labeled as a white (empty) or black (solid) circle. We use x to denote the feature value of each element. Our goal is to construct a decision tree that classifies a given number into white or black.

For simplicity, we assume that we can only build trees of depth 1, like the one shown at the bottom Figure 2. At each step of building a decision tree, the learning algorithm is looking for a predicate φ with the best score, with the goal of splitting the dataset into two pieces with *least diversity*, i.e., most elements have the same class (formally defined usually using a notion of entropy). This is what we see in our example: using the predicate $x \leqslant 10$, we split the dataset into two sets, one that is mostly white (left) and one that is completely black (right). This is the best split we can have for our data, assuming we can only pick predicates of the form $x \leqslant c$, for an integer c.[1]

Given a new element for a classification, we check if it is $\leqslant 10$, in which case we say it is white with probability 7/9—i.e., the fraction of white elements such that $\leqslant 10$. Otherwise, if the element is > 10, we say it is black with probability 1.

Data-Poisoning Robustness. Imagine we want to classify an input x but want to make sure the classification would not have changed had the training data been slightly different. For example, maybe some percentage of the data was maliciously added by an attacker to sway the learning algorithm, a problem known as *data poisoning*. Our goal is to check whether the classification of x is robust to data poisoning.

A Naïve Approach. Consider our running example and imagine we want to classify the number 5. Additionally, we want to prove that *removing up to two elements* from the training set would not change the classification of 5—i.e., we assume that up to ~15% (or 2/13) of the dataset is contributed maliciously. The naïve way to do this is to consider every possible training dataset with up to two elements removed and retrain the decision tree. If all trees classify the input 5 as white, the classification is robust to this level of poisoning.

Unfortunately, this approach is intractable. Even for our tiny example, we have to train 92 trees ($\binom{13}{2} + \binom{13}{1} + 1$). For a dataset of 1000 elements and a poisoning of up to 10 elements, we have $\sim 10^{23}$ possibilities.

An Abstract Approach. Our approach to efficiently proving poisoning robustness exploits a number of insights. First, we can perform decision-tree learning *abstractly* on a *symbolic set of training sets*, without having to deal with a combinatorial explosion. The idea is that the operations in decision-tree learning, e.g., selecting a predicate and splitting the dataset, do not need to look at every concrete element of a dataset, but at aggregate statistics (counts).

Recall our running example in Figure 2. Let us say that up to two elements have been removed. No matter what two elements you choose, the predicate $x \leqslant 10$ remains one that gives *a* best split for the dataset. In cases of ties between predicates, our algorithm abstractly represents all possible splits. For each predicate, we can symbolically compute best- and worst-case scores in the presence of poisoning as an *interval*. Similarly, we can also compute an interval that overapproximates the set of possible classification probabilities. For instance, in the left branch of the decision-tree, the probability will be $[0.71, 1]$ instead of 0.78 (or 7/9). The best case probability of 1 is when we drop the black points 0 and 4; the worst-case probability of 0.71 (or 5/7) is when we drop any two white points.

The next insight that enables our approach is that we *do not need to explicitly build the tree*. Since our goal is to prove robustness of a single input point, which effectively takes a single trace through the tree, we mainly need to keep track of the abstract training sets as they propagate along those traces. This insight drastically simplifies our approach; otherwise, we would need to somehow abstractly represent sets of elements of a tree data structure, a non-trivial problem in program analysis.

[1] Note that, while the set of predicates $x \leqslant c$ is infinite, for this dataset (and in general for any dataset), there exists only finitely many inequivalent predicates—e.g., $x \leqslant 4$ and $x \leqslant 5$ split the dataset into the same two sets.

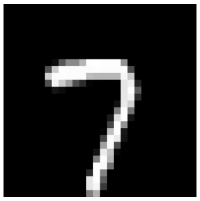


Figure 3. Example MNIST-1-7 digit that is proven poisoning-robust by Antidote.

Abstraction and Imprecision. We note that our approach is sound but necessarily incomplete; that is, when our approach returns "robust" the answer is correct, but there are robust instances for which our approach will not be able to prove robustness. The are numerous sources of imprecision due to overapproximation, for example, we use the *intervals domain* (or disjunctive intervals) to capture real-valued entropy calculations of different training set splits, as well as the final probability of classification.

An Involved Example. To further illustrate our technique, we preview one of our experiments. We applied our approach to the MNIST-1-7 dataset, which has been used to study data-poisoning for deep neural networks [29] and support vector machines [3]. In our experiments, we checked whether Antidote could prove data-poisoning for the inputs used in the same dataset when training a decision tree. For example, when applying Antidote to the image of the digit in Figure 3, Antidote proves that it is poisoning robust (always classified as a seven) for up to 192 poisoned elements in 90 seconds. This is equivalent to training on $\sim 10^{432}$ datasets!

3 Poisoning and Decision Tree Learning

In this section, we begin by formally defining the *data-poisoning-robustness problem.* Then, we present a *trace-based* view of decision-tree learning, which will pave the way for a poisoning-robustness proof technique.

3.1 The Poisoning Robustness Problem

In a typical supervised learning setting, we are given a learning algorithm L and a training set $T \subseteq \mathcal{X} \times \mathcal{Y}$ comprised of elements of some set $\mathcal{X}$, each with its classification label from a finite set of classes $\mathcal{Y}$. Applying L to T results in a classifier (or model): $M : \mathcal{X} \to \mathcal{Y}$. For now, we assume that both the learning algorithm L and the models it learns are deterministic functions.[2]

A *perturbed set* $\Delta(T) \subseteq 2^{\mathcal{X}\times\mathcal{Y}}$ defines a set of possible *neighboring* datasets of T. Our robustness definitions are relative to some given perturbation Δ. (In Section 4.1, we define a specific perturbed set that captures a particular form of data poisoning.)

Definition 3.1 (Poisoning Robustness)**.** Fix a learning algorithm L, a training set T, and let $\Delta(T)$ be a perturbed set. Given an element $x \in \mathcal{X}$, we say that x is robust to poisoning T if and only if

$$\forall T' \in \Delta(T).\ L(T')(x) = L(T)(x)$$

When T and Δ are clear from context, we will simply say that x is robust.

In other words, no matter what dataset $T' \in \Delta(T)$ we use to construct the model $M = L(T')$, we want M to always return the same classification for x. returned for x by the model $L(T)$ learned on the original training set T.

Example 3.2. Imagine we suspect that an attacker has contributed 10 training points to T, but we do not know which ones. We can define $\Delta(T)$ to be T as well as every subset of T of size $|T| - 10$. If an input x is robust for this definition of $\Delta(T)$, then no matter whether the attacker has contributed 10 training items or not, the classification of x does not change.

3.2 Decision Trees: A Trace-Based View

We now formally define decision trees. We will formalize a tree as the *set of traces* from the root to each of the leaves. As we will see, this trace-based view will help enable our proof technique. The idea of representing an already-learned decision tree as a set of traces is not new and has often been explored in the context of extracting interpretable rules from decision trees [24].

A decision tree R is a finite set of traces, where each trace is a tuple (σ, y) such that σ is a sequence of Boolean predicates and $y \in \mathcal{Y}$ is the classification.

Semantically, a tree R is a function in $\mathcal{X} \to \mathcal{Y}$. Given an input $x \in \mathcal{X}$, applying $R(x)$ results in a classification y from the trace $(\sigma, y) \in R$ where x satisfies all the predicates in the sequence $\sigma = [\varphi_1, \ldots, \varphi_n]$, that is, $\bigwedge_{i=1}^{n} x \models \varphi_i$ is true. We say a tree R is *well-formed* if for every $x \in \mathcal{X}$ there exists exactly one trace $(\sigma, y) \in R$ such that x satisfies all predicates in σ. In the following we assume all trees are well-formed.

Example 3.3 (Decision tree traces)**.** Consider the decision-tree in Figure 2. It contains two traces, each with a sequence of predicates containing a single predicate: $([x \leqslant 10], \textit{white})$ and $([x > 10], \textit{black})$.

3.3 Decision-Tree Learning: A Trace-Based View

We now present a simple decision-tree learning algorithm, DTrace. Then, in Section 4, we abstractly interpret DTrace with the goal of proving poisoning robustness.

