EnclaveTree: Privacy-preserving Data Stream Training and Inference Using TEE

Qifan Wang  
The University of Auckland  
Auckland, New Zealand  
qwan301@aucklanduni.ac.nz

Shujie Cui  
Monash University  
Melbourne, Australia  
shujie.cui@monash.edu

Lei Zhou  
Southern University of Science and Technology  
Shenzhen, China  
zhou6@sustech.edu.cn

Ocean Wu  
The University of Auckland  
Auckland, New Zealand  
hwu344@aucklanduni.ac.nz

Yonghua Zhu  
The University of Auckland  
Auckland, New Zealand  
yzhu970@aucklanduni.ac.nz

Giovanni Russello  
The University of Auckland  
Auckland, New Zealand  
g.russello@auckland.ac.nz

ABSTRACT

The classification service over a stream of data is becoming an important offering for cloud providers, but users may encounter obstacles in providing sensitive data due to privacy concerns. While Trusted Execution Environments (TEEs) are promising solutions for protecting private data, they remain vulnerable to side-channel attacks induced by data-dependent access patterns. We propose a Privacy-preserving Data Stream Training and Inference scheme, called EnclaveTree, that provides confidentiality for user’s data and the target models against a compromised cloud service provider. We design a matrix-based training and inference procedure to train the Hoeffding Tree (HT) model and perform inference with the trained model inside the trusted area of TEEs, which provably prevent the exploitation of access-pattern-based attacks. The performance evaluation shows that EnclaveTree is practical for processing the data streams with small or medium number of features. When there are less than 63 binary features, EnclaveTree is up to ~10× and ~9× faster than naïve oblivious solution on training and inference, respectively.

CCS CONCEPTS

• Security and privacy → Software and application security.

KEYWORDS

Data stream, hoeffding tree, SGX enclave, data-oblivious

1 INTRODUCTION

Machine learning (ML) applications such as remote healthcare and activity recognition, have attracted a lot of attention as a major breakthrough in the practice of ML. These specific applications of ML are characterized by data streams: data is generated by various devices and usually arrive in a timely manner. For either training the ML model or inferring (i.e., predicting or evaluating) an unlabelled instance by the model, the data stream should be processed efficiently on-the-fly as data might arrive rapidly. The Hoeffding Tree (HT) model [15], a variation of the decision tree model, has become the standard for processing data streams.

To process data streams efficiently, a promising solution is to outsource the HT training and inference to cloud platforms [38, 55]. However, this poses a severe threat to data privacy and model confidentiality. For privacy-sensitive applications, such as in the healthcare domain, all data samples, the model, the inference output, and any intermediate data generated during the model training and inference should be protected from the Cloud Service Provider (CSP). In particular, when training a HT, the main operation is to classify each newly arriving data sample with the current tree and to count the frequency of different feature values. The access path over the tree and the statistical information generated when training the model, should be protected as they can be leveraged by an adversary to construct a near-equivalent HT [51].

Privacy-preserving data mining (PPDM) aims to protect the privacy of outsourced ML tasks by employing cryptographic primitives, such as Secure Multi-Party Computation (SMC) [13, 17, 32, 52, 59, 63] or Homomorphic Encryption (HE) [3, 6, 33, 58]. Nevertheless, most of the existing PPDM approaches cannot be adopted to process data streams, because they: ❶ cannot process complicated functions such as logarithm and exponential operations in an efficient way, which are fundamental to the HT model training; ❷ impose too heavy computation and communication overheads on the clients; ❸ leak statistical information and tree structures.

Table 1 summarizes the related work in this area. First of all, note that most of the existing approaches focus on generic decision trees and none of them can securely process data streams (column DS in Table 1). The approaches given in [14, 17, 18, 32, 47, 52, 59] are impractical for data streams because they require multiple rounds of interactions between client and server. While the approaches proposed in [6, 13, 58, 63] leak information about the model, such as the structure and the number of nodes of the tree. To the best of our knowledge, [55, 61] are the only approaches that focus on data streams and can provide some level of protection for the data, the target model and the inference results. The reason these works are not included in the table is because they do not focus on decision trees. Moreover, the main idea of these approaches is to randomly perturb the data distribution with noise. This approach is usually efficient but at the cost of accuracy loss due to a large amount of...
perturbations. Furthermore, since only part of information is perturbed, the attacker can still compromise the user’s privacy by retrieving the features through inference attacks [1].

Our goals. In this work, we aim to design an outsourced approach to train and infer data streams with HT model in a secure and efficient manner. Specifically, our approach should not only protect all the data samples and the model from the CSP but also any intermediate data generated during the training and inference, such as the frequency of different feature values and the access pattern.

Challenges. To achieve the goals, we propose a privacy-preserving data stream classification scheme called EnclaveTree. The basic idea of EnclaveTree is to employ the Intel Software Guard Extension (SGX) [11] to process privacy-sensitive operations on the CSP. In this section, we provide background information on Intel SGX, side-channel attacks, and the oblivious primitives we use in the rest of this paper.

2 BACKGROUND

In this section, we provide background information on Intel SGX, side-channel attacks, and the oblivious primitives we use in the rest of this paper.

---

### Table 1: Comparison of decision tree training and inference protocols

| Scheme                      | Support | Communication | Privacy |
|-----------------------------|---------|---------------|---------|
|                            | DS      | Training      | Inference | Rounds | Bandwidth | Complexity on Client | Data | IR | Model | AP |
| Du et al. [17], Vaidya et al. [52], Samet et al. [47] | ✓       | ✓             | ✓         | Ω(t_d) | Ω(t_m log n + n) | Ω(t_m + 1)n           | ●   | □ | ●      | □  |
| Xiao et al. [59], Emekci et al. [18] | ✓       | ✓             | ✓         | Ω(t_d) | Ω(t_m log n + n) | Ω(t_m + 1)n           | ●   | □ | ●      | □  |
| Hoogh et al. [14], Lindell et al. [52] | ✓       | ✓             | ✓         | Ω(t_d) | Ω(t_m log n + n) | Ω(t_m + 1)n           | ●   | □ | ●      | □  |
| Bost et al. [6], Wu et al. [58], Tai et al. [50], Kiss et al. [27] | ✓       | ✓             | ✓         | Ω(t_d) | Ω(t_m log n + n) | Ω(t_m + 1)n           | ●   | □ | ●      | □  |
| Cock et al. [13]           | ✓       | ✓             | ✓         | Ω(t_d) | Ω(t_m log n + n) | Ω(t_m + 1)n           | ●   | □ | ●      | □  |
| Akavia et al. [3]          | ✓       | ✓             | ✓         | Ω(t_d) | Ω(t_m log n + n) | Ω(t_m + 1)n           | ●   | □ | ●      | □  |
| Liu et al. [33]            | ✓       | ✓             | ✓         | Ω(t_d) | Ω(t_m log n + n) | Ω(t_m + 1)n           | ●   | □ | ●      | □  |
| **EnclaveTree**            | ✓       | ✓             | ✓         | Ω(t_d) | Ω(t_m log n + n) | Ω(t_m + 1)n           | ●   | ● | ●      | ●  |

DS denotes data stream. Privacy of data, intermediate results, model and access patterns are denoted by Data, IR, Model and AP, respectively. Ω(·) and O(·) denote the computation complexity of each party in the distributed setting and client, respectively. ●, □, and ○ denote the target is protected, part of the parameters of the target are leaked, and fails to protect the target, respectively. t_d, t_m, t_b, c and n represents the tree’s depth, the number of nodes, the binary representation length of data samples, constants and the number of data samples, respectively.
2.1 Intel SGX

A Trusted Execution Environment (TEE), such as the Intel Software Guard Extensions (Intel SGX) [11], protects sensitive data and code from privileged attackers who may control all the software, including the operating system and hypervisor. In Intel SGX-enabled machines, the CPU protects the confidentiality and integrity of code and data by storing them in an isolated memory region, called enclave. Intel SGX also supports remote attestation of an initialized enclave. It enables a remote party to verify an enclave identity and the integrity of the code and data inside the enclave.

