Secure Collaborative Training and Inference for XGBoost

Andrew Law[*]
UC Berkeley

Chester Leung
UC Berkeley

Rishabh Poddar
UC Berkeley

Raluca Ada Popa
UC Berkeley

Chenyu Shi
UC Berkeley

Octavian Sima
UC Berkeley

Chaofan Yu
Ant Financial

Xingmeng Zhang
Ant Financial

Wenting Zheng
UC Berkeley

Abstract

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 has the potential to greatly benefit all parties involved, but organizations have also encountered obstacles in sharing sensitive data due to business, regulatory, and liability concerns.

We propose Secure XGBoost, a privacy-preserving system that enables multiparty training and inference of XGBoost models. Secure XGBoost protects the privacy of each party's data as well as the integrity of the computation with the help of hardware enclaves. Crucially, Secure XGBoost augments the security of the enclaves using novel *data-oblivious* algorithms that prevent access side-channel attacks on enclaves induced via access pattern leakage.

CCS Concepts

• **Security and privacy → Distributed systems security**; **Side-channel analysis and countermeasures**.

Keywords

collaborative learning; hardware enclaves; data-obliviousness

ACM Reference Format:
Andrew Law, Chester Leung, Rishabh Poddar, Raluca Ada Popa, Chenyu Shi, Octavian Sima, Chaofan Yu, Xingmeng Zhang, and Wenting Zheng. 2020. Secure Collaborative Training and Inference for XGBoost. In *2020 Workshop on Privacy-Preserving Machine Learning in Practice (PPMLP'20), November 9, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3411501.3419420

1 Introduction

Secure XGBoost is a platform for secure collaborative gradient-boosted decision tree learning, based on the popular XGBoost library. In a nutshell, multiple clients (or data owners) can *collaboratively* use Secure XGBoost to train an XGBoost model on their collective data in a cloud environment while preserving the privacy of their individual data. Even though training is done on the cloud, Secure XGBoost ensures that the data of individual clients is revealed to neither the cloud environment nor other clients. Clients collaboratively orchestrate the training pipeline remotely, and Secure XGBoost guarantees that each client retains control of the computation that runs on its individual data.

At its core, Secure XGBoost leverages the protection offered by *secure hardware enclaves* to preserve the privacy of the data and the integrity of the computation even in the presence of a hostile cloud environment. On top of enclaves, Secure XGBoost adds a second layer of security that additionally protects the enclaves against a large class of *side-channel attacks*—namely, attacks induced by access pattern leakage (see §2.2). Even though the attacker cannot directly observe the data protected by the enclave, it can still infer sensitive information about the data by monitoring the enclave's memory access patterns during execution. To prevent such leakage, we redesign the training and inference algorithms in XGBoost to be *data-oblivious*, guaranteeing that the memory access patterns of enclave code does not reveal any information about sensitive data.

In implementing Secure XGBoost, we strived to preserve the XGBoost API as much as possible so that our system remains easy to use for data scientists. Our implementation has been adopted by multiple industry partners, and is available at https://github.com/mc2-project/secure-xgboost.

2 Background

2.1 Hardware enclaves

Secure enclaves are a recent advance in computer processor technology that enable the creation of a secure region of memory (called an enclave) on an otherwise untrusted machine. Any data or software placed in the enclave is isolated from the rest of the system, and no other process on the same processor (not even privileged software like the OS or hypervisor) can access or tamper with that memory. Examples of secure enclave technology include Intel SGX [21] and AMD Memory Encryption [16].

An important feature of hardware enclaves is remote attestation [1], which allows a remote client system to cryptographically verify that specific software has been securely loaded into an enclave. As part of attestation, the enclave can also bootstrap a secure channel with the client by generating a public key and returning it with the signed report.