One of our key insights is that we do not need to explicitly represent the learned trees (i.e., the set of all traces), since our goal is to prove robustness of a *single input* point, which effectively takes a *single trace* through the tree. Therefore, in this section, we will define a *trace-based decision-tree learning algorithm.* This is inspired by standard algorithms—like CART [4], ID3 [25], and C4.5 [26]—but *it is input-directed, in*

[2] Our approach, however, needs to handle non-determinism in decision-tree learning, which arises when breaking ties for choosing predicates with equal scores and choosing labels for classes with equal probabilities.

```
**Input:** training set $T$ and input $x \in \mathcal{X}$
**Initialize:** $\varphi \leftarrow \diamond$, $\sigma \leftarrow$ empty trace
repeat $d$ times
    **if** $\mathsf{ent}(T) = 0$ **then** return
    $\varphi \leftarrow \mathsf{bestSplit}(T)$
    **if** $\varphi = \diamond$ **then** return
    $T \leftarrow \mathsf{filter}(T, \varphi, x)$
    **if** $x \models \varphi$ **then** $\sigma \leftarrow \sigma\varphi$ **else** $\sigma \leftarrow \sigma\neg\varphi$
**Output:** $\mathrm{argmax}_{i \in [1,k]}\, p_i$, where $\mathsf{cprob}(T) = \langle p_1, \ldots, p_k \rangle$
```

Figure 4. Trace-based decision-tree learner DTrace

the sense that it only builds the trace of the tree that a given input x will actually traverse.

A Trace-Based Learner. Our trace-based learner DTrace is shown in Figure 4. It takes a training set T and an input x and computes the trace traversed by x in the tree learned on T. Intuitively, if we compute the set of all traces $\mathsf{DTrace}(T, x)$ for each $x \in T$, we get the full tree, the one that we would have traditionally learned for T.

The learner DTrace repeats two core operations: (*i*) selecting a predicate φ with which to split the dataset (using bestSplit) and (*ii*) removing elements of the training set based on whether they satisfy the predicate φ (depending on x, using filter).[3] The number of times the loop is repeated (d) is the maximum depth of the trace that is constructed. Throughout, we assume a fixed set of classes $\mathcal{Y} = \{1, \ldots, k\}$.

The mutable state of DTrace is the triple (T, φ, σ):

- T is the training set, which will keep getting refined (by dropping elements) as the trace is constructed.
- φ is the most recent predicate along the trace, which is initially undefined (denoted by $\diamond$).
- σ is the sequence of predicates along the trace, which is initially empty.

Predicate Selection. We assume that DTrace is equipped with a finite set of predicates Φ with which it can construct a decision-tree classifier; each predicate in Φ is a Boolean function in $\mathcal{X} \rightarrow \mathbb{B}$.

$\mathsf{bestSplit}(T)$ computes a predicate $\varphi^\star \in \Phi$ that splits the current dataset T—usually minimizing a notion of entropy. Ideally, the learning algorithm would consider every possible sequence of predicates to partition a dataset in order to arrive at an optimal classifier. For efficiency, a decision-tree-learning algorithms does this greedily: it selects the best predicate it can find for a single split and moves on to the next split. To perform this greedy choice, it measures how diverse the two datasets resulting from the split are. We formalize this below:

$$\mathsf{ent}(T) = \sum_{i=1}^{k} p_i(1 - p_i), \quad \text{where } \mathsf{cprob}(T) = \langle p_1, \ldots, p_k \rangle$$

$$\mathsf{cprob}(T) = \left\langle \frac{|\{(x, y) \in T \mid y = i\}|}{|T|} \right\rangle_{i \in [1,k]}$$

Figure 5. Auxiliary operator definitions. ent is Gini impurity; cprob returns a vector of classification probabilities, one element for each class $i \in [1, k]$.

We use $T{\downarrow_\varphi}$ to denote the subset of T that satisfies φ, i.e.,

$$T{\downarrow_\varphi} = \{(x, y) \in T \mid x \models \varphi\}$$

Let Φ' be the set of all predicates that do not trivially split the dataset: $\Phi' = \{\varphi \in \Phi \mid T{\downarrow_\varphi} \neq \emptyset \wedge T{\downarrow_\varphi} \neq T\}$. Finally, $\mathsf{bestSplit}(T)$ is defined as follows:

$$\mathsf{bestSplit}(T) = \underset{\varphi \in \Phi'}{\mathrm{argmin}}\ \mathsf{score}(T, \varphi)$$

where $\mathsf{score}(T, \varphi) = |T{\downarrow_\varphi}| \cdot \mathsf{ent}(T{\downarrow_\varphi}) + |T{\downarrow_{\neg\varphi}}| \cdot \mathsf{ent}(T{\downarrow_{\neg\varphi}})$. Informally, $\mathsf{bestSplit}(T)$ is the predicate that splits T into two sets with the lowest entropy, as defined by the function ent shown in Figure 5. Formally, ent computes *Gini impurity*, which is used, for instance, in the CART algorithm [4]. Note that if $\Phi' = \emptyset$, we assume $\mathsf{bestSplit}(T)$ is undefined (returns $\diamond$). Further, if multiple predicates are possible values of $\mathsf{bestSplit}(T)$, we assume one is returned nondeterministically. Later, in Section 4, our abstract interpretation of DTrace will actually capture all possible predicates in the case of a tie.

Example 3.4. Recall our example from Section 2 and Figure 2. For readability, we use T instead of T_{bw} for the name of the dataset. Let us compute $\mathsf{score}(T, \varphi)$, where φ is $x \leqslant 10$. We have $|T{\downarrow_\varphi}| = 9$ and $|T{\downarrow_{\neg\varphi}}| = 4$. For the classification probabilities, defined by cprob (Figure 5), we have $\mathsf{cprob}(T{\downarrow_\varphi}) = \langle 7/9, 2/9 \rangle$ and $\mathsf{cprob}(T{\downarrow_{\neg\varphi}}) = \langle 0, 1 \rangle$ assuming the first element represents white classification; e.g., in $T{\downarrow_\varphi}$, there's a $7/9$ chance of being classified as white. For ent, we have $\mathsf{ent}(T{\downarrow_\varphi}) \approx 0.35$ and $\mathsf{ent}(T{\downarrow_{\neg\varphi}}) = 0$. Since $T{\downarrow_\varphi}$ is solely composed of black points, its Gini impurity is 0.

The score of $x \leqslant 10$ is therefore ~3.1. For the predicate $x \leqslant 11$, we get the higher (worse) score of ~3.2, as it generates a more diverse split.

Filtering the Dataset. The operator filter removes elements of T that evaluate differently than x on φ. Formally,

$$\mathsf{filter}(T, \varphi, x) = \begin{cases} T{\downarrow_\varphi} & \text{if } x \models \varphi \\ T{\downarrow_{\neg\varphi}} & \text{otherwise} \end{cases}$$

[3] Note that (*ii*) is what distinguishes our trace-based learning from conventional learning of a full tree. (*i*) selects predicates that would comprise the tree, while (*ii*) directs us to recurse *only* along the path that the specific x would take, as opposed to recursing down both (and without affecting how predicates in the tree are selected).

Learner Result. When DTrace terminates in a state $(T_r, \varphi_r, \sigma_r)$, we can read the classification of x as the class i with the highest number of training elements in T_r.

Using cprob, in Figure 5, we compute the probability of each class i for a training set T as a vector of probabilities. Finally, DTrace returns the class with the highest probability:

$$\operatorname*{argmax}_{i \in [1,k]} p_i \qquad \text{where } \mathrm{cprob}(T_r) = \langle p_1, \ldots, p_k \rangle$$

As before, in case of a tie in probabilities, we assume a nondeterministic choice.

Example 3.5. Following the computation from Ex. 3.4, DTrace$(T, 18)$ terminates in state $(T{\downarrow_{x>10}},\ x \leqslant 10,\ [x > 10])$. Point 18 is associated with the trace $[x > 10]$ and is classified as black because $\mathrm{cprob}(T{\downarrow_{x>10}}) = \langle 0, 1 \rangle$.

4 Abstractions of Poisoned Semantics

In this section, we begin by defining a data-poisoning model in which an attacker contributes a number of malicious training items. Then, we demonstrate how to apply the trace-based learner DTrace to *abstract sets of training sets*, allowing us to efficiently prove poisoning-robustness.

4.1 The n-Poisoning Model

For our purposes, we will consider a poisoning model where the attacker has contributed up to n elements of the training set—we call it *n-poisoning*. Formally, given a training set T and a natural number $n \leqslant |T|$, we define the following perturbed set:

$$\Delta_n(T) = \{T' \subseteq T \ : \ |T \setminus T'| \leqslant n\}$$

In other words, $\Delta_n(T)$ captures every training set the attacker could have possibly started from to arrive at T.

This definition of dataset poisoning matches many settings studied in the literature [7, 29, 35]. The idea is that an attacker has contributed a number of malicious data points into the training set to influence the resulting classifier. For example, Chen et al. [7] consider poisoning a facial recognition model to enable bypassing authentication, and Xiao et al. [35] consider poisoning a malware detector to allow the attacker to install malware.

We do not know which n points in T are the malicious ones, or if there are malicious points at all. Thus, the set $\Delta_n(T)$ captures every possible subset of T where we have removed up to n (potentially malicious) elements. Our goal is to prove that our classification is robust to up to n possible poisoned points added by the attacker. So if we try every possible dataset in $\Delta_n(T)$ and they all result in the same classification on x, then x is robust regardless of the attacker's potential contribution.

