Weijie Zhao, Yingjie Lao, and Ping Li. 2022. Integrity Authentication in Tree Models. In Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD ’22), August 14–18, 2022, Washington, DC, USA. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3534678.3539428

1 Introduction

Along with the evolution of machine learning, the growing cost of model building in terms of computational power, data annotation, and human expertise calls for intellectual property (IP) protection methods to secure the innovations and creative endeavors in this field. The problem is clearly aggravated by recent studies demonstrating that machine learning models are vulnerable to various categories of adversarial attacks [14, 15, 31, 44, 50, 56, 61] and model stealing via reverse engineering or model extraction [27–29]. Especially under the machine learning as a service (MLaaS) paradigm where the supply chain of models may involve multiple parties and vendors, the proprietary assets of the model IP owner, including data, algorithm, and computation infrastructure, are vulnerable to breach. On the other hand, maliciously altered models, e.g., by poisoning or backdoor attacks [11, 14, 15, 25, 45, 48, 54, 55], will also impair the integrity, reputation, and profit of the model owner. These attacks have been extensively studied on various machine learning models, including collaborative filtering [36], logistic regression [49], clustering [6], support vector machine [5], and also deep neural networks [24, 51, 54, 57, 59]. Very recently, such attacks on tree models have also been investigated [3, 10, 23, 62], which further demands techniques for protecting tree models.

To this end, signature embedding and fingerprinting have emerged as promising directions for protecting the advanced machine learning models [1, 9, 13, 17, 32–34, 37, 43, 58, 60, 63, 64]. Conceptually, these techniques embed a unique signature into a model that will behave differently from other models. The presence of such signatures should be easily verifiable in a later stage, which can serve as an ownership or integrity proof. However, almost all of these prior methods are only targeting deep neural networks, which is understandable as deep neural networks have shown superior performance across many fields while requiring large costs to build.

In this paper, we take the first step to investigate signature embedding for (ensemble) tree models [8]. We follow the direction of embedding fragile watermarks [33] for the purpose of verifying the integrity of the model, which is different from robust watermarks [1, 13, 17, 32, 37, 43, 58, 60, 63] that are used to trace the IP ownership. In particular, we focus on boosted trees in our experiments, although the ideas are also applicable to (e.g.,) bagging [7].

1.1 Boosted Tree Models

Boosting [4, 18–21, 38–40, 52, 53] is a successful learning paradigm widely used in practice. Boosting is typically integrated with trees [8] to produce powerful tools for classification and regression; readers are also referred to some interesting discussions at https://hunch.net/?p=1467 for comparing deep neural networks with boosted trees. As summarized in [16], two implementation tricks have made boosted tree models considerably more practical:

- Li et al. [42] developed an adaptive binning strategy to effectively transform numerical features into (non-negative) integers. It is adaptive because, for each feature, it only assigns bins in the regions where there are data points. This simple trick has simplified the implementation (including parallelization) and improved the efficiency.
• Li [39] derived the explicit (and robust) formula for tree-split criterion using the second-order gain information (i.e., "Robust LogitBoost"), which typically improves the accuracy, compared to using only the first-order gain information [20]. This formula for the tree-split gain has resolved the numerical issue in the original logitboost [20–22] and becomes the standard implementation in popular tree platforms.

Another major progress is "ABC-Boost" [38–41] for multi-class classification by re-writing the derivatives of the classical multi-class logistic regression loss function. ABC-Boost often considerably improves the accuracy of multi-class classification tasks.

In this paper, we develop novel integrity authentication schemes for the "Robust LogitBoost" algorithm [39, 40] using the code of [41].

1.2 Challenges, Goals, and Contributions
Signature embedding or watermarking has been widely studied in the deep learning community [1, 9, 13, 17, 32–34, 37, 43, 58, 60, 63, 64]. Many of these works embed a desired behavior into the learned function by using backdoor techniques [1] or enforcing a specific representation in a latent space [13]. However, similar techniques are not readily available for tree models in order to facilitate effective signature embedding, e.g., backdoor on tree models entails little study in the literature. In addition, tree models have different architectures and applications from deep neural networks, which also demands careful customization of tree model signature embedding. In general, it is not straightforward to directly apply the signature embedding process of deep learning to tree models:

• Deep learning methods require gradients. However, tree models are not differentiable.

• Many deep learning signature embedding algorithms require to retrain the network. In the context of boosted tree models, each training iteration constructs a new tree that attempts to correct the misclassifications of previous trees. Appending more trees not only increases the model size (to store additional trees) but also damages the inference performance (to evaluate additional trees).

• Meanwhile, retraining boosted tree models by only replacing a subset of existing trees is still an open research since each tree is generated on the results of the previous trees: eliminating previous trees invalidates the boosting dependency.

As a result, it is impractical to inplace retrain the tree models.

Therefore, this paper designs a new signature embedding framework for non-differentiable and non-inplace-retrainable tree models. We target to have a signature embedding framework to "sign" tree models. After distributing the model to the host services, we are able to remotely access the model in a black-box fashion: we query the model with several secret inputs (we call them signature keys) and verify the output predictions. We have three main objectives: (i) The signature embedding procedure should generate a vast number of distinct signs for different model host services. (ii) The signature can verify the integrity of the model that has not been tampered with. Whenever the model host modified the model, the output predictions of signature keys should be changed, i.e., the signature should be fragile against malicious modifications. (iii) The signature embedding algorithm should not degrade the prediction accuracy.

