Oblivious Multi-Party Machine Learning on Trusted Processors

Olga Ohrimenko, Felix Schuster, and Cédric Fournet, *Microsoft Research;*
Aastha Mehta, *Microsoft Research and Max Planck Institute for Software Systems (MPI-SWS);*
Sebastian Nowozin, Kapil Vaswani, and Manuel Costa, *Microsoft Research*

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/ohrimenko

This paper is included in the Proceedings of the
25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

Oblivious Multi-Party Machine Learning on Trusted Processors

Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta,[*] Sebastian Nowozin
Kapil Vaswani, Manuel Costa
Microsoft Research

Abstract

Privacy-preserving multi-party machine learning allows multiple organizations to perform collaborative data analytics while guaranteeing the privacy of their individual datasets. Using trusted SGX-processors for this task yields high performance, but requires a careful selection, adaptation, and implementation of machine-learning algorithms to provably prevent the exploitation of any side channels induced by data-dependent access patterns.

We propose data-oblivious machine learning algorithms for support vector machines, matrix factorization, neural networks, decision trees, and k-means clustering. We show that our efficient implementation based on Intel Skylake processors scales up to large, realistic datasets, with overheads several orders of magnitude lower than with previous approaches based on advanced cryptographic multi-party computation schemes.

1 Introduction

In many application domains, multiple parties would benefit from pooling their private datasets, training precise machine-learning models on the aggregate data, and sharing the benefits of using these models. For example, multiple hospitals might share patient data to train a model that helps in diagnosing a disease; having more data allows the machine learning algorithm to produce a better model, benefiting all the parties. As another example, multiple companies often collect complementary data about customers; sharing such data would allow machine learning algorithms to make joint predictions about customers based on a set of features that would not otherwise be available to any of the companies. This scenario also applies to individuals. For example, some systems learn an individual's preferences to make accurate recommendations [9]; while users would like to keep their data private, they want to reap the rewards of correlating their preferences with those of other users.

Secure multi-party computation [17, 24, 44] and fully homomorphic encryption [23] are powerful cryptographic tools that can be used for privacy preserving machine learning. However, recent work [38, 47, 48, 54] reports large runtime overheads, which limits their practical adoption for compute-intensive analyses of large datasets. We propose an alternative privacy-preserving multi-party machine learning system based on trusted SGX processors [45]. In our system, multiple parties agree on a joint machine learning task to be executed on their aggregate data, and on an SGX-enabled data center to run the task. Although they do not trust one another, they can each review the corresponding machine-learning code, deploy the code into a processor-protected memory region (called an enclave), upload their encrypted data just for this task, perform remote attestation, securely upload their encryption keys into the enclave, run the machine learning code, and finally download the encrypted machine learning model. The model may also be kept within the enclave for secure evaluation by all the parties, subject to their agreed access control policies. Figure 1 provides an overview of our system.

While we rely on the processor to guarantee that only the machine learning code inside the enclave has direct access to the data, achieving privacy still requires a careful selection, adaptation, and implementation of machine-learning algorithms, in order to prevent the exploitation of any side channels induced by disk, network, and memory access patterns, which may otherwise leak a surprisingly large amount of data [39, 49, 70]. The robust property we want from these algorithms is *data-obliviousness*: the sequence of memory references, disk accesses, and network accesses that they perform should not depend on secret data. We propose data-oblivious machine learning algorithms for support vector machines (SVM), matrix factorization, neural networks, decision trees, and k-means clustering. Our data-oblivious algorithms are based on careful elimination of data-dependent accesses (SVM, neural networks and k-means), novel algorithmic techniques (matrix factorization) and deployment of platform specific hardware features (decision trees). We provide strong, provable confidentiality guarantees, similar to those achieved by purely cryptographic solutions: we ensure that an attacker that observes the sequence of I/O operations, including their addresses and their encrypted contents, cannot distinguish between two datasets of the same size that yield results of the same size.

We implemented and ran our algorithms on off-the-shelf Intel Skylake processors, using several large ma-

[*] Work done while at Microsoft Research; affiliated with Max Planck Institute for Software Systems (MPI-SWS), Germany.