Observe that $|\Delta_n(T)| = \sum_{i=1}^{n} \binom{|T|}{i}$. So even for relatively small datasets and number n, the set of possibilities is massive, e.g., for MNIST-1-7 dataset (§6), for $n = 50$, we have about 10^{141} possible training sets in $\Delta_n(T)$.

4.2 Abstract Domains for Verifying n-Poisoning

Our goal is to efficiently evaluate DTrace on an input x for all possible training datasets in $\Delta_n(T)$. If all of them yield the same classification y, then we know that x is a robust input. Our insight is that we can abstractly interpret DTrace on a symbolic set of training sets without having to fully expand it into all of its possible concrete instantiations. This allows us to train on an enormous number of datasets, which would be impossible via enumeration.

Recall that the state of DTrace is (T, φ, σ); for our purposes, we do not have to consider the sequence of predicates σ, as we are only interested in the final classification, which is a function of T. In this section, we present the *abstract domains* for each component of the learner's state.

Abstract Training Sets. Abstracting training sets is the main novelty of our technique. We use the abstract element $\langle T', n' \rangle$ to denote a set of training sets and it captures the definition of $\Delta_{n'}(T')$: For every training set T' and number n', the concretization function is $\gamma\left(\langle T', n' \rangle\right) = \Delta_{n'}(T')$. Therefore, we have that initially the *abstraction* function $\alpha(\Delta_n(T)) = \langle T, n \rangle$ is precise. Note that an abstract element $\langle T', n' \rangle$ succinctly captures a large number of concrete sets, $\Delta_{n'}(T')$. Further, all operations we perform on $\langle T', n' \rangle$ will only modify T' and n', without resorting to concretization.

We can define an *efficient* join operation on two elements in the abstract domain[4] as follows:

Definition 4.1 (Joins)**.** Given two training sets T_1, T_2 and $n_1, n_2 \in \mathbb{N}$, $\langle T_1, n_1 \rangle \sqcup \langle T_2, n_2 \rangle := \langle T', n' \rangle$ where $T' = T_1 \cup T_2$ and $n' = \max(|T_1 \setminus T_2| + n_2, |T_2 \setminus T_1| + n_1)$.

Notice that the join of two sets is an overapproximation of the union of the two sets. The following proposition formalizes the soundness of this operation:

Proposition 4.2. *For any T_1, T_2, n_1, n_2, the following holds:*

$$\gamma\left(\langle T_1, n_1 \rangle\right) \cup \gamma\left(\langle T_2, n_2 \rangle\right) \subseteq \gamma(\langle T_1, n_1 \rangle \sqcup \langle T_2, n_2 \rangle).$$

Example 4.3. For any training set T_1, if we consider the abstract sets $\langle T_1, 2 \rangle$ and $\langle T_1, 3 \rangle$, because the second set represents strictly more concrete training sets, we have

$$\langle T_1, 2 \rangle \sqcup \langle T_1, 3 \rangle = \langle T_1, 3 \rangle$$

Now consider the training set $T_2 = \{x_1, x_2\}$. We have

$$\langle T_2, 2 \rangle \sqcup \langle T_2 \cup \{x_3\}, 2 \rangle = \langle T_2 \cup \{x_3\}, 3 \rangle$$

Notice how the join increased the poisoned elements from 2 to 3 to accommodate for the additional element x_3.

[4] Elements in the domain are ordered so that $\langle T_1, n_1 \rangle \sqsubseteq \langle T_2, n_2 \rangle$ if and only if $T_1 \subseteq T_2 \wedge n_2 \leqslant n_1 - |T_1 \setminus T_2|$. In the text, we define the concretization function, a special case of the abstraction function, and the join operation; note that we do not require an explicit meet operation for the purposes of this paper—although one is well-defined:

$$\langle T_1, n_1 \rangle \sqcap \langle T_2, n_2 \rangle := \text{if } |T_1 \setminus T_2| > n_1 \vee |T_2 \setminus T_1| > n_2 \text{ then } \bot$$
$$\text{else } \langle T_1 \cap T_2, \min(n_1 - |T_1 \setminus T_2|, n_2 - |T_2 \setminus T_1|) \rangle$$

Abstract Predicates and Numeric Values. When abstractly interpreting what predicates the learner might choose for different training sets, we will need to abstractly represent sets of possible predicates. Simply, a set of predicates is abstracted *precisely* as the corresponding set of predicates Ψ—i.e., for every set Ψ, we have $\alpha(\Psi) = \Psi$ and $\gamma(\Psi) = \Psi$. Moreover, $\Psi_1 \sqcup \Psi_2 = \Psi_1 \cup \Psi_2$. For certain operations, it will be handy for Ψ to contain a special null predicate $\diamond$.

When abstractly interpreting numerical operations, like cprob and ent, we will need to abstract sets of numerical values. We do so using the standard *intervals* abstract domain (denoted $[l, u]$). For instance, $\alpha(\{0.2, 0.4, 0.6\}) = [0.2, 0.6]$ and $\gamma([0.2, 0.6]) = \{x \mid 0.2 \leqslant x \leqslant 0.6\}$. The join of two intervals is defined as $[l_1, u_1] \sqcup [l_2, u_2] = [\min(l_1, l_2), \max(r_1, r_2)]$. Interval arithmetic follows the standard definitions and we thus elide it here.[5]

4.3 Abstract Learner DTrace$^\#$

We are now ready to define an abstract interpretation of the semantics of our decision-tree learner, denoted DTrace$^\#$.

Abstract Domain. Recall that the state of DTrace is (T, φ, σ); for our purposes, we do not have to consider the sequence of predicates σ, as we are only interested in the final classification, which is a function of T. Using the domains described in Section 4.2, at each point in the learner, our abstract state is a pair $(\langle T', n' \rangle, \Psi')$ (i.e., in the product abstract domain) that tracks the current set of training sets and the current set of possible most recent predicates the algorithm has split on (for all considered training sets).

When verifying n-poisoning for a training set T, the initial abstract state of the learner will be the pair $(\langle T, n \rangle, \{\diamond\})$. In the rest of the section, we define the abstract semantics (i.e., our abstract transformers) for all the operations performed by DTrace$^\#$. For operations that only affect one element of the state, we assume that the other component is left unchanged.

4.4 Abstract Semantics of Auxiliary Operators

We will begin by defining the abstract semantics of the auxiliary operations in the algorithm before proceeding to the core operations, filter and bestSplit. This is because the auxiliary operators are simpler and highlight the nuances of our abstraction.

Let us begin by considering $\langle T, n \rangle{\downarrow}^\#_\varphi$, which is the abstract analog of $T{\downarrow}_\varphi$.

$$\langle T, n \rangle{\downarrow}^\#_\varphi := \langle T{\downarrow}_\varphi, \min(n, |T{\downarrow}_\varphi|) \rangle \tag{1}$$

Simply, it removes elements not satisfying φ from T; since the size of $T{\downarrow}_\varphi$ can go below n, we take the minimum of the two.

Proposition 4.4. *Let $T' \in \gamma(\langle T, n \rangle)$. For any predicate φ, we have $T'{\downarrow}_\varphi \in \gamma(\langle T, n \rangle{\downarrow}^\#_\varphi)$.*

Now consider cprob(T), which returns a vector of probabilities for different classes. Its abstract version returns an interval for each probability, denoting the lower and upper bounds based on the training sets in the abstract set:[6]

$$\mathrm{cprob}^\#(\langle T, n \rangle) := \left\langle \frac{[\max(0, c_i - n), c_i]}{[|T| - n, |T|]} \right\rangle_{i \in [1,k]}$$

where $c_i = |\{(x, i) \in T\}|$. In other words, for each class i, we need to consider the best- and worst-case probability based on removing n elements from the training set, as denoted by the denominator and the numerator. Note that in the corner case where $n = |T|$, we set $\mathrm{cprob}^\#(\langle T, n \rangle) = \langle [0, 1] \rangle_{i \in [1,k]}$.

Proposition 4.5. *Let $T' \in \gamma(\langle T, n \rangle)$. Then,*

$$\mathrm{cprob}(T') \in \gamma\left(\mathrm{cprob}^\#(\langle T, n \rangle)\right)$$

where $\gamma\left(\mathrm{cprob}^\#(\langle T, n \rangle)\right)$ is the set of all possible probability vectors in the vector of intervals.

Example 4.6. Consider the training set on the left side of the tree in Figure 2; call it T_ℓ. It has 7 white elements and 2 black elements. $\mathrm{cprob}(T_\ell) = \langle 7/9, 2/9 \rangle$, where the first element is the white probability. $\mathrm{cprob}^\#(\langle T_\ell, 2 \rangle)$ produces the vector $\langle [5/9, 1], [0, 2/7] \rangle$. Notice the loss of precision in the lower bound of the first element. If we remove two white elements, we should get a probability of 5/7, but the interval domain cannot capture the relation between the numerator and denominator in the definition of cprob$^\#$.

