Certifying Decision Trees Against Evasion Attacks by Program Analysis

Stefano Calzavara, Pietro Ferrara, and Claudio Lucchese

Università Ca' Foscari Venezia

Abstract. Machine learning has proved invaluable for a range of different tasks, yet it also proved vulnerable to evasion attacks, i.e., maliciously crafted perturbations of input data designed to force mispredictions. In this paper we propose a novel technique to verify the security of decision tree models against evasion attacks with respect to an expressive threat model, where the attacker can be represented by an arbitrary imperative program. Our approach exploits the interpretability property of decision trees to transform them into imperative programs, which are amenable for traditional program analysis techniques. By leveraging the abstract interpretation framework, we are able to soundly verify the security guarantees of decision tree models trained over publicly available datasets. Our experiments show that our technique is both precise and efficient, yielding only a minimal number of false positives and scaling up to cases which are intractable for a competitor approach.

Keywords: Adversarial machine learning · Decision trees · Security of machine learning · Program analysis.

1 Introduction

Machine learning (ML) learns predictive models from data and has proved invaluable for a range of different tasks, yet it also proved vulnerable to *evasion attacks*, i.e., maliciously crafted perturbations of input data designed to force mispredictions [25]. To exemplify, let us assume a credit company decides to use a ML model to automatically assess whether customers qualify for a loan or not. A malicious customer who somehow realises or guesses that the model privileges unmarried people over married people could cheat about her marital status to improperly qualify for a loan.

The research community recently put a lot of effort in the investigation of *adversarial* ML, e.g., techniques to train models which are resilient to attacks or assess the security properties of models. In the present paper we are interested in the security certification of a popular class of models called *decision trees*, i.e., we investigate formally sound techniques to quantify the resilience of such models against evasion attacks. Specifically, we propose the first *provably sound* certification technique for decision trees with respect to an expressive threat model, where the attacker can be represented by an arbitrary imperative program. Verifying ML techniques with respect to highly expressive threat models is nowadays one of the most compelling research directions of adversarial ML [16, 12]. This is an important step forward over previous work, which either proposed empirical techniques without formal guarantees or only focused on artificial attackers expressed as mathematical distances (see Section 6 for full details).

Our approach exploits the *interpretability* property of decision trees, i.e., their amenability to be easily understood by human experts, which makes their translation into imperative programs a straightforward task. Once a decision tree is translated into an imperative program, it is possible to leverage state-of-the-art program analysis techniques to certify its resilience to evasion attacks. In particular we leverage the *abstract interpretation* framework [9, 10] to automatically extract a sound abstraction of the behaviour of the decision tree under attack. This allows us to efficiently compute an over-approximated, yet precise, estimate of the resilience of the decision tree against evasion attacks.

Contributions. We specifically contribute as follows:

1. We propose a general technique to certify the security guarantees of decision trees against evasion attacks attempted by an attacker expressed as an arbitrary imperative program. We exemplify the technique at work on an expressive threat model based on rewriting rules (Section 3).
2. We implement our technique into a new tool called TreeCert. Given a decision tree, an attacker and a test set of instances used to estimate prediction errors, TreeCert outputs an over-approximation of the error rate that the attacker can force on the decision tree. TreeCert implements a *context-insensitive* analysis computing a single over-approximation of the attacker's behavior and reuses it in the analysis of all the test instances, thus boosting efficiency without missing attacks (Section 4).
3. We experimentally prove the effectiveness of TreeCert against publicly available datasets. Our results show that TreeCert is extremely precise, since it can compute tight over-approximations of the actual error rate under attack, with a difference of at most 0.02 over it on cases which are small enough to be analyzed without approximated techniques. Moreover, TreeCert is much faster than a competitor approach [5] and scales to intractable cases, avoiding the exponential blow-up of non-approximated techniques (Section 5).

2 Background

2.1 Security of Supervised Learning

In this paper, we deal with the security of *supervised learning*, i.e., the task of learning a classifier from a set of labeled data. Formally, let $\mathcal{X} \subseteq \mathbb{R}^d$ be a d-dimensional space of real-valued features and $\mathcal{Y}$ be a finite set of class labels; a *classifier* is a function $f : \mathcal{X} \to \mathcal{Y}$ which assigns a class label to each element of the vector space (also called *instance*). The correct label assignment for each instance is modeled by an unknown function $g : \mathcal{X} \to \mathcal{Y}$, called *target* function.

Given a *training set* of labeled data $\mathcal{D}_{train} = \{(\mathbf{x}_1, g(\mathbf{x}_1)), \ldots, (\mathbf{x}_n, g(\mathbf{x}_n))\}$ and a *hypothesis space* $\mathcal{H}$, the goal of supervised learning is finding the classifier $\hat{h} \in \mathcal{H}$ which best approximates the target function g. Specifically, we let $\hat{h} = \text{argmin}_{h \in \mathcal{H}} \mathcal{L}(h, \mathcal{D}_{train})$, where $\mathcal{L}$ is a *loss* function which estimates the cost of the prediction errors made by h on $\mathcal{D}_{train}$. Once $\hat{h}$ is found, its performance is assessed by computing $\mathcal{L}(\hat{h}, \mathcal{D}_{test})$, where $\mathcal{D}_{test}$ is a *test set* of labeled, held-out data drawn from the same distribution of $\mathcal{D}_{train}$.

