Towards Privacy-Preserving Collaborative Gradient Boosted Decision Tree Learning

Chester Leung

May 29, 2020

Towards Privacy-Preserving Collaborative Gradient Boosted Decision Tree Learning

by Chester Leung

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

RPopa

Professor Raluca Ada Popa
Research Advisor

May 29, 2020

(Date)

* * * * * * *

Joseph E. Gonzalez

Professor Joseph Gonzalez
Second Reader

May 27, 2020

(Date)

Towards Privacy-Preserving Collaborative Gradient Boosted Decision Tree Learning

by

Chester Leung

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Raluca Ada Popa, Chair
Professor Joseph Gonzalez

Spring 2020

Towards Privacy-Preserving Collaborative Gradient Boosted Decision Tree Learning

To my family, and especially my parents, who have inspired me in countless ways.

Contents

List of Figures

List of Tables

Acknowledgments

I'd like to thank my advisor, Professor Raluca Ada Popa, for her unwavering support, for providing me numerous opportunities, and for dedicating time to guide my growth. I'd also like to thank my mentors – Ankur Dave, Rishabh Poddar, and Wenting Zheng – who have made my time in RISE an absolutely incredible and delightful experience. It's been an immense joy working with you four, and you've taught me a tremendous amount in a number of technical and non-technical areas. I'll be forever grateful for having the opportunity to work with you.

Thanks as well to Professors Joey Gonzalez, Ion Stoica, and Vern Paxon, who have been kind enough to share feedback, insights, and their experiences.

To my good friends Ryan and Rohen: what a ride these last few years have been. Thank you both for sharing your drive, your knowledge, and your humor.

Lastly, an enormous thank you to my family. You've given me more than I could ever ask for.

Abstract

Towards Privacy-Preserving Collaborative Gradient Boosted Decision Tree Learning

by

Chester Leung

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Raluca Ada Popa, Chair

In recent years, gradient boosted decision tree learning has proven to be an effective method of training robust models. Moreover, collaborative learning among multiple parties can yield great benefits for all parties involved, but parties must be cautious of how they share sensitive data due to regulatory, business, and liability concerns.

We propose Secure XGBoost, an oblivious gradient boosting system that enables multiparty computation. Secure XGBoost is the first system of its kind, and builds on XGBoost, a state-of-the-art gradient boosting library with no security for the single party setting, to execute pre-agreed upon queries from multiple parties in hardware enclaves. Notably, Secure XGBoost introduces (i) a new system design facilitating secure collaboration tailored toward the outsourced computation model and (ii) oblivious algorithms for gradient boosted decision tree training and inference.

We find that our implementation of Secure XGBoost providing data encryption, authentication, and computation integrity is 0.23 to 12.5x slower than XGBoost; obliviousness comes with 2-3 orders of magnitude overhead.

Chapter 1

Introduction

There has recently been growing interest in collaborative machine learning, where multiple parties work together to perform a task over their collective data. Collaboration among parties often yields significant benefits; training on more data tends to produce higher quality models [31], and having complementary data from multiple parties may even enable new applications that were previously infeasible with only one party's data. However, parties may be unwilling to share their data due to legal regulation or business competition, establishing the need for systems that facilitate collaborative computation on sensitive datasets while hiding the data contents. Below, we illustrate two concrete use cases that outline the benefits of jointly computing on sensitive data.

Fraud Detection and Anti-Money Laundering: Banks today detect fraud by training models on customer transaction data, but criminals often mask their actions by moving assets across banks. As a result, trained models will be much weaker if only trained on one bank's data – models that can harness multiple banks' data will be much better at recognizing criminal patterns and detecting wrongdoing. However, customer financial data is sensitive and cannot be shared in plaintext. Consequently, any joint computation can occur only if the shared data is kept confidential.

Disease Diagnosis and Treatment: Hospitals may also want to collaborate to train more effective models for diagnoses or treatment plans, requiring the exchange of sensitive patient data that should not be done in plaintext. Secure collaboration is ideal for this situation – hospitals can leverage the data of other hospitals to improve their own models and consequently patient care, but will not see the medical data of patients from other hospitals.

A popular machine learning techique today is decision trees, a powerful algorithm that can efficiently, explainably, and accurately model non-linear relationships in data. An extension to decision trees is gradient boosted decision trees (GBDT), which has produced state-of-the-art results in production environments and machine learning competitions. Facebook uses it to predict clicks on ads [32], while XGBoost [15], an existing gradient boosted decision tree framework, produced 17 of 29 challenge-winning solutions on ML competition site Kaggle in 2015. Other GBDT libraries [38, 22] have also become immensely popular.

Our work aims to bring the benefits of GBDT to the secure collaborative setting. Prior work in this setting uses either specialized cryptography or hardware enclaves. Cryptography offers either functionality too limited for the gradient boosting, or comes with overhead too high for any practical system [17, 20, 6]. Hardware enclave technology, however, provides a trusted execution environment [48] that enables practical performance while maintaining confidentiality and integrity, and serves as a promising starting point for secure collaborative gradient boosting.

Unfortunately, as discussed in §2.3, hardware enclaves are prone to side channel attacks, many of which exploit access pattern leakage. To eliminate side channel vulnerabilities, an enclave should execute in a data-oblivious manner by performing computation that makes memory accesses independent of input data. Thus, to achieve secure multiparty gradient boosting on sensitive data, we need a solution that guarantees no visibility into sensitive data and no undesired computation on that sensitive data.

In this report, we propose Secure XGBoost, a distributed system for gradient boosted decision tree learning that securely computes on sensitive data in hardware enclaves. Importantly, Secure XGBoost offers data-oblivious computation to protect against side channel attacks. To our knowledge, Secure XGBoost is the only existing open source system to provide a secure GBDT pipeline. Keeping in mind the immense and continually growing popularity of cloud computing, Secure XGBoost assumes an outsourced computation model: one in which multiple mutually distrustful parties transfer their sensitive data to an untrusted cloud running a cluster of enclaves, where learning obliviously executes inside and only inside trusted enclaves. The major contributions of Secure XGBoost are in both systems and security, and are listed as follows:

- We combine existing techniques in cryptography and systems to design and build a distributed system that supports encrypted and oblivious computation on sensitive data belonging to different owners.
- We propose a party–enclave handshake in the multiparty setting to authenticate all trusted entities and provide building blocks for confidentiality and integrity in all future communications.
- We extend the AES-GCM authenticated encryption algorithm and introduce a novel scheme tailored for the distributed setting.
- We introduce algorithms for data-oblivious gradient boosted decision tree training and inference that eliminate memory access pattern leakage during execution.

We implemented Secure XGBoost on top of the existing XGBoost [15] library that provides gradient boosting, and evaluate it with three datasets using SGX-enabled virtual machines on the Azure Confidential Computing cloud. We find that Secure XGBoost minus side channel protection is up to 12.5x slower than XGBoost, while Oblivious Secure XGBoost is 2-3 orders of magnitude slower. We recognize that this overhead is significant and are actively looking for ways to address it.

Chapter 2

Background

2.1 Gradient Boosted Decision Trees

Decision trees are a machine learning technique that build a model in a tree-like manner. Decision tree training occurs by iteratively extending the depth of the tree by adding nodes; internal nodes represent a rule that partitions records on a feature according to optimal information gain [61], while leaves represent labels for all remaining records at that node. Decision trees evaluate a data instance and produce a prediction by traversing a path from root to leaf according to the feature values of the data instance.

GBDT is a technique used to build a tree ensemble by sequentially improving on the error of the model from the previous iteration. The key insight is that the residual, i.e., the error in predictions, is the negative gradient of the squared error loss. In each iteration, GBDT adds a weak learner trained on the loss of the previous iteration's model to the existing ensemble of weak learners.

2.2 Hardware Enclaves

A hardware enclave provides a private region of memory isolated from the rest of the host. Nothing other than that particular enclave, including other processes, the hypervisor, and even the host kernel, can access this secure region of memory or tamper with execution inside the enclave. The trusted execution environment that a secure enclave creates provides confidentiality and integrity guarantees, making it fit for sensitive data processing.

Hardware enclaves provide remote attestation [4], a procedure that allows a remote client to cryptographically verify that an enclave has loaded specific code. During attestation, an enclave generates a report containing a hash of the enclave, enabling remote client verification. The enclave may also generate a public key and send it to the client with the attestation report; the public key can subsequently be used to establish TLS channels.

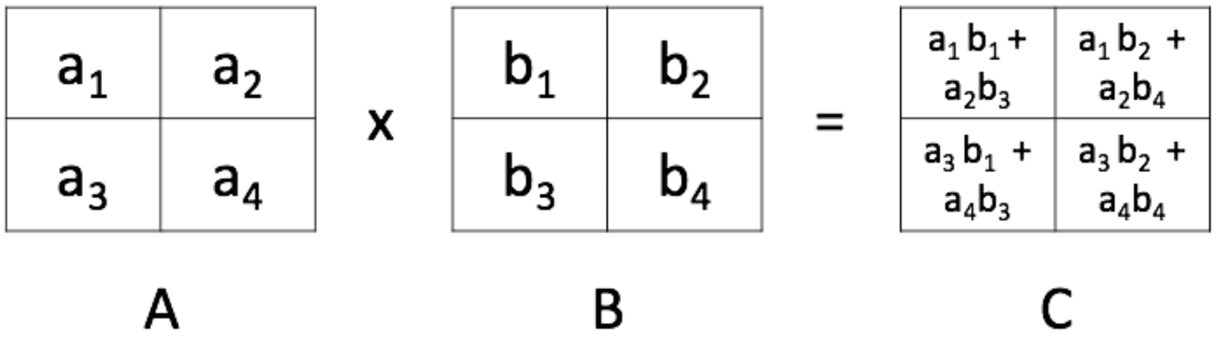

Figure 2.1: 2x2 matrix multiplication. Matrix multiplication is oblivious because its execution follows the same pattern regardless of the values within the input matrices.