2.2 Side-channel Attacks on Intel SGX

One issue of Intel SGX is that it still shares many resources with untrusted programs, e.g., CPU cache and branch prediction units, and relies on the underlying OS for resource management. As a result, Intel SGX is susceptible to side-channel attacks. In recent years, various side channels have been extensively exploited to infer secrets from enclaves, such as L1 cache [22, 36], page tables [7, 60], branch predictor [19, 25, 30], and the transient execution mechanism [28, 53, 54]. They infer secrets by mainly exploiting the data-dependent enclave access pattern at different granularity. For instance, with cache-timing attacks, the adversary can learn the enclave access pattern at cache line granularity.

Existing countermeasures are either hardware-based [42, 49] or software-based [2, 9, 41]. Hardware-based solutions, such as cache partitioning [62] and enclave self-paging [42], are efficient yet they require hardware modifications, which take a long period to be applied and cannot be retrofitted to existing hardware. In contrast, software-based solutions are more flexible. However, they generally leverage expensive normalisation or randomisation techniques, making them impractical. For instance, OBfuscuro [2] leverages ORAM operations to perform secure code execution and data access, which adds about 51× overhead to enclaves. It is desirable to protect sensitive data and operations from side-channel attacks with techniques that are specific to the enclave.

2.3 Oblivious Primitives

A library of general-purpose oblivious primitives, operating solely on registers whose contents are restricted to the code outside the enclave, has been introduced in previous work [29, 40, 46] and experimentally demonstrated that it is several orders of magnitude faster than previous ORAM-based approaches. In this work, we will use the following oblivious primitives:

- **Oblivious comparison**. ogreater and oequal, are used to compare variables and implemented with x86 instruction cmp.
- **Oblivious selection**. oselect, allows to conditionally select an element.
- **Oblivious assignment**. oassign, allows to conditionally assign variables. It specifically uses CMOVZ for equality comparisons and subsequently combine it with oassign to assign a value to the destination register.
- **Oblivious array access**. oaccess scans the array at cache-line granularity and obliviously load one element based on oassign, and then is optimized with AVX2 vector instructions [10].

3 OVERVIEW OF OUR APPROACH

In this section, we will describe the system model and design overview. We will conclude the section with discussing the threat model.

3.1 System Model

As shown in Fig. 1, EnclaveTree consists of 2 entities: the Data Owner (DO) and the Cloud Service Provider (CSP).

- The DO continuously receives data from devices, encrypts them, and outsources them to the CSP. The data samples could be labelled samples or unlabelled instances. In particular, labelled samples are used to train the model, while the unlabelled instances will be inferred with a label value by the model. The inference results are sent from the CSP to the DO.
- The CSP considered in EnclaveTree should have Intel SGX support, (e.g., Microsoft Azure [34] and Alibaba Cloud [4]). The CSP consists of a trusted and an untrusted component. The trusted component is represented by the SGX enclave (as shown in Fig. 1). This is where the models are trained and where the inference is performed. The untrusted component is any computational resources in the CSP Host that is outside the SGX Enclave. With the assistance of an enclave, the CSP trains the model with the data samples outsourced from the DO and classify them with the model.
do process the remaining operations in order to hide the enclave memory access pattern.

3.3 Threat Model

We assume the DO and the SGX enclave are fully trusted. The CSP host is untrusted and attempts to infer secrets, such as the tree structure and statistical information, by observing and analysing memory access pattern of the enclave. Moreover, the CSP can eavesdrop on the communication between the DO and the enclave. Note that rollback attacks [43], denial-of-service attacks [23] and other attacks based on physical information, such as electromagnetic, power consumption and acoustic are out of our scope.

The security analysis of EnclaveTree is given in Appendix A.

4 DATA AND MODEL REPRESENTATION

In this section, we provide some details on how a Hoeffding Tree (HT) is originally built and used for inference. Then, we will describe how the data and the HT are represented in our approach. To make things more concrete, we will use a simple running example throughout this paper. The example consists of building a HT to decide whether it is suitable to play tennis based on a weather dataset [18]. The example tree consists of 4 features and each feature has 2 or 3 possible values listed as follows: Outlook (Sunny, Overcast, Rain), Windy (True, False), Humidity (High, Normal), and Temperature (Hot, Cool). Each internal node of the tree is associated with a feature, and its possible values determine the branches of the node. The leaf nodes represent the label that has 2 values: either Yes or No. Fig. 2 shows how a HT is built for this example. In the rest of this paper we will use the notation shown in Table 2.

4.1 Hoeffding Tree

A decision tree consists of internal nodes (including the root) and leaves, where each internal node is associated with a test on a feature, each branch represents the outcome of the test, and each leaf represents a class label which is the decision taken after testing all the features on the corresponding path.

![Figure 2: A HT example and its extension after one round of training. The orange nodes are internal nodes which have been assigned with features. The green nodes are leaves. Each leaf has a label value for inference, and it stores statistical information for the features that have not been assigned to the path for training. Each branch is assigned with a feature value.](https://via.placeholder.com/150)
for each feature and label. The IG of a feature is derived from the frequencies of its possible \((value, label)\) pairs. For instance, for the left-most leaf of the tree in Fig. 2a, to compute \(G(T\text{emp})\) we need to count how many samples have been classified into the leaf-most leaf. These samples might contain the following pairs: \((\text{Hot, Yes}), (\text{Hot, No}), (\text{Cold, Yes}), (\text{Cold, No})\). Similarly, to compute \(G(\text{Windy})\), we need to count how many samples have got the following pairs: \((\text{True, Yes}), (\text{True, False}), (\text{False, Yes}), (\text{False, Yes})\). Each leaf records the frequencies of the pairs for unassigned features and updates them when receiving new samples.

Computing IG values is expensive due to the complex logarithm and exponentiation operations. Therefore, feature IGs of each leaf are computed when the leaf receives every \(n_{\text{min}}\) samples, where \(n_{\text{min}}\) is a pre-defined parameter.

Overall, we can summarize the main operations of building a HT model with the following steps:

1. classifying new arrivals into leaves with current HT model; and performing steps 2 and 3 for each leaf that gets new data samples;
2. updating the frequency of each \((value, label)\) pair for unassigned features;
3. checking if the leaf has received \(n_{\text{min}}\) data samples, and performing steps 4-6 if true;
4. computing the IG value for each unassigned feature;
5. checking if the top two highest IG values satisfy the Hoeffding Bound;
6. if true, converting the leaf node into an internal one using the feature with the highest IG value.

Inference operations start from the root of the tree. An unlabelled instance is tested with the feature at each internal node and then moved down the tree along the edge corresponding to the instance’s value for that feature. When a leaf node is reached on the path, the label associated with it is assigned to the instance.

### 4.2 Data Representation

We assume that \(S = (s_1, s_2, \cdots, s_d)\) represents a sequence of \(d\) features, with each feature \(s_i\) having \(m_i\) possible values: \(V_{s_i} = \{v_{i,1}, v_{i,2}, \cdots, v_{i,m_i}\}\), where \(1 \leq i \leq d\). We use the one-hot encoding technique [24] to encode each value \(v_{i,j}\) into a bit string, where \(1 \leq j \leq m_i\). More precisely, a \(m_i\)-bit string is used to represent a value \(v_{i,j}\), where the \(j\)-th bit of the string is 1 and all the other bits are 0. For labelled data samples, the last feature \(s_d\) is the label, and we will use the same bit representation for possible label values. Therefore, a data sample \(D\) is represented as a bit string with \(M = \sum_{i=1}^{d} m_i\) bits.