2.2 Side-channel leakage

A large class of known side-channel attacks on enclaves exploit data-dependent access patterns—i.e., the sequence of accesses made by the executing program to disk, network, or memory. The attacker

[*] Authors ordered alphabetically

PPMLP'20, November 9, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8088-1/20/11.
https://doi.org/10.1145/3411501.3419420

```
void max(int x, int y,
         int* z) {
  if (x > y)
    *z = x;
  else
    *z = y;
}
```

Figure 1: Regular code

```
void max(int x, int y,
         int* z) {

  bool cond = ogreater(x, y);
  oassign(cond, x, y, z);

}
```

Figure 2: Oblivious code

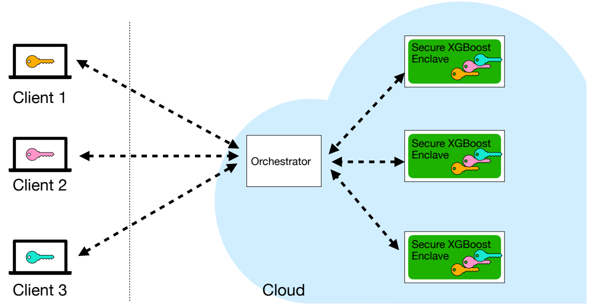

Figure 3: Parties invoke an orchestrator service at the cloud, which waits for calls from all parties before relaying the commands to the enclave cluster. Enclave inputs and outputs are always encrypted, and are decrypted only within the enclave or at client premises.

can observe the access sequence in a variety of ways: e.g., cache-timing attacks [4, 9, 10, 13, 22, 30], branch prediction attacks [18], page monitoring [6, 32], or snooping on the memory bus [17].

Example. As an example, consider the code in Figure 1 that determines the maximum of two integers using a non-oblivious if-else statement. An attacker observing the memory addresses of accessed program instructions can identify whether $x > y$, depending on whether the code within the if-block or the else-block gets executed.

2.3 Data-obliviousness

Oblivious computation is a type of cryptographic computation that prevents the aforementioned attacks by removing data-dependent access patterns. Consequently, a data-oblivious enclave program prevents an attacker from inferring information about the underlying data by observing memory, disk, or network accesses.

In Secure XGBoost, we design and implement data-oblivious algorithms for model training and inference. In particular, our algorithms produce an *identical sequence* of disk, network and memory accesses that depend only on the public information, and are *independent* of the input data. Hence, they provably prevent all side-channels induced by access pattern leakage.

3 Overview

3.1 System model

In this section, we describe the different entities in a Secure XGBoost deployment. The entities consist of: (i) multiple data owners (or clients) who wish to collaboratively train a model on their individual data; and (ii) an untrusted cloud service that hosts the Secure XGBoost platform within a cluster of machines. The general architecture of Secure XGBoost is depicted in Figure 3.

Clients. A client refers to a party who wants to jointly train a model with other clients. The clients collectively execute the computation pipeline on the Secure XGBoost platform by remotely invoking its APIs.

Cloud service with enclaves. The cloud service consists of a cluster of virtual machines, each with hardware enclave support. Secure XGBoost distributes the computation across the cluster of hardware enclaves, which communicate with each other over TLS channels that begin and end inside the enclaves.

Additionally, an orchestrator service at the cloud mediates communication between clients and the Secure XGBoost platform deployed within enclaves.

3.2 Workflow

The following describes an end-to-end example workflow for using Secure XGBoost. We use the term 'command' to refer to a client's desired execution of a step in the computation process, i.e., the APIs exposed by Secure XGBoost for data loading, training, etc.

(1) The clients agree on a pre-determined sequence of commands that will be executed on Secure XGBoost (§5.1).
(2) Clients attest the enclaves on the cloud (via the remote attestation procedure) to verify that the expected Secure XGBoost code has been securely loaded within each enclave (§5.2).
(3) Each client C_i encrypts its data with a symmetric key k_i and uploads it to cloud storage (§5.3).
(4) The clients submit signed commands to the orchestrator. The orchestrator aggregates all the client signatures and relays each command to Secure XGBoost. Secure XGBoost authenticates the signatures, ensuring that every client indeed issued the same command, and executes the command (§5.4).
(5) Secure XGBoost returns the results of the command (e.g., an encrypted trained model, or encrypted prediction results) to the orchestrator, who relays it to the clients. The process continues until all commands have been executed.