2.3 Obliviousness

While hardware enclaves provide confidentiality and integrity guarantees, they do not protect against side channel attacks. In particular, an attacker can infer sensitive information about the data by observing auxillary channels like the paging and caching mechanisms during computation. Side channel leakage is often dependent on access patterns – an attacker can gain information by inspecting the sequence of accesses a program makes to disk, memory, or the network.

We can protect against a large majority of side channel attacks by employing *data-oblivious algorithms*. These algorithms make the same series of accesses independently of the input. Consequently, any execution of these algorithms will yield the same access patterns, preventing an attacker from gleaning any additional information by observing side channels.

A simple example of an oblivious algorithm is matrix multiplication – the inputs to matrix multiplication are always aggregated according to a predetermined pattern regardless of their values. Figure 2.1 depicts 2x2 matrix multiplication. On the other hand, quicksort is not oblivious – in each iteration, all records smaller than the pivot move to one memory region, while all records larger than the pivot move to another memory region. In particular, the value of each element relative to the value of the pivot affects its movement.

Chapter 3

Related Work

3.1 Tree-Based Learning

sklearn [60] supports decision trees, random forests, AdaBoost, and gradient boosted decision trees for relatively small datasets. Spark MLlib [50] builds on top of Spark [76] and enables scalable machine learning, supporting decision trees, random forests, and gradient boosted decision trees.

Other work has focused solely on gradient boosting. XGBoost [15] improves upon existing tree-boosting frameworks by contributing an approximate histogram algorithm to identify optimal feature splits, enabling scalability and greater performance with little cost in accuracy. LightGBM [38] further improves performance and scalability specifically for high dimensional large datasets by leveraging novel techniques to reduce the number of features and accurately estimate information gain without significantly hurting accuracy. CatBoost [22] provides gradient boosting with categorical feature support. SecureBoost [16] extends gradient boosted decision trees to the federated setting, enabling collaborative training on vertically partitioned data.

While the works mentioned above have grown in popularity and are widely used in the community, they provide little to no security and are not suited for our threat model and in particular outsourced computation to the untrusted cloud.

3.2 Cryptographic Approaches

Prior work has explored applying cryptographic techniques to secure the training and inference of decision trees and random forests. Wu et al. [73] leverages homomorphic encryption for the two party setting, while other work [44, 23, 71] focuses on applying secure multiparty computation (MPC) techniques for both $n = 2$ and $n \geq 3$ parties. MPC has also been used as a tool in privacy-preserving AdaBoost [26, 45].

In general, cryptographic techniques are prohitively slow, and are mostly impractical for deployed systems. Our work supports gradient boosting and provides orders of magnitude speedup compared to existing cryptographic techniques.

3.3 Hardware Approaches

SCONE [3], Graphene [70], Ryoan [33], Haven [5], Cipherbase [1], VC3 [63] and Opaque [78] offer computation with trusted hardware, but are confined to the single-party setting. OCQ [21] enables oblivious coopetitive analytics with hardware enclaves but does not support gradient boosting.

Ohrimenko et al. [56] provides oblivious inference with pre-trained decision trees. Similar to our work, it developed a new algorithm using oblivious primitives to hide access patterns during evaluation, preventing a side channel attacker from inferring test records and tree structure.

Chandra et al. [12] also enables secure decision tree inference by appending dummy records to sensitive test data and obliviously shuffling the resulting set before evaluating each record on a decision tree trained offline. The dummy records serve to confound an attacker – a side channel attacker is unsure whether the information inferred from access patterns of an evaluation reflects a real or dummy record.

While both works tackle only decision tree inference in the single party setting and are confined to using Intel SGX, our work contributes oblivious algorithms for both the training and inference of gradient boosted decision trees and supports an abstract enclave model.

3.4 Differential Privacy

Jagannathan et al. [35] and Friedman et al. [25] propose ϵ-differentially private mechanisms for training decision trees, while Li et al. [43] introduces differential privacy for gradient boosted decision trees. These methods are complementary to Secure XGBoost, and can be combined with our techniques to achieve even greater privacy and security.

Chapter 4

System Overview

4.1 System Architecture

In this section we discuss the actors in Secure XGBoost. We discuss (i) clients, collaborating parties with sensitive data; (ii) an untrusted cloud service that hosts Secure XGBoost on a cluster of enclaves; and (iii) an RPC orchestrator, which mediates communication and relays messages between clients and the cloud. The general architecture of Secure XGBoost is depicted in Figure 4.1.

Clients

A client refers to a party that participates in the collaborative learning process with other parties. Clients collectively execute the computation pipeline on cloud-hosted Secure XGBoost by remotely invoking agreed-upon APIs.

The client is trusted – data originates at the client and rests in unencrypted form. Any computation that occurs at the client is secure even without enclaves or obliviousness.

Cloud

The cloud consists of a cluster of virtual machines, each with hardware enclave support. The VMs are arranged in two possible ways for communication: a tree structure or a ring structure. In a tree structure, depicted in Figure 4.2, nodes may have up to one parent and up to two children, and consequently communicate with up to three other machines. In a ring structure, depicted in Figure 4.3, the nodes are arranged consecutively as in a linked list; each node has up to one parent and up to one child.

During execution of client commands, Secure XGBoost distributes computation over the cluster; enclaves communicate over TLS channels that begin and end inside the enclaves.

On each VM, we distinguish between the host – the untrusted portion of the VM – and the enclave – the trusted execution environment. Any data or command sent by a client to the cloud must first pass through the host before reaching the enclave. Moreover, anything

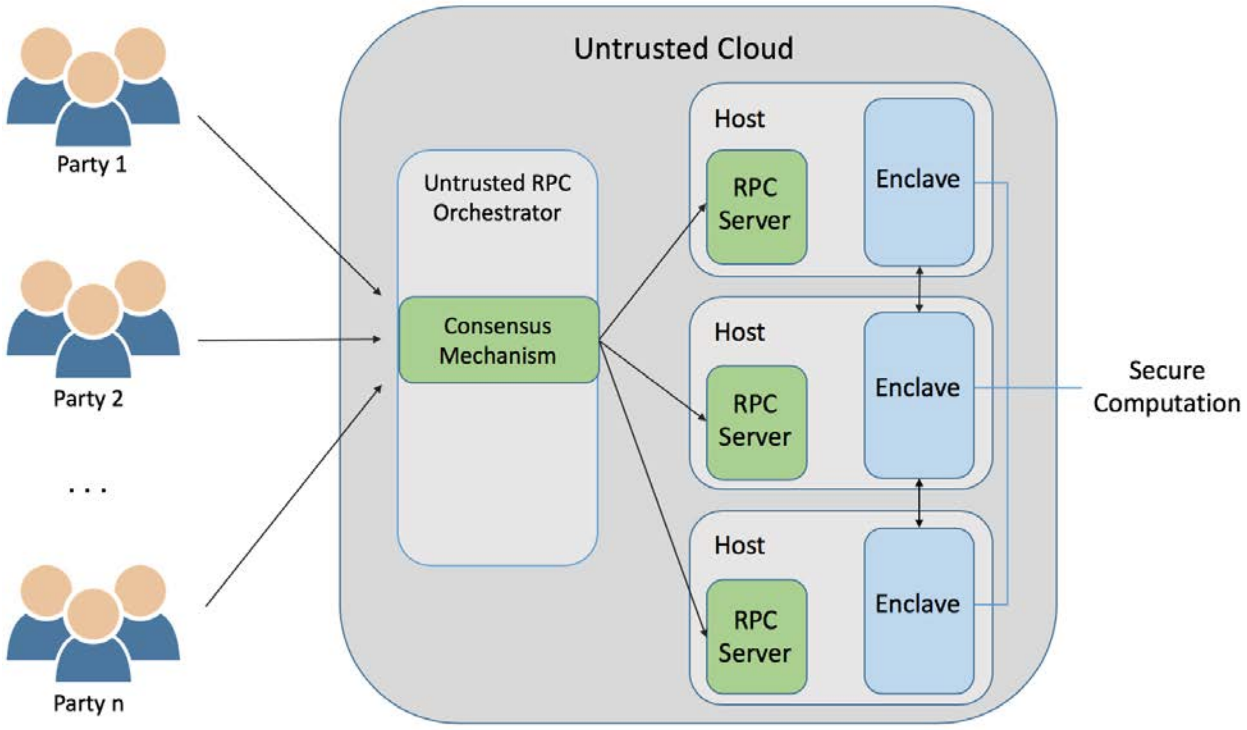

Figure 4.1: Parties make calls to an orchestrator service, which waits for calls from all parties before relaying the commands to the Secure XGBoost enclave cluster. Secure XGBoost distributes computation across the cluster of hardware enclave. Enclaves communicate over TLS channels that begin and end inside the enclaves. Enclave inputs and outputs are always encrypted, and are decrypted only within the enclave or at client premises.

that resides on the host outside the enclave is visible to and vulnerable to tampering by the untrusted host.

