Data-oblivious training for XGBoost models

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 Group
Xingmeng Zhang Ant Group
Wenting Zheng UC Berkeley

1 Introduction

Secure XGBoost [16, 28] 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). 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.

In this paper, we describe how we prevent such leakage by redesigning the training algorithms in XGBoost to be *data-oblivious*, guaranteeing that the memory access patterns of enclave code does not reveal any information about sensitive data.

2 Background

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 [20] and AMD Memory Encryption [15]. 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.

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 can observe the access sequence in a variety of ways: e.g., cache-timing attacks [10, 4, 27, 21, 12, 9], branch prediction attacks [18], page monitoring [30, 6], 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

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

```
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

program instructions can identify whether $x > y$, depending on whether the code within the if-block or the else-block gets executed.

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.

3 System setup and threat model

System setup. 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 our platform on a compute cluster and trains the clients' combined dataset.

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 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., [10, 4, 27, 21, 12, 18, 30, 6, 17]). Other attacks that violate our abstract enclave model—such as attacks based on timing analysis or power consumption [22, 29], denial-of-service attacks [14, 11], or rollback attacks [24] (which have complementary solutions [3, 19])—are out of scope. Transient execution attacks (e.g., [5, 26, 7]) 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.

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 §3.

4 Data-oblivious training

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 [23, 25], and build our oblivious algorithms on top of these primitives.

4.1 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. The set of primitives we use are oblivious comparison (`oless`, `ogreater`, `oequal`), assignment (`oassign`), sort (`osort`), and array access (`oaccess`) operations. Details are presented in Appendix A.

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.

4.2 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, 13], 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.

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.

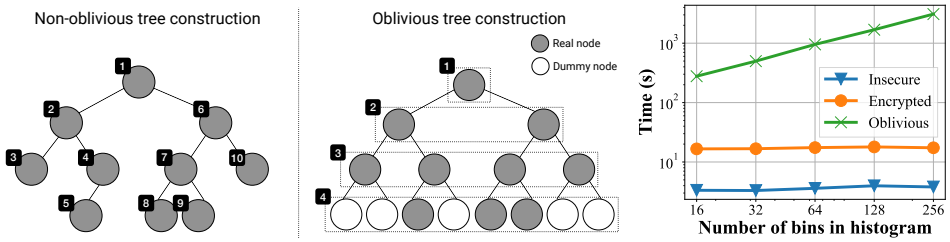

Figure 3: a) Illustration of oblivious training in Secure XGBoost. Numbers indicate the order in which nodes are added. Non-oblivious training adds nodes sequentially to the tree, while our algorithm constructs a full binary tree while adding nodes level-wise. b) Evaluation comparison among the insecure baseline, and encrypted as well as oblivious Secure XGBoost

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 3a illustrates how nodes are added to the tree during our oblivious training routine.

5 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 machines with support for Intel SGX enclaves, and that are equipped with 4 vCPUs, 16 GiB of memory, and a 112 MiB enclave page cache.

Figure 3b 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.

A Oblivious primitives

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.

References

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

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

[3] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza. 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)*, 2017.

[4] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[5] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the USENIX Security Symposium*, 2018.

[6] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proceedings of the USENIX Security Symposium*, 2017.

[7] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[8] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. *CoRR*, abs/1603.02754, 2016.

[9] F. Dall, G. D. Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. In *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2018.

[10] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache Attacks on Intel SGX. In *Proceedings of the European Workshop on Systems Security (EuroSec)*, 2017.

[11] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. Another Flip in the Wall of Rowhammer Defenses. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2017.

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

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

[14] Y. Jang, J. Lee, S. Lee, and T. Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *Proceedings of the Workshop on System Software for Trusted Execution (SysTEX)*, 2017.

[15] D. Kaplan, J. Powell, and T. Woller. AMD Memory Encryption. Whitepaper, 2013.

[16] A. Law, C. Leung, R. Poddar, R. A. Popa, C. Shi, O. Sima, C. Yu, X. Zhang, and W. Zheng. Secure collaborative training and inference for XGBoost. In *ACM CCS Workshop on Privacy-Preserving Machine Learning in Practice*, 2020.

[17] D. Lee, D. Jung, I. T. Fang, C.-C. Tsai, and R. A. Popa. An Off-Chip Attack on Hardware Enclaves via the Memory Bus. In *Proceedings of the USENIX Security Symposium*, 2020.

[18] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the USENIX Security Symposium*, 2017.

[19] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun. ROTE: Rollback Protection for Trusted Execution. In *Proceedings of the USENIX Security Symposium*, 2017.

[20] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[21] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[22] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.

[23] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proceedings of the USENIX Security Symposium*, 2016.

[24] B. Parno, J. Lorch, J. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical State Continuity for Protected Modules. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2011.

[25] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa. Visor: Privacy-Preserving Video Analytics as a Cloud Service. In *Proceedings of the USENIX Security Symposium*, 2020.

[26] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[27] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2017.

[28] Secure XGBoost. 2020. `https://github.com/mc2-project/secure-xgboost`.

[29] A. Tang, S. Sethumadhavan, and S. Stolfo. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *Proceedings of the USENIX Security Symposium*, 2017.

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