4 Threat model and security guarantees

We describe the aims and capabilities of the attackers that Secure XGBoost protects against.

4.1 Threat model for the cloud and hardware enclaves

The cloud service provider and the orchestrator service are untrusted. The trusted computing base includes the CPU package and its hardware enclave implementation, as well as our implementation of Secure XGBoost.

The design of Secure XGBoost is not tied to any specific hardware enclave; instead, Secure XGBoost builds on top of an *abstract* model of hardware enclaves where the attacker controls the server's software stack outside the enclave (including the OS), but cannot perform any attacks to glean information from inside the processor (including processor keys). The attacker can additionally observe the contents and access patterns of all (encrypted) pages in memory, for both data and code. We assume that the attacker can observe the enclave's memory access patterns at cache line granularity.

Secure XGBoost provides protection against *all channels of attack that exploit data-dependent access patterns at cache-line granularity*, which represent a large class of known attacks on enclaves (e.g., [4, 6, 10, 13, 17, 18, 22, 30, 32]). Other attacks that violate our abstract enclave model—such as attacks based on timing analysis or power consumption [23, 31], denial-of-service attacks [12, 15], or rollback attacks [26] (which have complementary solutions [3, 19])—are out of scope. Transient execution attacks (e.g., [5, 7, 29]) are also out of scope; these attacks violate the threat model of SGX and are typically patched promptly by the enclave vendor via microcode updates.

4.2 Threat model for the clients

Each client expects to protect its data from the cloud service hosting the enclaves, as well as the other clients in the collaboration. Malicious clients may collude with each other and/or the cloud service to try and learn a victim client's data. They may also attempt to subvert the integrity of the computation by tampering with the computation steps (i.e., the commands submitted for execution). Secure XGBoost protects the client data and computation in accordance with the threat model and guarantees from §4.1.

5 System Design

5.1 System setup

Secure XGBoost is launched at the cloud service within enclaves. It contains an embedded list of client names, along with the public key of a trusted certificate authority (CA), which it uses to verify a client's identity before establishing a connection with the client (described in §5.2). A single "master" enclave generates a 2048-bit RSA key pair (pk, sk) and a nonce N. The public key will be used to establish a secure channel of communication with the clients, and the nonce to ensure freshness of communicated messages.

Each client C_i generates a 256-bit symmetric key k_i. Each client also has its own 2048-bit RSA key pair (pk_i, sk_i), along with a certificate signed by a certificate authority (CA); the CA's public key is embedded in Secure XGBoost. The clients will use the certificate to authenticate themselves to Secure XGBoost.

5.2 Client-server attestation

Clients authenticate the Secure XGBoost deployment within the enclave cluster via remote attestation (as described in §2.1). More precisely, we logically arrange the enclaves in a tree topology; the enclave at the root of the tree is the "master" enclave. During attestation, each client attests only the "master" enclave to verify that the expected Secure XGBoost code has been securely loaded; in turn, each enclave in the tree (including the master) attests its children enclaves. As part of the attestation process, the enclaves establish TLS sessions with their neighboring enclaves. In addition, the master enclave sends the generated public key pk and a nonce N to the clients along with the signed attestation report.

Each client encrypts its key k_i using the enclave's public key pk, and signs the message. It then sends the signed message to the master enclave along with its certificate. The master enclave verifies each client's signed message, decrypts the symmetric key k_i, and percolates k_i to all attested enclaves in the cluster, giving each enclave the ability to decrypt data belonging to the client.

5.3 Data preparation and transfer

Each client uploads its encrypted data to cloud storage; enclaves retrieve the encrypted data from storage before training. To enable distributed data processing, each enclave must retrieve only a partition of the encrypted training data. This requirement precludes each client from encrypting its data as a single blob. Instead, to facilitate distributed processing of the encrypted data, clients encrypt each row in their data separately, which enables each enclave to retrieve, decrypt, and process only a subset of the rows.