Contributions. We summarize our contributions as follows:

• We introduce a model authentication scheme and signature embedding algorithm for tree models. As far as we know, this is the first study that investigates tree model authentication.

• We propose a novel searching and selection algorithm to generate signature keys and manipulate tree models.

• We empirically evaluate the proposed algorithm on various public datasets. The results confirm the effectiveness and high authentication rate of our methods.

• Since this is the first work on signature embedding for tree models, we design baseline attacks to assess the quality of our technique, which also provides qualitative and quantitative metrics and comparisons for future tree model protection.

2 AUTHENTICATION FRAMEWORK
2.1 Threat Model
The signature embedding process can be conducted by the model builder or a trusted party. Without loss of generality, we assume a pre-trained model could be received from a model builder who builds the model architecture $F$ and corresponding parameters $\Theta_{pr}$, with the training dataset $D_{tr}$, and a held-out validation dataset $D_{v}$ for evaluating the performance. We then apply the proposed methodology to this tree model to embed a desired signature. Only the legitimate model owner will have knowledge of the signature and the corresponding signature keys. In the authentication process, the model owner can verify the presence of the signature by using the signature keys via the prediction API—the model owner only needs access to the predicted class during the authentication. Thus, memorizing the tree weight hashing value or the predicted probability of some inputs may not be applicable.

For signature embedding on DNNs, transformation attacks such as model compression, model fine-tuning, and signature overwriting are often used to evaluate the performance of an embedded signature. Since this paper is the first work to investigate signature embedding on tree models, we design similar transformation attacks as a baseline countermeasure to assess the quality of our proposed technique, which also provides qualitative and quantitative metrics and comparisons for future tree model protection methods. Note that the target of this work is a fragile signature, which could verify if a tree model has been tampered with and serve as a proactive defense against malicious model modifications, similar to [33]. In other words, the objective of this signature is integrity verification as opposed to IP tracing as in some of prior DNN watermarking and fingerprinting methods [1, 9, 13, 26, 32, 60]. We thus expect embedded signatures would disappear after transformation attacks. Note that ambiguity attack is not a concern for fragile signature [33].

2.2 Flow of Enrollment and Authentication
We formally express our proposed scheme as two phases:

• Signature embedding $R^{msg} \leftarrow \text{Embed}(R, msg)$ that generates a signed version of regression trees $R^{msg}$, signature key $key$ from the given original regression trees $R$ with a target signature message $msg$.

• Signature extraction $msg' \leftarrow \text{Extract}(R^{msg}, key)$ that collects the signature message $msg'$ using $key$ from the given $R^{msg}$. $msg'$ will then be used for authentication.
Figure 1 provides an illustration of the signature enrollment and authentication workflow. Built upon the proposed signature embedding technique, an authentication process can be established. After successfully embedding the signatures, the signature key and the correct signature message msg are stored on a secure server. We can also choose to register the model ID during enrollment to enable authentication for multiple legitimate users. In this case, each version of the model needs to be generated by applying the proposed technique with a different set of signature keys and messages. Then, the model builder will sell, distribute, or deploy the model. Each customer may receive a different version of the original tree model (i.e., with a different signature). Later, when an authentication process is initiated, based on the enrolled model ID, the server sends the signature key to the model API and collects the corresponding response. The server determines the authenticity of the unknown model by checking how this response matches the stored signature message. Thus, another advantage of this method is its small computational and communication overhead: only one authentication process, especially when comparing to those countermeasures that require periodically checking the accuracy [48, 59]. We focus on the steps of embedding signatures on tree models in this paper. Modern authentication techniques and protocols [2, 35, 47] can be implemented on top of the generated signatures.

3 SIGNATURE EMBEDDING

In this section, after a brief introduction of boosted trees, we present the proposed two-stage algorithm to embed signatures: 1) locating the signature key candidates; 2) selecting independent signature keys and manipulating the prediction value on the terminal nodes.

3.1 Booted Trees: MART & Robust LogitBoost

Denote a training dataset by \(\{(y_i, x_i)\}_{i=1}^N\), where \(N\) is the number of training samples, \(x_i\) is the \(i\)-th feature vector, and \(y_i \in \{1, 2, ..., K\}\) is the \(i\)-th class label. Consider the same classical framework as in [20, 21], which assumes the class probabilities \(p_{i,k}\), to be:

\[
p_{i,k} = \Pr(y_i = k | x_i) = \frac{e^{F_{i,k}(x_i)}}{\sum_{k=1}^K e^{F_{i,k}(x_i)}} , \quad i = 1, 2, ..., N, \tag{1}
\]

where \(F_{i,k}(x_i)\) is an additive model of \(M\) terms:

\[
F^{(M)}(x) = \sum_{m=1}^M \rho_m h(x; a_m), \tag{2}
\]

where \(h(x; a_m)\) is typically a regression tree, and \(\rho_m\) and \(a_m\) are parameters learned by minimizing the negative log-likelihood loss:

\[
L = \sum_{i=1}^N L_i, \quad L_i = -\sum_{k=1}^K r_{i,k} \log p_{i,k} \tag{3}
\]

where \(r_{i,k} = 1\) if \(y_i = k\) and \(r_{i,k} = 0\) otherwise. The optimization procedures would need the derivatives of \(L\) with respect to \(F_{i,k}\). The classical multi-class logistic regression textbooks give

\[
\frac{aL_i}{aF_{i,k}} = -(r_{i,k} - p_{i,k}), \quad \frac{a^2 L_i}{a^2 F_{i,k}} = p_{i,k}(1 - p_{i,k}). \tag{4}
\]

Algorithm 1 Robust LogitBoost.