Within the context of security certification, one should measure the accuracy of $\hat{h}$ by taking into account all the actions that an attacker could take to fool the classifier into mispredicting, i.e., the so-called *evasion attacks* [1, 2]. To provide a more accurate evaluation of the performance of the classifier under attack, the loss $\mathcal{L}$ can thus be replaced by the *loss under attack* $\mathcal{L}^A$ [21]. Formally, the attacker can be modeled as a function $A : \mathcal{X} \to 2^{\mathcal{X}}$ mapping each instance into a set of *perturbed* instances which might fool the classifier. The test set $\mathcal{D}_{test}$ can thus be corrupted into any dataset obtained by replacing each $(\mathbf{x}_i, y_i) \in \mathcal{D}_{test}$ with any $(\mathbf{x}'_i, y_i)$ such that $\mathbf{x}'_i \in A(\mathbf{x}_i)$; we let $A(\mathcal{D}_{test})$ stand for the set of all such datasets. The loss under attack $\mathcal{L}^A$ is thus defined by making the pessimistic assumption that the attacker is able to craft the most damaging perturbations, as follows:

$$\mathcal{L}^A(\hat{h}, \mathcal{D}_{test}) = \max_{\mathcal{D}' \in A(\mathcal{D}_{test})} \mathcal{L}(\hat{h}, \mathcal{D}').$$

Unfortunately, computing $\mathcal{L}^A$ by enumerating $A(\mathcal{D}_{test})$ is intractable, given the huge number of perturbations available to the attacker: for example, if the attacker can flip K binary features, then each instance can be perturbed in 2^K different ways, leading to $2^K \cdot |\mathcal{D}_{test}|$ possible attacks.

2.2 Decision Trees

A powerful set of hypotheses $\mathcal{H}$ is the set of the *decision trees* [4]. We focus on traditional binary decision trees, whose internal nodes perform thresholding over feature values. Such trees can be inductively defined as follows: a decision tree t is either a leaf $\lambda(\hat{y})$ for some label $\hat{y} \in \mathcal{Y}$ or a non-leaf node $\sigma(f, v, t_l, t_r)$, where $f \in [1, d]$ identifies a feature, $v \in \mathbb{R}$ is the threshold for the feature f and t_l, t_r are decision trees. At test time, an instance $\mathbf{x} = (x_1, \ldots, x_d)$ traverses the tree t until it reaches a leaf $\lambda(\hat{y})$, which returns the *prediction* $\hat{y}$, denoted by $t(\mathbf{x}) = \hat{y}$. Specifically, for each traversed tree node $\sigma(f, v, t_l, t_r)$, $\mathbf{x}$ falls into the left tree t_l if $x_f \leq v$, and into the right tree t_r otherwise.

Figure 1 represents an example decision tree, which assigns the instance (6,8) with label -1 to its correct class. In fact, (i) the first node checks whether the second feature (whose value is 8) is less than or equal to 10 and then takes the left sub-tree, and (ii) the second node checks whether the first feature (whose value is 6) is less than or equal to 5 and then takes the right leaf, classifying the instance with label -1. However, note that an attacker who was able to corrupt (6,8) into (5,8) could force the decision tree into changing its output, leading to the prediction of the wrong class $+1$.

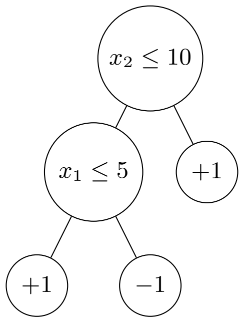

Fig. 1. Example of decision tree.

2.3 Abstract Interpretation

In the abstract interpretation framework, the behavior of a program is approximated through *abstract values* of a given *abstract domain* with a lattice structure, rather than concrete values. For example, the Sign domain abstracts numbers just with their sign, as formalized by the following *abstraction* and *concretization* functions (α and γ respectively):

$$\alpha(V) = \begin{cases} \bot & \text{if } V = \emptyset \\ + & \text{if } \forall v \in V : v > 0 \\ 0 & \text{if } \forall v \in V : v = 0 \\ - & \text{if } \forall v \in V : v < 0 \\ \top & \text{otherwise} \end{cases} \qquad \gamma(a) = \begin{cases} \mathbb{R} & \text{if } a = \top \\ \{n \in \mathbb{R} \mid n > 0\} & \text{if } a = + \\ \{0\} & \text{if } a = 0 \\ \{n \in \mathbb{R} \mid n < 0\} & \text{if } a = - \\ \emptyset & \text{if } a = \bot \end{cases}$$

Notice that for all sets of concrete values $V \subseteq \mathbb{R}$ we have $V \subseteq \gamma(\alpha(V))$, i.e., the abstraction function provides an over-approximation of the concrete values. Operations over concrete values like the sum operation $+$ are over-approximated by abstract counterparts $\oplus$ over the abstract domain, which define the *abstract semantics*. For example, the sum of two positive numbers is certainly positive, while the sum of a positive number and a negative number can be positive, negative or 0; this lack of information is modeled by $\top$. Hence, $\oplus$ is defined such that $+ \oplus + = +$ and $+ \oplus - = \top$. A sound definition of $\oplus$, here omitted, must ensure that $\forall V_1, V_2 \subseteq \mathbb{R} : \{v_1 + v_2 \mid v_1 \in V_1 \wedge v_2 \in V_2\} \subseteq \gamma(\alpha(V_1) \oplus \alpha(V_2))$, i.e., abstract operations must over-approximate operations over concrete values. By simulating the program over the abstract domain, abstract interpretation ensures a fast convergence to an over-approximation of all the reachable program states. In particular, the analysis consists in computing the fixpoint of the abstract semantics over the abstract domain, making use of a *widening* operator – usually if the upper bound operator does not converge within a given threshold [9, 10].

Thanks to its modular approach, abstract interpretation allows one to define multiple abstractions of the same concrete domain. Therefore, several abstract domains approximating numerical values have been proposed in the literature.

For instance Octagons [22] and Polyhedra [11] track different types of (linear) relations among numerical variables, and have been fruitfully applied to different contexts. Apron [18] is a library of numerical abstract domains comprising the main domains leveraged in this work.

3 Security Verification of Decision Trees

3.1 Threat Model

Our approach is general enough to be applied to attackers represented as arbitrary imperative programs. To exemplify it, we show how it can be applied to an expressive threat model based on *rewriting rules* [5]. This relatively new threat model goes beyond traditional distance-based models, which are plausible for perceptual tasks like image recognition, but are inappropriate for non-perceptual tasks (e.g., loan assignment) where mathematical distances do not capture useful semantic properties of the domain of interest.