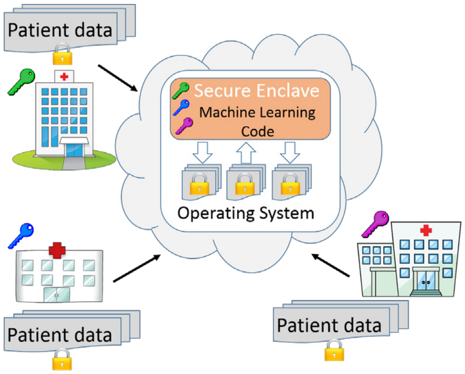


Figure 1: Sample privacy-preserving multi-party machine learning system. Multiple hospitals encrypt patient datasets, each with a different key. The hospitals deploy an agreed-upon machine learning algorithm in an enclave in a cloud data center and share their data keys with the enclave. The enclave processes the aggregate datasets and outputs an encrypted machine learning model.

chine learning datasets. Our results show that our approach scales to realistic datasets, with overheads that are several orders of magnitude better than with previous approaches based on advanced cryptographic multi-party computation schemes. On the other hand, our approach trusts the processor to protect the confidentiality of its internal state, whereas these cryptographic approaches do not rely on this assumption.

2 Preliminaries

Intel SGX SGX [45] is a set of new x86 instructions that applications can use to create protected memory regions within their address space. These regions, called enclaves, are isolated from any other code in the system, including operating system and hypervisor. The processor monitors all memory accesses to the enclaves: only code running in an enclave can access data in the enclave. When inside the physical processor package (in the processor's caches), the enclave memory is available in plaintext, but it is encrypted and integrity protected when written to system memory (RAM). External code can only invoke code inside the enclave at statically-defined entry points. SGX also supports attestation and sealing [2]: code inside an enclave can get messages signed using a per-processor private key along with a digest of the enclave. This enables other entities to verify that these messages originated from a genuine enclave with a specific code and data configuration.

Using SGX instructions, applications can set up fine-grained trusted execution environments even in (potentially) hostile or compromised hosts, but application developers who write code that runs inside enclaves are still responsible for maintaining confidentiality of secrets managed by the enclave. In this paper, we focus on guaranteeing that the machine learning algorithms that we load into enclaves do not leak information through memory, disk, or network access patterns.

Adversary Model We assume the machine learning computation runs in an SGX-enabled cloud data center that provides a convenient 'neutral ground' to run the computation on datasets provided by multiple parties. The parties do not trust one another, and they are also suspicious about the cloud provider. From the point of view of each party (or any subset of parties), the adversary models all the other parties and the cloud provider.

The adversary may control all the hardware in the cloud data center, except the processor chips used in the computation. In particular, the adversary controls the network cards, disks, and other chips in the motherboards. She may record, replay, and modify network packets or files. The adversary may also read or modify data after it left the processor chip using physical probing, direct memory access (DMA), or similar techniques.

The adversary may also control all the software in the data center, including the operating system and hypervisor. For instance, the adversary may change the page tables so that any enclave memory access results in a page fault. This active adversary is general enough to model privileged malware running in the operating or hypervisor layers, as well as malicious cloud administrators who may try to access the data by logging into hosts and inspecting disks and memory.

We assume that the adversary is unable to physically open and manipulate the SGX processor chips that run the machine learning computation. Denial of-service and side-channel attacks based on power and timing analysis are outside our scope. We consider the implementation of the machine learning algorithms to be benign: the code will never intentionally try to leak secrets from enclaves. We assume that all parties agree on the machine learning code that gets access to their datasets, after inspecting the code or using automated verification [60] to ascertain its trustworthiness. We assume that all parties get access to the output of the machine learning algorithm, and focus on securing its implementation—limiting the amount of information released by its correct output [19] is outside the scope of this paper.

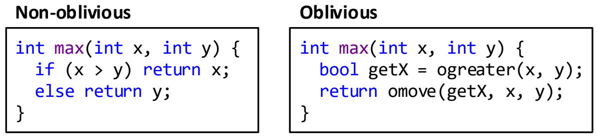
Security Guarantees We are interested in designing algorithms with strong provable security guarantees. The attacker described above should not gain any side information about sensitive data inputs. More precisely, for each machine learning algorithm, we specify *public parameters* that are allowed to be disclosed (such as the input sizes and the number of iterations to perform) and we treat all other inputs as private. We then say that an algorithm is *data-oblivious* if an attacker that interacts with it and observes its interaction with memory, disk and network learns nothing except possibly those public parameters. We define this interaction as a trace execution τ of I/O events, each recording an access type (read or write), an address, and some contents, controlled by the adversary for all read accesses. Crucially, this trace leaks accurate information about access to code as well as data; for example, a conditional jump within an enclave may reveal the condition value by leaking the next code address [70].