The abstract version of the Gini impurity is identical to the concrete one, except that it performs interval arithmetic:

$$\mathrm{ent}^\#(T) = \sum_{i=1}^{k} \iota_i([1, 1] - \iota_i), \quad \text{where } \mathrm{cprob}^\#(T) = \langle \iota_1, \ldots, \iota_k \rangle$$

Each term ι_i denotes an interval.

[5] While we choose intervals as our numerical abstract domain in this paper, any numerical abstract domain could be used.

[6] Note that this transformer can be more precise: for example, the interval division as written is not guaranteed to be a subset of $[0, 1]$, despite the fact that all concrete values would be. Throughout this section, many of the transformers are simply the "natural" lifting of numerical arithmetic to interval arithmetic; while this may not be optimal, we do so to make it easier to see the correctness of the approach (and to make proofs and implementation straightforward).

In the case of cprob$^\#$, we can compute the optimal transformer inexpensively: it is equivalent to write that cprob(T) computes, for each class $i \in [1, k]$, the *average* of the multiset $S_i = [\text{if } y = i \text{ then } 1 \text{ else } 0 \mid (x, y) \in T]$. We can then have $\mathrm{cprob}^\#(\langle T, n \rangle)$ perform a similar computation for each component: let L_i denote the m-many least elements of S_i, and let U_i denote the m-many greatest elements of S_i, where $m = |T| - n$. These L_i and U_i exhibit extremal behavior of averaging, so we can directly compute the endpoints of the interval assigned to each class as $[\frac{1}{m} \sum_{b \in L_i} b, \frac{1}{m} \sum_{b \in U_i} b]$.

Note that our implementation used for the evaluation (Section 6) *does* employ this optimal transformer for cprob$^\#$, while the other transformers match what is presented.

4.5 Abstract Semantics of filter

We are now ready to define the abstract version of filter. Since we are dealing with abstract training sets, as well as a set of predicates Ψ, we need to consider for each $\varphi \in \Psi$ all cases where $x \models \varphi$ or $x \models \neg\varphi$, and take the join of all the resulting training sets (Definition 4.1). Let

$$\Psi_x = \{\varphi \in \Psi \mid x \models \varphi\} \text{ and } \Psi_{\neg x} = \{\varphi \in \Psi \mid x \models \neg\varphi\}$$

Then,

$$\mathsf{filter}^{\#}(\langle T, n\rangle, \Psi, x) := \left(\bigsqcup_{\varphi \in \Psi_x} \langle T, n\rangle\downarrow^{\#}_{\varphi}\right) \sqcup \left(\bigsqcup_{\varphi \in \Psi_{\neg x}} \langle T, n\rangle\downarrow^{\#}_{\neg\varphi}\right)$$

Proposition 4.7. *Let $T' \in \gamma(\langle T, n\rangle)$ and $\varphi' \in \Psi$. Then,*

$$\mathsf{filter}(T', \varphi', x) \in \gamma\left(\mathsf{filter}^{\#}(\langle T, n\rangle, \Psi, x)\right)$$

Example 4.8. Consider the full dataset T_{bw} from Figure 2. For readability, we write T instead of T_{bw} in the example. Let x denote the input with numerical feature 4, and let $\Psi = \{x \leqslant 10\}$. First, note that because $\Psi_{\neg x}$ is the empty set, the right-hand side of the result of applying the filter$^{\#}$ operator will be the bottom element $\langle\emptyset, 0\rangle$ (i.e., the identity element for $\sqcup$). Then,

$$\begin{aligned}
\mathsf{filter}^{\#}(\langle T, 2\rangle, \Psi, x) &= \langle T, 2\rangle\downarrow^{\#}_{x\leqslant 10} \sqcup \langle\emptyset, 0\rangle && (\text{def. of filter}^{\#}) \\
&= \langle T\downarrow_{x\leqslant 10}, 2\rangle \sqcup \langle\emptyset, 0\rangle && (\text{def. of } \langle T, n\rangle\downarrow^{\#}_{\varphi}) \\
&= \langle T\downarrow_{x\leqslant 10}, 2\rangle && (\text{def. of } \sqcup).
\end{aligned}$$

4.6 Abstract Semantics of bestSplit

We are now ready to define the abstract version of bestSplit. We begin by defining bestSplit$^{\#}$ without handling trivial predicates, then we refine our definition.

Minimal Intervals. Recall that in the concrete case, bestSplit returns a predicate that minimizes the function score(T, φ). To lift bestSplit to the abstract semantics, we define score$^{\#}$, which returns an interval, and what it means to be a *minimal* interval—i.e., the interval corresponding to the abstract minimal value of the objective function score$^{\#}(T, \varphi)$.

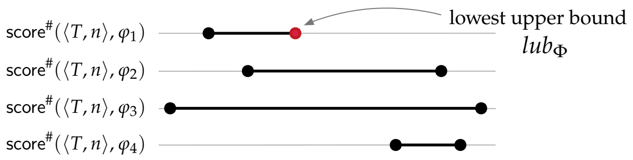
Lifting score(T, φ) to score$^{\#}(\langle T, n\rangle, \varphi)$ can be done using the sound transformers for the intermediary computations:

$$\begin{aligned}
\mathsf{score}^{\#}(\langle T, n\rangle, \varphi) := {}& |\langle T, n\rangle\downarrow^{\#}_{\varphi}| \cdot \mathsf{ent}^{\#}(\langle T, n\rangle\downarrow^{\#}_{\varphi}) \\
&+ |\langle T, n\rangle\downarrow^{\#}_{\neg\varphi}| \cdot \mathsf{ent}^{\#}(\langle T, n\rangle\downarrow^{\#}_{\neg\varphi})
\end{aligned}$$

where $|\langle T, n\rangle| := [|T| - n, |T|]$.

However, given a set of predicates Φ, bestSplit$^{\#}$ must return the ones with the minimal scores. Before providing the formal definition, we illustrate the idea with an example.

Example 4.9. Imagine a set of predicates $\Phi = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$ with the following intervals for score$^{\#}(\langle T, n\rangle, \varphi_i)$.



Notice that φ_1 has the lowest upper bound for score (denoted in red and named lub_Φ). Therefore, we call score$^{\#}(\langle T, n\rangle, \varphi_1)$ the *minimal interval* with respect to Φ. bestSplit$^{\#}$ returns all the predicates whose scores *overlap* with the minimal interval score$^{\#}(\langle T, n\rangle, \varphi_1)$, which in this case are φ_1, φ_2, and φ_3. This is because there is a chance that φ_1, φ_2 and φ_3 are indeed the predicates with the best score, but our abstraction has lost too much precision for us to tell conclusively.

Let lb/ub be functions that return the lower/upper bound of an interval. First, we define the lowest upper bound among the abstract scores of the predicates in Φ as

$$lub_\Phi = \min_{\varphi \in \Phi} \mathsf{ub}(\mathsf{score}^{\#}(\langle T, n\rangle, \varphi))$$

We can now define the set of predicates whose score overlaps with the minimal interval as:

$$\{\varphi \in \Phi \mid \mathsf{lb}(\mathsf{score}^{\#}(\langle T, n\rangle, \varphi)) \leqslant lub_\Phi\}$$

Dealing with Trivial Predicates. Our formulation above considers the full set of predicates, Φ. To be more faithful to the concrete semantics, bestSplit$^{\#}$ needs to eliminate trivial predicates from this set. In the concrete case, we only considered φ as a possible best split if φ performed a non-trivial split on T, which we denoted $\varphi \in \Phi'$. (Recall that a trivial split of T is one that returns $\emptyset$ or T.)

This is a little tricky to lift to our abstract case, since a predicate φ could non-trivially split *some* of the concrete datasets but not others. We lift the set Φ' in two ways:

- *Universal predicates*: the predicates that are non-trivial splits *for all* concrete training sets in $\gamma(\langle T, n\rangle)$[7]

$$\Phi_\forall = \{\varphi \in \Phi \mid \emptyset \notin \gamma(\langle T, n\rangle\downarrow^{\#}_{\varphi}) \wedge \emptyset \notin \gamma(\langle T, n\rangle\downarrow^{\#}_{\neg\varphi})\}$$

- *Existential predicates*: the predicates that are non-trivial splits *for at least one* concrete training set in $\gamma(\langle T, n\rangle)$

$$\Phi_\exists = \{\varphi \in \Phi \mid \langle\emptyset, \cdot\rangle \neq \langle T, n\rangle\downarrow^{\#}_{\varphi} \wedge \langle\emptyset, \cdot\rangle \neq \langle T, n\rangle\downarrow^{\#}_{\neg\varphi}\}$$

Finally, the definition of bestSplit$^{\#}$ considers two cases:

$$\begin{aligned}
\mathsf{bestSplit}^{\#}(\langle T, n\rangle) := {}& \text{if } \Phi_\forall = \emptyset \text{ then } \Phi_\exists \cup \{\diamond\} \text{ else} \\
& \{\varphi \in \Phi_\exists : \mathsf{lb}(\mathsf{score}^{\#}(\langle T, n\rangle, \varphi)) \leqslant lub_{\Phi_\forall}\}
\end{aligned}$$

The first case captures when no single predicate is non-trivial for all sets: we then return all predicates that succeed on at least one training set in $\langle T, n\rangle$, since we cannot be sure one is strictly better than another. To be sound, we also assume the cause of $\Phi_\forall$ being empty is a particular concrete training set for which every predicate forms a trivial split, hence we include $\diamond$ as a possibility. The second case corresponds to returning the predicates with minimal scores.