We model the attacker A as a pair (R, K), where R is a set of *rewriting rules*, defining how instances can be corrupted, and $K \in \mathbb{R}^+$ is a *budget*, limiting the amount of alteration the attacker can apply to each instance. Each rule $r \in R$ has form:

$$[a, b] \xrightarrow{f}_k [\delta_l, \delta_u],$$

where: (i) $[a, b]$ and $[\delta_l, \delta_u]$ are intervals on $\mathbb{R} \cup \{-\infty, +\infty\}$, with the former defining the *precondition* for the application of the rule and the latter defining the *magnitude* of the perturbation enabled by the rule; (ii) $f \in [1, d]$ is the index of the feature to perturb; and (iii) $k \in \mathbb{R}^+$ is the *cost* of the rule. The semantics of the rewriting rule can be explained as follows: if an instance $\mathbf{x} = (x_1, \ldots, x_d)$ satisfies the condition $x_f \in [a, b]$, then the attacker can corrupt it by adding any $v \in [\delta_l, \delta_u]$ to x_f and spending k from the available budget. The attacker can corrupt each instance by using as many rewriting rules as desired in any order, possibly multiple times, up to budget exhaustion.

According to this attacker model, we can define $A(\mathbf{x})$, the set of the attacks against the instance $\mathbf{x}$, as follows.

Definition 1 (Attacks). *Given an instance* $\mathbf{x}$ *and an attacker* $A = (R, K)$*, we let* $A(\mathbf{x})$ *be the set of the* attacks *that can be obtained from* $\mathbf{x}$*, i.e., the set of the instances* $\mathbf{x}'$ *such that there exists a sequence of rewriting rules* $r_1, \ldots, r_n \in R$ *and a sequence of instances* $\mathbf{x}_0, \ldots, \mathbf{x}_n$ *where:*

1. $\mathbf{x}_0 = \mathbf{x}$ *and* $\mathbf{x}_n = \mathbf{x}'$*;*
2. *for all* $i \in [1, n]$*, the instance* $\mathbf{x}_{i-1}$ *can be corrupted into the instance* $\mathbf{x}_i$ *by using the rewriting rule* r_i*, as described above;*
3. *the sum of the costs of* $r_1, \ldots, r_n$ *is not greater than* K*.*

Notice that $\mathbf{x} \in A(\mathbf{x})$ *for any* A *by picking an empty sequence of rewriting rules, i.e., the attacker can always leave the instance uncorrupted.*

Example 1. Consider the attacker $A = (\{r_1, r_2\}, 10)$, where:

```
int predict (float[] x) {
    if (x[2] <= 10) {
        if (x[1] <= 5)
            return +1;
        else
            return -1;
    }
    else
        return +1;
}
```

Fig. 2. Translation of the decision tree in Figure 1 into an imperative program.

- $r_1 = [0, 10] \xrightarrow{1}_5 [-1, 0]$ allows the attacker to corrupt the first feature by adding any value in $[-1, 0]$, provided that the feature value is in $[0, 10]$ and the available budget is at least 5;
- $r_2 = [5, 10] \xrightarrow{2}_4 [0, 1]$ allows the attacker to corrupt the second feature by adding any value in $[0, 1]$, provided that the feature value is in $[5, 10]$ and the available budget is at least 4.

The attacker A can force the decision tree in Figure 1 to change its original prediction (-1) on the instance $(6, 8)$. In particular, we can show that $(5, 8)$ is a possible attack against $(6, 8)$, since A can apply r_1 once by spending 5 from the budget, and $(5, 8)$ is classified as $+1$ by the decision tree.

3.2 Conversion to Imperative Program

Our analysis technique exploits the *interpretability* property of decision trees, i.e., their amenability to be easily understood by human experts. In particular, it is straightforward to convert any decision tree into an equivalent, loop-free imperative program. To exemplify, Figure 2 shows the translation of the decision tree in Figure 1 into an equivalent function.

We can then model the attacker as an imperative program which has access to the function representing the decision tree to analyse. In particular, we observe that the attacker $A = (R, K)$ can be represented by means of a *non-deterministic* program which behaves as follows:

1. Select a random rewriting rule $r \in R$.
2. Let $[a, b] \xrightarrow{f}_k [\delta_l, \delta_u]$ be the selected rule r and let $\mathbf{x} = (x_1, \ldots, x_d)$ be the instance to perturb. If $x_f \in [a, b]$ and the available budget is at least k, then select a random $\delta \in [\delta_l, \delta_u]$, replace x_f with $x_f + \delta$ and subtract k from the available budget.
3. Non-deterministically go to step 1 or terminate the process. This stop condition allows the attacker to spare part of the budget, which is needed to enforce termination when the entire budget cannot be spent (or does not need to be spent).

```
 1  float[] attack (float[] x) {
 2      float K = 10;
 3      boolean done = false;
 4      while (!done) {
 5          int rule = random_int(1,3);
 6          switch (rule) {
 7              case 1:
 8                  if (x[1] >= 0 && x[1] <= 10 && K >= 5) {
 9                      float delta = random_float(-1,0);
10                      x[1] = x[1] + delta;
11                      K = K - 5;
12                  }
13                  break;
14              case 2:
15                  if (x[2] >= 5 && x[2] <= 10 && K >= 4) {
16                      float delta = random_float(0,1);
17                      x[2] = x[2] + delta;
18                      K = K - 4;
19                  }
20                  break;
21              case 3:
22                  // this models non-deterministic termination
23                  done = true;
24          }
25      }
26      return x;
27  }
28
29  int predict_under_attack (float[] x) {
30      float[] x' = attack(x);
31      return predict(x');
32  }
```

Fig. 3. Encoding predictions under attack into an imperative program.