Specifically, client C_i encrypts each row in its data (using its symmetric key k_i) as follows:

$$j,\ \ n_i,\ \ \mathtt{Enc}(\mathtt{row}_j),\ \ \mathtt{MAC}(j||n_i||\mathtt{Enc}(\mathtt{row}_j))$$

Here, j is the index number of the row being encrypted; n_i is the total number of rows in C_i's data; $\mathtt{Enc}(\mathtt{row}_j)$ is an AES-GCM ciphertext over the j-th row; and $\mathtt{MAC}(j||n_i||\mathtt{Enc}(\mathtt{row}_j))$ is an AES-GCM authentication tag computed over the ciphertext, the index number j and the total number of rows n_i. 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).

While processing a client's data, each enclave retrieves a subset of the encrypted rows. The enclaves then communicate to ensure that they together loaded n_i rows, and that all row indices from $j = 1 \ldots n_i$ were present in the retrieved data.

5.4 Collaborative API execution

Once all clients have uploaded their data to the cloud, they collectively invoke the APIs exposed by Secure XGBoost. Each API invocation requires consensus—Secure XGBoost executes an API call only if it receives the command from every client. This ensures that no processing can be performed on a particular client's data without that client's consent.

To make an API call, each client submits a signed command to the orchestrator:

$$\mathtt{cmd} = \mathtt{<seqn, func, params>},\ \ \mathtt{Sign(cmd)}$$

A command contains three fields: (i) a sequence number $\mathtt{seqn} = (N||\mathtt{ctr})$ that consists of the nonce N (obtained from the enclaves during attestation) concatenated with an incrementing counter; (ii) the API function `func` being invoked; and (iii) the function parameters `params`. Including the sequence number ensures the freshness of the command, and prevents replay attacks on the system. The orchestrator aggregates the signed commands and relays them to the enclave cluster. Each enclave verifies that an identical command was submitted by every client before executing the corresponding function.

Once the function completes, Secure XGBoost produces a signed response and returns it to the clients via the orchestrator:

$$\mathtt{resp} = \mathtt{<seqn, result>},\ \ \mathtt{Sign(resp)}$$

The response contains the sequence number of the request (to cryptographically bind the response to the request), along with the results of the function (which are potentially encrypted with the clients' keys, depending on the function that was invoked).

6 Data-oblivious training and inference

To prevent side-channel leakage via access patterns, we design data-oblivious algorithms for training and inference. To implement the algorithms, we use a small set of data-oblivious primitives, based on those from prior work [24, 27]. In this section, we first describe the primitives, and then show their usage in our algorithms.

6.1 Oblivious primitives

Our oblivious primitives operate solely on registers whose contents are loaded from and stored into memory using deterministic memory accesses. Since registers are private to the processor, any register-to-register operations cannot be observed by the attacker.

1) Oblivious comparisons (`oless`, `ogreater`, `oequal`). These primitives can be used to obliviously compare variables, and are wrappers around the x86 `cmp` instruction.

2) Oblivious assignment (`oassign`). The `oassign` primitive performs conditional assignments, moving a source to a destination register if a condition is true.

3) Oblivious sort (`osort`). The `osort` primitive obliviously sorts a size n array by passing its inputs through a bitonic sorting network [2], which performs an identical sequence of $O(n \log^2(n))$ carefully arranged compare-and-swap operations regardless of the input array values.

4) Oblivious array access (`oaccess`). The `oaccess` primitive accesses the i-th element in an array without leaking i itself by scanning the array at cache-line granularity while performing `oassign` operations, setting the condition to true only at index i.

Example. To show how these primitives can be used to implement higher-level data-oblivious code, Figure 2 depicts the data-oblivious version of the `max` program from Figure 1. In this version, all instructions are executed sequentially, without any secret-dependent branches, causing the program to have identical memory access patterns regardless of the inputs values.

6.2 Oblivious training

Each enclave in the cluster loads a subset of the collected data, and then uses a distributed algorithm to train the model. In particular, we use XGBoost's histogram-based distributed algorithm (`hist`) for training an approximate model [8, 14], but redesign the algorithm in order to make it data-oblivious. In this algorithm, the data samples always remain distributed across all the enclave machines in the cluster, and the machines only exchange data summaries with each other. The summaries are used to construct a single tree globally and add it to the model's ensemble.