Fig. 3a shows a concrete example for the encoding of 5 features. In the example, \(d = 5\) and \(S = (\text{Outlook, Windy, Humidity, Temp, Label})\). The first feature \(s_1 = \text{Outlook}\) has 3 values (i.e., \(m_1 = 3\)): \(v_{1,1} = \text{Sunny}, v_{1,2} = \text{Overcast}\) and \(v_{1,3} = \text{Rain}\), and they will be encoded to: 001, 010, and 100, respectively. The last feature \(s_5 = \text{Label}\) has 2 values (i.e., \(m_5 = 2\)): \(v_{5,1} = \text{Yes}\) and \(v_{5,2} = \text{No}\), and they will be encoded to 01 and 10, respectively. A data sample \(D = (\text{Sunny}, \text{True}, \text{High}, \text{Hot, No})\) will be encoded into \((0010101110)\), consisting of 11 bits (i.e., \(M = 11\)).

Based on the bit-wise representation, we can query if a data sample contains \(x\) given feature values by calculating the inner product between its encoding and a \(M\)-bit mask. Specifically, for each value \(v_{i,j}\) to be queried, we set its corresponding bits in the mask to 1 and set all the other bits to 0. In this way, the inner product should be equal to \(x\) if the sample contains all the \(x\) values. In our example, if we want to check that a sample \(D\) contains \((\text{Sunny, Yes})\), the mask will be set to \((00100000001)\), the inner product will be equal to 2 if both values are contained in \(D\).

Here, we stress that our work focuses on training categorical features. Numerical features can be converted into categorical ones using methods such as discretization [16]. Specifically, numerical values of a feature can be grouped into discrete bins. For example, if we wanted to group the values for \(\text{Temp}\) 2 categories this could be a possible discretization: \(\text{Cool}\) for temperatures below \(25^\circ\text{C}\), \(\text{Hot}\) for temperatures equal or above \(25^\circ\text{C}\).

### 4.3 Model Representation

One of the main contributions of \(\text{EnclaveTree}\) is the novel way in which we represent the model as a matrix, and perform the HT training and inference as a matrix multiplication to hide the access pattern. Fig. 3b shows a simplified matrix representation of the model with the value expressed as strings of characters and its corresponding tree representation. Columns in the matrix map to paths of the tree. Each column contains \(d - 1\) elements where the
i-th element is the value of feature $s_i$ assigned to the corresponding path\(^1\). In particular, if a feature $s_i$ has not been assigned to the specific path, the i-th element of the column is set to ‘*’. This will be converted into specific feature values with the subsequent training.

The last two columns in the matrix are dummy columns. In order to hide the number of tree paths from side-channel attacks, i.e., the number of columns in the matrix, we add a number of dummy columns into the matrix. The elements in dummy columns can be of any value. More details on how dummy columns are generated will be provided in Section 5.2.

To make things more concrete, let’s look at the example in Fig. 3b. The matrix representing our model consists of 4 real columns and 2 dummy columns. The first column contains the elements (Sunny, *, High, *). This indicates that the value Sunny for feature $s_1$ (i.e., Outlook) and value High for feature $s_3$ (i.e., Humidity) are assigned to the first path of the tree. Likewise, the third column (Overcast, *, *, *) indicates that only the value Overcast has been assigned to the third path for feature $s_1$, while the remaining 3 features have not been assigned. The right-hand side of Fig. 3b depicts the model currently stored in the matrix if it were represented as a tree.

As we said, the matrix in Fig. 3b is a simplified representation of how the model is stored in EnclaveTree. Fig. 3c shows how the matrix is actually stored in the enclave as a collection of bit strings. Using the one-hot encoding technique, the matrix $M_t$ only contains 0 and 1 bit. For instance, looking at the first column in the matrix, the values Sunny and High are encoded into 001 and 01, respectively; while the value ‘*’ for feature is embedded into a string with 0 bits.

Each column of matrix only contains $d-1$ values: these are the values that could be assigned to features excluding the values for the labels. Thus each column of $M_t$ has $M - m_d$ bits, where $m_d$ is the number of values for labels. Assuming the model has $P_{\text{real}}$ real columns and EnclaveTree inserts $P_{\text{dummy}}$ dummy columns into $M_t$, the total number of columns in the matrix is $P = P_{\text{real}} + P_{\text{dummy}}$. Therefore, the size of $M_t$ is $(M - m_d) \times P$.

For HT training, EnclaveTree also stores the statistical information for each leaf, which is required for computing the IG value. In EnclaveTree, the statistical information of each leaf is stored in a 2d array Leaf. Because the number of leaves of the model should also be protected, we store in Leaf some dummy values representing dummy leaves. Considering that each column in the model could represent a HT path with a leaf, then Leaf contains $P$ 1d arrays: $P_{\text{real}}$ arrays for real leaves and $P_{\text{dummy}}$ arrays for dummy leaves. Precisely, the $p$-th array, $\text{Leaf}[p]$, contains all features for the $p$-th leaf, where $p \in [1, P]$. The actual values stored in Leaf are the frequency values defined as $c_{\text{value, label}}$, for each (value, label) pair. Note that only the (value, label) pair of the features that have not been assigned to a path will be updated and used for computing IG. Storing the pairs of all features for all leaves ensures $\text{Leaf}[p]$ is the same for all leaves, which is $L = \sum_{i=1}^{d-1} m_i \times m_d$. In this way, the entire model structure is protected from side-channel attacks.

Both $M_t$ and Leaf are stored within the enclave in plaintext.

5 HT TRAINING AND INFERENCE IN ENCLAVETREE

In this section, we explain how EnclaveTree obliviously trains the HT model and securely inferences unlabelled data instances.

Although the main focus of this section is about training and inference for a single HT model, EnclaveTree can be easily extended to support a Random Forest (RF) model by performing the HT training and inference over several trees. We give the details of this extension for RF in Appendix B.2.

5.1 Setup

As the first step in the setup, the DO establishes a secure channel with an enclave instance in the CSP to share a secret key $sk$. For HT training and inference, all the data transmitted between the DO and the enclave will be encrypted with $sk$ and a semantically secure symmetric encryption primitive, e.g., AES-GCM. During the setup, DO also securely shares the features $S$ and the values $V_s$ of each feature to the enclave.

5.2 Oblivious HT Training

In Section 4.1, we have summarised the 6 steps for performing the HT training. In order to hide the tree structure during the training, the 6 steps are modified in EnclaveTree as below:

1. classifying new arrivals into leaves with current HT model;
2. updating the frequency of each (value, label) pair for each feature for all leaves, not only for the leaves that receive new data samples;
3. checking if each leaf has received $n_{\text{min}}$ data samples, and performing steps 4-6 if true;
4. computing the IG value for all features, not just for unassigned features;
5. checking if the top two highest IG values satisfy the Hoeffding Bound;
6. if true, converting the leaf node into an internal node using the feature with the highest IG value; otherwise, performing indistinguishable dummy operations.

Here we present how each step is performed obliviously in EnclaveTree in details. We will use as an example the case illustrated in Fig. 4.

To protect the access pattern from side-channel attacks, EnclaveTree performs the first two steps with a matrix multiplication. Basically, EnclaveTree converts a batch of data samples to a matrix $M_d$, generates a query matrix $M_q^p$ for each column $p$ in matrix $M_t$, and computes $M_q^p \leftarrow M_d \times M_q^p$. The elements of the resulting matrix $M_q^p$ will be used to update the frequency information in the $\text{Leaf}[p]$ array.

In the following, we take the first column of $M_t$, denoted as $M_t[:, 1]$, as an example. The different steps are shown in Fig. 4.