One RPC server runs on each host outside the enclave, listening for commands from the RPC orchestrator. Each RPC server accepts requests from the orchestrator and makes calls to the enclave according to the request.

The enclave, on the other hand, is trusted. As a result, it provides confidentiality and integrity guarantees on all computation within it and all data loaded inside it. However, as mentioned previously, enclaves are prone to access pattern leakage, which can be addressed by running data-oblivious algorithms.

RPC Orchestrator

The RPC orchestrator mediates communication between clients and the cloud and serves as a central rendezvous point for all clients. Clients submit signed commands to the RPC orchestrator, which waits for all clients to submit the same command before relaying the

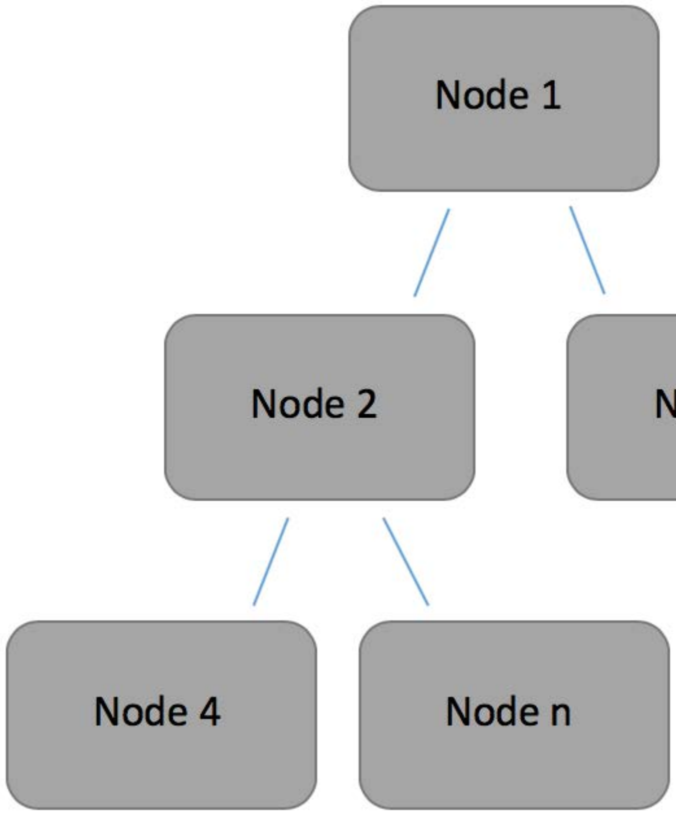

Figure 4.2: Depiction of the tree topology, a possible arrangement of nodes in the untrusted cloud in Secure XGBoost.

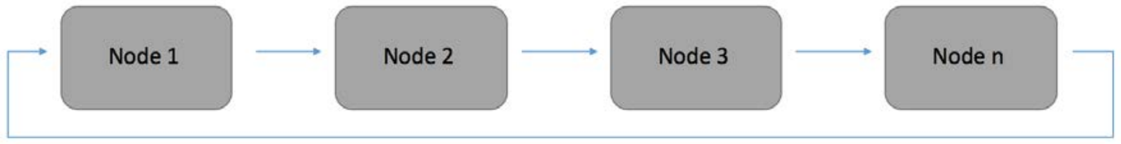

Figure 4.3: Depiction of the ring topology, a possible arrangement of nodes in the untrusted cloud in Secure XGBoost.

command and a list of client signatures to Secure XGBoost. Secure XGBoost then executes the command in a distributed fashion, and returns the results to the RPC orchestrator. The RPC orchestrator lastly distributes the results back to each client.

4.2 General Workflow

In the following we present an end-to-end outsourced computation workflow for multiple clients. To collaborate, clients leverage the cloud for its computation and storage resources, transferring their sensitive data to a remote cluster for processing. The term "cloud" refers to an enclave cluster running Secure XGBoost, and the term "command" to a client's desired execution of a step in the computation pipeline, e.g., data loading or training.

1. The clients agree on a sequence of commands to be executed by Secure XGBoost.

2. Clients attest the enclaves on the cloud via remote attestation to verify that each enclave loads the proper code. Running remote attestation will also return to each client an enclave public key pk.

3. Each client C_i encrypts its data with its symmetric key k_i and transfers its encrypted data to cloud storage.

4. The clients issue a signed command to the RPC orchestrator. The orchestrator's consensus mechanism waits for all clients to submit the same command before relaying the command and a list of signatures to Secure XGBoost in the cloud. Secure XGBoost authenticates the command, ensuring that every client in the cluster indeed issued the command, and executes the command over the cluster.

5. Secure XGBoost then returns the result of the command (e.g., an encrypted serialized model or encrypted predictions) back to the RPC orchestrator, which then relays it to the clients.

6. Steps 4 and 5 repeat as needed.

Chapter 5

Threat Model

5.1 Cloud Threat Model

Secure XGBoost considers an attacker who has completely compromised the host in the cloud save for the enclave itself, including the operating system, hypervisor, and other processes. Moreover, we assume that the attacker has visibility into memory accesses, which can lead to attacks leveraging page faults [74, 11], the branch predictor [42], cache timing [27, 9, 64, 19, 30, 52], the memory bus [39], and others.

Moreover, Secure XGBoost builds on top of an abstract enclave model. While our implementation in Chapter 8 builds on Intel SGX [34], our design is not tied to it and is compatible with any enclave. To date, there have been a growing number of enclave offerings, including SGX, AMD Memory Encryption [37], Keystone [40], Sanctum [18], MI6 [7], and others [2, 24]. We assume that an attacker cannot access or undetectably modify data or code inside an enclave, that it cannot undermine the enclave attestation process, and that it cannot abuse side channels others than the ones exploiting the specific memory access patterns mentioned above. In particular, prior work has shown that Intel SGX suffers from other side channel attacks based on speculative execution [10, 14, 65], power consumption [54, 67], rollback [59], intra-cache line memory accesses [53, 75], denial-of-service attacks [29, 36], and others [41, 72]. While some defenses have been proposed [46, 57, 8, 13], and some enclave designs (e.g., Keystone, MI6) completely void a subset of these attacks, we do not protect against these attacks, but hope that our design will be combined with stronger enclaves to yield an even more secure system.

In short, the clients only need to trust that the hardware enclave implementation is correct and that the code underlying Secure XGBoost is safe. Since Secure XGBoost is open source, clients can verify it before use.

5.2 Client Threat Model

Each client trusts no entity other than itself. We assume that malicious clients can collude with each other, the cloud service, and/or the RPC orchestrator to learn information about a victim client's data. They may also attempt to tamper with the computation steps or inspect cloud computation.

We note that Secure XGBoost does not protect against a malicious party inputting garbage data or data intentionally crafted to reveal aspects of another party's data. To address this, we note that Secure XGBoost contains a consensus mechanism that forces every party to authorize any action by any party. In particular, to ensure fairness, every party can check that it indeed benefited from the joint computation before releasing any results to any party.

Chapter 6

System Design

In this chapter we discuss the design of Secure XGBoost. We combine existing techniques in cryptography and distributed systems to build a system that limits information leakage according to a policy agreed upon by all clients before computation.

6.1 Initial Setup

In this section we present what each entity in our system initially holds. Notably, Secure XGBoost focuses only on key usage and not key distribution; we assume that there exists a public key infrastructure or alternative method of key management.

Client: Each client holds 3 items upon startup:

1. A 2048-bit RSA public/private keypair (pk_i, sk_i), used to sign commands.
2. A certificate signed by a trusted certificate authority to authenticate the client to Secure XGBoost.
3. A 256-bit symmetric key k_i used to encrypt sensitive data.

Enclave: Each enclave holds 4 items upon startup:

1. A 2048-bit RSA public/private keypair (pk_j, sk_j). The master enclave uses this keypair to securely obtain each client's symmetric key, as explained in §6.3.
2. The public key of a trusted certificate authority, used to verify each client's identity.
3. A nonce N to ensure freshness of messages.
4. An embedded list of client usernames.

RPC Orchestrator: Since it simply acts as an intermediary, the RPC orchestrator holds nothing.

6.2 Client-Cluster Attestation

The client-cluster attestation procedure authenticates all trusted entities in Secure XGBoost. Before any computation can occur, each client performs a party-enclave handshake to authenticate the Secure XGBoost deployment in the cloud as follows. Clients attest the master enclave to verify that the proper code has been loaded inside the master enclave via the remote attestation procedure outlined in §2.2. Each enclave subsequently attests all neighboring enclaves in the cluster, a process we term inter-enclave attestation. If any attestation attempt fails, the node with the enclave that fails to authenticate is excluded from the cluster, and attestation again occurs. Once inter-enclave attestation has completed, enclaves establish TLS sessions with one another, and the master enclave returns its public key pk_{master} and the nonce N along with the attestation report to each attesting client. The nonce is used in all ensuing communication to prevent cross-session replay attacks.

Each client then encrypts its symmetric key k_i with pk_{master}, signs the resulting ciphertext with its public key pk_i, and sends the ciphertext and the signature through the RPC orchestrator to the master enclave. The master enclave verifies the signature, decrypts the ciphertext to obtain k_i and broadcasts k_i over the secure channels established during inter-enclave attestation, giving every enclave in the cluster k_i and thus the ability to decrypt client C_i's encrypted data.