At a high level, the `hist` algorithm builds a tree in rounds, adding a node to the tree per round. Given a data sample $x \in \mathbb{R}^d$, at each node the algorithm chooses a feature j and a threshold t according to which the data samples are partitioned (i.e., if $x(j) < t$, the sample is partitioned into the left subtree, otherwise the right). To add a node to the tree, each enclave in the cluster builds a histogram over its data for each feature; the boundaries of the bins in the histogram serve as potential splitting points for the corresponding feature. The algorithm combines the histograms across enclaves, and uses the aggregate statistics to find the best feature and splitting point. Note that in the absence of data-obliviousness the algorithm reveals a large amount of information via access-pattern leakage: e.g., it leaks which feature was chosen at each node in the tree, as well the complete ordering of the data samples. We now describe the oblivious algorithm in more detail.

Oblivious histogram initialization. Before a tree can be constructed, all the enclaves in the cluster first align on the boundaries of the histograms per feature. These boundaries are computed once and re-used for adding all the nodes in the tree, instead of computing new histogram boundaries per node.

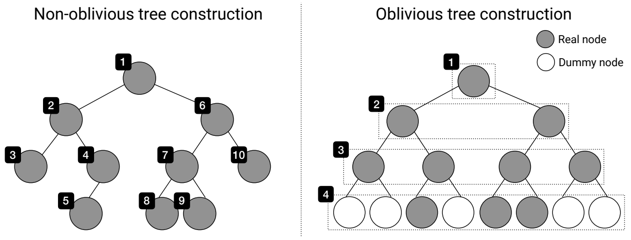

Figure 4: Illustration of oblivious training in Secure XGBoost. Numbers indicate the order in which nodes are added. Non-oblivious training adds nodes sequentially to the tree, while our algorithm constructs a full binary tree while adding nodes level-wise.

(1) Each enclave first 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 samples using `osort`. Then, it initializes an empty array S of size equal to the number of samples. 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 `oselect`, either setting it to 0 (if $x_{i-1} = x_i$), 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 $b + 1$ (where b is a user-defined parameter for the maximum number of bins in the histogram). The aim of the pruning operation is to select $b + 1$ elements from the list with ranks $0, \frac{|S|}{b}, \frac{2|S|}{b} \ldots |S|$, where $|S|$ is the size of the summary. We do this obliviously as follows. First, the enclave sorts the summary using `osort`. Next, it scans the sorted summary, and for each element in the summary, it selects the element (using `oassign`) if its rank matches the next rank to be selected, otherwise it selects a dummy. Finally, it sorts the selected elements (which includes dummies), pushing the dummy elements to the end, and truncates the list.

(3) Next, each enclave broadcasts its summary S. The summaries are pairwise combined 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 tuples in the merged summary are then scanned to identify adjacent values that are duplicates; the 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.

The global summary per feature computed in this manner represents the bins of a histogram, with the constituent values in the summary as the boundaries of different bins.

Oblivious node addition. The algorithm uses the feature histograms to construct a tree, adding nodes to the tree starting with the root. As nodes get added to the tree, the data gets partitioned at each node across its children. Here, we describe an oblivious subroutine for obliviously adding a node by finding the optimal split for the node, using the data samples that belong to the node.

(1) Each enclave computes a histogram for each feature by scanning its data samples to compute a gradient per sample, followed by updating a single bin in each histogram using `oaccess` combined with `oassign`. The enclaves then broadcast their histograms.
(2) The enclaves collectively sum up the histograms. Each enclave then computes a score function over the aggregated histogram, deterministically identifying the best feature to split by, as well as the split value.
(3) Finally, each enclave partitions its data based on the split value: it simply updates a marker per sample (using `oassign`) that identifies which child node the sample belongs to.