**Data Samples Matrix.** To improve efficiency, we perform the 6 steps of HT training when a batch of $N$ data samples has been stored in the Training Buffer on the CSP. The Training Buffer stores the data samples outside the enclave. Note that the buffer size could be larger than $N$. When $N$ data samples are cached in the Training Buffer, EnclaveTree loads these samples into the enclave, and for each round of training, converts them into a matrix $M_d$. Recall

\(^1\)Note that the order of features in each column is fixed and same to the order defined in $S$, i.e., the i-th value of each column must be a value of feature $s_i$.\(^\)}
that EnclaveTree represents the data sample as an M-bit string. After the N data samples are imported in the enclave and decrypted, EnclaveTree packs them into a NxM matrix \( M' \), where each row of \( M' \) is a data sample encoded as a bit string. Fig. 4a shows an example where \( N = 4 \), and each data sample is represented as a 11-bit string. Thus, the resulting size of the matrix \( M' \) is 4x11.

**Query Matrix.** Assume column \( M'_i[:, p] \) contains \( r_p \) assigned feature values and \( u_p \) unassigned features. Our next step is to query whether any data sample in the current batch contains (i) the \( r_p \) assigned feature values in the column \( M_i[:, p] \), and (ii) a \((\text{value}, \text{label})\) pair for any of the \( u_p \) features that are not still assigned.

We perform this query by means of a matrix multiplication and the result of this multiplication will be another matrix \( M''_i \). The elements in \( M''_i \) are then used to update the frequencies of the queried \((\text{value}, \text{label})\) pairs in the array Leaf[p].

The process of generating a query matrix \( M''_i \) for a given column \( M'_i[:, p] \) is then reduced to define a set of M-bit masks which form the columns in \( M''_i \). Each mask can only check one case. Thus the number of masks, i.e., the number of columns of matrix \( M''_i \), is determined by the possible values of unassigned features and the possible values of the label. In more detail, for \( M'_i[:, p] \), \( M''_i \) are determined by (i) the \( r_p \) assigned feature values (these will be the same across all the columns of the query matrix); and (ii) all the possible combinations of the \((\text{value}, \text{label})\) pairs for the \( u_p \) unassigned features.

To make things more concrete, let us look at Fig. 4b, where both the model matrix \( M_i \) and the query matrix \( M''_i \) for column \( M_i[:, 1] \) are presented in human-readable and bit-string forms. As we can see from the figure, \( M_i[:, 1] \) includes 2 assigned feature values (i.e., Sunny and High), and 2 unassigned features (i.e., Windy and Temp). This means that \( r_1 = 2 \) and \( r_1 = 2 \).

The number of columns (i.e., masks) in \( M''_i \) is defined as \( L' = \sum_{i=1}^{n} m_i \) which \( m_i \) are the possible values of each unassigned feature and \( m_d \) are the possible values of the label. This means the size of \( M''_i \) is \( M \times L' \).

In the example in Fig. 4b, as both the unassigned features, Windy and Temp, and the label Label have 2 possible values (i.e., \( V_{\text{Windy}} = \{\text{True, False}\} \), \( V_{\text{Temp}} = \{\text{Hot, Cool}\} \), and \( V_{\text{Label}} = \{\text{Yes, No}\} \)), the total number of masks that we need to query is given by the following: \( |V_{\text{Windy}}| + |V_{\text{Label}}| + |V_{\text{Temp}}| + |V_{\text{Label}}| = 2 \). In other words, for column \( M_i[:, 1] \) we need a query matrix \( M''_i \) of 8 columns with the values for each column shown in Fig. 4b.

**Matrix multiplication.** By computing \( M' \times M''_i \), we get a \( N \times L' \) result matrix \( M''_i \). We use \( M''_i[n, k] \) to represent its element at the \( n \)-th row and \( k \)-th column, where \( n \in [1, N] \) and \( k \in [1, L'] \). \( M''_i[n, k] \) is the inner product between the \( n \)-th data sample and the \( k \)-th mask. This value represents the number of values in \( n \)-th data sample that match the values in the \( k \)-th column of the query matrix. We are interested in finding the data samples that fully match the values defined in \( M''_i[n, k] \): the \( r_p \) assigned feature values in \( M_i[:, p] \) and the \((\text{value}, \text{label})\) pair that we are querying for. In other words, if \( M_i[n, k] = r_p + 2 \) the \( n \)-th sample matches the mask \( M''_i[k] \). To be more concrete, let us look at a specific case presented in Fig. 4c. Recall that we are querying for \( M_i[:, 1] \): this column has two fixed values Sunny and High. Thus we are looking for a matching value of \( r_1 = 2 \). In Fig. 4c, we can see all the elements \( M''_i[n, k] \) = 4 highlighted in red boxes.

The next step is to update the frequency information of each \((\text{value}, \text{label})\) pair contained in the Leaf arrays. This is performed by scanning each column of the result matrix \( M''_i \) and checking how many elements in each column is equal to \( r_p + 2 \). For instance, in Fig. 4c, the first column of \( M''_i \) contains two matches. The corresponding frequency value \( c_{(\text{True, No})} \) in Leaf[1] is increased by
2. Here the enclave uses a mapping $\sigma$ to map the columns of $M^f_t$ to the elements in $Leaf[p]$.

EnclaveTree executes these operations obliviously, otherwise an adversary could use side-channel attacks to learn which data sample contains which pair. Precisely, EnclaveTree linearly scans each column of $M^f_t$, using $\text{seq1}$ to check how many elements in $M^f_t[\cdot, k]$ equal to $r_k + 2$. At the last step, the frequency counts are added to the corresponding $c(\text{value, label})$ value in the relevant leaf array using assign.

During the training, EnclaveTree requires to access all the columns of $M_t$ and generate a query matrix for each column. Even if this operation is executed in the enclave, with side-channel attacks, an adversary could infer information about the model (e.g., the number of columns, which maps to the number of HT paths). Likewise, when accesses are made to $Leaf$ for updating the frequency information, the adversary could also infer the number of leaves. To prevent such a leakage, EnclaveTree inserts dummy columns and dummy arrays into $M_t$ and $Leaf$ during the setup. EnclaveTree uses a $P$-bit string $\text{isDummy}$ to mark if $M_t[\cdot, p]$ and $Leaf[\cdot]$ is real or dummy.

![Figure 5: The model after one round training.](image)

The parts set in red are those modified after one round of training.

Oblivious model construction. Once the frequency of each pair has been updated, the IGs of those leaves that have received $n_{\text{min}}$ data samples can be securely computed within the enclave. However, the last 2 steps should be performed obliviously as they involve memory access.

For step 5, the enclave uses $\text{ogreater}$ and $\text{seq1}$ to obliviously find out the two features with the highest IG values for each leaf. Assume the two features are $s_a$ and $s_b$ for $Leaf[p]$, where $\overline{G(s_a)} > \overline{G(s_b)}$. The enclave uses $\text{ogreater}$ to check if $\overline{G(s_a)} - \overline{G(s_b)} > \epsilon$. If true, the enclave selects the feature $s_a$ using $\text{select}$ and performs the last step, i.e., converting $Leaf[p]$ into an internal node with $s_a$.

In terms of the tree structure, converting a leaf into an internal node means assigning $s_a$ to the leaf, outputting $m_a$ branches with $m_a$ new leaves, and assigning the $m_a$ values of feature $s_a$ to the new branches. In terms of the matrix model in EnclaveTree, the enclave modifies $M_t$ and $Leaf$ with the following extensions.

$M_t$ extension: To hide whether the model is extended after each round of training, EnclaveTree converts $m_a - 1$ dummy columns into real ones by resetting $\text{isDummy}$, rather than adding new columns into $M_t$. In more details, EnclaveTree first copies the values of $M_t[\cdot, p]$ to $m_a - 1$ dummy columns, and then assigns the $m_a$ values of feature $s_a$ to $M_t[\cdot, p]$ and the $m_a - 1$ dummy columns with assign. Fig. 5 shows how $M_t$ is changed when $Leaf[p]$ is converted into an internal node with feature $Temp$. In the example, $m_{\text{Temp}} = 2$, thus only one dummy column, $M_t[\cdot, 5]$, is converted into a real one. The last 2 bits of $M_t[\cdot, 1]$ and $M_t[\cdot, 5]$ are changed to 01 and 10, respectively (the encoding for $\text{Hot}$ and $\text{Cool}$, respectively).