[7] Note that checking $\emptyset \notin \gamma(\langle T, n\rangle)$ is equivalent to checking $n \neq |T|$.

Lemma 4.10. *Let $T' \in \gamma(\langle T, n\rangle)$. Then,*

$$\mathsf{bestSplit}(T') \in \gamma(\mathsf{bestSplit}^{\#}(\langle T, n\rangle))$$

4.7 Abstracting Conditionals

We abstractly interpret conditionals in DTrace, as is standard, by taking the join of all abstract states from the feasible *then* and *else* paths. In DTrace, there are two branching statements of interest for our purposes, one with the condition $\mathsf{ent}(T) = 0$ and one with $\varphi = \diamond$.

Let us consider the condition $\varphi = \diamond$. Given an abstract state $(\langle T, n\rangle, \Psi)$, we simply set $\Psi = \{\diamond\}$ and propagate the state to the *then* branch (unless, of course, $\diamond \notin \Psi$, in which case we omit this branch). For $\varphi \neq \diamond$, we remove $\diamond$ from Ψ and propagate the resulting state through the *else* branch.

Next, consider the conditional $\mathsf{ent}(T) = 0$. For the *then* branch, we need to *restrict* an abstract state $(\langle T, n\rangle, \Psi)$ to training sets with 0 entropy: intuitively, this occurs when all elements have the same classification. We ask: *are there any concretizations composed of elements of the same class?*, and we proceed through the *then* branch with the following training set abstraction:

$$\bigsqcup_{i \in [1,k]} \mathit{pure}(\langle T, n\rangle, i)$$

where

$$\begin{aligned} \mathit{pure}(\langle T, n\rangle, i) \coloneqq\ & \text{Let } T' = \{(x, y) \in T \mid y = i\} \text{ in} \\ & \text{if } |T \setminus T'| \leqslant n \text{ then } \langle T', n - |T \setminus T'|\rangle \\ & \text{else } \bot \end{aligned}$$

The idea is as follows: the set T' defines a subset of T containing only elements of class i. But if we have to remove more than n elements from T to arrive at T', then the conditional is not realizable by a concrete training set of class i, and so we return the empty abstract state.

In the case of $\mathsf{ent}(T) \neq 0$ (the *else* branch), we soundly (imprecisely) propagate the original state without restriction.

4.8 Soundness of Abstract Learner

Finally, DTrace# soundly overapproximates the results of DTrace and can therefore be used to prove robustness to n-poisoning.

Theorem 4.11. *Let $T' \in \gamma(\langle T, n\rangle)$, let $(T'_f, \cdot, \cdot)$ be the final state of DTrace(T', x), and let $(\langle T''_f, n_f\rangle, \cdot)$ be the final abstract state of DTrace$^{\#}(\langle T, n\rangle, x)$. Then $T'_f \in \gamma(\langle T''_f, n_f\rangle)$.*

It follows from the soundness of DTrace# that we can use it to prove n-poisoning robustness. Let $I = ([l_1, u_2], \ldots, [l_k, u_k])$ be a set of intervals. We say that interval $[l_i, u_i]$ dominates I if and only if $l_i > u_j$ for every $j \neq i$.

Corollary 4.12. *Let $\langle T', n'\rangle$ be the final abstract state of DTrace$^{\#}(\langle T, n\rangle, x)$. If $I = \mathsf{cprob}^{\#}(\langle T', n'\rangle))$ and there exists an interval in I that dominates I (i.e., same class is selected for every $T \in \gamma\langle T, n\rangle$), then x is robust to n-poisoning of T.*

5 Extensions

In this section, we present two extensions that make our abstract interpretation framework more practical. First, we show how our abstract domain can be modified to accommodate real-valued features (§ 5.1). Second, we present a disjunctive abstract domain that is more precise than the one we discussed, but more computationally inefficient (§ 5.2).

5.1 Real-Valued Features

Thus far, we have assumed that DTrace and DTrace# operate on a finite set of predicates Φ. In real-world decision-tree implementations, this is not quite accurate: for real-valued features, there are infinitely many possible predicates of the form $\lambda x_i.\, x_i \leqslant \tau$ (where $\tau \in \mathbb{R}$), and the learner chooses a finite set of possible τ values dynamically, based on the training set T. We will use the subscript $\mathbb{R}$ to denote the real-valued versions of existing operations.

From DTrace to DTrace$_{\mathbb{R}}$. The new learner DTrace$_{\mathbb{R}}$ is almost identical to DTrace. However, each invocation of $\mathsf{bestSplit}_{\mathbb{R}}$ first computes a finite set of predicates $\Phi_{\mathbb{R}}$. Consider all of the values appearing in T for the ith feature in $\mathcal{X}$, sorted in ascending order. For each pair of adjacent values (a, b) (i.e., such that there exists no c in T such that $a < c < b$), we include in $\Phi_{\mathbb{R}}$ the predicate $\varphi = \lambda x_i.\, x_i \leqslant \frac{a+b}{2}$.

Example 5.1. In our running example from Figure 2, we have training set elements in T_{bw} whose features take the numeric values $\{0, 1, 2, 3, 4, 7, \ldots, 14\}$. $\mathsf{bestSplit}_{\mathbb{R}}(T_{bw})$ would pick a predicate from the set $\Phi_{\mathbb{R}} = \{\lambda x.\, x \leqslant \tau \mid \tau \in \{\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \frac{7}{2}, \frac{11}{2}, \frac{15}{2}, \ldots, \frac{27}{2}\}\}$.

From DTrace# to DTrace$^{\#}_{\mathbb{R}}$. To apply the abstract learner in the real-valued setting, we can follow the idea above and construct a finite set $\Phi_{\mathbb{R}}$. Because our poisoning model assumes dropping up to n elements of the training set, this results in roughly $(n+1) \cdot |T|$ predicates in the worst case—i.e., we need to account for every pair (a, b) of adjacent feature values or that are adjacent after removing up to n elements between them.

Example 5.2. Continuing Example 5.1. Say we want to compute $\Phi_{\mathbb{R}}$ for $\langle T_{bw}, 1\rangle$. Then, for every pair of values that are 1 apart we will need to add a predicate to accommodate the possibility that we drop the value between them. E.g., in $T_{bw} = \{\ldots, 3, 4, 7, \ldots\}$, we will additionally need the predicate $\lambda x.\, x \leqslant (3{+}7)/2$, for the case where we drop the element with value 4 from the dataset.

To avoid a potential explosion in the size of the predicate set and maintain efficiency, we compactly represent sets of similar predicates symbolically. We describe this detail in the full version of our paper [12].

5.2 Disjunctive Abstraction

The state abstraction used by DTrace$^\#$ can be *imprecise*, mainly due to the join operations that take place, e.g., during filter$^\#$. The primary concern is that we are forced to perform a very imprecise join between possibly quite dissimilar training set fragments. Consider the following example:

Example 5.3. Let us return to T_{bw} from Figure 2, but imagine we have continued the computation after filtering using $x \leqslant 10$ and have selected some best predicates. Specifically, consider a case in which we have $x = 4$ and

- $\langle T, 1\rangle$, where $T = \{0, 1, 2, 3, 4, 7, 8, 9, 10\}$
- $\Psi = \{x \leqslant 3, x \leqslant 4\}$ (ignoring whether this is correct)

Let us evaluate filter$^\#(\langle T, 1\rangle, \Psi, x)$. Following the definition of filter$^\#$, we will compute

$$\langle T', n'\rangle = \langle T_{\leqslant 4}, 1\rangle \sqcup \langle T_{>3}, 1\rangle$$

where $T_{\leqslant 4} = \{(4, b), (3, w), (2, w), (1, w), (0, b)\}$ and $T_{>3} = \{(4, b), (7, w), (8, w), (9, w), (10, w)\}$, thus giving us $T' = T$ (the set we began with) and $n' = 5$ (much larger than what we began with). This is a large loss in precision.