1: \(F_{i,k} = 0, \quad p_{i,k} = \frac{1}{K}, \quad k = 1 \text{ to } K, \quad i = 1 \text{ to } N\)
2: for \(m = 1 \text{ to } M\) do
3:   for \(k = 1 \text{ to } K\) do
4:     \(\{R_{j,k,m}\}_{j=1}^J\) is \(j\)-terminal node regression tree from \((r_{i,k} - p_{i,k}, x_i)_{i=1}^N\), with weights \(p_{i,k}(1 - p_{i,k})\), using the tree split gain formula Eq. (5).
5:     \(\beta_{j,k,m} = \frac{\sum_{i \in B_{j,k,m}} r_{i,k} - p_{i,k}}{\sum_{i \in B_{j,k,m}} (1 - p_{i,k}) p_{i,k}}\)
6:     \(f_{i,k} = \sum_{j=1}^J \beta_{j,k,m} x_i R_{j,k,m}, \quad F_{i,k} = F_{i,k} + \nu f_{i,k}\)
7:   end for
8:   \(p_{i,k} = \text{exp}(F_{i,k}) / \sum_{s=1}^K \text{exp}(F_{i,s})\)
9: end for

Algorithm 1 describes Robust LogitBoost [39]. It fixes the previously thought ‘numerical instability’ problem [21], as discussed in [20, 22]. Robust LogitBoost [39] uses the 2nd-order formula in (5) for computing the gains when deciding the splits.

Given \(N\) data points which are assumed to be sorted according to the corresponding feature values. The tree-split procedure is to find the index \(s, 1 \leq s \leq N\), such that the weighted square error (SE) is reduced the most if split at \(s\). That is, we seek the \(s\) to maximize

\[
\text{Gain}(s) = \frac{\left|\sum_{i=1}^s (r_{i,k} - p_{i,k})\right|^2}{\sum_{i=1}^N p_{i,k}(1 - p_{i,k})^2} \tag{5}
\]

\[
+ \frac{\left|\sum_{i=s+1}^N (r_{i,k} - p_{i,k})\right|^2}{\sum_{i=s+1}^N p_{i,k}(1 - p_{i,k})^2}
\]

Because the computations involve \(\sum_{i=1}^N p_{i,k}(1 - p_{i,k})\) as a group, this procedure is numerically stable. In comparison, MART [20] used the first order information to construct the trees, i.e.,

\[
\text{MartGain}(t) = \frac{1}{s} \left|\sum_{i=1}^s (r_{i,k} - p_{i,k})\right|^2 + \frac{1}{N-s} \left|\sum_{i=s+1}^N (r_{i,k} - p_{i,k})\right|^2 - \frac{1}{N} \left|\sum_{i=1}^N (r_{i,k} - p_{i,k})\right|^2. \tag{6}
\]

To avoid repetition, we do not provide the pseudo code for MART [20], which is in fact almost identical to Algorithm 1. The
only difference is in Line 4, which for MART becomes

\[ \{ R_{i,k,m} \}_{i=1}^{I} = j \text{-terminal node regression tree from } \{ r_{i,k} - p_{i,k}, \ x_i \}_{i=1}^{N} \text{ using the tree split gain formula Eq. (6).} \]

**Inference.** After the training, we obtain \( K \times M \) regression trees \( f_{i,k} \) (M trees for each class). In the inference, we just follow the trained regression trees to obtain \( f_{i,k} \). For each instance, it ends up at a terminal node in the tree, and \( f_{i,k} \) is the prediction value stored at the node. We aggregate them to \( f_{i,k} \) and apply a softmax over all classes to compute the multi-class classification probability. Figure 2 depicts an example: we have 3 classes and build the tree models for 2 iterations. For each tree, the inference procedure follows the tree to a terminal node. The prediction value for each class is aggregated across iterations and then softmaxed to generate the probability.

\[
\begin{array}{c|c|c|c}
\text{Iter 1} & \text{Class 1} & \text{Class 2} & \text{Class 3} \\
0.2 & 2.1 & 2.7 & 0.1 \\
-3.7 & -2.6 & -2.0 & 0.0 \\
-1.6 & 1.1 & -0.4 & 0.0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
\text{Iter 2} & \text{Class 1} & \text{Class 2} & \text{Class 3} \\
2.1 + 0.6 & 2.7 & 0.986 & 0.005 & 0.009 \\
-3.7 + 1.1 & -2.6 & 0.005 & 0.009 \\
-1.6 - 0.4 & -2.0 & 0.009 & 0.009 \\
\end{array}
\]

**Figure 2:** Inference example for 2 iterations and 3 classes. (For simplicity, the learning rate is assumed to be \( \nu = 1 \) here.)

### 3.2 Signature Key Candidates Locating

As described in Section 2.2, our proposed authentication framework leverages a collection of signature keys to check the embedded signature. Here a signature key is a legal input of the model, e.g., an image for vision datasets. For a subset of the signature keys, we manipulate the tree prediction value so that their prediction flips to a new class other than the original one. On the other hand, the manipulation must be carefully performed to preserve the model functionality. Intuitively, the new class should be the second most likely class of an input: We tend to have the key input whose highest class prediction is close to its second-highest class. With \( S \) signature keys, we can generate \( 2^S \) unique signatures (by either using the original class or flipping it to a new class).