This encoding is exemplified in Figure 3, where lines 1-27 show how the attacker of Example 1 can be modeled as an imperative program, using standard functions for random number generation. Once the attacker has been modeled, we can finally encode the behavior of the decision tree under attack: this is shown in lines 29-32, where we let the attacker corrupt the input instance before it is fed to the decision tree for prediction.

3.3 Proving Security by Program Analysis

Given a decision tree t, an attacker A and a test set $\mathcal{D}_{test}$, we can compute an over-approximation of $\mathcal{L}^A(t, \mathcal{D}_{test})$ as follows.

We first translate the decision tree t together with the attacker A into an imperative program P modeling the decision tree under attack, as discussed in Section 3.2. For each instance $(\mathbf{x}_i, y_i) \in \mathcal{D}_{test}$, we build an abstract state $\alpha(\{\mathbf{x}_i\})$ representing $\mathbf{x}_i$ in the chosen abstract domain and we analyze P with such entry state. Then, the output of the analysis might be either of the following:

1. only leaves of the decision tree with the correct class label y_i are reachable. This means that, for all possible attacks against $\mathbf{x}_i$, the decision tree always classifies the instance correctly;
2. leaves with the wrong label are reachable as well. If t correctly classifies the instance in the unattacked setting, this might happen either because there is indeed an attack leading to a misprediction or for a loss of precision due to the over-approximation performed by the static analysis.

Since our approach relies on sound static analysis engines, it is not possible to miss attacks, i.e., every instance which can be mispredicted upon attack must fall in the second case of our analysis. Let $P^{\#}(\mathbf{x}_i) = Y_i$ stand for the set of labels Y_i returned by the analysis of P on the instance $\mathbf{x}_i$.

By using this information, we can construct an *abstraction* of the behaviour of t under attack on $\mathcal{D}_{test}$ defined as follows:

$$\forall (\mathbf{x}_i, y_i) \in \mathcal{D}_{test} : t^{\#}(\mathbf{x}_i) = \begin{cases} y_i & \text{if } P^{\#}(\mathbf{x}_i) = \{y_i\} \\ y \neq y_i & \text{otherwise} \end{cases}$$

By construction, we have that $\mathcal{L}^A(t, \mathcal{D}_{test}) \leq \mathcal{L}(t^{\#}, \mathcal{D}_{test})$ for any loss function which depends just on the number of mispredictions, like the *error rate*, i.e., the fraction of wrong predictions among all the performed predictions. This means that after building $t^{\#}$ we have an efficient way to over-approximate the loss under attack $\mathcal{L}^A$ by computing just a traditional loss $\mathcal{L}$, which does not require the computation of the set of attacks.

3.4 Extensions

We discuss here possible extensions of our approach to different popular settings. We leave the implementation of these extensions to future work, since they are essentially an engineering effort.

Regression. The *regression* task requires one to learn a regressor rather than a classifier from the training data. The key difference between a regressor and a classifier is that the former does not assign a class from a finite set $\mathcal{Y}$, but rather infers a numerical quantity from an unbound set, e.g., estimates the salary of an employee based on her features. Regression can be modeled by revising the abstraction $t^{\#}$ such that it returns an abstract value over-approximating all the values of the predictions found in the leaves which are reachable upon attack. Formally, this means requiring $t^{\#}(\mathbf{x}_i) = \sqcup_{y_i \in P^{\#}(\mathbf{x}_i)} \alpha(\{y_i\})$, where $\sqcup$ stands for the upper bound operator on the abstract domain.

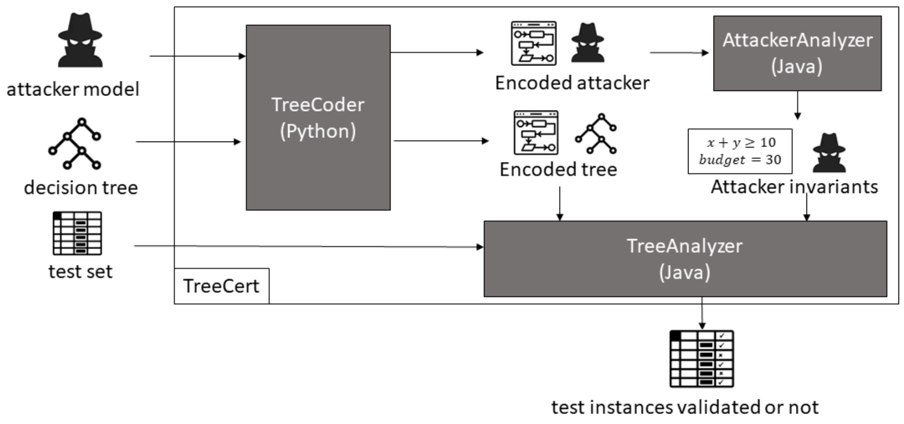

Fig. 4. The architecture of TreeCert.

Tree Ensembles. Ensemble methods train multiple decision trees and combine them to improve prediction accuracy. Traditional ensemble approaches include random forest [3] and gradient boosting [14]. Irrespective of how an ensemble is trained, its final predictions are performed just by aggregating the predictions of the individual trees, e.g., using majority voting or averaging. This means that it is possible to readily generalize our analysis technique to ensembles by translating each tree therein and by aggregating their predictions in the generated imperative program.

4 Implementation

Figure 4 depicts the architecture of TreeCert. The inputs are: (i) the attacker, expressed in the threat model of Section 3.1 using a JSON file, (ii) a decision tree to analyse, serialized through the `joblib` library, and (iii) a test set in CSV format. TreeCert reports for each test instance whether it is correctly classified for each possible attack or it might be wrongly classified. The analysis is performed along three different modules, called TreeCoder, AttackerAnalyzer and TreeAnalyzer respectively, which we detail in the following.

4.1 TreeCoder

The first step of TreeCert is to encode the attacker and the decision tree as Java programs through the module TreeCoder, as described in Section 3.2. TreeCoder is a Python script that, given an attacker model and a decision tree, produces two distinct Java files encoding the attacker (see method `attack` in Figure 3) and the decision tree (see method `predict` in Figure 2).