To address this imprecision, we will consider a *disjunctive* version of our abstract domain, consisting of unboundedly many disjuncts of this previous domain, which we represent as a set $\{(\langle T, n\rangle_i, \Psi_i)\}_i$. Our join operation becomes very simple: it is the union of the two sets of disjuncts.

Definition 5.4 (Joins)**.** Given two disjunctive abstractions $D_I = \{(\langle T, n\rangle_i, \Psi_i)\}_{i\in I}$ and $D_J = \{(\langle T, n\rangle_j, \Psi_j)\}_{j\in J}$, we define

$$D_I \sqcup D_J \coloneqq D_I \cup D_J$$

Adapting DTrace$^\#$ to operate on this domain is immediate: each of the transformers described in the previous section is applied to each disjunct.

Because our disjunctive domain eschews memory- and time-efficiency for precision, we are able to prove more things, but at a cost (we explore this in our evaluation, § 6). Note that, by construction, the disjunctive abstract domain is at least as precise as our standard abstract domain.

6 Implementation and Evaluation

We implemented our algorithms DTrace and DTrace$^\#$ in C++ in a (single-threaded) prototype we call Antidote. Our evaluation[8] aims to answer the following research questions:

RQ1 Can Antidote prove data-poisoning robustness for real-world datasets? (§6.2)

RQ2 How does the performance of Antidote vary with respect to the scale of the problem and the choice of abstract domain? (§6.3)

6.1 Benchmarks and Experimental Setup

We experiment on 5 datasets (Table 1). We obtained the first three datasets from the UCI Machine Learning Repository [13]. Iris is a small dataset that categorizes three related flower species; Mammographic Masses and Wisconsin Diagnostic Breast Cancer are two datasets of differing complexities related to classifying whether tumors are cancerous. We also evaluate on the widely-studied MNIST dataset of handwritten digits [18], which consists of 70,000 grayscale images (60,000 training, 10,000 test) of the digits zero through nine. We consider a form of MNIST that has been used in the poisoning literature and create another variant for evaluation:

- We make the same simplification as in other work on data poisoning [3, 29] and restrict ourselves to the classification of ones versus sevens (13,007 training instances and 2,163 test instances), which we denote MNIST-1-7-Real. Steinhardt et al. [29], for example, recently used this to study poisoning in support vector machines.
- Each MNIST-1-7-Real image's pixels are 8-bit integers (which we treat as real-valued); to create a variant of the problem with reduced scale, we *also* consider MNIST-1-7-Binary, a black-and-white version that uses each pixel's most significant bit (i.e. our predicates are Boolean).

For each dataset, we consider a decision-tree learner with a maximum tree depth (i.e. number of calls to bestSplit) ranging from 1 to 4. Table 1 shows that test set[9] accuracies of the decision trees learned by DTrace are reasonably high—affirmation that when we prove the robustness of its results, we are proving something worthwhile.

Experimental Setup. For each test element, we explore the amount of poisoning (i.e. how large of a n from our Δ_n model) for which we can prove the robustness property as follows.

1. For each combination of dataset T and tree depth d, we begin with a poisoning amount $n = 1$, i.e. a single element could be missing from the training set.
2. For each test set element x, we attempt to prove that x is robust to poisoning T using any set in $\Delta_n(T)$. Let S_n be the test subset for which we do prove robustness for poisoning amount n. If S_n is non-empty, we double n and again attempt to verify the property on elements in S_n.
3. If at a depth n all instances fail, we binary search between n and $n/2$ to find an $n/2 < n' < n$ at which some instances terminate. This approach allows us to better illustrate the experiment trends in our plots.

Failure occurs due to any of three cases: (i) the computed over-approximation does not conclusively prove robustness,

[8] We use a machine with a 2.3GHz processor and 160GB of RAM throughout.

[9] The UCI datasets come as a single training set. We selected a random 80%-20% split of the data, saving the 20% as the test set to use in our experiments. The scale of the MNIST dataset is large; for pragmatic reasons, we fix a random subset of 100 of the original 2,163 test set elements for robustness proving, and we run our DTrace$^\#$ experiments only on this subset.

Table 1. Detailed metrics for the benchmark datasets considered in our evaluation. * Test set accuracy for MNIST is computed on the full 2,163 instances; robustness experiments are performed on 100 randomly chosen test set elements.

Data Set	Size		Features $\mathcal{X}$	Classes $\mathcal{Y}$	DT Test-Set Accuracy (%)			
	Training	Test			Depth 1	2	3	4
Iris	120	30	$\mathbb{R}^4$	{Setosa, Versicolour, Virginica}	20.0	90.0	90.0	90.0
Mammographic Masses	664	166	$\mathbb{R}^5$	{benign, malignant}	80.7	83.1	81.9	80.7
Wisconsin Diagnostic Breast Cancer	456	113	$\mathbb{R}^{30}$	{benign, malignant}	91.2	92.0	92.9	94.7
MNIST-1-7-Binary	13,007	100*	$\{0,1\}^{784}$	{one, seven}	95.7	97.4	97.8	98.3
MNIST-1-7-Real	13,007	100*	$\mathbb{R}^{784}$	{one, seven}	95.6	97.6	98.3	98.7

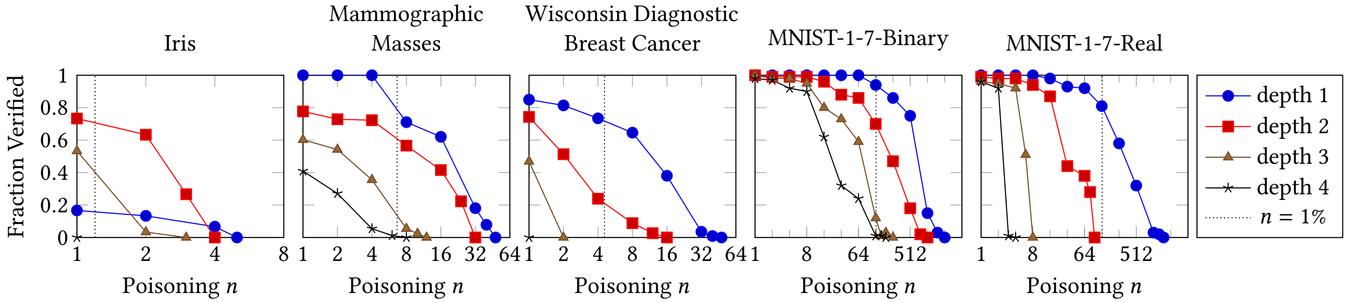


Figure 6. Fraction of test instances proven robust versus poisoning parameter n (log scale). The dotted line is a visual aid, indicating n is 1% of the training set size.

(*ii*) the computation runs out of memory, or (*iii*) the computation exceeds a one-hour timeout. We run the entire procedure for the non-disjunctive and disjunctive abstract domains.

6.2 Effectiveness of Antidote

We evaluate how effective Antidote is at proving data-poisoning robustness. In this experiment, we consider a run of the algorithm on a single test element successful if either the non-disjunctive or disjunctive abstract domain succeeds at proving robustness (mimicking a setting in which two instances of DTrace$^\#$, one for each abstract domain, are run in parallel)—we will contrast the results for the different domains in §6.3. Figure 6 shows these results.

To exemplify the power of Antidote, draw your attention to the depth-2 instance of DTrace$^\#$ invoked on MNIST-1-7-Real. For 38 of the 100 test instances, we are able to verify that even if the training set had been poisoned by an attacker who contributed up to 64 poisoned elements ($\approx \frac{1}{2}$%), the attacker would not have had any power to change the resulting classification. Conventional machine learning wisdom says that, in decision tree learning, small changes to the training set can cause the model to behave quite differently. Our results verify nuance—sometimes, there is some stability.[10] These 38 verified instances average ~800s run time. $\Delta_{64}(T)$ consists of over 10^{174} concrete training sets; This is staggeringly efficient compared to a naïve enumeration baseline, which would be unable to verify robustness at this scale.

To answer **RQ1**, *Antidote can verify robustness for real-world datasets with extremely large perturbed sets and decision-tree learners with high accuracies.*

6.3 Performance of Antidote

We evaluate how the performance of Antidote is affected by the complexity of the problem, e.g., the size of the training set and its number of features, the number of poisoned elements, and the depth of the learned decision tree. Due to the large number of parameters involved in our evaluation, this section only provides a number of representative statistics. In particular, although the reader can find plots describing all the metrics evaluated on each dataset in the full version of our paper [12], most of our analysis will focus on MNIST-1-7-Binary (see Figure 7), since it exhibits the most illustrative behavior.