6.3 Data Preparation

The distributed setting renders the typical file-wise encryption infeasible; a data file may be partitioned at any row for distributed computation according to Secure XGBoost's partitioning patterns, but data encrypted as a file cannot be properly decrypted when arbitrarily partitioned. We introduce an extension of the AES-GCM authenticated encryption scheme tailored for the distributed setting.

Using AES-GCM, each client performs row-wise encryption on its sensitive data with k_i. However, this leaves the data vulnerable to row deletion tampering. To ensure integrity of the data when at rest on the untrusted cloud, the client appends the row number j and the total number of rows N_i to each row in its data file.

In particular, each client C_i encrypts each row in its data as follows:

$$j, \quad N_i, \quad \mathsf{Enc}(\mathsf{row}_j), \quad \mathsf{MAC}(j||N_i||\mathsf{Enc}(\mathsf{row}_j))$$

Here, j is the index number of the row being encrypted; N_i the total number of rows in C_i's data; $\mathsf{Enc}(\mathsf{row}_j)$ an AES-GCM ciphertext over the j-th row; and $\mathsf{MAC}(j||N_i||\mathsf{Enc}(\mathsf{row}_j))$ an AES-GCM authentication tag computed over j, N_i and $\mathsf{Enc}(\mathsf{row}_j)$. Including j and N_i within the authentication tag prevents the untrusted cloud service from tampering with the data (e.g., by deleting or duplicating rows).

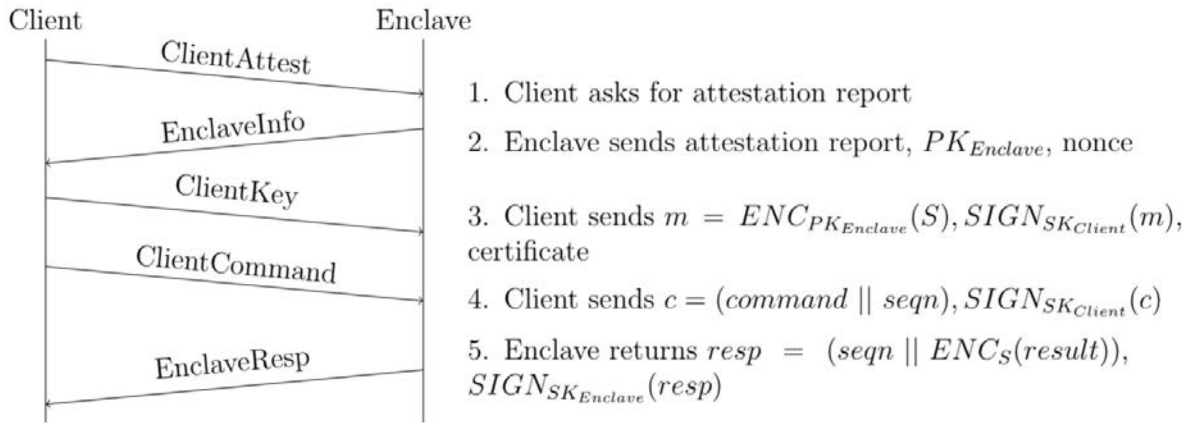
Client
Enclave
ClientAttest
EnclaveInfo
ClientKey
ClientCommand
EnclaveResp
1. Client asks for attestation report
2. Enclave sends attestation report, $PK_{Enclave}$, nonce
3. Client sends $m = ENC_{PK_{Enclave}}(S), SIGN_{SK_{Client}}(m)$, certificate
4. Client sends $c = (command \mid\mid seqn), SIGN_{SK_{Client}}(c)$
5. Enclave returns $resp = (seqn \mid\mid ENC_S(result))$, $SIGN_{SK_{Enclave}}(resp)$

Figure 6.1: The communication between a client and the master enclave, where S is the symmetric key used to encrypt a client's sensitive data. The RPC orchestrator sits in the middle and relays each message from the sender to the recipient, but is excluded here for simplicity.

Each client uploads its encrypted data to the cloud for processing. Each node in the cluster will process a partition of the data; consequently, in a federation of n parties, each node processes n partitions of data, one for each party.

To decrypt each client's data, each enclave follows the following steps. First, each enclave loads in its assigned rows by examining the appended row index j. The enclaves then communicate to ensure that they together loaded t rows. Finally, each enclave checks the `MAC` tag of each row and decrypts its partition of the data with the symmetric key k_i corresponding to C_i.

6.4 Privacy Preserving Distributed Data Processing

Once all enclaves are able to decrypt all clients' data, the cluster is ready to begin computation on the aggregated training data. However, each step in the pipeline requires consensus from all parties to ensure that no party is operating on another party's data without consent.

As an example, consider a consortium of banks that have properly configured a cluster and are ready to begin training. All banks transfer their sensitive data to the cloud and load it inside the enclave. A malicious bank then trains a model on all banks' data, saves the model, makes the serialized model accessible to only itself, and withdraws its sensitive data from the cloud. In this scenario, the lack of consensus has allowed a malicious party to defraud other banks; a consensus mechanism would have prevented the malicious party from proceeding with training and saving the joint model.

Parties achieve consensus when they all submit a signed command invoking the same API to the RPC orchestrator. In particular, each party sends the following to the RPC orchestrator:

$$\texttt{cmd} = (\texttt{seqn || func || params}),\ \ \texttt{Sign}(\texttt{cmd})$$

A command contains three fields: (i.) a sequence number $\texttt{seqn} = (N||\texttt{ctr})$ that consists of the nonce N (obtained from the enclaves during attestation) concatenated with an incrementing counter; (ii.) `func`, the API function being invoked; and (iii.) `params`, the function parameters. Including the sequence number ensures the freshness of the command and prevents replay attacks on the system.

Once the RPC orchestrator receives a command from every client, it aggregates the signatures and relays the command and a list of signatures to each enclave in the cluster. Each enclave verifies the signature, ensures the nonce is as expected, and checks that all commands are identical. After checking, each enclave proceeds with its portion of the distributed computation.

During computation, enclaves normally return a pointer to an object inside the enclave to the user. However, doing so in the outsourced computation model with an untrusted cloud and orchestrator leaves each returned pointer vulnerable to tampering. Moreover, in the distributed setting pointers from different enclaves may have different values. To address this, we introduce pointer aliasing: each enclave instead returns to the user an alias for the pointer; the alias is mapped to the pointer value inside the enclave.

We did not contribute to the *distributed* portion of the learning – we based our implementation of distributed GBDT on the weighted quantile sketch algorithm split finding algorithm described in [15] – but rather made the learning algorithms oblivious, i.e. resistant to memory access pattern leakage. More on our oblivious algorithms is in Chapter 7.

All computation and unencrypted sensitive information is visible only in a trusted area. The usual split finding and split aggregation that happens in GBDT all executes within an enclave. For example, each machine is broadly tasked with finding the optimal split for a partition of the training data. It loads its partition of the encrypted training data into the enclave, decrypts it accordingly, and runs the split finding algorithm. The optimal splits are then sent over a secure channel, and updates to the global model are returned over a secure channel.

Any pointer aliases, predictions, or serialized models that leave the enclave are individually encrypted. For example, if all n parties agree to run code that saves a model, Secure XGBoost encrypts the serialized trained model n times, each time with a different party's symmetric key to prevent one party from tampering with another party's model.

Once processing for each command has finished, the cluster returns any results, along with the sequence number, to the clients by way of the RPC orchestrator. In particular, Secure XGBoost returns the following to each client C_i:

$$\texttt{resp} = (\texttt{seqn || Enc}_{k_i}(\texttt{result})),\ \ \texttt{Sign}(\texttt{resp})$$

The response contains `seqn`, the sequence number of the request (to cryptographically bind the response to the request); the results of the function, which are encrypted with each client's key k_i; and a signature over the two.

The full communication between a party and the master enclave is shown in Figure 6.1. While the RPC orchestrator lies between the party and the master enclave, we exclude it for simplicity.

6.5 Privacy Policy

In our experience, even when ensuring organizations that their sensitive data will not be leaked, many have expressed reluctance to collaborate in fear that they will only marginally benefit or that their competitors will benefit more. We recognize this as a tussle between privacy and transparency.

While we leave a specific policy framework to future work, we note that collaborators can agree to run code after training to ensure that the jointly computed model meets each party's expectations and that the collaboration has benefited all parties. For example, all parties can upload some validation data not used during training and test the trained model with the data. If the accuracy of the jointly computed model is below a certain threshold, that party can choose to abort, at which point none of the collaborators receive anything from the collaboration process.

Moreover, collaborators can establish what the cloud can leak. If side channel attacks are outside their scope, collaborators can choose to train a joint model without obliviousness. If they are not concerned about an arbitrary party reverse engineering the model to extract another party's sensitive data [62, 66, 55, 69], they can choose to distribute a copy of the model to each party. If they fear that allowing each party to query the joint model an arbitrary number of times may help the party find patterns in other parties' data, they can limit the number of predictions served to each party.

Chapter 7

Obliviousness

In the context of decision trees, non-data-oblivious training and inference leaks significant information. Even when done in an enclave, a traversal of a tree will leak its structure to an adversary observing memory access patterns. Consider the evaluation of a data instance that follows the path $a \to c \to f$ in the tree illustrated in Figure 7.1. An adversary will observe that the memory at a's address is a node at the first layer of the tree, the memory at c's address is a node at the second layer of the tree, and the memory at f is a node at the third layer of the tree. Repeatedly observing the evaluation of multiple data instances may allow an adversary to infer the entire structure of the tree.