$Leaf$ extension: As $m_a$ new leaves are added, the leaf array $Leaf$ should also be updated. Similarly, EnclaveTree first converts $m_a - 1$ dummy arrays into real ones by initializing all the possible $c(\text{value, label})$ of unassigned features to 0. The original leaf $Leaf[p]$ will be used to store the statistical information of the new $p$-th leaf, and its each $c(\text{value, label})$ is set to 0.

During the setup, the enclave generates a number of dummy columns and leaves in $M_t$ and $Leaf$, respectively. As dummy values in both the model and the $Leaf$ arrays are processed as real values, a large number of dummies will degrade the performance. To balance efficiency with security, EnclaveTree periodically generates new dummies. In detail, after $\gamma$ extensions, EnclaveTree checks the number of remaining dummy values, and if this value is below a given threshold $T$, EnclaveTree generates new dummies. The threshold $T$ should ensure there are enough dummies for $\gamma$ extensions. In the worst case, all of the $\gamma$ leaves are split and generate $\gamma \times (m_{\text{max}} - 1)$ new leaves, where $m_{\text{max}} = \max(m_1, \ldots, m_d)$. We thus set $T = \gamma \times (m_{\text{max}} - 1)$.

5.3 Other oblivious HT Inference

One of the features of data stream classifications is that unlabeled data instances can be received for inference at any time. In other words, there is not a clear separation between a training and an inference phase. As such, EnclaveTree has to be able to support inference operations while the model is being trained.

The target of HT inference is to return a classifying label value for each data instance to the DO. Before classifying any data instance, EnclaveTree has to define the label values in the current model. Data samples with different label values could be classified into the same leaf during the training. The label value with the highest frequency will be used as the label value of the leaf. For the $p$-th leaf, the label value that has the highest frequency can be obtained by checking the $c(\text{value, label})$ in $Leaf[p]$ with oblivious primitives.

To protect the enclave access pattern, EnclaveTree also performs the HT inference with a matrix multiplication. In more detail, the Oblivious Inference sub-component of EnclaveTree processes...
a batch of instances each time. Assume the batch size for HT inference is \( N' \). After loading and decrypting \( N' \) data instances, \( \text{EnclaveTree} \) converts the instances into a matrix \( M_i \). \( \text{EnclaveTree} \) also represents each data instance with a bag of bits. Compared with data samples, the bit string of a data instance only has \( M - m_d \) bits as the data instance does not have label values. Thus, the size of \( M_i \) is \( N' \times (M - m_d) \). For instance, in Fig 6, each column of \( M_i \) has 9 bits.

\( \text{EnclaveTree} \) performs the inference by computing \( M'_r \leftarrow M_i \times M_L \). Since the size of \( M_i \) and \( M_L \) are \( N' \times (M - m_d) \) and \( (M - m_d) \times P \) respectively, the size of \( M'_r \) is \( N' \times P \). The element \( M'_r(n,p) \) indicates whether the \( n \)-th data instance belongs to the \( p \)-th path, where \( n \in [1, N'] \). If this is the case, then \( M'_r(n,p) = \tau_p \). \( \tau_p \) can be easily obtained by checking how many 1 bits\(^2\) are in the \( p \)-th column of \( M_i \).

To check which path the \( n \)-th data instance belongs to, the enclave scans the \( n \)-th row of \( M'_r \) and checks if \( M_i[n,p] = \tau_p \) with equal. If this is true, then the label value of the \( p \)-th leaf will be the inference result for the \( n \)-th data instance. Finally, the enclave encrypts the \( N' \) labels with sk and sends them to the DO.

### 6 IMPLEMENTATION AND EVALUATION RESULTS

In this section, we first describe the implementation of \( \text{EnclaveTree} \). We then describe the evaluation test-bed we used for running our experiments. Finally, we conclude this section with a detailed performance analysis.

#### 6.1 Implementation

The prototype of \( \text{EnclaveTree} \) is implemented in C++ based on the machine learning library mlpack \[12\]. Mlpack implements the original HT algorithm (also known as Very Fast Decision Tree, VFDT) given in \[15\]. We modify both the training and inference into matrix-based processes according to our approach. To make the algorithm oblivious, we implemented oblivious primitives with in-line assembly code (as done in \[29, 40, 44\]).

#### 6.2 Experiment Setup

**Testbed.** We evaluated the prototype of \( \text{EnclaveTree} \) on a desktop with AVX2 and SGX support, where AVX2 feature is required for oaccess. The desktop contains 8 Intel i9-9900 3.1GHZ cores and 32GB of memory (~93 MB EPC memory), and runs Ubuntu 18.04.5 LTS and OpenEnclave 0.16.0.

**Baselines.** To the best of our knowledge, there is no other approach in the wild that can be used for a performance comparison with \( \text{EnclaveTree} \). Therefore, to better show the performance of \( \text{EnclaveTree} \), we implemented and evaluated 3 baseline cases named Insecure, SGX, and Oblivious SGX. Insecure baseline does not provide any protection and performs the traditional HT training and inference in plaintext where each data sample is classified level by level from the root to a leaf node \[15\]. Note that this baseline is performed without using SGX enclaves and in plaintext therefore it does not provide any security. SGX baseline performs the traditional HT training and inference within an enclave but without protecting the access pattern. By comparing the performance of the first two baselines, we can see the overhead incurred by using SGX. To protect the enclave access pattern, Oblivious SGX baseline obviously performs the traditional HT training and inference within the enclave with oblivious primitives. We leverage the strategy used in \[29\] for implementing Oblivious SGX, where the nodes of each level are stored in an array and the target node is obliviously accessed with oaccess. Moreover, dummy nodes are generated to hide the real number of nodes in each level.

When outside the enclave, the data samples are encrypted with 128-bit AES-GCM in SGX, Oblivious SGX, and \( \text{EnclaveTree} \).

All the experiment results presented in the following are average over 100 runs.

**Batch size.** The batch size \( N \) affects the performance of HT training and also the inference accuracy. As shown in Fig. 7, the performance of HT training improves at the increase of \( N \), whereas the accuracy of the model decreases with the increase of \( N \). \( \text{EnclaveTree} \) processes each batch of data samples in one step, which means the 6 steps of the HT training are performed once every \( N \) data samples. As a result, less computation is required when \( N \) gets larger, yet the best moment to covert leaves to internal nodes could be missed. From Fig. 7, we can also notice that when \( N < 128 \) the accuracy of the model decreases very slightly (Fig. 7a) but the decrease in runtime overhead is much more dramatic especially when considering 63 features (Fig. 7b).

For \( N = 100 \), the accuracy of the model is almost the same as for \( N = 1 \). Thus, in the following experiments, we set \( N = 100 \).

#### 6.3 Evaluation on Real Datasets

We first evaluated the performance of HT training with 3 real datasets that are widely used in the literature: Adult dataset, Record Linkage Comparison Patterns (REC) dataset, and Covertype dataset. They are obtained from UCI Machine Learning Repository \[3\]. The details of each dataset are shown in Table 3. In particular, we use the Adult and REC datasets to evaluate the performance of HT training, and use the REC dataset to test the performance of RF training, where 100 trees are trained and each tree consists of 7 features. The results are shown in Table 4. For Adult and REC, \( \text{EnclaveTree} \) outperforms Oblivious SGX by \( \sim 1.7 \times \) and

---

\[2\] The one-hot encoding ensure that the encoded value for each feature has only one bit set 1.

\[3\] https://archive.ics.uci.edu/ml/
datasets, we can see that the datasets usual contain dozens of features. The observation presented in [39, 57] also shows that dozens of features, e.g., 10, 20, or 30, are usually enough to reflect the distribution of the dataset. However, to better analyze the performance of EnclaveTree, in our tests we set the number of features to range between 3 and 127.