There are only two small technical differences over the previous presentation. First, given that all instances of the same dataset share the same set of features, instances are not encoded as arrays, but rather modeled using a distinct local variable for each feature, which simplifies the static analysis; specifically, we let variable $\mathsf{x_i}$ represent the initial value of the i-th feature and variable $\mathsf{x'_i}$ represent its value after the attack. In addition, each time a rewriting rule r is applied, we increment a counter r_counter, initially set to 0, which allows one to capture useful analysis invariants. Clearly, these changes do not affect the semantics of the generated program, so we did not include them in Figure 3 for simplicity.

4.2 AttackerAnalyzer

The encoded attacker is then passed to the AttackerAnalyzer module, a static analyzer based on abstract interpretation. The analyzer interfaces with Apron, a standard library implementing many popular abstract domains. The analyzer then computes a fixpoint over the Java program representing the attacker, using the Polka implementation[1] of the Polyhedra domain [11].

Polka tracks linear equalities and inequalities over an arbitrary number of variables. These invariants allow AttackerAnalyzer to infer the upper and lower bounds of each attacked feature, based on how many times a feature can be attacked using the available budget. To exemplify, pick the attacker in Figure 3. AttackerAnalyzer infers on such program that, after the attack has been performed: (i) the value of the first feature may have been decreased by at most r1_counter (formally, $\mathsf{x'_1} \in [\mathsf{x_1} - 1 * \mathsf{r1\_counter}, \mathsf{x_1}]$), (ii) the second feature may have been increased by at most r2_counter ($\mathsf{x'_2} \in [\mathsf{x_2}, \mathsf{x_2} + 1 * \mathsf{r2\_counter}]$), (iii) both the counters are non-negative ($\mathsf{r1\_counter} \geq 0 \wedge \mathsf{r2\_counter} \geq 0$), and (iv) the budget spent in the application of the two rewriting rules is less than or equal to the initial budget ($5 * \mathsf{r1\_counter} + 4 * \mathsf{r2\_counter} \leq 10$). Note that the last invariant is inferred only if the calculation of a fixpoint over the abstract semantics did not require to apply the Polyhedra widening operator to convergence. Otherwise, the analysis would drop such information to ensure termination.

4.3 TreeAnalyzer

The attacker invariants are then passed to the TreeAnalyzer module together with the test set. Like AttackerAnalyzer, TreeAnalyzer performs a static analysis using the Polka implementation of the Polyhedra abstract domain. For each test instance $\mathbf{x}$, TreeAnalyzer (i) adds the initial values of the features of $\mathbf{x}$ to the attacker invariants, (ii) computes the fixpoint over the program encoding the decision tree t under attack, and (iii) uses it to return the output $t^{\#}(\mathbf{x})$.

To clarify, consider again Example 1, where the test instance $(6, 8)$ is correctly classified as -1 by the decision tree in Figure 1, but can be misclassified upon attack. First of all, TreeAnalyzer adds the invariants $\mathsf{x_1} = 6$ and $\mathsf{x_2} = 8$ to the inferred attacker invariants, leading to an initial Polyhedra state tracking that $\mathsf{x'_1} \in [6 - \mathsf{r1\_counter}, 6]$ and $\mathsf{x'_2} \in [8, 8 + \mathsf{r2\_counter}]$ with $5 * \mathsf{r1\_counter} + 4 * \mathsf{r2\_counter} \leq 10$. Then the static analysis of the encoded tree starts with

[1] `http://apron.cri.ensmp.fr/library/0.9.10/mlapronidl/Polka.html`

the evaluation of the condition $\mathsf{x}_2' \leq 10$, inferring that such condition is always evaluated to true: indeed, x_2 could become greater than 10 only if $\mathsf{r2\_counter}$ was strictly greater than 2, but then $5 * \mathsf{r1\_counter} + 4 * \mathsf{r2\_counter} \leq 10$ could not hold since $\mathsf{r1\_counter} \geq 0$. TreeAnalyzer then analyzes the condition $\mathsf{x}_1' \leq 5$. In this case, it cannot definitely conclude that the condition is always evaluated to false, since x_1 can become less than or equal to 5 if $\mathsf{r1\_counter} \geq 1$, which is allowed by the invariant $5 * \mathsf{r1\_counter} + 4 * \mathsf{r2\_counter} \leq 10$. TreeAnalyzer then concludes that the test instance might be wrongly classified, since a branch that classifies it as $+1$ could be reached.

5 Experimental Evaluation

5.1 Methodology

We evaluate our proposal on three public datasets: Census, House and Wine, which are described in Section 5.2. Our methodology includes multiple steps. We start with a preliminary *threat modeling* phase, where we define the attacker's capabilities by means of a set of rewriting rules R and a set of possible budgets $\{K_1, \ldots, K_n\}$, as explained in Section 3.1. Our attackers are primarily designed to perform an experimental evaluation of TreeCert, yet they are representative of plausible attack scenarios which do not fit traditional distance-based models and are instead readily supported by the expressiveness of our threat model.

Datasets are divided into $\mathcal{D}_{train}$ and $\mathcal{D}_{test}$ by using a 90-10 splitting with stratified sampling (80-20 splitting is used for the smaller Wine dataset). We first train a decision tree t on $\mathcal{D}_{train}$ using the popular `scikit-learn` library, tuning the maximum number of leaves in the set $\{2^1, 2^2, \ldots, 2^{10}\}$ through cross validation on $\mathcal{D}_{train}$. We then evaluate the tree resilience to attacks against each attacker $A = (R, K_i)$ on $\mathcal{D}_{test}$, using a non-approximated technique. Given the expressiveness of our threat model, the only available solution for this is the algorithm in [5]. In particular, the algorithm computes $\mathbb{A}(\mathbf{x}_i)$, the set of *representative* attacks against t, for each instance $\mathbf{x}_i$ in $\mathcal{D}_{test}$. This is a comparatively small subset of the attacks $A(\mathbf{x}_i)$, which suffices to detect the successful evasions attacks without any loss of soundness or precision. We refer to this method as *Representative Attacks*. We observe and we experimentally confirm that computing even the representative attacks is intractable in general, which motivates the need for approximated analyses like ours; yet, being able to deal with this in a few cases is useful to assess the precision of TreeCert against a ground truth.