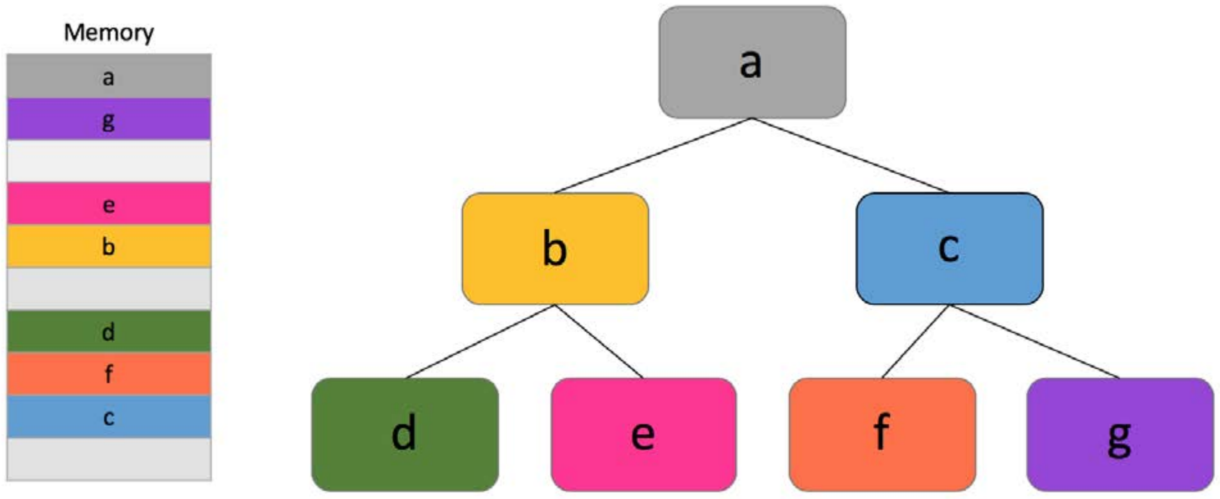

Figure 7.1: Without data-obliviousness, observing memory access patterns during a root-to-leaf traversal of a tree leaks the structure of the tree in memory.

```
template <typename T>
void OAssign(bool pred, const T& t_val, const T& f_val, T* out) {
    T result;
    __asm__ volatile (
        "mov %2, %0;"
        "test %1, %1;"
        "cmovz %3, %0;"
        : "=&r" (result)
        : "r" (pred), "r" (t_val), "r" (f_val), "m" (out)
        : "cc"
        );
    *out = result;
}
```

Figure 7.2: An implementation of the `oassign` primitive, written in x86 assembly.

7.1 Oblivious Primitives

Similar to prior work [56], our oblivious algorithms for gradient boosting rely on four oblivious primitives implemented with x86 assembly. Fundamental to these primitives is the `CMOV` instruction, which takes as input two registers — a source and destination — and moves the source to the destination if a condition is true. Once the operands have been loaded into registers, the instructions are immune to memory-access-based pattern leakage because registers are private to the processor, making any register-to-register operations oblivious.

Oblivious Assignment (`oassign`): The `oassign` primitive leverages the `CMOV` instruction to obliviously assign a value to the destination operand. Dummy writes are performed by setting the input condition to false. Because registers are private to their processor, only code inside the enclave has visibility into the contents of the registers, meaning that any operations that involve only registers are not recorded in the trace. Consequently, register-to-register operations are by default data-oblivious. Figure 7.2 shows `oassign` implemented in our codebase.

Oblivious Comparison (`oless`, `ogreater`, `oequal`): These primitives wrap around the `cmp` instruction to obliviously compare two variables. They take as input two registers and update the processor's flags register with the result of the comparison.

Oblivious Sorting (`osort`): The `osort` primitive uses a bitonic sorting network, and in particular a series of predefined compare-and-swap operations implemented using `oassign`, to obliviously sort an array. Given an input size n, the network layout is fixed and the network performs $O(nlog^2n)$ compare-and-swap operations. Thus, the memory accesses are dependent only on n and not the contents of the array, as opposed to other sorting algorithms like quicksort that rely on the value of the array elements to determine access patterns.

Oblivious Access (`oaccess`): `oaccess` obliviously indexes an array. One way to easily implement `oaccess` is to iterate through an entire array and access each element but load just one. However, our threat model assumes that an adversary can only observe memory access patterns at cache-line granularity, enabling us to scan arrays at cache-line granularity using `oassign`. For `oaccess`, we use AVX2 vector instructions to optimize the primitive.

7.2 Oblivious Training

Secure XGBoost uses a data-oblivious revision of XGBoost's approximate algorithm [15] to perform training. The approximate algorithm builds a tree in rounds, adding a node to the tree per round. Given data samples $x \in \mathbb{R}^d$, at each node the algorithm chooses a feature f and a threshold t to partition data samples (i.e., if $x(f) < t$, the algorithm partitions samples into the left subtree, otherwise the right). Each round of training thus requires choosing the feature to split by and the corresponding threshold.

To add a node to the tree, each enclave in the cluster first proposes candidate splitting points according to percentiles of feature distribution. The algorithm then maps the continuous features into buckets split by these candidate points, aggregates the statistics, and finds the best solution among proposals based on the aggregated statistics.

In the absence of data-obliviousness, the algorithm reveals a large amount of information via access-pattern leakage. For example, it leaks which feature was chosen at each node in the tree as well as the complete ordering of the data samples. To prevent this leakage, Secure XGBoost uses oblivious subroutines for adding nodes to the tree in each round, as follows:

1. Each enclave obliviously creates a summary S of its data (one summary per feature): each element in the summary is a tuple (y, w), where y_j are the unique feature values in the list of data samples, and w_j are the sum of the weights of the corresponding samples.

   To create the summary, the enclave sorts its n samples using `osort` and initializes an empty array S of size n. Next, it scans the samples to identify unique values while maintaining a running aggregate of the weights: for each sample x_i it updates $S[i]$ using `oassign`, either setting it to 0 (if $x_i(f) = x_{i+1}(f)$), or to the aggregated weight. At the end, it sorts S using `osort` to push all 0 values to the end of the list.

2. Each enclave then obliviously prunes its summary to a size b, shrinking the summary to size b by averaging out the elements. To compute this obliviously, we scan the original summary b times, using `oassign` operations to perform the averaging.

3. Next, each enclave broadcasts its summary S. Summaries are pairwise aggregated into a "global" summary (one summary per feature) as follows:

   i. Each pair of summaries is first merged into a single list using `osort`. The merged summary is then scanned to identify duplicate adjacent values; duplicates are zeroed out using `oaccess` while aggregating the weights. The merged summary is then sorted using `osort` to push all 0 values to the end of the list, and then truncated.

ii. Next, the merged summary is pruned as before into a summary of size b.

4. The global summary represents a histogram, with the constituent values as the boundaries of different bins. Each enclave computes gradients over its data samples per bin. That is, it scans its data samples to compute a gradient per sample, and then updates a single bin using `oaccess` and `oassign`. The enclaves then broadcast their histograms.

5. Each enclave sums up the histograms. Note that there is a histogram per feature. Each enclave then computes a score function over each histogram, identifying the best feature to split by, as well as the split value.

6. Finally, each enclave partitions its data based on the split value: it sorts the samples using `osort`, and obliviously zeros out elements less than / larger than the split using `oassign`.

The oblivious split finding routine runs a fixed number of times (depending on the desired depth of the tree) until the entire tree is constructed.

7.3 Oblivious Inference

Normally, inference on a data instance executes by traversing each tree in the model from root to leaf and comparing the feature value of each interior node with the corresponding feature value in the data instance. In particular, the decision d on a data instance $x \in \mathbb{R}^n$ at a node for feature f with value v is as follows:

$$d(x; f, v) = \begin{cases} \text{left}, & \text{if } x(f) < v, \\ \text{right}, & \text{otherwise} \end{cases}$$

However, this series of decisions leaks information on both the data instance x and the structure of the trees. To prevent this information leakage, we follow [56] and store each tree layer as an array of nodes. Inference then proceeds as follows: as we progress down a tree, Secure XGBoost obliviously looks up the proper node using `oaccess`. The corresponding feature in the data instance $x(f)$ is found using `oaccess`, and the comparison performed with `oless`. If a leaf is found before reaching the maximum depth of the tree, Secure XGBoost obliviously stores the leaf value with `oassign` and performs dummy accesses until reaching the maximum depth. Predictions from all trees are lastly aggregated to give the final prediction.

Chapter 8

Implementation

Our implementation is an extension of the original XGBoost [15] codebase, written mainly in Python and C++. We also wrote x86 assembly to implement the oblivious primitives. In total, our codebase contains approximately 118,000 lines of C++ and 11,000 lines of Python. We note that because our implementation is an extension of the original XGBoost codebase, not all the code that is present is called.

Secure XGBoost provides a Python frontend similar to that of XGBoost – we aim to enable data scientists, developers, and other researchers with less security background to write secure applications. By offering a Python frontend, we hope to ease the barrier of entry, and to help the community build applications that are secure by design.

The Python frontend uses gRPC [28] for client—RPC Orchestrator—cloud communication. At each party and on the cloud, Secure XGBoost calls a C++ backend that performs cryptography and that runs the code within the enclave. In particular, we use MbedTLS [47] for cryptography and Open Enclave [58] for enclave interfacing.