**Performance of HT training.** To measure the training performance of EnclaveTree, we performed two sets of experiments: 1) first we fixed the number of features while we changed the number of data samples; and 2) we fixed the number of data samples while we changed the number of features.

In the first set of experiments, we set the number of features to 31, which is large enough to cover most of the data stream scenarios, and changed the number of data samples from \(1 \times 10^4\) to \(5 \times 10^4\) samples. In the second test, we fixed the number of data samples to \(5 \times 10^4\) and increased the number of features from 3 to 127. For the same settings, we compare the performance of EnclaveTree with the other three baselines and the results are presented in Fig. 8.

Fig. 8a shows the execution time in seconds to perform the training with fixed features. From the results we can see that
EnclaveTree needs less time than Oblivious SGX but more time than SGX. Precisely, EnclaveTree outperforms Oblivious SGX by $4.03\times$, $3.02\times$, $2.86\times$, $2.58\times$, $2.29\times$, but incurs $\sim 6\times$, $\sim 9\times$, $\sim 10\times$, $\sim 11\times$, $\sim 13\times$ overhead for protecting the access pattern when compared to SGX for the five cases, respectively.

Fig. 8b shows the results when we fix the data sample size and vary the number of features. As expected, the training time increases with the increase of the number of features. It is interesting to note that for less than 63 feature, EnclaveTree execution time is better than Oblivious SGX. However, with more than 63 features, Oblivious SGX outperforms EnclaveTree in terms of execution times. The main reason of this increase in execution time is the increase in size for the matrices $M_p$, $M_q$ and $M^p$. These matrices become larger at the increase of the number of features, and this increases the running time for performing the matrix multiplication to get $M_r$.

Performance of HT inference. To evaluate the performance of inference, we also conducted two sets of experiments: 1) first, we fixed the number of features to 31 and changed the number of data samples from $1\times10^4$ to $5\times10^4$; and 2) then we fixed the data samples to $5\times10^4$ and changed the number of features from 3 to 127. In both sets of experiments, we set the batch size $N^\prime = 100$, i.e., 100 data instances are classified with one matrix multiplication. The results for both experiment sets are shown in Table 5 and Fig. 9, respectively.

From both Table 5 and Fig. 9, we can see that despite being the most secure of all the other baselines, the HT inference in EnclaveTree is very comparable to that of SGX (EnclaveTree performance is even better than SGX in some cases). The results also show that EnclaveTree is faster than Oblivious SGX (up to $\sim 7.23\times$ times).

7 RELATED WORK

In this section, we review existing privacy-preserving approaches for general ML algorithms and for data stream classification.

7.1 Privacy-preserving Machine Learning

Cryptography-based Solutions. Most of the existing privacy-preserving works [3, 14, 17, 18, 32, 33, 47, 52, 56, 59] rely on cryptographic techniques, such as SMC and HE. Compared with EnclaveTree, these schemes require multiple rounds of interaction between different participants. The schemes proposed in [17, 18, 32, 47, 52, 59] leak the statistical information and/or tree structures to the CSP. Moreover, as shown in [40], these cryptographic solutions incur heavy computational overheads. None of these works is suitable for data stream classification.

TEE-based Solutions. In recent years, advances in TEE technology have enabled a set of exciting ML applications such as Haven [5] and VC3 [48]. However, TEE solutions (e.g., Intel SGX) are vulnerable to a large number of side-channel attacks. Decision tree is vulnerable to those attacks as it induces data-dependent access patterns when performing training and inference tasks inside the enclave. Raccoon [46] proposes several mechanisms for data-oblivious execution for TEE to prevent these attacks. Ohrimenko et al. [40] propose to make the decision tree inference oblivious with oblivious primitives. Motivated by [40], Secure XGBoost [29] makes both the XGBoost model (a variant of the decision tree) training and inference oblivious with oblivious primitives. Combining TEE with oblivious primitives can prevent side-channel attacks and achieve better performance than cryptographic-based solutions. However, the use of oblivious primitives still leads to prohibitive performance overheads. EnclaveTree significantly reduces the need of using oblivious primitives because the access pattern to the model is hidden by the use of matrix multiplication. We only use oblivious primitives to process the results of the result matrices (i.e., $M_r$ and $M^\prime_r$) and to access to the Leaf array. Another issue is that both these approaches have not been designed to process data streams. Ohrimenko et al.’s solution only focuses on inferences. Secure XGBoost supports generic decision tree models and is not designed for HT.

7.2 Privacy-preserving Data Stream Mining

In the literature, several works have focused on protecting data stream privacy [8, 26, 31, 64]. However, they mainly focus on protecting the data distribution by adding noise. In more detail, these works leverage anonymization and data perturbation techniques to perturb the data and thus defend against attacks exploring the relationships across many features in data stream.

Few works have considered protecting the training process and the generated model in data stream classification. For instance, the solution proposed in [61] works on multiple stream sources to build a Naïve Bayesian model. They minimize the privacy leakage that could be incurred in the data exchange among data owners and do not consider the model privacy. [55] provides privacy protection for CNN inference with data stream but similarly the privacy of model and training process is not their focus. While these two works focus on data streams, neither of these two schemes focus on data stream classification using HT. Moreover, the main drawback of both approaches is that frequently adding noise reduces the model accuracy which may require frequent reconstructions of the model. Another issue is that an attacker could infer sensitive information from the data stream, such as the user’s identity, the locations a commuter visits and the type of illness a patient suffers from, by deploying various inference-based attacks [1, 8, 26].

8 CONCLUSION AND FUTURE WORK

We presented EnclaveTree, a practical, the first privacy-preserving data stream classification framework, which protects user’s private information and the target model against access-pattern-based attacks. EnclaveTree adopts novel matrix-based data-oblivious algorithms for the SGX enclave and uses x86 assembly oblivious primitives. EnclaveTree supports strong privacy guarantees while achieving acceptable performance overhead in privacy-preserving training and inference over data streams. As future work to improve EnclaveTree performance, we will investigate two potential solutions: (a) distribute the computation across multiple enclaves on different machines to perform matrix multiplications in parallel, and (b) securely outsource the matrix multiplication to GPUs.
ACKNOWLEDGMENTS
Russello would like to acknowledge the MBIE-funded programme STRATUS (UOWX1503) for its support and inspiration for this research.