Finally, we compute the abstraction $t^{\#}$ on $\mathcal{D}_{test}$ for each attacker $A = (R, K_i)$ by using TreeCert. This allows us to classify each $(\mathbf{x}_i, y_i) \in \mathcal{D}_{test}$ as follows:

- *True Positive* (*TP*): TreeCert states that the instance $\mathbf{x}_i$ can be misclassified upon attack and this conclusion is correct. Formally, $t^{\#}(\mathbf{x}_i) \neq y_i \wedge \exists \mathbf{x}_i' \in \mathbb{A}(\mathbf{x}_i) : t(\mathbf{x}_i') \neq y_i$.
- *False Positive* (*FP*): TreeCert states that the instance $\mathbf{x}_i$ can be misclassified upon attack, but this conclusion is wrong. Formally, $t^{\#}(\mathbf{x}_i) \neq y_i \wedge \forall \mathbf{x}_i' \in \mathbb{A}(\mathbf{x}_i) : t(\mathbf{x}_i') = y_i$.

– *True Negative* (*TN*): TreeCert states that the instance $\mathbf{x}_i$ cannot be misclassified upon attack and this conclusion is correct. Formally, $t^{\#}(\mathbf{x}_i) = y_i \wedge \forall \mathbf{x}'_i \in \mathbb{A}(\mathbf{x}_i) : t(\mathbf{x}'_i) = y_i$.
– *False Negative* (*FN*): TreeCert states that the instance $\mathbf{x}_i$ cannot be misclassified upon attack, but this conclusion is wrong. Formally, $t^{\#}(\mathbf{x}_i) = y_i \wedge \exists \mathbf{x}'_i \in \mathbb{A}(\mathbf{x}_i) : t(\mathbf{x}'_i) \neq y_i$.

Since our analysis is sound, we cannot have *FN*. We then assess the quality of TreeCert by computing its *False Positive Rate FPR* and *False Discovery Rate FDR* as follows:

$$FPR = \frac{FP}{FP + TN}, \qquad FDR = \frac{FP}{FP + TP}.$$

We also compare the value of the loss under attack $\mathcal{L}^A(t, \mathcal{D}_{test})$ against its over-approximation $\mathcal{L}(t^{\#}, \mathcal{D}_{test})$, focusing on the *error rate*, i.e., the fraction of wrong predictions. Finally, we compare the execution time of TreeCert against the time spent in the computation of the set of the representative attacks.

5.2 Datasets

We perform our experiments on three publicly available datasets, whose key statistics are shown in Table 1. The preconditions of the rewriting rules and the magnitude of the perturbations have been set after a preliminary data exploration step, based on the observed data distribution in the dataset. A real-world application of our analysis technique would require input from domain experts to define the relevant threats, which is beyond the scope of our evaluation.

Census. The Census[2] dataset includes demographic information about American citizens. The prediction task is estimating whether the income of a citizen is above 50,000$ per year. For this dataset, we define four rewriting rules:

– cost 5: if the capital gain is in [0,100000], a citizen can raise it by 200;
– cost 5: if the capital loss is in [0,100000], a citizen can lower it by 200;
– cost 10: if the number of work hours is in [0,40], a citizen can raise it by 1;
– cost 10: if the age is in [0,40], a citizen can raise it by 1.

We consider 20, 40, 60, 80 as possible values of the attacker's budget.

House. The House[3] dataset contains house sale prices for the King County area. The prediction task is inferring whether a house costs at least as the median house price. For this dataset, we define four rewriting rules:

– cost 5: if the square footage of the living space of the house is in [0,3000], it can be increased by 50;

[2] `http://archive.ics.uci.edu/ml/machine-learning-databases/adult`
[3] `https://www.kaggle.com/harlfoxem/housesalesprediction`

Table 1. Properties of datasets used in the experiments.

Dataset	#Instances	#Features	Maj. class
Census	29169	51	0.75
House	21613	19	0.51
Wine	6497	12	0.63

- cost 5: if the square footage of the land space is in [0,2000], it can be increased by 50;
- cost 5: if the average square footage of the living space of the 15 closest houses is in [0,2000], it can be increased by 50;
- cost 5: if the construction year is in [1900,1970], it can be increased by 10.

We consider 10, 20, 30, 40 as possible values of the attacker's budget.

Wine. The Wine[4] dataset represents different types of wines. The prediction task is detecting whether a wine has quality score at least 6 on a scale 0–10. For this dataset, we define four rewriting rules:

- cost 2: if the residual sugar is in [2,4], it can be lowered by 0.01;
- cost 5: if the alcohol level is in [0,11], it can be increased by 0.01;
- cost 5: if the volatile acidity is in [0,1], it can be lowered by 0.01;
- cost 5: if the free sulfur dioxide is in [20,40], it can be lowered by 0.1.

We consider 20, 30, 40, 50, 60 as possible values of the attacker's budget.

5.3 Experimental Results

Precision. Table 2 reports for all datasets and budgets a number of measures computed for the trained decision tree t:

1. the traditional loss in absence of attacks $\mathcal{L}(t, \mathcal{D}_{test})$. This is the fraction of wrong predictions returned by t on $\mathcal{D}_{test}$ in the unattacked setting;
2. the loss under attack $\mathcal{L}^A(t, \mathcal{D}_{test})$, computed by enumerating all the representative attacks using the algorithm in [5]. This is the fraction of wrong predictions returned by t on $\mathcal{D}_{test}$ upon attack;
3. the over-approximation of the loss under attack $\mathcal{L}(t^\#, \mathcal{D}_{test})$, computed using the program analysis of TreeCert;
4. the false positive rate of TreeCert, noted *FPR*;
5. the false discovery rate of TreeCert, noted *FDR*.