We express our security properties using a simulation-based technique: for each run of an algorithm given some input that yields a trace τ, we show that there exists a *simulator program* given *only* the public parameters that simulates the interaction of the original algorithm with the memory by producing a trace τ' indistinguishable from τ. Intuitively, if the algorithm leaked any information depending on private data, then the simulator (that does not have the data) would not be able to adapt its behavior accordingly. Beside the public parameters, the simulator may be given the result of the algorithm (e.g., the machine learning model) in scenarios where the result is revealed to the parties running the algorithm. We rely on indistinguishability (rather than simple trace equivalence $\tau' = \tau$) to account for randomized algorithms, and in particular for encryption. For instance, any private contents in write events will be freshly encrypted and thus (under some suitable semantic-encryption security assumption) will appear to be independently random in both τ and τ', rather than equal. More precisely, we define indistinguishability as usual in cryptography, using a game between a system that runs the algorithm (or the simulator) and a computationally bounded adversary that selects the inputs, interacts with the system, observes the trace, and attempts to guess whether it interacts with the algorithm or the simulator. The algorithm is data-oblivious when such adversaries guess correctly with probability at most $\frac{1}{2}$ plus a negligible advantage.

3 Data-Oblivious Primitives

Our algorithms rely on a library of general-purpose oblivious primitives. We describe them first and then show how we use them in machine learning algorithms.

Oblivious assignments and comparisons These primitives can be used to conditionally assign or compare integer, floating point, or 256-bit vector variables. They are implemented in x86-64 assembly, operating solely on registers whose content is loaded from and stored to memory using deterministic memory accesses. The registers are private to the processor; their contents are not accessible to code outside the enclave. As such, evaluations that involve registers only are not recorded in the trace τ, hence, any register-to-register data manipulation is data-oblivious by default.

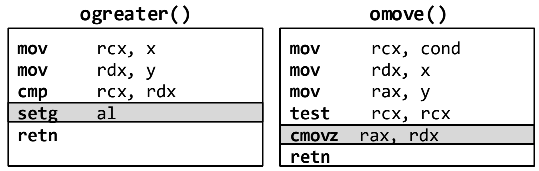


```
Non-oblivious
int max(int x, int y) {
  if (x > y) return x;
  else return y;
}
```

```
Oblivious
int max(int x, int y) {
  bool getX = ogreater(x, y);
  return omove(getX, x, y);
}
```

Figure 2: **Left:** C++ function determining the maximum of two integers using a non-oblivious if-else statement; **right:** oblivious variant of the function using oblivious primitives.

We choose omove() and ogreater() as two representative oblivious primitives. In conjunction, they enable the straightforward, oblivious implementation of the max() function, as shown in Figure 2. In the oblivious version of max(), ogreater() evaluates the guard x > y and omove() selects either x or y, depending on that guard. In our library, similar to related work [53], both primitives are implemented with conditional instructions cmovz and setg. For example, in simplified form, omove() and ogreater() for 64-bit integers comprise the following instructions:



```
ogreater()
mov    rcx, x
mov    rdx, y
cmp    rcx, rdx
setg   al
retn
```

```
omove()
mov    rcx, cond
mov    rdx, x
mov    rax, y
test   rcx, rcx
cmovz  rax, rdx
retn
```

On top of such primitives for native C++ types, our library implements more complex primitives for user-defined types. For example, most of our oblivious algorithms rely on omoveEx(), an extended version of the basic omove(), which can be used to conditionally assign any type of variable; depending on the size of the given type, omoveEx() iteratively uses the 64-bit integer or 256-bit vector version of omove().