Box vs Disjuncts. In this section we use Disjuncts to refer to the disjunctive abstract domain and Box to refer to the non-disjunctive one. Disjuncts is more precise than Box and, as expected, it can verify more instances. However, Disjuncts is slower and more memory-intensive. Consider the MNIST-1-7-Binary dataset (see Figure 7). For depth 3 and $n = 64$ (approximately 0.5% of the dataset), Disjuncts can

[10] The Iris dataset has an interesting quirk—we're unable to prove much at depth 1 because in the concrete case, one of the leaves is a 50/50 split between two classes, thus changing one element could make the difference for any of the test set instances taking that path. At depth 2, a predicate is allowed to split that leaf further, making decision-tree learning more stable.

verify 52 instances while Box can only verify 15. However, Disjuncts takes on average 32s to terminate (0 timeouts) and uses 1,650MB of memory, while Box takes on average 0.7s to terminate (0 timeouts) and uses 150MB of memory. It is worth noting that Box can verify certain instances that Disjunct cannot verify due to timeouts. For example, at depth 4 and $n = 128$, Box is able to verify 1 problem instance,[11] while Disjuncts always times out. An interesting direction for future research would be to consider strategies that capitalize on the precision of tracking many disjuncts while incorporating the efficiency of allowing some to be joined.

Number of Poisoned Elements. It is clear from the plots that the number of poisoned elements greatly affects the performance and efficacy of Antidote. We do not focus on particular numbers, since the trends are clear from the plots : The memory consumption and running times of Disjuncts grow exponentially with n, but are still practical and Disjuncts is effective up to high depths. The memory consumption and running times of Box grow more slowly: 95% of all experiments we ran using Box finished within 20 seconds, and none timed out (the longest took 232 seconds).[12] However, Box is less effective than Disjuncts as the depths increase; this is expected, as the loss of precision with more operations is more severe for Box.

Size of Dataset and Number of Features. We measure whether the size of the dataset (which in our benchmarks is quite correlated with the number of features) affects the performance. Consider the case of verifying a decision-tree learner of depth 3 using the disjunctive domain and a perturbed set where 0.5% of the points[13] are removed from the dataset (similar trends are observed when varying these parameters). The average running time of Antidote is 0.1s for Iris, 0.2s for Mammographic Masses, 26s for Wisconsin Diagnostic Breast Cancer, and 32s for MNIST-1-7-Binary. For MNIST-1-7-Real, 100% of the benchmarks TO at 0.05% poisoning. As expected, the size of the dataset and the number of features have an effect on the verification time. However, it is hard to exactly quantify this effect, given how differently each dataset behaves; an obvious comparison we can make is the difference between MNIST-1-7-Binary and MNIST-1-7-Real. These two datasets have identical sizes, but the former uses binary features and the latter uses real features. As we can see, handling real features results in a massive slowdown and in proving fewer instances robust. This is not surprising since real features can result in more predicates, which affect both running time and the discrimination power of individual nodes in the decision tree.

Depth of the Tree. Consider the case of verifying a decision-tree learner for MNIST-1-7-Binary using the disjunctive domain, and a perturbed set where up to 64 of the points have been added maliciously to the dataset (similar trends are observed when varying these parameters and for other datasets). The average running time of Antidote is 0.3s at depth 1, 0.5s at depth 2, 32s at depth 3, and 933s at depth 4. As expected, the depth of the tree is an important factor in the performance of the disjunctive domain, as each abstract operation expands the set of disjuncts.

We summarize the results presented in this section and answer **RQ2**: *in general, the disjunctive domain is more precise but slower than the non-disjunctive domain, and the depth of the learned trees and the number of poisoned elements in the dataset are the greatest factors affecting performance.*

7 Related Work

Instability in Decision Trees. Decision-tree learning has a long and storied history. A particular thread of work that is relevant to ours is the analysis of decision-tree *instability* [14, 19, 23, 31]. These works show that decision-tree learning algorithms are in general susceptible to small data-poisoning attacks—although they do not phrase it in those terms. For the most part, the works are motivated from the perspective that a decision tree represents a set of "rules," and they are concerned with conditions under which those rules will not change (either by quantifying forms of invariance or providing novel learning algorithms). Our work is different in that it *proves* that no poisoning attack exists on a formalization of very basic decision-tree learning, and we can often precisely allow for the "rules" to change so long as the ultimate classification does not.

Data Poisoning. Data-poisoning robustness has been studied extensively from an attacker perspective [3, 20, 22, 36, 37]. This body of work has demonstrated attacks that can degrade classifier accuracy, sometimes dramatically. These works phrase the problem of identifying a poisoned set as a constraint optimization problem. To make the problem tractable, they typically focus on support vector machines (SVMs) and forms of regression for which existing optimization techniques are readily available. Our approach differs from these works in multiple ways: (*i*) Our work focuses on *decision trees*. The greedy, recursive nature of decision-tree learning is fundamentally different from the optimization problem solved in learning SVMs. (*ii*) While our technique is general, in this paper we consider a poisoning model in which training elements have been added [7, 35]. Some works instead focuses on a model in which elements of the training set can be modified [1]. (*iii*) Final and most important, our work *proves* that no poisoning attack exists using abstract interpretation, while existing techniques largely provide search techniques for finding poisoned training sets.

[11] The 14 other instances that succeeded at $n = 64$ similarly terminated after 0.7s on average, but their final state did not prove robustness.

[12] This data must be taken with a grain of salt: Box is generally less effective than Disjuncts; due to the incremental nature of our experiments, it did not attempt as many of the "harder" problems as Disjuncts did.

[13] We round to the closest n for which the tool can verify at least one instance

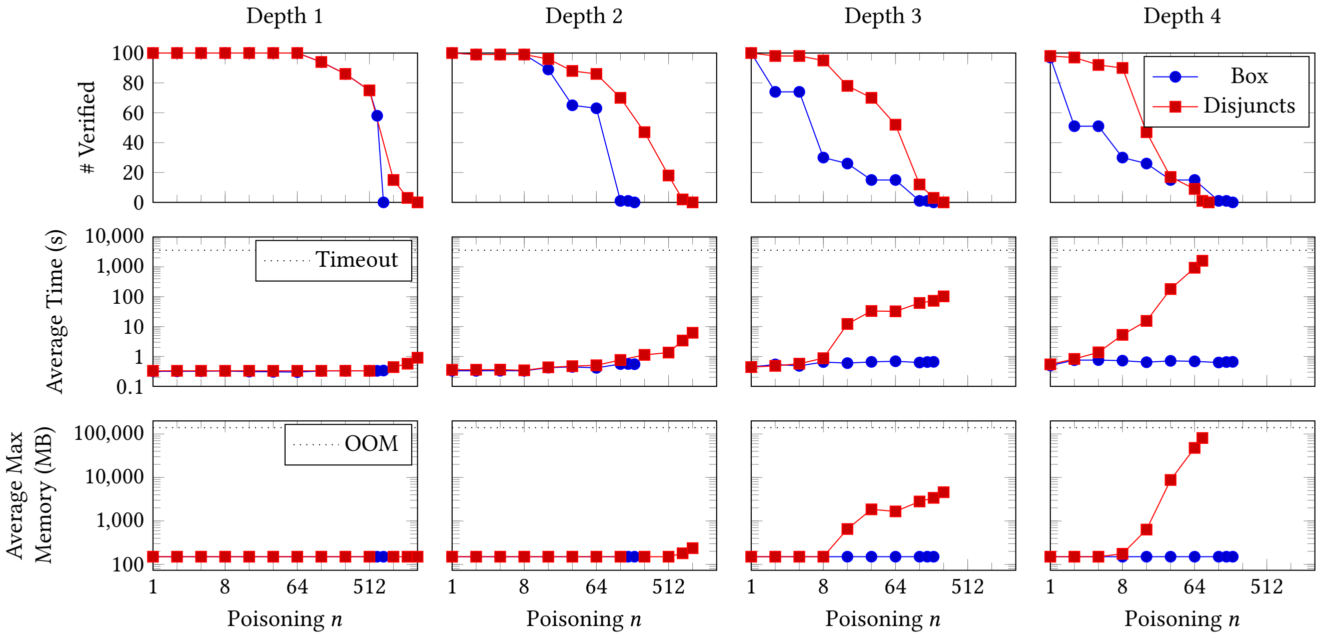


Figure 7. Efficacy, performance, and memory usage for MNIST-1-7-Binary

Recently, techniques have been proposed to modify the training processes of machine learning models to make them robust to various data-poisoning attacks (while remaining computationally efficient). These techniques [9, 10, 17, 29] are often based on robust estimation, e.g. outlier removal; see [11] for a survey. In general, these approaches provide limited probabilistic guarantees about certain kinds of attacks; the works are orthogonal to ours, though they raise an interesting question for future work: Can one verify that, on a given training set, these models actually make the training process resistant to data poisoning?