**Signature key locating problem.** Recall that each internal node of the trees represents a split condition, i.e., whether feature \( x \) is greater than \( y \). For any terminal node (leaf node), we can extract the possible input space to arrive at this node by intersecting the split conditions of its ancestors. Thus, even without the training data, we can construct a valid input space by searching the split conditions. Given \( M \times K \) trees, we are going to find \( S \) distinct signature keys, such that the maximum gap for each signature key is minimized, where the gap denotes the difference between the largest \( f_{i,k} \) and the second largest \( f_{i,k'} \) (class \( k \) is the original prediction and class \( k' \) is the class we are going to flip to after embedding the signature).

This problem is NP-Hard as we can straightforwardly reduce the partition problem [30] to the signature key locating problem within polynomial time. The good news is that we are not required to have the exact best \( S \) signature keys for the signature embedding procedure. As long as the gap is sufficiently small, changing the prediction value on a terminal node (i.e., flipping the prediction class) will not dramatically affect predictions for other test instances.

Thus, we introduce a heuristic random search algorithm to locate the “sufficiently good” signature key candidates in Algorithm 2. We call them candidates since they are not guaranteed to be independent—flipping one of them may affect other keys. We introduce a selection procedure in Section 3.3 below to generate the final signature keys from the candidate set. We have a scaling factor \( \alpha \) to generate sufficient candidates for the follow-up selection. Algorithm 2 maintains a heap that stores the best signature keys (the ones with the least gaps) found currently. Since tree models commonly quantize data into natural numbers [42], we assume all the data are non-negative integers—all constraints are initialized to \([0, +\infty)\) before each search. The sub-procedure Algorithm 3 performs a depth-first search (DFS) while considering all the terminal nodes in a random order (Line 12). When we reach Line 3, any instance satisfies \( \text{cons} \) is a possible signature key. We construct one instance from the constraints, compute its gap, and insert it into

**Algorithm 2 Signature Key Candidates Locating**

| Input: | \( M \times K \) decision trees \( f_{i,k} \) |
| Output: | \( S \times \alpha \) signature keys |
| 1. | initialize a global heap that stores the found best \( S \times \alpha \) signature keys |
| 2. | for repeat \( \leftarrow 1 \) to \( S \times \alpha \) do |
| 3. | \( \text{cons} \leftarrow [0, +\infty) \forall d \in [1..\#Features] \) |
| 4. | Random-DFS(1,1,\text{cons}) |
| 5. | end for |

**Algorithm 3 Random-DFS**

| Input: | current searching iteration \( i \), class \( k \) and constraints \( \text{cons} \) |
| Output: | a heap with updated signature keys |
| 1. | if \( i > M \) then |
| 2. | if \( k > K \) then |
| 3. | update signature key heap with \( \text{cons} \) |
| 4. | if reach max search step then |
| 5. | stop all Random-DFS |
| 6. | end if |
| 7. | return |
| 8. | else |
| 9. | return Random-DFS(1, \( k + 1 \), \text{cons}) |
| 10. | end if |
| 11. | end if |
| 12. | for each terminal node \( n \) of tree \( f_{i,k} \) in random order do |
| 13. | if \( \text{cons} \cap \text{condition}(n) \neq \emptyset \) then |
| 14. | Random-DFS(i + 1, \( k, \text{cons} \cap \text{condition}(n) \)) |
| 15. | end if |
| 16. | end for |
After obtaining $S \times \alpha$ signature key candidates, we are required to select $S$ independent signature keys. The definition of the "independence" is as follows:

**Definition 3.1.** Given a collection of instances, they are independent if and only if: for each instance, there exists a terminal node on its highest and second-highest prediction classes such that the terminal node is not referenced by any other instances in this collection.

Our signature embedding process will add a perturbation to the prediction value on terminal nodes of the signature keys (details are presented in Section 3.4). We can add the perturbation to the "independent" terminal node on its highest or second-highest prediction class. In this case, the perturbation will not affect the prediction result of other instances since no other instance references this terminal node. The independence guarantees that we can pick any subset of the signature keys as the signature message independently—the prediction class flip of one chosen signature key does not affect others.

![Iterative Algorithm](image)

**Figure 3: An example for signature key selection.**

Figure 3 depicts an example for the signature key selection. We have 3 signature key candidate instances represented in 3 colors in the example. Flipping class 3 or class 1 for the green instance may change the prediction value of other instances. Thus, the collection of red, blue, and green signature candidates are not independent. However, we can choose a subset of instances to form a collection of independent signature keys: Here we can manipulate the prediction value of the 3rd terminal node on the (iter 1, class 2) tree to flip the prediction of the red instance and manipulate the prediction value of the 2nd terminal node on the (iter 2, class 1) tree to flip the blue candidate. The flipping of the red and blue signature candidates does not affect any other instances.

However, maximizing the number of selected independent signature keys in a collection is an NP-Hard problem—it is equivalent to the maximum independent set problem [12]. Since we only need to obtain a "reasonable" number of independent signature keys to generate a sufficient number of key combinations, a collection with 10 to 20 independent signature keys is sufficient for practical authentication usage. We do not need to pursue an optimal but time-consuming independent signature key selection process. Thus, we propose a greedy algorithm to select the signature keys.

According to the independence definition, we construct a histogram ref_freq that counts the frequency of all terminal nodes: how many signature key candidates reference the node. As illustrated in Algorithm 4, we find the terminal node that only has one referencing signature key candidate and is on the highest or second-highest prediction class of the candidate instance. Then we put this candidate into the final set of signature keys. Because the terminal node is only referenced by this signature key, manipulating the prediction value of this node does not affect other signature keys’ highest and second-highest predicted classes.