We've tested and deployed our implementation on Intel SGX [34] on Microsoft's Azure Confidential Computing cloud service [51]. However, our design and implementation is enclave agnostic; Open Enclave currently supports both Intel SGX and ARM Trustzone [2], while our design is compatible with all enclaves. Our codebase is open source and available at: `https://github.com/mc2-project/mc2-xgboost`.

```
import securexgboost as sxgb

# Setup
sxgb.init_user(username, sym_key, pub_key, cert)
enclave = sxgb.Enclave("sxgb_enclave.signed")
enclave.attest()
enclave.add_key()

# Load data
dtrain = sxgb.DMatrix({username: "train.enc"})
dtest = xgb.DMatrix({username: "test.enc"})

# Train and evaluate
params = {
        "tree_method": "hist",
        "objective": "binary:logistic",
        "min_child_weight": "1",
        "gamma": "0.1",
        "max_depth": "3"
}

num_rounds = 5
booster = sxgb.train(params, dtrain, num_rounds)

# Get encrypted results
booster.save_model("modelfile.model")
predictions, num_preds = booster.predict(dtest)
booster.decrypt_predictions(predictions, num_preds)
booster.get_fscore()
```

Figure 8.1: Sample client code of Secure XGBoost. The sample shows calls for user initialization, enclave creation, attestation, key sharing, data loading, training, prediction, and feature importance retrieval.

Chapter 9

Evaluation

9.1 Datasets and Setup

In our experiments we used three datasets, summarized in Table 9.1: the Allstate Insurance Claim Prediction dataset, the Higgs boson dataset, and the Beijing Pollution dataset.

Allstate Insurance: Each row in the Allstate Insurance Claim Prediction dataset [1] represents the risk factors of a driver, such as the vehicle model and the vehicle model year. The task is to predict bodily injury insurance claim payments. We random selected 1 million rows for our experiments, and one-hot encoded the relevant of the original 33 features to obtain 3,788 features. The resulting dataset is 5.7 GB.

Higgs boson: The Higgs boson dataset [2], from high energy physics, contains particle data from signal processes. 21 of the 28 features are kinematic properties of particles in the process; the remaining are functions of those kinematic properties. The task is to predict whether a signal process produces Higgs boson particles. We randomly selected 2 million rows for our experiments. The resulting dataset is 422 MB.

Beijing Pollution: The Beijing Pollution dataset [3] contains hourly air quality data from 12 air-quality monitoring sites in Beijing, China. The task is to predict PM2.5 readings – PM2.5 are atmospheric particulate matter with a diameter of less than 2.5 micrometers, and

[1] https://www.kaggle.com/c/ClaimPredictionChallenge

[2] https://archive.ics.uci.edu/ml/datasets/HIGGS

[3] https://archive.ics.uci.edu/ml/datasets/Beijing+Multi-Site+Air-Quality+Data

Dataset	Rows	Features	Task
Allstate	1M	3,788	Claim Classification
Higgs boson	2M	30	Process Classification
Beijing Pollution	420k	31	Pollution Regression

Table 9.1: Summary of datasets used in our experiments.

	Encryption Time (sec)	Decryption Time (sec)
Allstate	82.1379	23.7213
Higgs boson	14.0419	4.1167
Beijing Pollution	1.1530	0.6591

Table 9.2: Encryption and decryption times of the three datasets used.

are used to measure pollution. We used all 420k rows in the dataset. After one-hot encoding, the dataset contains 31 features, and is 41 MB.

Our experiments compare three systems: Vanilla XGBoost, an out of the box implementation of XGBoost [4]; Encrypted Secure XGBoost, a version of Secure XGBoost with no side channel protection; and Oblivious Secure XGBoost, Secure XGBoost with obliviousness enabled.

We ran our experiments on Microsoft's cloud-based Azure Confidential Computing [5] service; we used DC4s_V2 machines, which provide Intel SGX along with 4 vCPUs, 16 GiB of memory, and a 112 MiB enclave page cache.

9.2 Results

Allstate Insurance: The majority of features in the Allstate Insurance dataset are sparse, enabling us to leverage XGBoost's built-in sparsity-aware algorithm [15] for high performance. We measured the average time to train one tree over all 1 million rows, shown in Figure 9.1, and the overall training time of 50 trees, shown in Figure 9.3. We also looked at the overall training times of vanilla XGBoost, Encrypted Secure XGBoost, and Oblivious Secure XGBoost for a varying number of rows, shown in Figure 9.2.

Higgs boson: We measured the average time to train one tree over all 2 million rows, shown in Figure 9.1, and the overall training time of 50 trees, shown in Figure 9.3.

Beijing Pollution: We measured the average time to train one tree over all 420k rows, shown in Figure 9.1, and the overall training time of 50 trees, shown in Figure 9.3.

We lastly measured the time taken to encrypt each dataset at the client and to decrypt each dataset inside the enclave; Table 9.2 shows the results.

In general, Encrypted Secure XGBoost incurs 0.2x–12.5x overhead compared to vanilla XGBoost, which provides no security. Oblivious Secure XGBoost incurs two orders of magnitude overhead compared to Encrypted Secure XGBoost.

[4]https://github.com/dmlc/xgboost

[5]https://azure.microsoft.com/en-us/solutions/confidential-compute/

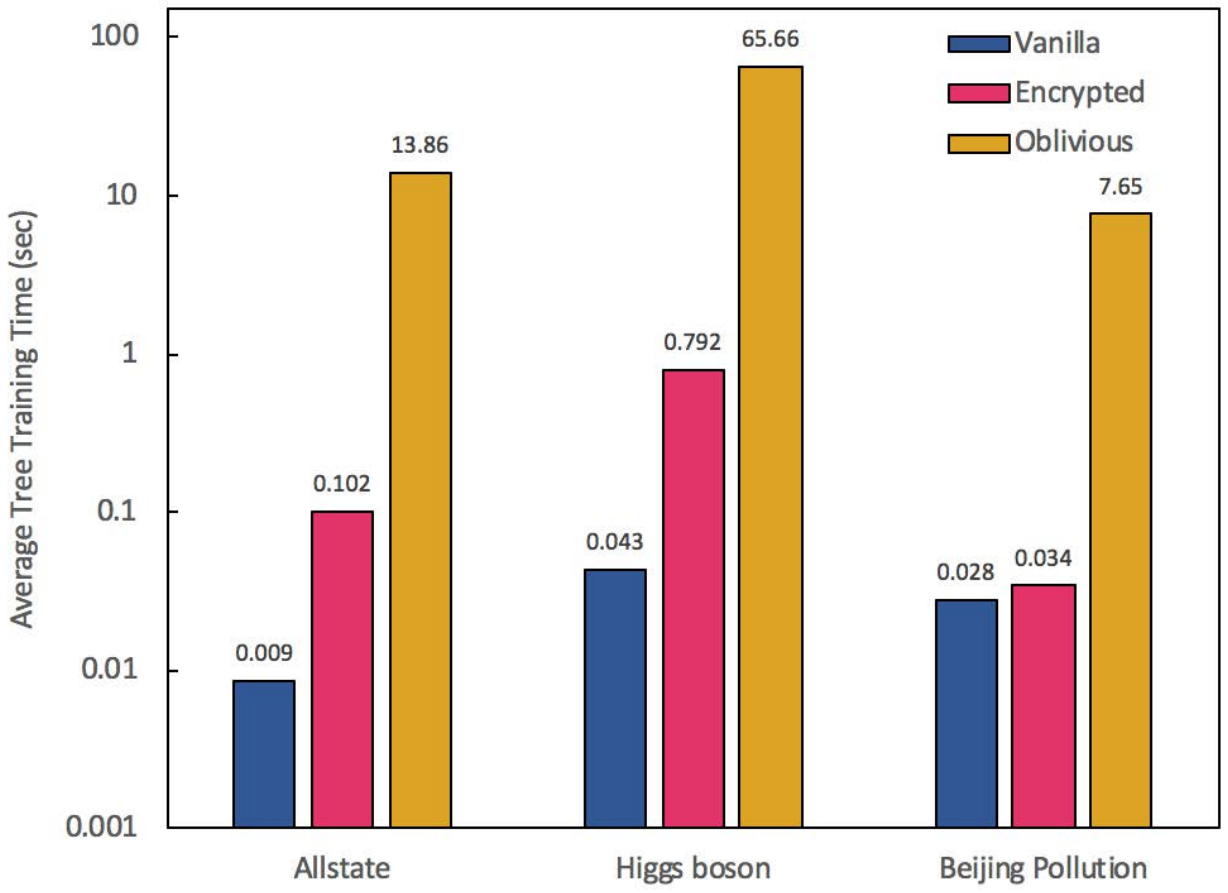

Figure 9.1: Average time to train one tree in each system, per dataset.