Abstract Interpretation for Robustness. Abstract interpretation [8] is one of the most popular models for static program analysis. Our work is inspired by that of Gehr et al. [15], where abstract interpretation is used to prove input-robustness for neural networks. (Recently, Ranzato and Zanella have done similar work for decision tree ensembles [27].) Many papers have followed improving on this problem [2, 28]. The main difference between these works and ours is that we tackle the problem of verifying training-time robustness, while existing works focus on test-time robustness. The former problem requires abstracting sets of training sets, while the latter only requires abstracting sets of individual inputs. In particular, Gehr et al. rely on well-known abstract domains—e.g., intervals and zonotopes—to represent sets of real vectors, while our work presents entirely new abstract domains for reasoning about sets of training sets. To our knowledge, our work is the first that even tries to tackle the problem of verifying data-poisoning robustness.

Other works have focused on *provable training* of neural networks to exhibit test-time robustness by construction [21, 34]: this is done by using abstract interpretation to overapproximate the worst-case loss formed by any adversarial perturbation to any element in the training set. One can think of these techniques as performing a form of symbolic training, which is conceptually similar to our core idea. Note, however, two important distinctions: (*i*) These works address the problem of adversarial changes to test inputs, while we address adversarial changes to the training set; (*ii*) These works construct a different, robust model, while we verify a property of an unchanged model (or rather, the learner).

8 Conclusion

We presented Antidote, the first tool that can verify data-poisoning robustness for decision tree learners, where an attacker may have contributed malicious training items. Antidote is based on abstract interpretation and introduces a new abstract domain for representing sets of training sets. We showed that Antidote can verify robustness for real-world datasets in cases where an enumeration approach would be intractable. To our knowledge, this paper is the first to verify data-poisoning robustness for any kind of machine learning model. A natural future direction is to extend our ideas to neural networks, where the learning algorithm is stochastic.

Acknowledgements. We would like to thank Jerry Zhu, Ilias Diakonikolas, and Paris Koutris for their feedback, as well as our shepherd, Martin Vechev. This material is based upon work supported by the National Science Foundation under grant numbers 1652140, 1704117, 1750965, and 1918211.

References

[1] Scott Alfeld, Xiaojin Zhu, and Paul Barford. 2016. Data poisoning attacks against autoregressive models. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[2] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. 2019. Optimization and abstraction: a synergistic approach for analyzing neural network robustness. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 731–744.

[3] Battista Biggio, Blaine Nelson, and Pavel Laskov. 2012. Poisoning Attacks Against Support Vector Machines. In *Proceedings of the 29th International Coference on International Conference on Machine Learning* (Edinburgh, Scotland) *(ICML'12)*. Omnipress, USA, 1467–1474. http://dl.acm.org/citation.cfm?id=3042573.3042761

[4] Leo Breiman. 2017. *Classification and regression trees*. Routledge.

[5] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 39–57.

[6] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 785–794.

[7] 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).

[8] P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.

[9] Ilias. Diakonikolas, Gautam. Kamath, Daniel. Kane, Jerry. Li, Ankur. Moitra, and Alistair. Stewart. 2019. Robust Estimators in High-Dimensions Without the Computational Intractability. *SIAM J. Comput.* 48, 2 (2019), 742–864. https://doi.org/10.1137/17M1126680 arXiv:https://doi.org/10.1137/17M1126680

[10] Ilias Diakonikolas, Gautam Kamath, Daniel Kane, Jerry Li, Jacob Steinhardt, and Alistair Stewart. 2019. Sever: A Robust Meta-Algorithm for Stochastic Optimization. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 1596–1606. http://proceedings.mlr.press/v97/diakonikolas19a.html

[11] Ilias Diakonikolas and Daniel M. Kane. 2019. Recent Advances in Algorithmic High-Dimensional Robust Statistics. *CoRR* abs/1911.05911 (2019). arXiv:1911.05911 http://arxiv.org/abs/1911.05911

[12] Samuel Drews, Aws Albarghouthi, and Loris D'Antoni. 2019. Proving Data-Poisoning Robustness in Decision Trees. *CoRR* abs/1912.00981 (2019). arXiv:1912.00981 http://arxiv.org/abs/1912.00981

[13] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[14] Kenneth Dwyer and Robert Holte. 2007. Decision Tree Instability and Active Learning. In *Machine Learning: ECML 2007, 18th European Conference on Machine Learning, Warsaw, Poland, September 17-21, 2007, Proceedings*. 128–139. https://doi.org/10.1007/978-3-540-74958-5_15

[15] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 3–18. https://doi.org/10.1109/SP.2018.00058

[16] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*. Springer, 97–117.

[17] Ricky Laishram and Vir Virander Phoha. 2016. Curie: A method for protecting SVM Classifier from Poisoning Attack. *CoRR* abs/1606.01584 (2016). arXiv:1606.01584 http://arxiv.org/abs/1606.01584

[18] Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. [n.d.]. The MNIST Database of handwritten digits. http://yann.lecun.com/exdb/mnist

[19] Ruey-Hsia Li and Geneva G. Belford. 2002. Instability of decision tree classification algorithms. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada*. 570–575. https://doi.org/10.1145/775047.775131

[20] Shike Mei and Xiaojin Zhu. 2015. Using Machine Teaching to Identify Optimal Training-set Attacks on Machine Learners. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (Austin, Texas) *(AAAI'15)*. AAAI Press, 2871–2877. http://dl.acm.org/citation.cfm?id=2886521.2886721

[21] Matthew Mirman, Timon Gehr, and Martin T. Vechev. 2018. Differentiable Abstract Interpretation for Provably Robust Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research)*, Jennifer G. Dy and Andreas Krause (Eds.), Vol. 80. PMLR, 3575–3583. http://proceedings.mlr.press/v80/mirman18b.html

[22] Andrew Newell, Rahul Potharaju, Luojie Xiang, and Cristina Nita-Rotaru. 2014. On the Practicality of Integrity Attacks on Document-Level Sentiment Analysis. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop* (Scottsdale, Arizona, USA) *(AISec '14)*. ACM, New York, NY, USA, 83–93. https://doi.org/10.1145/2666652.2666661

[23] Jesús M. Pérez, Javier Muguerza, Olatz Arbelaitz, Ibai Gurrutxaga, and José Ignacio Martín. 2005. Consolidated Trees: Classifiers with Stable Explanation. A Model to Achieve the Desired Stability in Explanation. In *Pattern Recognition and Data Mining, Third International Conference on Advances in Pattern Recognition, ICAPR 2005, Bath, UK, August 22-25, 2005, Proceedings, Part I*. 99–107. https://doi.org/10.1007/11551188_11

[24] J.R. Quinlan. 1987. Simplifying decision trees. *International Journal of Man-Machine Studies* 27, 3 (1987), 221 – 234. https://doi.org/10.1016/S0020-7373(87)80053-6

[25] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.

[26] J Ross Quinlan. 1993. C4.5: Programs for machine learning. *The Morgan Kaufmann Series in Machine Learning, San Mateo, CA: Morgan Kaufmann,| c1993* (1993).

[27] Francesco Ranzato and Marco Zanella. 2020. Abstract Interpretation of Decision Tree Ensemble Classifiers. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI'20)*, V. Conitzer and F. Sha (Eds.).

[28] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. 2019. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 41.

[29] Jacob Steinhardt, Pang Wei W Koh, and Percy S Liang. 2017. Certified defenses for data poisoning attacks. In *Advances in neural information processing systems*. 3517–3529.

[30] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. 2014. Intriguing properties of neural networks. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*.

[31] Peter D. Turney. 1995. Technical Note: Bias and the Quantification of Stability. *Machine Learning* 20, 1-2 (1995), 23–33. https://doi.org/10.1007/BF00993473

[32] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal security analysis of neural networks using symbolic intervals. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1599–1614.

[33] Yizhen Wang, Somesh Jha, and Kamalika Chaudhuri. 2018. Analyzing the Robustness of Nearest Neighbors to Adversarial Examples. In

Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. 5120–5129.

[34] Eric Wong and J. Zico Kolter. 2018. Provable Defenses against Adversarial Examples via the Convex Outer Adversarial Polytope. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research)*, Jennifer G. Dy and Andreas Krause (Eds.), Vol. 80. PMLR, 5283–5292. http://proceedings.mlr.press/v80/wong18a.html

[35] 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*. 1689–1698.

[36] Huang Xiao, Battista Biggio, Blaine Nelson, Han Xiao, Claudia Eckert, and Fabio Roli. 2015. Support Vector Machines Under Adversarial Label Contamination. *Neurocomput.* 160, C (July 2015), 53–62. https://doi.org/10.1016/j.neucom.2014.08.081

[37] Han Xiao, Huang Xiao, and Claudia Eckert. 2012. Adversarial Label Flips Attack on Support Vector Machines. In *Proceedings of the 20th European Conference on Artificial Intelligence* (Montpellier, France) *(ECAI'12)*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 870–875. https://doi.org/10.3233/978-1-61499-098-7-870