Level-wise oblivious tree construction. A simple way to construct a tree is to sequentially add nodes to the tree as described above, until the entire tree is constructed. To prevent leaking information about the data or the tree: (i) the order in which nodes are added needs to be independent of the data; and (ii) a fixed number of nodes need to be added to the tree. At the same time, adding nodes sequentially by repeatedly invoking the node addition subroutine above is sub-optimal for performance. This is because oblivious node addition only uses the data that belongs to the node; however, concealing which data samples belong to the node either requires accessing each sample using `oaccess`, or scanning all the samples while performing dummy operations for those that do not belong to the node. Both these options impact performance adversely.

We simultaneously solve all the problems above by sequentially adding entire levels to the tree, instead of individual nodes. That is, we obliviously add all the nodes at a particular level of a tree in a *single scan* of all the data samples, as follows. For each data sample, we first use `oaccess` to obliviously fetch the histograms of the node that the sample belongs to. We then update the histograms as described in the subroutine above, and then obliviously write back the histogram to the node using `oaccess`.

Note that as a result of level-wise tree construction, we always build a full binary tree (unlike the non-oblivious algorithm) and some nodes in the tree are "dummy" nodes. These nodes are ignored during inference. Figure 4 illustrates how nodes are added to the tree during our oblivious training routine.

6.3 Oblivious inference

Inference normally occurs by traversing a tree from root to leaf and comparing the feature value of each interior node with the corresponding feature in the test data instance. To obliviously evaluate an XGBoost model on a data instance, we follow [24]. In summary, we store each layer in the tree as an array, use the `oaccess` primitive to obliviously select the proper node at that layer, and use the `oless` primitive for comparison.

7 Implementation

In this section we describe our implementation of Secure XGBoost and discuss a few use cases in industry.

7.1 Implementation

We implemented a prototype of Secure XGBoost based on XGBoost version 0.9. Following XGBoost's implementation model, we provide a Python API on top of a core C++ library, imitating the XGBoost API as much as possible. An example of the Secure XGBoost API is shown in Figure 5. We used the Open Enclave SDK [25] to interface between the untrusted host and the enclave and to enable Secure XGBoost to run agnostic of a specific hardware enclave; Mbed TLS [20] for cryptography and for secure communication between enclaves; and gRPC [11] for client-server communication.

```
import securexgboost as xgb

# Initialize client and connect to enclave cluster
xgb.init_client(user_name="user1",
            sym_key_file="key.txt",
            priv_key_file="user1.pem",
            cert_file="user1.crt")

# Client side remote attestation to authenticate enclaves
xgb.attest()

# Load the encrypted data and associate it with a user
dtrain = xgb.DMatrix({"user1": "train.enc"})
dtest = xgb.DMatrix({"user1": "test.enc"})

params = {
   "objective": "binary:logistic",
   "gamma": "0.1",
   "max_depth": "3"
}

# Train a model
num_rounds = 5
booster = xgb.train(params, dtrain, num_rounds)

# Get encrypted predictions and decrypt them
predictions, num_preds = booster.predict(dtest)
```

Figure 5: Example client code in Secure XGBoost. Functions highlighted in red are additions to the existing XGBoost library. Functions highlighted in blue exist in XGBoost but were modified for Secure XGBoost.

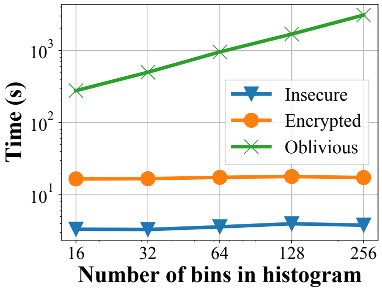

Figure 6: Evaluation comparison among the insecure baseline, and encrypted as well as oblivious Secure XGBoost

Our codebase is open source and available at https://github.com/mc2-project/secure-xgboost.

7.2 Adoption

We've been fortunate enough to work with several collaborators in industry, each of whom has been using our system for a different purpose. Ericsson used Secure XGBoost to bring together mutually distrustful network operators to collaborate in applications such as predicting cell tower hardware faults [28]. Scotiabank has been leading an effort with other Canadian banks to use Secure XGBoost to fight money laundering. Finally, Ant Financial is using Secure XGBoost in production for credit loan risk modeling.