Oblivious array accesses Scanning entire arrays is a commonly used technique to make data-dependent memory accesses oblivious. In the simplest case, we use omoveEx() iteratively to access each element when actually just a single element is to be loaded or stored.[1] However, our adversary model implies that, for enclave code, the attacker can only observe memory accesses at cache-line granularity. Accordingly, the x least significant bits[2] of memory addresses are not recorded in a trace τ. It is hence sufficient to scan arrays at cache-line granularity rather than element or byte granularity. We implement accordingly optimized array access primitives that leverage AVX2 vector instructions [31]. In particular, the vpgatherdd instruction can load each of the eight 32-bit (4-byte) components of a 256-bit vector

[1] Dummy writes without actual effect are made to all but one element in case of a store. Modern processors treat such writes in the same way as real writes and mark corresponding cache lines as dirty.

[2] The value of x depends on the actual hardware implementation; for Skylake processors, where cache lines are 64 bytes long, $x = 6$.

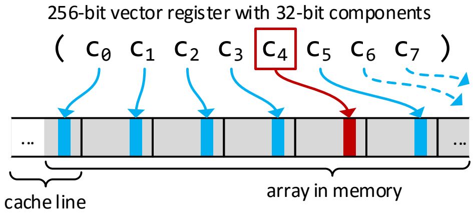

256-bit vector register with 32-bit components
(C0 C1 C2 C3 C4 C5 C6 C7)
...
...
cache line
array in memory

Figure 3: Optimized array scanning using the `vpgatherdd` instruction; here, the value of interest is read into `C4`. The other components perform dummy reads.

register from a different memory offset. Hence, by loading each component from a different cache line, 4 bytes can be read obliviously from an aligned 512-byte array with a single instruction as depicted in Figure 3 (i.e., a 4-byte read is hidden among 8 cache lines accessed via `vpgatherdd`). On top of this, the `oget()` primitive is created, which obliviously reads an element from an unaligned array of arbitrary form and size. `oget()` iteratively applies the `vpgatherdd` instruction in the described way while avoiding out-of-bounds reads. Depending on the dynamic layout of the caches, `oget()` can significantly speed-up oblivious array lookups (see Section 6.6). The construction of `oget()` is conservative in the sense that it assumes (i) that the processor may load vector components in arbitrary, possibly parallel order[3] and (ii) that this order is recorded precisely in τ. For cases where (i) or (ii) do not apply, e.g., for software-only attackers, a further optimized version of `oget()` is described in Appendix B.

Oblivious sorting We implement oblivious sorting by passing its elements through a network of carefully arranged compare-and-swap functions. Given an input size n, the network layout is fixed and, hence, the memory accesses fed to the functions in each layer of the network depend only on n and not the content of the array (as opposed to, e.g., quicksort). Hence, its memory trace can be easily simulated using public parameter n and fixed element size. Though there exists an optimal sorting network due to Ajtai *et al.* [1], it incurs high constants. As a result, a Batcher's sorting network [7] with running time of $O(n(\log n)^2)$ is preferred in practice. Our library includes a generic implementation of Batcher's sort for shuffling the data as well as re-ordering input instances to allow for efficient (algorithm-specific) access later on. The sorting network takes as input an array of user-defined type and an oblivious compare-and-swap function for this type. The oblivious compare-and-swap usually relies on the `ogreater()` and `omoveEx()` primitives described above.

4 Machine Learning Algorithms

We describe five machine learning algorithms: four training and one prediction method, and their data-oblivious counter-parts. The algorithms vary in the complexity of access patterns, from randomly sampling the training data to input-dependent accesses to the corresponding model. Hence, we propose algorithm-specific mitigation techniques that build on the oblivious primitives from the last section.

4.1 K-Means

The goal of k-means clustering is to partition input data points into k clusters such that each point is assigned to the cluster closest to it. Data points are vectors in the Euclidean space $\mathbb{R}^d$. To implement clustering, we chose a popular and efficient Lloyd's algorithm [40, 41, 43].

During its execution, k-means maintains a list of k points that represent the current cluster centroids: for $i = 1..k$, the ith point is the mean of all points currently assigned to the ith cluster. Starting from random centroids, the algorithm iteratively reassigns points between clusters: (1) for each point, it compares its distances to the current k centroids, and assigns it to the closest cluster; (2) once all points have been processed, it recomputes the centroids based on the new assignment. The algorithm ends after a fixed number of iterations, or once the clustering is stable, that is, in case points no longer change their cluster assignments. Depending on the application, k-means returns either the centroids or the assignment of data points to clusters.