### Algorithm 4 Signature Key Selection

**Input:** $S \times \alpha$ signature key candidates  
**Output:** $S$ signature keys

1. selected ← ∅  
2. for each terminal nodes $n$ do  
3. if ref_freq[$n$] = 1 then  
4. $c$ ← the signature key candidate referencing $n$  
5. if $n \in$ the highest/second-highest prediction of $c$ then  
6. if $|\text{selected}| < S$ and $c$ is not selected then  
7. selected ← selected ∪ {$c$}  
8. end if  
9. end if  
10. end if  
11. end for  
12. return selected

Now we have obtained $S$ independent signature keys. The signature key candidate locating algorithm (Algorithm 2) tries to find an instance such that its aggregated prediction values of the highest possible class ($F_{i,k}$) and the second-highest class ($F_{i,k'}$) are close. In order to flip the prediction of a signature key without affecting other keys, we can simply add the difference between $F_{i,k}$ and $F_{i,k'}$ to the terminal node whose ref_freq is 1 and then add a small perturbation $\epsilon$, e.g., $10^{-3}$, onto the prediction value of the node to obtain a desired flipped prediction. The small perturbation we added flips the order of the aggregated prediction values of the highest and the second-highest class. In addition, our independence constraint ensures that the perturbation does not affect the prediction value for other signature keys. We assume the prediction value of $F_{i,k}$ and $F_{i,k'}$ are significantly greater than the $F$ values of other classes. The assumption empirically holds for a well-trained converged model. With this assumption, after we apply a small perturbation on the prediction value of class $k$ and $k'$, it should not make the prediction values of them smaller than other classes. Therefore, we only consider the terminal nodes on classes $k$ and $k'$. This completes the proposed signature embedding algorithm.
4 EXPERIMENTAL EVALUATION

The objective of the experimental evaluation is to investigate the performance of our algorithm, based on Robust LogitBoost [21, 39, 40]. Specifically, we target to answer the following questions:

- How many signature keys can be generated in one pass?
- How does the signature embedding procedure affect the model functionality, i.e., test accuracy?
- Is there any correlation among the generated signature keys?
- How effective is the embedded signature in detecting malicious modification, i.e., when the attacker adds/removes decision trees?

Implementation. We use the code base from [40]. The code is compiled with g++-5.4.0 enabling the "O3" optimization. We execute the experiments on a single node server with one Intel Xeon Processor Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz and 128 GB of memory. The OS is Ubuntu 16.04.4 LTS 64-bit.

Datasets. We evaluate our proposed algorithm on 20 public datasets.

See Table 1 for the dataset specifications.

Table 1: Dataset specifications.

| Dataset | #Train | #Test | #Class | #Dim |
|---------|--------|-------|--------|------|
| CIFAR10 | 50,000 | 10,000 | 10     | 3,072|
| connect4| 54,045 | 13,512 | 3      | 126  |
| covtype | 464,809| 116,203| 7      | 54   |
| glass   | 171    | 43    | 6      | 9    |
| letter  | 15,000 | 5,000  | 26     | 16   |
| MNIST   | 60,000 | 10,000 | 10     | 780  |
| news20  | 15,935 | 3,993  | 20     | 62,061|
| pendants | 7,494 | 4,984  | 10     | 16   |
| poker   | 25,010 | 1,000,000 | 10 | 10  |
| protein | 17,766 | 6,621  | 3      | 357  |
| satimage| 4,435  | 2,000  | 6      | 36   |
| segment | 1,848  | 462    | 7      | 19   |
| Sensorless | 48,309 | 10,000 | 11     | 48   |
| SVHN    | 73,257 | 26,032 | 10     | 3,072|
| svmguide2 | 312  | 79    | 3      | 20   |
| svmguide4 | 300  | 312   | 6      | 10   |
| usps    | 7,291  | 2,007  | 10     | 256  |
| acoustic| 78,823 | 19,705 | 3      | 50   |
| vehicle | 676    | 170    | 4      | 18   |
| vowel   | 528    | 462    | 11     | 10   |

Hyperparameters. We use the code base of Robust LogitBoost from Li and Zhao [40]. The model mainly has 3 hyperparameters: the number of terminal nodes for each decision tree ($J$), the number of training iterations ($M$), and the learning rate ($\alpha$). The learning rate does not affect the tree structure. Since our signature embedding algorithm does not depend on how well the model is trained, we fix the learning rate as 0.1 in the following experiments. We enumerate the remaining two hyperparameters, i.e., $J$ and #Iterations, to examine the performance of our proposed algorithm.

4.1 Number of Independent Signature Keys

In Table 2, our signature key candidate size is $S = 40$, and we perform $S \times \alpha = 40 \times 8 = 320$ Random-DFS searching. When $J$ is sufficiently large, it is easy to find the terminal node with only one referenced signature key (Algorithm 4). In most cases, all candidates can be selected in the procedure. When $J$ is small (e.g., $J = 4$) and #Iteration is small (it is not common in practice due to its inferior accuracy), since each candidate has to select a terminal node from the limited choices, fewer nodes with 1 referencing frequency can be found. The signature space (the number of unique signatures) is exponential to the number of selected signature keys, i.e., $2^x$ where $x$ is the number of selected signature keys. As seen in Table 2, for most configurations, we can have more than 20 selected independent signature keys. With 20 keys, the signature space goes beyond 1 million. Even with the rare cases when $J$ is small and 50 iterations, we have at least 9 independent signature keys—it yields more than 500 unique signatures and is well sufficient for real-world authentication applications.