The experimental results clearly confirm the quality of the analysis performed by TreeCert. In particular, we observe that the *FPR* is remarkably low, standing well below 5%, where 10% is considered a state-of-the-art reference for static analysis techniques [24]. Indeed, in Census we measured an absolute number of

[4] `https://www.openml.org/data/get\_csv/49817/wine\_quality.arff`

Table 2. Accuracy results across datasets.

Dataset	Budget	$\mathcal{L}(t, \mathcal{D}_{test})$	$\mathcal{L}^A(t, \mathcal{D}_{test})$	$\mathcal{L}(t^\#, \mathcal{D}_{test})$	FPR	FDR
Census	20	0.14	0.17	0.17	0.00	0.00
	40	0.14	0.17	0.17	0.00	0.01
	60	0.14	0.18	0.18	0.00	0.01
	80	0.14	0.20	0.21	0.00	0.01
House	10	0.10	0.12	0.12	0.00	0.02
	20	0.10	0.14	0.15	0.01	0.04
	30	0.10	0.16	0.17	0.01	0.06
	40	0.10	0.18	0.19	0.02	0.08
Wine	20	0.24	0.30	0.31	0.01	0.02
	30	0.24	0.34	0.35	0.02	0.03
	40	0.24	0.36	0.37	0.02	0.04
	50	0.24	0.37	0.39	0.03	0.05
	60	0.24	0.38	0.40	0.03	0.05

false positives never greater than 5. This is interesting, because it shows that for many instances there is a simple security proof, i.e., TreeCert is able to prove that they cannot be successfully attacked (i.e., they are *TN*), which significantly drops the *FPR*. As to the *FDR*, we observe that it also scores extremely well on all datasets, though it tends to be slightly higher than *FPR*. However, this is not a major problem in our application setting: contrary to what happens in traditional program analysis, where users are forced to investigate all false alarms to identify possible bugs, here we are rather interested in the aggregated analysis results, i.e., the final over-approximation of the loss under attack. Even on the House dataset, where *FDR* tends to be higher, we observe that the loss under attack is appropriately approximated by TreeCert, since there is a difference of at most 0.01 between the actual value of the loss under attack and its over-approximation. Remarkably, our experiments also show that the quality of the over-approximation is not significantly affected by the attacker's budget, which is important because it suggests that TreeCert likely generalizes to cases where computing the actual value of the loss under attack is computationally intractable, which is the intended use case of our analysis tool.

Efficiency. To show the efficiency of our approach, we compare in Figure 5 the running time of TreeCert against the time taken to compute the full set of the representative attacks. It is possible to clearly see that the two curves exhibit completely different trends. The time taken to construct the representative attacks has an *exponential* trend: the approach is efficient and feasible when the attacker's budget is low, but blows up to intractability very quickly. For example, each increase in the attacker's budget multiplies the execution time of a 3x factor in the case of Census and we experimentally confirmed that more than 12 hours of computation are needed when the budget grows to 100 (not

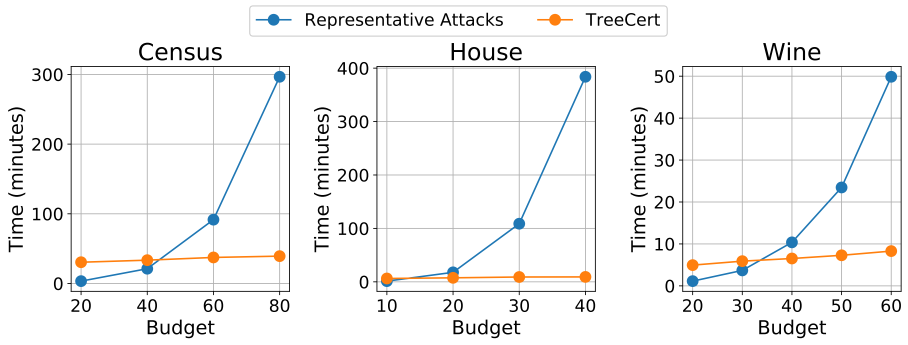

Fig. 5. Running time of TreeCert against the enumeration of representative attacks.

plotted). Conversely, the execution time of TreeCert is only marginally affected when increasing the attacker's budget, since the analysis always converges in less than one hour. In the case of the House dataset, computing the set of the representative attacks is even less feasible: even for small budgets, the running time is remarkably high, due to the fact that the trained decision tree uses many different thresholds, which makes the number of representative attacks blow up. Finally, also the Wine dataset shows similar figures, though the execution times there are lower due to its smaller size. This confirms that brute-force approaches based on the exhaustive enumeration of the representative attacks do not scale, yet luckily they can be replaced by more efficient abstraction techniques with very good precision.

6 Related Work

Verifying the security guarantees of machine learning models is an important task, which received significant attention by the research community in the last few years. In particular, many papers proposed techniques to verify the security of deep neural networks [28, 27, 17, 20, 15]; we refer to a recent survey for more work in this research area [29]. As of now, however, comparatively less attention has been received by the security verification of decision trees models.

The closest related work to our approach is a very recent paper by Ranzato and Zanella [23]. Their work also focuses on decision trees and builds on the abstract interpretation framework. However, their approach can only be applied to attackers who admit a simple mathematical characterization as a set of perturbations, e.g., based on distances. In particular, their soundness theorem relies on the hypothesis that, for each test instance $\mathbf{x}$, one has $A(\mathbf{x}) \subseteq \gamma(\alpha(\{\mathbf{x}\}))$, i.e., the abstraction of $\mathbf{x}$ must cover all the possible attacks. Checking this condition for distance-based attackers is straightforward, yet it is computationally infeasible in general. For example, in the case of the rewriting rules we considered, $A(\mathbf{x})$ is unknown *a priori*, but is induced by the application of the rules. Indeed, their tool *silva* only supports attackers based on the infinity-norm L_∞, which has a compact mathematical characterization as a set, but falls short of representing realistic threats. Instead, our approach is general enough to work on attackers modeled as arbitrary imperative programs.