REFERENCES

[1] Charu C Aggarwal. 2005. On k-anonymity and the curse of dimensionality. In VLDB, Vol. 5, 901–909.
[2] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinjun Zhang, Insik Shin, and Byoungyoun Lee. 2019. OBFSUCRU: A Commodity Obfuscation Engine on Intel SGX. In 26th Annual Network and Distributed System Security Symposium. NDSS 2019, San Diego, California, USA, February 24-27, 2019. The Internet Society.
[3] Adi Akavia, Max Leibovich, Yehezkel S Resheff, Roey Ron, Moni Shahar, and Margarita Vald. 2019. Privacy-Preserving Decision Tree Training and Prediction against Malicious Server. IACR Cryptol. ePrint Arch. 2019 (2019), 1282.
[4] Alibaba. 2020. Alibaba Cloud Security White Paper. https://www.alibabacloud.com/
[5] Andrew Baumann, Marcus Feinando, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. ACM Transactions on Computer Systems (TOCS) 33, 3 (2015), 1–26.
[6] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. 2015. Machine learning classification over encrypted data. In NDSS, Vol. 4324. 4325.
[7] Jo van Budek, Nico Wiesbrodt, Rüdiger Kapitzka, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In USENIX Security 2017. USENIX Association, 1041–1056.
[8] Mahowaga Arachchige Pathum Chamikara, Peter Bertok, Dongxi Liu, Seyit Camtepe, and Ibrahim Khalil. 2019. An efficient and scalable privacy preserving algorithm for big data and data streams. Computers & Security, 87 (2019), 101570.
[9] Guoxing Chen, Wenhao Wang, Taixuan Chen, Santu Chen, Yinjun Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. 2018. Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races. In 2018 IEEE Symposium on Security and Privacy. SP, 178–194.
[10] Intel Corporation. 2016. Intel (r) 64 and ia-32 architectures software developer’s manual. Combined Volumes, Dec (2016).
[11] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. IACR Cryptol. ePrint Arch. 2016, 6 (2016), 1–118.
[12] Ryan R. Curtin, Marcus Edel, Mikhail Likhomanok, Yannis Mentekidis, Sumedh Ghaisas, and Shangtong Zhang. 2018. mlpack 3: A fast, flexible machine learning library. Journal of Open Source Software 3 (2018), 726. Issue 26.
[13] Martine De Cock, Rafael Dowsley, Caleb Horst, Raj Katti, Anderson CA Nascimento, Sebastiaan de Hoogh, Berry Schoenmakers, Ping Chen, and Harm op den Akker. 2020. Bthestunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2020, 1 (2020), 321–347.
[14] David Harris and Sarah Harris. 2010. Digital design and computer architecture. Morgan Kaufmann.
[15] Tszman Huo, XiaoMeng Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhao, and Minghui Li. 2020. Blbethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2020, 1 (2020), 321–347.
[16] Georgios Kellaris, Stavros Papadopoulos, Xiaokui Xiao, and Dimitris Papadatos. 2014. Differentially private event sequences over infinite streams. Proceedings of the VLDB Endowment 7, 12 (2014), 1155–1166.
[17] Ágnes Kiss, Masoud Naderpour, Jian Liu, N Atakan, and Thomas Schneider. 2019. SoK: Modular and efficient private decision tree evaluation. Proceedings on Privacy Enhancing Technologies 2019, 2 (2019), 187–208.
[18] Paul Kocher, Jann horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Peschler, Michael Schwarz, and Yuval Yarom. 2020. Spectre attacks: exploiting speculative execution. Commun. ACM 63, 7 (2020), 93–101.
[19] Andrew Law, Chester Leung, Rishabh Poddar, Raluca Ada Popa, Chenyu Shi, Octavian Sima, Chaofan Yu, Xingzeng Zhang, and Wenting Zheng. 2020. Secure Collaborative Training and Inference for XGBoost. In Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice: 21–26.
[20] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Feinando. 2017. Inferencing Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. USENIX Association, 557–574.
[21] Feifei Li, Jimeng Sun, Spirou Papadimitriou, George A Mihaila, and Ioana Stanoi. 2007. Hiding in the crowd: Privacy preservation on evolving streams through correlation tracking. In 2007 IEEE 23rd International Conference on Data Engineering, IEEE, 686–695.
[22] Yehuda Lindell and Benny Pinkas. 2000. Privacy preserving data mining. In Annual International Cryptology Conference. Springer, 36–54.
[23] Lin Liu, Rongmao Chen, Xinmeang Liu, Jinshu Su, and Linbo Qiao. 2020. Towards practical privacy-preserving decision tree training and evaluation in the cloud. IEEE Transactions on Information Forensics and Security 15 (2020), 2914–2929.
[24] Microsoft. 2021. Microsoft Azure Confidential Computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/.
[25] Microsoft. 2021. Open Enclave SDK. https://openenclave.io Accessed July 1, 2021.
[26] Ahmad Moghimi, Gorka Irarrazoi, and Thomas Eisenbarth. 2017. Cachezoomer: How SGX amplifies the power of cache attacks. In International Conference on Cryptographic Hardware and Embedded Systems. Springer, 69–90.
[27] Jacob Montiel, Jesse Read, Albert Bifet, and Taleb Abdessalem. 2018. Sckit-multline: A Multi-output Streaming Framework. Journal of Machine Learning Research 19, 72 (2018), 1–3.
[28] Hai-Long Nguyen, Yew-Kwong Woon, and Wee-Keong Ng. 2015. A survey on privacy-preserving multiple party computation. Knowledge and information systems 45, 3 (2015), 533–569.
[29] Peiming Nie, Heng Huang, Xiao Cai, and Chris Ding. 2010. Efficient and robust feature selection via joint l2, 1-norms minimization. Advances in neural information processing systems 23 (2010).
[30] Olga Ohriemenko, Felix Schuster, Cédric Fourment, Aastha Mehta, Sebastian Nowozin, Kapil Varwani, and Manuel Costa. 2019. Off-the-Shelf Privacy-Preserving Machine Learning on Intel SGX. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2019, 2 (2019), 187–208.
[31] Ökçe Karakorkmaz, Mevlüt Ince, and Manuel Costa. 2016. Oblivious multi-party machine learning on trusted processors. In 25th {USENIX} Security Symposium {(USENIX) Security 16} 619–636.
[32] Oleskii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Versy: Protecting SGX Enclaves from Practical Side-Channel Attacks. In 2018 {USENIX} Annual Technical Conference, {USENIX} ATC'18: 227–240.
[33] Meni Orenbach, Andrew Baumann, and Mark Silberstein. 2020. Autarky: closing controlled channels with self-publishing enclaves. In EuroSys’20: Fifteenth EuroSys Conference 2020, Halkidik, Greece, April 27-30, 2020. ACM, 7:1–7:16.
[34] Bryan Parno, Jacob R Lorch, John R Douceur, James McKens, and Jonathan M McCune. 2011. Memor: Practical state continuity for protected modules. In 2011 IEEE Symposium on Security and Privacy. IEEE, 379–394.
[35] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. 2020. Visor: Privacy-preserving video analytics as a cloud service. In 29th {USENIX} Security Symposium {(USENIX) Security 20}. 1039–1056.
[36] J Ross Quinlan. 1986. Induction of decision trees. Machine learning 1, 1 (1986), 81–106.
[37] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Racoom: Closing digital side-channels through obfuscated execution. In 24th {USENIX} Security Symposium {(USENIX) Security 15} 431–446.
A SECURITY ANALYSIS

In this section, we analyse how EnclaveTree protects the enclave access pattern along with detailed pseudocode.

Definition A.1 (Data-oblivious). As defined in [44], we say that an algorithm is data-oblivious if an adversary that observes its interaction with memory, disk or network during the executions learns only the public information.

In the following, we prove both the HT training and HT inference in EnclaveTree is data-oblivious.

A.1 Oblivious HT Training

Algorithm 1: Oblivious HT Training

Input: $N$ encrypted data samples $Enc\cdot D$, $S$, $V$, $m$, $M$

1. Initialize the model matrix $M_0$ and leaves array $Leaf$ with $P$ dummy objects. Initialize the bit string $isDummy$. Initialize a list node, where $node[p] = (idx, tp)$ for $p \in [0, P]$. $node[p].idx$ stores the indices of the features that have not assigned on $M[\cdot; p]$, and $node[p].tp$ stores the number of feature values assigned to $M[\cdot; p]$.

2. Decrypt $Enc\cdot D$ and pack them into a $N \times M$ matrix $M$

3. Generate query matrix $M_Q$

4. foreach $p \in [0, P]$ do

5. $T[p] = success(node[p].idx, S, V)$. Where $V = (V_1, \ldots, V_M)$

6. $M_Q[p] = GenerateMasks(M_0[p], isDummy, T[p])$

7. $M_0[p] = M_Q[p] \| \cdots \| M_Q[P]$

8. $output[]$

9. endforeach

10. $output = RecordStat(M_Q[p], T[p])$

11. $UpdateStat(isDummy, output, Leaf)$

12. $Check$ for a split

13. $splitidxs = SplitCheck(isDummy, S, Leaf)$

14. $Generate$ new leaf nodes, update $I$, $Leaf$, and $M_0$

15. $isSplit = (splitidxs == -1)$

16. $CreateChildren(isSplit, node, M_0, Leaf)$

THEOREM A.2. The oblivious HT training of EnclaveTree (Algorithm 1) is data-oblivious with public parameters: $N$, $P$, $d$, and $M$.