Table 2: Numbers of selected independent signature keys with $S = 40$ candidates, which are generated with $\alpha = 8$ and max search step = 1,000 for each Random-DFS. $J$ is the number of terminal nodes for a tree and #Iteration is the number of training iterations (the number of trees for a class).

| $J$   | 4 | 8 | 12 | 20 | 4 | 8 | 12 | 20 | 4 | 8 | 12 | 20 |
|-------|---|---|----|----|---|---|----|----|---|---|----|----|
| Iteration | 50 | 100 | 200 |
| CIFAR10 | 21 | 40 | 40 | 40 | 33 | 40 | 40 | 40 | 40 | 40 | 40 | 40 |
| connect4 | 17 | 33 | 40 | 40 | 19 | 39 | 40 | 40 | 23 | 40 | 40 | 40 |
| covtype | 23 | 37 | 39 | 40 | 30 | 40 | 40 | 40 | 27 | 40 | 40 | 39 |
| glass   | 23 | 36 | 37 | 35 | 22 | 33 | 36 | 39 | 32 | 33 | 28 | 35 |
| letter  | 38 | 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 |
| MNIST   | 34 | 40 | 40 | 40 | 37 | 40 | 40 | 40 | 30 | 40 | 40 | 31 |
| news20  | 38 | 39 | 40 | 40 | 40 | 40 | 37 | 40 | 28 | 40 | 40 | 30 |
| pendants | 23 | 35 | 40 | 40 | 28 | 37 | 39 | 40 | 36 | 40 | 40 | 33 |
| poker   | 9  | 24 | 21 | 38 | 14 | 31 | 34 | 40 | 25 | 38 | 40 | 38 |
| protein | 15 | 23 | 21 | 40 | 23 | 28 | 40 | 40 | 10 | 35 | 40 | 31 |
| satimage| 34 | 40 | 40 | 40 | 38 | 40 | 40 | 40 | 40 | 40 | 40 | 40 |
| segment | 33 | 35 | 38 | 38 | 37 | 39 | 40 | 34 | 31 | 37 | 40 | 38 |
| Sensorless | 29 | 40 | 40 | 40 | 34 | 39 | 40 | 40 | 36 | 28 | 22 | 20 |
| SVHN    | 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 | 40 |
| svmguide2 | 19 | 35 | 39 | 39 | 26 | 37 | 29 | 25 | 27 | 38 | 23 | 14 |
| svmguide4 | 24 | 32 | 37 | 40 | 26 | 32 | 40 | 39 | 31 | 37 | 39 | 30 |
| usps    | 37 | 38 | 40 | 40 | 32 | 36 | 40 | 40 | 29 | 40 | 34 | 38 |
| acoustic| 20 | 33 | 39 | 40 | 29 | 39 | 40 | 40 | 37 | 40 | 40 | 40 |
| vehicle | 21 | 40 | 40 | 40 | 20 | 39 | 40 | 40 | 25 | 40 | 40 | 40 |
| vowel   | 26 | 38 | 40 | 40 | 24 | 36 | 36 | 34 | 28 | 31 | 24 | 22 |

4.2 Searching Factor $\alpha$

The searching factor $\alpha$ balances the signature key candidate searching time and the number of selected independent signature keys. With a larger $\alpha$, we achieve more candidates at the cost of searching time. Table 3 shows that the searching time is almost linear to $\alpha$. As expected, the number of selected independent signature keys increases when we use a larger $\alpha$ factor. $\alpha = 8$ is sufficient for most datasets to generate enough signature keys. Since our signature key candidate searching only touches on the decision trees of the trained model, neither training nor testing data are required in the authentication workflow. As a result, the signature key candidate searching is very efficient (regardless of the size of the training/testing dataset). The execution time for most datasets is
around only one second. For the data with more classes, Gradient Boosting Machine generates more trees (it works in a one versus all multi-class classification fashion). Therefore, the letter dataset with 26 classes takes the most searching time. Because the searching only takes sub-minute time, users are free to increase $\alpha$ to obtain more independent signature keys for a larger signature space.

### 4.3 Model Functionality

Now we have obtained the signature keys. Our target is to embed these signatures without dramatically changing the model functionality, i.e., prediction accuracy. Table 4 presents the number of changed predictions after we embed all 20 signatures (by manipulating the prediction values on terminal nodes and flipping all the prediction classes of 20 signature keys). The prediction change is minimal (0%−0.08%) on all datasets in the experiments. Although our independent signature key selection ensures that one manipulated prediction value on a terminal node is only referenced by one signature key, the instance in the test dataset may touch multiple manipulated terminal nodes. The aggregated small perturbation may flip the prediction for some “sensitive” test instance (whose highest prediction class is very close to its second-highest prediction class). As shown in the experiment result, this aggregated flipping is very uncommon because our perturbation is small—at most 0.08% prediction values are changed for test datasets. Therefore, we can conclude that our proposed signature embedding algorithm preserves the original model functionality.

### 4.4 Attacking

At this moment, we have tested the signature key generation algorithm and the embedding procedure—the proposed technique can embed signatures into tree models without noticeable performance degradation. Here we present two possible attacks to verify the fragility of the signatures: we desire to detect unauthorized modification to the model so that the signature should be destroyed when the model is attacked. Due to the space limit, we only report the attacking result on 5 popular datasets (CIFAR10, letter, MNIST, pendigits, and poker). The attacking results on other datasets show a similar trend and thus are omitted.