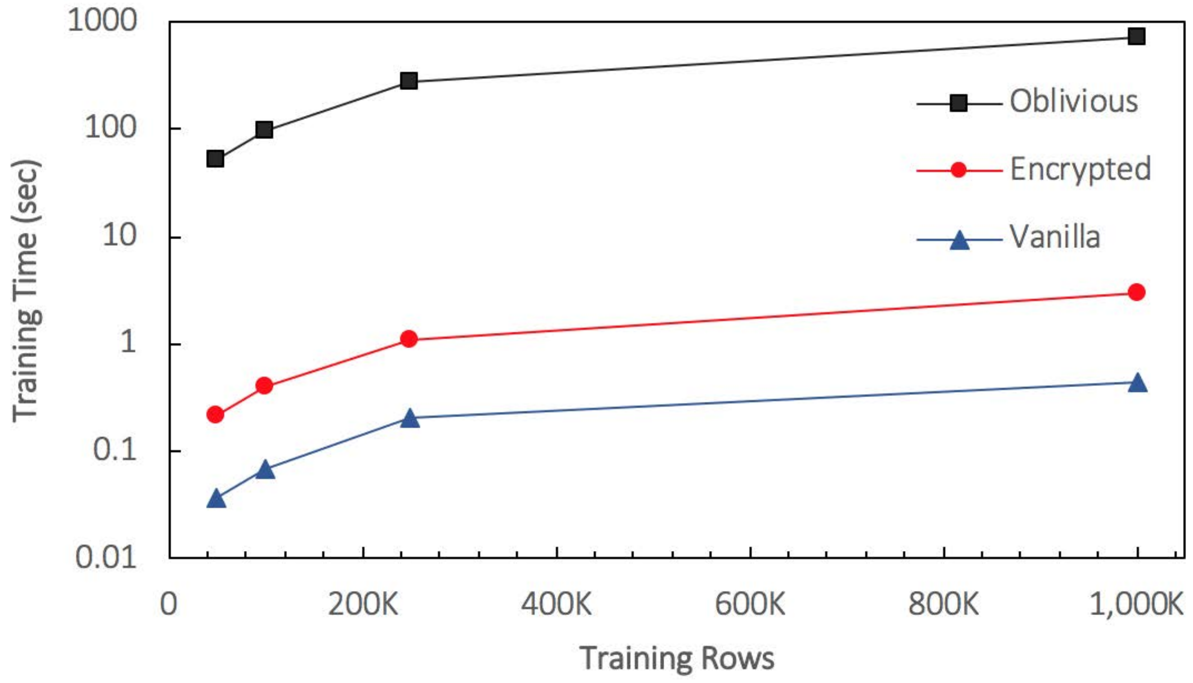

Figure 9.2: Comparison of training times for 50 trees over a varying number of rows of the Allstate Insurance dataset. We compare vanilla XGBoost, Encrypted Secure XGBoost, and Oblivious Secure XGBoost.

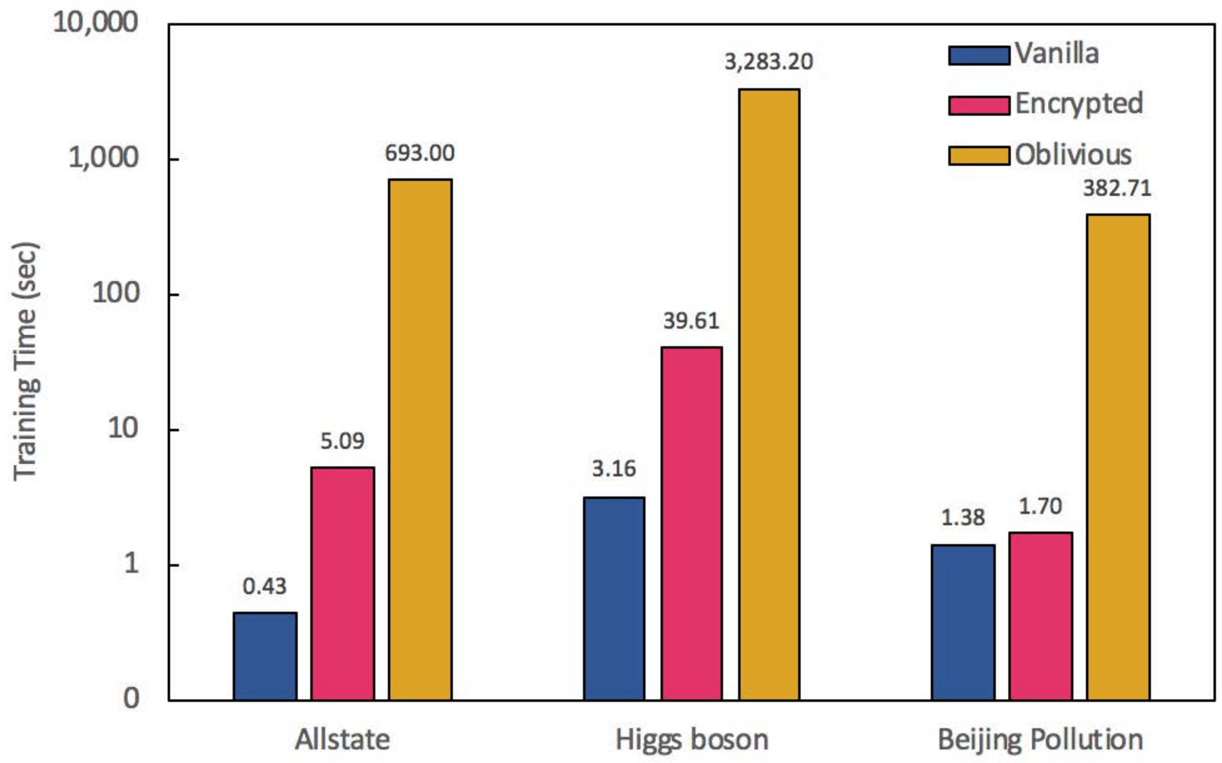

Figure 9.3: Overhead of Secure XGBoost's security. Encrypted Secure XGBoost incurs 0.2x–12.5x overhead compared to vanilla XGBoost, which provides no security. Oblivious Secure XGBoost incurs two orders of magnitude overhead compared to Encrypted Secure XGBoost.

Chapter 10

Deployment

We've been fortunate to collaborate with some teams in industry who have applied our work to real-world use cases. In this chapter we discuss each use case and the ongoing collaboration.

10.1 Scotiabank

Scotiabank is one of five institutions that dominate the Canadian banking industry, which together serve nearly 80% of all Canadians. We've been partnering with their fraud detection division for over a year to deploy Secure XGBoost and enable collaboration across all five banks in a joint anti-money laundering effort. This use case nearly identically matches one of our motivating scenarios: a group of banks want to collaborate to identify fraudsters and sex traffickers who attempt to dilute their trail by spreading their incriminating actions across multiple banks, but cannot simply share their data with one another because their customers' financial data is sensitive.

We've presented a demo of our open source implementation to Scotiabank's executive team, deployed Secure XGBoost as a library option in a Scotiabank-run hackathon for University of Toronto, and are currently undergoing a security review process to receive authorization to deploy to production. Scotiabank plans to deploy Secure XGBoost to production by early 2021.

10.2 Ant Financial

Ant Financial, a financial services provider affiliated with Alibaba, has been partnering with our team to deploy Secure XGBoost in production. They currently work with other financial institutions to approve loans, but the approval process is slow and requires manual effort. By using machine learning to model customer risk, the institutions can automate and expedite the entire process. Ant Financial has explored using gradient boosting for this purpose, and has been partnering with us specifically for side channel protection.

10.3 Ericsson

Ericsson has adopted Secure XGBoost to the federated setting. Machine learning, and in particular gradient boosting, has many applications in the telecommunications space, such as predicting hardware faults at cell sites [1]. However, because telecommunications systems are generally reliable, each operator owns little fault data. In addition, operators are hesitant to share their data due to competition, making this scenario a perfect fit for the secure collaborative setting.

Secure XGBoost served as a solution to these issues. We modified Secure XGBoost to this use case: we enabled Ericsson to compute a GBDT model over all operators data without revealing the data of any operator to any other operator or transferring the data away from its source by modifying the architecture of Secure XGBoost.

Even with no data leaving its source, Ericsson found that the accuracy of models trained in Secure XGBoost was comparable to the accuracy of models trained with an insecure library like XGBoost.

[1] https://medium.com/riselab/the-mc%C2%B2-platform-and-applications-in-telecom-24305fdcffa8

Chapter 11

Limitations and Future Work

11.1 Limitations

The current implementation of Secure XGBoost supports obliviousness with high overhead. The most immediate next step is to address this overhead by finding optimizations.

Secure XGBoost assumes that all organizations participating in the collaborative training process hold data with records containing the exact same features, i.e. that the training data fed into a model is horizontally partitioned. Currently, parties with vertically partitioned data, or that have data of different schemas, are unable to jointly compute a model.

Secure XGBoost also does not support any pre-processing, and assumes that any pre-processing is done locally at each party before data is transferred to the untrusted cloud. In particular, Secure XGBoost does not work with categorical features; categorical features must be first one-hot encoded during preprocessing. Existing solutions for local preprocessing include pandas [49, 68] and scikit-learn [60].

11.2 Future Work

An avenue to explore is designing a tailored MPC protocol specifically for GBDT. While general-purpose MPC protocols are prohibitively slow and impractical for large-scale systems, one can pick specific techniques that perform only certain types of computation to eliminate overhead [77]. Consequently, one can select and aggregate cryptographic techniques that enable the higher-performant computation of GBDT.

Another interesting area is to build a system that abstracts away the execution of the underlying computation, i.e., using hardware enclaves or MPC. While a tailored GBDT MPC protocol will have better performance than a general-purpose MPC protocol running GBDT, computing with hardware enclaves likely has lower latencies. However, the use of hardware enclaves requires (i) access to specialized hardware; and (ii) trust in the hardware vendor, e.g., Intel. If parties do not have one or the other, they may want to use MPC. Thus, a system that abstracts away the underlying compute technology and that enables users to choose enclaves or MPC by simply flipping a flag may be valuable for not only GBDT but also for other types of computation.