Other approaches also deal with the verification of decision trees, but are not based on abstract interpretation. For example, Einzinger et al. use SMT solving to verify the robustness of gradient-boosted models [13]. Their approach also requires to explicitly encode the set of attacks $A(\mathbf{x})$ in closed form, which is only easily doable for artificial distance-based attackers. Moreover, SMT solving suffers from scalability issues, which required the authors to develop custom optimizations to make their approach practical. It is unclear whether this line of work can be adapted and scale to more expressive attackers or not, also because their tool is not publicly available. Other notable work includes the robustness verification algorithm by Chen et al. [8], which only works for attackers based on the infinity-norm L_∞, and the abstraction-refinement approach by Törnblom and Nadjm-Tehrani [26], which is not proved sound.

Finally, it is worth mentioning adversarial learning algorithms which train decision trees more resilient to evasion attacks by construction [19, 5, 7, 6]. This line of work is orthogonal to the security verification of decision trees, i.e., our approach can also be applied to estimate the improved robustness guarantees of trees trained using such algorithms.

7 Conclusion

We proposed a technique to certify the security of decision trees against evasion attacks by leveraging the abstract interpretation framework. This is the first solution which is both sound and expressive enough to deal with sophisticated attackers represented as arbitrary imperative programs. Our experiments showed that our technique is both precise and efficient, yielding only a minimal number of false positives and scaling up to cases which are intractable for a competitor [5].

We foresee several avenues for future work. First, we plan to extend our approach to the analysis of regression tasks and tree ensembles: though this is straightforward from an engineering perspective, we want to analyze the precision and the efficiency of our solution in such settings. Moreover, we will investigate techniques to automatically infer the minimal attacker's budget required to induce a given error rate on the test set, so as to efficiently provide security analysts with this useful information. Finally, we will investigate the trade-off between the precision and the efficiency of TreeCert by testing more sophisticated abstract domains and analysis techniques, e.g., trace partitioning.

References

1. Biggio, B., Corona, I., Maiorca, D., Nelson, B., Srndic, N., Laskov, P., Giacinto, G., Roli, F.: Evasion attacks against machine learning at test time. In: Proceedings of ECML PKDD. pp. 387–402 (2013)

2. Biggio, B., Roli, F.: Wild patterns: Ten years after the rise of adversarial machine learning. Pattern Recognit. **84**, 317–331 (2018)
3. Breiman, L.: Random forests. Machine Learning **45**(1), 5–32 (2001)
4. Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: Classification and Regression Trees. Wadsworth (1984)
5. Calzavara, S., Lucchese, C., Tolomei, G.: Adversarial training of gradient-boosted decision trees. In: Proceedings of CIKM. ACM (2019)
6. Calzavara, S., Lucchese, C., Tolomei, G., Abebe, S.A., Orlando, S.: Treant: Training evasion-aware decision trees. Data Min. Knowl. Discov. (2020), to appear
7. Chen, H., Zhang, H., Boning, D.S., Hsieh, C.: Robust decision trees against adversarial examples. In: Proceedings of ICML. PMLR (2019)
8. Chen, H., Zhang, H., Si, S., Li, Y., Boning, D.S., Hsieh, C.: Robustness verification of tree-based models. In: Proceedings of NeurIPS. pp. 12317–12328 (2019)
9. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proceedings of POPL. ACM (1977)
10. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Proceedings of POPL. ACM (1979)
11. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of POPL. ACM Press (1978)
12. Dreossi, T., Jha, S., Seshia, S.A.: Semantic adversarial deep learning. In: Proceedings of CAV. Springer (2018)
13. Einziger, G., Goldstein, M., Sa'ar, Y., Segall, I.: Verifying robustness of gradient boosted models. In: Proceedings of AAAI. pp. 2446–2453. AAAI Press (2019)
14. Friedman, J.H.: Greedy function approximation: a gradient boosting machine. Annals of statistics pp. 1189–1232 (2001)
15. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: safety and robustness certification of neural networks with abstract interpretation. In: Proceedings of Security and Privacy. IEEE Computer Society (2018)
16. Goodfellow, I., McDaniel, P., Papernot, N.: Making machine learning robust against adversarial inputs. Commun. ACM **61**(7) (Jul 2018)
17. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: Proceedings of CAV. Springer (2017)
18. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Proceedings of CAV. Springer (2009)
19. Kantchelian, A., Tygar, J.D., Joseph, A.D.: Evasion and hardening of tree ensemble classifiers. In: Proceedings of ICML. JMLR.org (2016)
20. Katz, G., Barrett, C.W., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: Proceedings of CAV. Springer (2017)
21. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. In: Proceedings of ICLR. OpenReview.net (2018)
22. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation (2006)
23. Ranzato, F., Zanella, M.: Abstract interpretation of decision tree ensemble classifiers. In: Proceedings of AAAI. AAAI Press (2020)
24. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., Jaspan, C.: Lessons from building static analysis tools at google. Commun. ACM **61**(4) (Mar 2018)

25. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I.J., Fergus, R.: Intriguing properties of neural networks. In: Proceedings of ICLR (2014)
26. Törnblom, J., Nadjm-Tehrani, S.: An abstraction-refinement approach to formal verification of tree ensembles. In: Proceedings of SAFECOMP. Springer (2019)
27. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Efficient formal safety analysis of neural networks. In: Proceedings of NeurIPS 2018 (2018)
28. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: Proceedings of USENIX Security. USENIX Association (2018)
29. Xiang, W., Musau, P., Wild, A.A., Lopez, D.M., Hamilton, N., Yang, X., Rosenfeld, J.A., Johnson, T.T.: Verification for machine learning, autonomy, and neural networks survey. CoRR **abs/1810.01989** (2018), `http://arxiv.org/abs/1810.01989`