### Table 5: The percentage of the signature key outputs change when appending more training iterations on CIFAR10, letter, MNIST, pendigits, and poker with $J = 20$

| #Signed iterations | #Appended iterations |
|-------------------|----------------------|
| 1                 | 5                    |
| 10                | 50                   |
| 200               | 50                   |
| CIFAR10           | 65% 50% 50%          |
| letter            | 40% 55% 60%          |
| MNIST             | 60% 55% 50%          |
| pendigits         | 70% 50% 40%          |
| poker             | 45% 45% 35%          |

**Attack 1: Adding more training iterations.** One possible attack is to continue training the model by adding more training iterations. This attack appends more trees to the original model. From Table 5, we observe that even appending one iteration to the original model...
we may have fewer independent signatures for some datasets. However, the generated signature keys with $\alpha = 8$ are sufficient for practical use. The signature embedding procedure has minimal impact on model functionality. For most datasets, only a small number of predictions (< 10) are changed. In the percentage view, 0% – 0.08% predictions are changed for all datasets. We consider two possible attacks: the attacker removes some trees or appends some trees by training more iterations. In both attack scenarios, our authentication framework can successfully detect those changes—because the embedded signatures are changed even with adding or removing one iteration of decision trees.

5 CONCLUSION
In this paper, we introduce a novel model authentication framework and signature embedding algorithm for tree models. We propose a (largely) heuristic searching and selection algorithm to generate signature keys and manipulate tree models. We evaluate the proposed method on 20 public datasets. Experiments demonstrate that our proposed algorithm can efficiently locate signature keys in a few seconds. The signature embedding minimally affects the model functionality—the accuracy change is within 0.08% for all tested datasets and within 0.03% for most cases. As a fragile signature for model authentication, the empirical results confirm that adding/removing even a small number of trees to the model will destroy the embedded signatures. In summary, the generated signature by our proposed method is an effective tool for ensuring the integrity of a deployed model that has not been tampered with.

REFERENCES
[1] Yossi Adi, Carsten Baum, Moustapha Cissé, Benno Pinkas, and Joseph Keshet. Turning your weakness into a strength: Watermarking deep neural networks by backdooring. In Proceedings of the 27th USENIX Security Symposium (USENIX Security), pages 1615–1631, Baltimore, MD, 2018.
[2] Amjad Ali Alamr, Firdous Kausar, Jongsung Kim, and Changho Seo. A secure ecc-based rfid mutual authentication protocol for internet of things. The Journal of Supercomputing, 74(9):4281–4294, 2018.
[3] Maksym Andriushchenko and Matthias Hein. Provably robust boosted decision stumps and trees against adversarial attacks. In Advances in Neural Information Processing Systems (NeurIPS), pages 12997–13008, Vancouver, Canada, 2019.
[4] Peter Bartlett, Yoav Freund, Wee Sun Lee, and Robert E. Schapire. Boosting the margin: a new explanation for the effectiveness of voting methods. The Annals of Statistics, 26(5):1615–1666, 1998.
[5] Battista Biggio, Ignazio Pillai, Samuel Rota Bulò, Davide Ariu, Marcello Pelillo, and Fabio Roli. Is data clustering in adversarial settings secure? arXiv preprint arXiv:1811.09982, 2018.
[6] [12] Stephen A. Cook. An overview of computational complexity. Commun. ACM, 26(4):123–140, 1996.
[7] Leo Breiman. Bagging predictors. Mach. Learn., 24(2):123–140, 1996.
[8] [13] Bita Darvish Rouhani, Huili Chen, and Farinaz Koushanfar. DeepSigns: An end-to-end watermarking framework for ownership protection of deep neural networks. In Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLoS), pages 485–497, Providence, RI, 2019.
[9] Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Ipguard: Protecting intellectual property of deep neural networks via fingerprinting the classification boundary. In Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS), pages 14–25, Virtual Event, Hong Kong, 2021.
[10] Hongjie Chen, Huan Zhang, Duane S. Boning, and Cho-Jui Hsieh. Robust decision trees against adversarial examples. In Proceedings of the 36th International Conference on Machine Learning (ICML), pages 1122–1131, Long Beach, CA, 2019.
[11] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. arXiv preprint arXiv:1712.05526, 2017.
[12] [14] Amjad Ali Alamr, Firdous Kausar, Jongsung Kim, and Changho Seo. A secure ecc-based rfid mutual authentication protocol for internet of things. The Journal of Supercomputing, 74(9):4281–4294, 2018.
[13] [15] Khosra Doan, Yingjie Lao, Weijie Zhao, and Ping Li. LIRA: learnable, imperceptible and robust backdoor attacks. In Proceedings of the 2021 IEEE/CVF International
Conference on Computer Vision (ICCV), pages 11946–11956, Montreal, Canada, 2021.

Chenlin Fan and Ping Li. Classification acceleration via merging decision trees. In Proceedings of the ACM-IMS Foundations of Data Science Conference (FODS), pages 13–22, Virtual Event, 2020.

Lixin Fan, Kum Woh Ng, and Chee Seng Chan. Rethinking deep neural network ownership verification: Embedding passports to defend against backdooring attacks. Advances in Neural Information Processing Systems (NeurIPS), pages 4714–4723, Vancouver, Canada, 2019.

Yoav Freund. Boosting a weak learning algorithm by majority. Inf. Comput., 121(2):256–285, 1995.

Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of online learning and an application to boosting. J. Comput. Syst. Sci., 55(1):119–139, 1997.

Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. The Annals of Statistics, 29(5):1189–1232, 2001.

Jerome H. Friedman, Trevor J. Hastie, and Robert Tibshirani. Additive logistic regression: A statistical view of boosting. The Annals of Statistics, 28(2):337–407, 2000.

Jerome H. Friedman, Trevor J. Hastie, and Robert Tibshirani. Response to evidence contrary to the statistical view of boosting. Journal of Machine Learning Research, 9:175–180, 2008.

Vincent Gran, Boris Ruf, Sylvain Lamprier, and Marcyn Detyniecki. Achieving fairness with decision trees: An adversarial approach. Data Science and Engineering, 5:99–110, 2020.

Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Evaluating backdooring attacks on deep neural networks. IEEE Access, 7:47230–47244, 2019.

Zecheng He, Tianwei Zhang, and Ruby Lee. Sensitive-sample fingerprinting of deep learning. In The 35th AAAI Conference on Artificial Intelligence (AAAI), pages 6103–6113, Montreal, Canada, 2018.

Mika Juuti, Sebastian Szyller, Samuel Marchal, and N. Asokan. PRADA: protecting watermarks into deep neural networks. In Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, 2018.

Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of online learning and an application to boosting. J. Comput. Syst. Sci., 55(1):119–139, 1997.

Vincent Grari, Boris Ruf, Sylvain Lamprier, and Marcin Detyniecki. Achieving fairness with decision trees: An adversarial approach. Data Science and Engineering, 5:99–110, 2020.

Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. arXiv preprint arXiv:1708.06733, 2017.

Weizhe Hua, Zhiyu Zhang, and G. Edward Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In Proceedings of the 55th Annual Design Automation Conference (DAC), pages 4:1–4, San Francisco, CA, 2018.

Mika Juuti, Sebastian Slytter, Samuel Marchal, and N. Asokan. PRADA: protecting against DNN model stealing attacks. In Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P), pages 512–527, Stockholm, Sweden, 2019.

Richard E. Korf. A complete anytime algorithm for number partitioning. Artificial Intelligence, 106(2):181–203, 1998.

Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. Adversarial machine learning at scale. In Proceedings of the 5th International Conference on Learning Representations (ICLR), Toulon, France, 2017.

Ping Li, Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, and Ian Goodfellow. Adversarial training improves ImageNet classification. In Proceedings of the 32nd International Conference on Machine Learning (ICML), Beijing, China, 2015.

Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Evaluating backdooring attacks on deep neural networks. IEEE Access, 7:47230–47244, 2019.

Chen Lin, Yiming Wang, Weijie Zhao, and Ping Li. Identification for deep neural networks: Simply adjusting few weights! In Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI), pages 11957–11965, New York, NY, 2020.

Vincent Grari, Boris Ruf, Sylvain Lamprier, and Marcin Detyniecki. Achieving fairness with decision trees: An adversarial approach. Data Science and Engineering, 5:99–110, 2020.

Zecheng He, Tianwei Zhang, and Ruby Lee. Sensitive-sample fingerprinting of deep learning. In Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI), pages 6103–6113, Montreal, Canada, 2018.

Mika Juuti, Sebastian Szyller, Samuel Marchal, and N. Asokan. PRADA: protecting against DNN model stealing attacks. In Proceedings of the 55th Annual Design Automation Conference (DAC), pages 4:1–4, San Francisco, CA, 2018.

Ping Li, Yiming Wang, Weijie Zhao, and Ping Li. Identification for deep neural networks: Simply adjusting few weights! In Proceedings of the 38th IEEE International Conference on Data Engineering (ICDE), Virtual Event, 2022.

Ping Li, Yiming Wang, Weijie Zhao, Peng Yang, and Ping Li. Deepauth: A din authentication framework by model-unique and fragile signature embedding. In Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI), Virtual, 2022.

Erwan Le Meur, Patrick Perez, and Gilles Tredan. Adversarial frontier stitching for remote neural network watermarking. Neural Computing and Applications, 32(3):9233–9244, 2020.

Jun-Ya Lee, Wei-Cheng Lin, and Yu-Hung Huang. A lightweight authentication protocol for internet of things. In Proceedings of the 2014 International Symposium on Next-Generation Electronics (ISNE), pages 1–2, 2014.

Bo Li, Yining Wang, Aarti Singh, and Yevgeniy Vorobeychik. Data poisoning attacks on factorization-based collaborative filtering. In Advances in Neural Information Processing Systems (NeIPS), pages 1885–1893, Barcelona, Spain, 2016.

Huiying Li, Emily Willson, Haitao Zheng, and Ben Y Zhao. Piracy resistant watermarks for deep neural networks. arXiv preprint arXiv:1910.01226, 2019.

Ping Li. Adb-boost: Adaptive base class boost for multi-class classification. In Proceedings of the 26th Annual International Conference on Machine Learning (ICML), pages 625–632, Montreal, Canada, 2009.

Ping Li. Robust logitboost and adaptive base class (abc) logitboost. In Proceedings of the Twenty-Sixth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI), pages 302–311, Catalina Island, CA, 2010.

Ping Li and Weijie Zhao. Fast ABC-Boost: A unified framework for selecting the base class in multi-class classification. arXiv preprint arXiv:2205.10927, 2022.

Ping Li and Weijie Zhao. Package for Fast ABC-Boost. https://github.com/ptbeev/abcboost, 2022.