Chapter 12

Conclusion

In this paper we proposed Secure XGBoost, an oblivious distributed solution for gradient boosted decision trees using hardware enclaves. We presented our design of a system that supports the end-to-end collaborative learning of multiple parties following the outsourced computation model, and discussed methods to make gradient boosted decision trees data-oblivious. Over the lifetime of this project, we've partnered with multiple groups in industry to apply Secure XGBoost to their use cases, from which we've seen promising results.

Bibliography

[1] Arvind Arasu et al. "Orthogonal Security with Cipherbase". In: *CIDR*. 2013.

[2] ARM. *TrustZone*. https://developer.arm.com/ip-products/security-ip/trustzone.

[3] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX". In: *OSDI*. 2016.

[4] *Attestation Service for Intel SGX*. https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf.

[5] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *OSDI*. 2014.

[6] Raphael Bost et al. "Machine Learning Classification over Encrypted Data." In: *NDSS*. 2015.

[7] Thomas Bourgeat et al. "MI6: Secure Enclaves in a Speculative Out-of-Order Processor". In: *MICRO*. 2019.

[8] Marcus Brandenburger et al. "Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory". In: *DSN*. 2017.

[9] Ferdinand Brasser et al. "Software Grand Exposure: SGX Cache Attacks Are Practical". In: *WOOT*. 2017.

[10] Jo Van Bulck et al. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *USENIX Security*. 2018.

[11] Jo Van Bulck et al. "Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution". In: *USENIX Security*. 2017.

[12] Swarup Chandra et al. "Securing Data Analytics on SGX with Randomization". In: *ESORICS*. 2017.

[13] Guoxing Chen et al. "Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races". In: *IEEE S&P*. 2018.

[14] Guoxing Chen et al. "SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution". In: *EuroS&P*. 2019.

[15] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *KDD*. 2016.

[16] Kewei Cheng et al. "Secureboost: A Lossless Federated Learning Framework". In: *arXiv:1901.08755* (2019).

[17] Martine de Cock et al. "Fast, privacy preserving linear regression over distributed datasets based on pre-distributed data". In: *AISec*. 2015.

[18] Victor Costan, Ilia Lebedev, and Srinivas Devadas. "Sanctum: Minimal Hardware Extensions for Strong Software Isolation". In: *USENIX Security*. 2016.

[19] Fergus Dall et al. " CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks". In: *CHES*. 2018.

[20] Ivan Damgård et al. "Multiparty Computation From Somewhat Homomorphic Encryption". In: *CRYPTO*. 2012.

[21] Ankur Dave et al. "Oblivious Coopetitive Analytics Using Hardware Enclaves". In: *EuroSys*. 2020.

[22] Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. "CatBoost: Gradient Boosting with Categorical Features Support". In: *NeurIPS*. 2017.

[23] Wenliang Du and Zhijun Zhan. "Building Decision Tree Classifier on Private Data". In: *International Conference on Privacy, Security, and Data Mining*. 2002.

[24] Andrew Ferraiuolo et al. "Komodo: Using Verification to Disentangle Secure-Enclave Hardware From Software". In: *SOSP*. 2017.

[25] Arik Friedman and Assaf Schuster. "Data Mining with Differential Privacy". In: *KDD*. 2010.

[26] Sébastien Gambs, Balázs Kégl, and Esma Aïmeur. "Privacy-Preserving Boosting". In: *Data Mining and Knowledge Discovery* (2007).

[27] Johannes Götzfried et al. "Cache Attacks on Intel SGX". In: *EuroSec*. 2017.

[28] *gRPC*. https://grpc.io/.

[29] Daniel Gruss et al. "Another Flip in the Wall of Rowhammer Defenses". In: *IEEE S&P*. 2017.

[30] Marcus Hähnel, Weidong Cui, and Marcus Peinado. "High-Resolution Side Channels for Untrusted Operating Systems". In: *ATC*. 2017.

[31] Alon Halevy, Peter Norvig, and Fernando Pereira. "The Unreasonable Effectiveness of Data". In: *IEEE Intelligent Systems*. 2009.

[32] Xinran He et al. "Practical Lessons From Predicting Clicks on Ads at Facebook". In: *ADKDD*. 2014.

[33] Tyler Hunt et al. "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data". In: *OSDI*. 2018.

[34] *Intel Software Guard Extensions (SGX)*. https://software.intel.com/en-us/isa-extensions/intel-sgx/.

[35] Geetha Jagannathan, Krishnan Pillaipakkamnatt, and Rebecca N Wright. "A Practical Differentially Private Random Decision Tree Classifier". In: *ICDM*. 2009.

[36] Yeongjin Jang et al. "SGX-Bomb: Locking Down the Processor via Rowhammer Attack". In: *SysTEX*. 2017.

[37] David Kaplan, Jeremy Powell, and Tom Wolle. *AMD Memory Encryption*. 2016.

[38] Guolin Ke et al. "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In: *NeurIPS*. 2017.

[39] Dayeol Lee et al. "An Off-Chip Attack on Hardware Enclaves via the Memory Bus". In: *USENIX Security*. 2020.

[40] Dayeol Lee et al. "Keystone: An Open Framework for Architecting TEEs". In: *arXiv:1907.10119* (2019).

[41] Jaehyuk Lee et al. "Hacking in Darkness: Return-oriented Programming against Secure Enclaves". In: *USENIX Security*. 2017.

[42] Sangho Lee et al. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". In: *USENIX Security*. 2017.

[43] Qinbin Li et al. "Privacy-Preserving Gradient Boosting Decision Trees". In: *arXiv:1911.04209* (2019).

[44] Yehuda Lindell and Benny Pinkas. "Privacy Preserving Data Mining". In: *CRYPTO*. 2000.

[45] Zhuo Ma et al. "Lightweight Privacy-Preserving Ensemble Classification for Face Recognition". In: *IEEE Internet of Things Journal* (2019).

[46] Sinisa Matetic et al. "ROTE: Rollback Protection for Trusted Execution". In: *USENIX Security*. 2017.

[47] *MbedTLS*. https://github.com/ARMmbed/mbedtls.

[48] Frank McKeen et al. "Innovative Instructions and Software Model for Isolated Execution". In: *HASP*. 2013.

[49] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *SciPy*. 2010.

[50] Xiangrui Meng et al. "Mllib: Machine Learning in Apache Spark". In: *JMLR*. 2016.

[51] *Microsoft Azure Confidential Computing*. https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[52] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". In: *CHES*. 2017.

[53] Ahmad Moghimi et al. “MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations”. In: *CT-RSA*. 2018.

[54] Kit Murdock et al. “Plundervolt: Software-based Fault Injection Attacks Against Intel SGX”. In: *IEEE S&P*. 2020.

[55] Seong Joon Oh et al. “Towards Reverse-Engineering Black-Box Neural Networks”. In: *ICLR*. 2018.

[56] Olga Ohrimenko et al. “Oblivious Multi-Party Machine Learning on Trusted Processors”. In: *USENIX Security*. 2016.

[57] Oleksii Oleksenko et al. “Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks”. In: *ATC*. 2018.

[58] *Open Enclave*. https://openenclave.io/sdk/.

[59] Bryan Parno et al. “Memoir: Practical State Continuity for Protected Modules”. In: *IEEE S&P*. 2011.

[60] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *JMLR*. 2011.

[61] J. Ross Quinlan. “Induction of Decision Trees”. In: Springer, 1986.

[62] Ahmed Salem et al. “ML-Leaks: Model and Data Independent Membership Inference Attacks and Defenses on Machine Learning Models”. In: *NDSS*. 2019.

[63] Felix Schuster et al. “VC3: Trustworthy Data Analytics in the Cloud Using SGX”. In: *IEEE S&P*. 2015.

[64] Michael Schwarz et al. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: *DIMVA*. 2017.

[65] Michael Schwarz et al. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. In: *CCS*. 2019.

[66] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. “Machine Learning Models That Remember Too Much”. In: *CCS*. 2017.

[67] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. “CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management”. In: *USENIX Security*. 2017.

[68] The pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[69] Florian Tramèr et al. “Stealing Machine Learning Models via Prediction APIs”. In: *USENIX Security*. 2016.

[70] Chia-Che Tsai, Donald E. Porter, and Mona Vij. “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX”. In: *ATC*. 2017.

[71] Jaideep Vaidya et al. “Privacy-Preserving Decision Trees over Vertically Partitioned Data”. In: *TKDD*. 2008.

[72] Nico Weichbrodt et al. “AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves”. In: *ESORICS*. 2016.

[73] David J Wu et al. “Privately Evaluating Decision Trees and Random Forests”. In: *PETS*. 2016.

[74] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. “Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems”. In: *IEEE S&P*. 2015.

[75] Yuval Yarom, Daniel Genkin, and Nadia Heninger. “CacheBleed: a Timing Attack on OpenSSL Constant-Time RSA”. In: *CHES*. 2016.

[76] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *HotCloud*. 2010.

[77] Wenting Zheng et al. “Helen: Maliciously Secure Coopetitive Learning for Linear Models”. In: *IEEE S&P*. 2019.

[78] Wenting Zheng et al. “Opaque: An Oblivious and Encrypted Distributed Analytics Platform”. In: *NSDI*. 2017.