8 Evaluation results

We ran experiments on Secure XGBoost using a synthetic dataset obtained from Ant Financial, consisting of 100, 000 data samples with 126 features. Our experiments compare three systems: vanilla XGBoost; encrypted Secure XGBoost (a version of Secure XGBoost without obliviousness); and oblivious Secure XGBoost (Secure XGBoost with obliviousness enabled). We ran our experiments on Microsoft's Azure Confidential Computing service. We used DC4s_V2 machines, which have support for Intel SGX enclaves, and are equipped with 4 vCPUs, 16 GiB of memory, and a 112 MiB enclave page cache.

Figure 6 shows our training results. In general, encrypted Secure XGBoost incurs $4.5\times - 5.1\times$ overhead compared to vanilla XGBoost, which provides no security. Oblivious Secure XGBoost incurs $16.7\times - 178.2\times$ overhead over encrypted Secure XGBoost. The main takeaway is that one has to be careful in tuning the hyperparameters by adjusting the number of bins, the number of levels per tree and the number of trees. For example, decreasing the number of bins while increasing the number of trees could improve performance while maintaining the same accuracy.

9 Conclusion

In this paper we proposed Secure XGBoost, an oblivious distributed solution for gradient boosted decision trees using hardware enclaves. Our codebase is available at https://github.com/mc2-project/secure-xgboost, and we are currently working with industry collaborators to deploy our system.

Acknowledgments

This work was supported in part by the NSF CISE Expeditions Award CCF-1730628, and gifts from the Sloan Foundation, Bakar Program, Alibaba, Amazon Web Services, Ant Group, Capital One, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

References

[1] Ittai Anati, Shay Gueron, Simon P. Johnson, and Vincent R. Scarlata. 2013. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP).*

[2] K. E. Batcher. 1968. Sorting Networks and Their Applications. In *Proceedings of the Spring Joint Computer Conference.*

[3] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. 2017. Rollback and Forking Detection for Trusted Execution Environments using Lightweight Collective Memory. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems & Networks (DSN).*

[4] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT).*

[5] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the USENIX Security Symposium.*

[6] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proceedings of the USENIX Security Symposium.*

[7] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P).*

[8] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. *CoRR* abs/1603.02754 (2016). arXiv:1603.02754 http://arxiv.org/abs/1603.02754

[9] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. 2018. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. In *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES).*

[10] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. In *Proceedings of the European Workshop on Systems Security (EuroSec).*

[11] gRPC. 2020. https://grpc.io/docs/.

[12] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2017. Another Flip in the Wall of Rowhammer Defenses. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P).*

[13] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-Resolution Side Channels for Untrusted Operating Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC).*

[14] Histogram-based training in XGBoost. 2020. https://github.com/dmlc/xgboost/issues/1950.

[15] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *Proceedings of the Workshop on System Software for Trusted Execution (SysTEX).*

[16] David Kaplan, Jeremy Powell, and Tom Woller. 2013. AMD Memory Encryption. Whitepaper. (2013).

[17] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. 2020. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *Proceedings of the USENIX Security Symposium.*

[18] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the USENIX Security Symposium.*

[19] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *Proceedings of the USENIX Security Symposium.*

[20] Mbed TLS. 2020. https://tls.mbed.org/.

[21] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP).*

[22] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies The Power of Cache Attacks. In *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES).*

[23] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P).*

[24] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proceedings of the USENIX Security Symposium.*

[25] Open Enclave 2020. Open Enclave SDK. (2020). https://openenclave.io.

[26] Bryan Parno, Jay Lorch, John Douceur, James Mickens, and Jonathan M. McCune. 2011. Memoir: Practical State Continuity for Protected Modules. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P).*

[27] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. 2020. Visor: Privacy-Preserving Video Analytics as a Cloud Service. In *Proceedings of the USENIX Security Symposium.*

[28] Raluca Ada Popa and Anila Joshi. 2020. 3 ways to train a secure machine learning model. https://www.ericsson.com/en/blog/2020/2/training-a-machine-learning-model.

[29] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS).*

[30] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA).*

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

[32] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P).*