Although the algorithm data flow is largely independent of the actual points and clusters, its naive implementation may still leak much information in the conditional update in (1)—enabling for instance an attacker to infer some point coordinates from the final assignment, or vice-versa—and in the recomputation (2)—leaking, for instance, intermediate cluster sizes and assignments.

In the following, we treat the number of points (n), clusters (k), dimension (d) and iterations (T) as public. We consider efficient, streaming implementations with, for each iteration, an outer loop traversing all points once, and successive inner loops on all centroids for the steps (1) and (2) above. For each centroid, in addition to the d coordinates, we locally maintain its current cluster size (in $0..n$). To perform both (1) and (2) in a single pass, we maintain both the current and the next centroids, and we delay the division of coordinates by the cluster size in the latter. Thus, for a given point, inner loop (1) for $i = 1..k$ maintains the (square of the) current minimal distance δ_{min} and its centroid index i_{min}. And inner loop (2) performs k conditional updates on the next centroids, depending on $i = i_{min}$. Finally, a single pass over centroids recomputes their coordinates. An important detail is to uniformly handle the special case of empty clus-

[3] The implementation of the `vpgatherdd` instruction is microarchiteture-specific and undocumented.

ters; another to select the initial centroids, for instance by sampling random points from the shuffled dataset.

In our adapted algorithm, the "privacy overhead" primarily consists of oblivious assignments to loop variables in (1), held in registers, and to the next centroids, held in the cache. In particular, instead of updating statistics for only the centroid that the point belongs to, we make dummy updates to each centroid (using our `omoveEx()` primitive). In the computation of new centroids, we use a combination of `ogreater()` and `omoveEx()` to handle the case of empty clusters. These changes do not affect the algorithm's time complexity: in the RAM model the operations above can still be done in $O(T(nkd+kd)) = O(Tnkd)$ operations.

Theorem 1. *The adapted* k-means *algorithm runs in time $O(Tnkd)$ and is data-oblivious, as there exists a simulator for* k-means *that depends only on T, n, d, and k.*

Proof. The simulator can be trivially constructed as follows: given T, n, d and k, it chooses n random points from $\mathbb{R}^d$ and simply runs the algorithm above for k centroids and T iterations. □

It is easy to see that the subroutine for finding the closest centroid in the training algorithm can be also used to predict the trained cluster that an input point belongs to.

4.2 Supervised Learning Methods

In supervised machine learning problems, we are given a dataset $D = \{(x_i, y_i)\}_{i=1..n}$ of instances, where $x_i \in \mathscr{X}$ is an observation and $y_i \in \mathscr{Y}$ is a desired prediction. The goal then is to learn a predictive model $f : \mathscr{X} \rightarrow \mathscr{Y}$ such that $f(x_i) \approx y_i$ and the model generalizes to unseen instances $x \in \mathscr{X}$. Many machine learning methods learn such a model by minimizing an *empirical risk* objective function together with a regularization term [65]:

$$\min_w \Omega(w) + \frac{1}{n} \sum_{i=1}^{n} L(y_i, f_w(x_i)). \tag{1}$$

We will show secure implementations of support vector machines (SVM) and neural networks, which are of the form (1). Other popular methods such as linear regression and logistic regression are also instances of (1).

Most algorithms to minimize (1) operate iteratively on small subsets of the data at a time. When sampling these subsets, one common requirement for correctness is that the algorithm should have access to a distribution of samples with an unbiased estimate of the expected value of the original distribution. We make an important observation that an unbiased estimate of the expected value of a population can be computed from a subset of independent and identically distributed instances as well as from a subset of *pairwise-distinct* instances.

Thus, we can achieve correctness and security by adapting the learning algorithm as follows. Repeatedly, (1) securely shuffle all instances at random, using Batcher's sort, for example, or an oblivious shuffle [50]; (2) run the learning algorithm on the instances *sequentially*, rather than randomly, either individually or in small batches. Thus, the cost of shuffling is amortized over all n instances. Next, we illustrate this scheme for support vector machines and neural networks.