Proof. Here we analyse what the adversary can learn from each operation in Algorithm 1.

The memory access occurred due to the initialization (line 1) and $M_Q$ generation (line 2) is independent of the data, from which the adversary could only learn the size information $P$, $N$ and $M$, which are public.

The loop from line 3 to line 5 aims to traverse $M$ and $node$ and generate the query matrix for each column of $M$. This loop always runs $P$ times, which means all the columns and elements of $M$ are distributed. $node$ respectively are always accessed for each round of training, resulting the same access pattern no matter what the input is. Recall that the query matrix is generated based on the feature values assigned and those unassigned to the column. Within the loop, the enclave first fetches the values of unassigned features indexed by $node[p].idx$ from $S$ and $V$ using $isDummy$ and stores them into $T[p]$ (line 4). Although $node[p].idx$ is different for different columns, the access patterns over $S$ and $V$ occurred by $isDummy$ are oblivious and are independent of $node[p].idx$. The function $GenerateMasks$ in line 5 generates the query matrix based on the values in $T[p]$ and $M_0[p]$. $M_0[p]$ is obtained with $isDummy$, which is also oblivious. Here the enclave generates query matrix in the same way for real and dummy columns. The difference is that the enclave assigns null to the masks for dummy columns, but the values in $T[p]$ for real columns, however there is no way for the adversary to learn that. After the loop, the query matrix $M_Q$ of the whole tree is generated by combining the matrix of each path together.
Once $M_d$ and $M_0$ are ready, the next step is to perform the matrix multiplication, which is inherently oblivious, and obliviously access the result matrix $M_e$ with oblivious primitives.

The second loop (line 10-14) is used to update the statistic information stored in Leaf and update $M_t$ and Leaf if they are leaves that need to be converted. The function RecordStat in line 10 checks the elements in each column of $M_t$ with node[$p$].tp and records the counts into a vector output. This process is performed obliviously with oequal and oselect, resulting the access pattern over $M_e$ and output independent of any value. In line 11, the enclave uses oassign to update Leaf based on output. Here no matter whether the array is real or dummy, the enclave processes it with oassign. The difference is that dummy arrays are assigned with 0, but real arrays are assigned with the values recorded in output. What the adversary observes from this process is all the same.

In line 12, the enclave checks whether to split the $p$-th leaf based on the updated Leaf. Precisely, the enclave first calculates the IG for all unassigned features. The enclave next uses ogreater, oequal and oselect to select the feature with the highest and second-highest IG, return a value splitIdx that indicates if $p$-th leaf is split by comparing with Hoeffding Bound (using oselect). Its access patterns are thus independent of $S$.

If node[$p$] is real and its IG values satisfy the Hoeffding Bound, line 14 converts the $p$-th leaf into internal nodes by updating node, $M_t$ and Leaf accordingly. The main idea is to convert node[$p$], $M_t[; p]$, and Leaf[$p$] into dummies by resetting isDummy. Moreover, assume the best feature selected for converting the $p$-th leaf has $m$ values, $m$ dummies in node, $M_t$ and Leaf are converted into real ones by setting their values based on the new leaves and paths with oblivious primitives. If either node[$p$] is dummy or it is not ready to be converted, the enclave similarly performs dummy write operations on node, $M_t$ and Leaf, which is indistinguishable from the operations performed for the former case due to the oblivious primitives.

Overall, from Algorithm 1 the adversary can only learn the public information $N$, $P$, $d$ and $M$.

A.2 Oblivious HT Inference

In this section, we provide pseudocode along with proofs of security for the oblivious HT inference in Algorithm 2.

**Algorithm 2: Oblivious HT Inference**

| Input: $N'$ encrypted data instances Enc.D, $m_i$, $d$, $M_6$, Leaf |
|---------------------------------|
| 1 Decrypt the unlabelled instances and pack them into a $N' \times (M - m_d)$ matrix $M_i$ |
| 2 Initialize label array $A_{label}$ of size $P$ for storing labels |
| % Store labels in an array |
| 3 foreach $p \in [1, P]$ do |
| 4 $A_{label} = $MajorityLabel(Leaf[$p$]) |
| % Record counts for each instance using $M_i$ |
| $M'_i = $KthMax($M_i$, $M_i$) |
| 6 output$=[]$ |
| 7 output = RecordStat($M_i$) |
| % Compare values in output and assign labels to instances |
| 8 Result$=[]$ |
| 9 Result = Predict(output,$A_{label}$) |
| 10 return Result |

**Theorem A.3.** The oblivious HT inference of EnclaveTree (Algorithm 2) is data-oblivious, with public parameters $N'$, $P$ and $M$.

**Proof.** The access patterns of line 1 depend only on the number of instances $N'$ and $M - m_d$. Line 2 depends on $P$.

The loop in line 3 and line 4 is used to determine each leaf’s label of the current tree, which executes $P$ times. Within function MajorityLabel, the enclave only uses oblivious primitives, which does not leak any access patterns. Thus, the adversary could only learn $P$.

In line 5, the access patterns occurred by the matrix multiplication is inherently oblivious.

The function RecordStat in line 7 checks the elements of each column in $M'_i$ and records the counts into output. Similarly, the two operations are both performed with oblivious primitives, which do not leak access patterns. The function Predict in line 9 first compares the values in output using oequal. It then accesses the $A_{label}$ to get the target label and assigns it to the corresponding instances using oassign. In this process, the adversary could only learn $N'$ and $P$.

\[\square\]

B PERFORMANCE OF ENCLAVETREE

B.1 More Results for HT Training and Inference

Here we show the performance of HT training and inference with 15 features in Fig. 10 and 63 features in Fig. 11. It is indicated that HT training performs better with less number of features, which is close to SGX when there are 15 features. However, when the number of features increases to 63, the runtime of HT training is close to Oblivious SGX. The fact is that EnclaveTree is efficient to process the data streams in most scenarios as they generally involve about a dozen of features. Regarding the inference, as shown in Fig. 11, our solution always outperforms Oblivious SGX baseline by several orders of magnitude.

B.2 RF Training and Inference

One concern of training data streams with HT is that the underlying data distribution of the stream might change over time, which leads to the accuracy degradation of the model, known as concept drift [20]. Ensemble models such as Random Forest (RF) with adaptive mechanisms [21] is a promising way to cope with the problem of concept drifts.

RF consists of a set of trees, and each tree is trained over a $\sqrt{d}$ subset of $S$ features. EnclaveTree uses the HT training component to train each tree in the RF. The features used to train a tree is randomly selected from $S$. To make the selection oblivious, the enclave accesses $S$ using oaccess. Assigning a label to a data instance with RF inference means classifying the instance with each tree and getting a set of labels. The final result is the label that is output by the majority of trees.

EnclaveTree performs the RF inference in a way similar to the HT inference using matrix multiplication. In particular, the data instances can be classified by multiple trees with one matrix multiplication by combining the matrices of the trees together.
We also evaluated the performance of RF training and inference, and the results are shown in Fig. 12. Fig. 12a shows the runtime in seconds to perform the RF training with 31 features and $5 \times 10^4$ samples. For all the test cases, every tree of the RF is trained with 6 features. The results show that, with the increase of trees, EnclaveTree is much faster than Oblivious SGX, which is up to $\sim 3.2 \times$. We also see that the performance of EnclaveTree is close to SGX.

With the same setting, we compare the inference performance of EnclaveTree with the other three baselines and the result is shown in Fig. 12b. We can see that EnclaveTree also performs better than Oblivious SGX by roughly 3.8x. Compared with SGX, EnclaveTree inference incurs more overhead when there are less than about 150 trees but is better when there are more than 150 trees. The reason is that the inference process requires EPC memory to store data, and it causes EPC paging when the EPC is exhausted. EnclaveTree simply performs matrix multiplication, and this operation involves much less memory access than SGX, which means less EPC paging occurred than SGX.