4.3 Support Vector Machines (SVM)

Support Vector Machines are a popular machine learning model for classification problems. The original formulation [12] applies to problems with two classes. Formally, SVM specializes (1) by using the linear model $f_w(x) = \langle w, x \rangle$, the regularization $\Omega(w) = \frac{\lambda}{2}\|w\|^2$ for $\lambda > 0$, and the loss function $L(y_i, f_w(x_i)) = \max\{0, 1 - y_i f_w(x_i)\}$.

The SVM method is important historically and in practice for at least four separate reasons: *first*, it is easy to use and tune and it performs well in practice; *second*, it is derived from the principle of structural risk minimization [65] and comes with excellent theoretical guarantees in the form of generalization bounds; *third*, it was the first method to be turned into a non-linear classifier through application of the kernel trick [12, 56], *fourth*, the SVM has inspired a large number of generalizations, for example to the multi-class case [67], regression [61], and general pattern recognition problems [63].

Here we only consider the linear (primal) case with two classes, but our methods would readily extend to multiple classes or support vector regression problems. There are many methods to solve the SVM objective and for its simplicity we adapt the state-of-the-art *Pegasos* method [58]. The algorithm proceeds in iterations $t = 1..T$ and, at each iteration, works on small subsets of l training instances at a time, $A^{(t)}$. It updates a sequence of weight vectors $w^{(1)}, w^{(2)}, \ldots, w^{(T)}$ converging to the optimal minimizer of the objective function (1).

Let us now consider in detail our implementation of the algorithm, and the changes that make it data-oblivious. We present the pseudo-code in Algorithm 1 where our changes are highlighted in blue, and indented to the right. As explained in Section 4.2, SVM samples input data during training. Instead, we obliviously (or privately) shuffle the data and process it sequentially. The original algorithm updates the model using instances that are mispredicted by the current model, $A_+^{(t)}$ in Line 5 of the pseudo-code. As this would reveal the state of the current model to the attacker, we make sure that the computation depends on every instance of $A^{(t)}$. In particular, we generate a modified set of instances in $B^{(t)}$ which has the original (x, y) instance if x is mispredicted and $(x, 0)$ otherwise, assigning either 0 or y using our `ogreater()` and `omove()` primitives (see Figure 2).

Algorithm 1 SVM Original with changes (starting with ▷) and additional steps required for the Oblivious Version indicated in blue.

```
1:  INPUT: $I = \{(x_i, y_i)\}_{i=1,\dots,n}$, $\lambda$, $T$, $l$
2:  INITIALIZE: Choose $w^{(0)}$ s.t. $\|w^{(0)}\| \leq 1/\sqrt{\lambda}$
3:      Shuffle $I$
4:  FOR $t = 1, 2, \dots, T \times n/l$
5:      Choose $A^{(t)} \subseteq I$ s.t. $|A^{(t)}| = l$
                                  ▷ Set $A^{(t)}$ to $t$th batch of $l$ instances
6:      Set $A_+^{(t)} = \{(x,y) \in A^{(t)} : y\langle w^t, x\rangle < 1\}$
              ▷ $B^{(t)} = \{(x, \mathbb{1}[y\langle w^t, x\rangle < 1]y) : \forall(x,y) \in A^{(t)}\}$
7:      Set $\eta = 1/\lambda t$
8:      Set $v = \sum_{(x,y)\in A_+^{(t)}} yx$            ▷ $v = \sum_{(x,z)\in B^{(t)}} zx$
9:      Set $v = (1-\eta\lambda)w^{(t)} + \frac{\eta}{l} v$
10:     Set $c = 1 < \frac{1}{\sqrt{\lambda}}\|w\|$
11:     Set $w^{(t+1)} = \min\left\{1, \frac{1}{\sqrt{\lambda}}\|w\|\right\} v$
                        ▷ Set $w^{(t+1)} = \left(c + (1-c) \times \frac{1}{\sqrt{\lambda}\|w\|}\right) v$
12: OUTPUT $w^{(t+1)}$
```

The second change is due to a Euclidean projection in Line 11, where v is multiplied by the minimum of the two values. In the oblivious version, we ensure that both values participate in the update of the model, again using our oblivious primitives. The modifications above are simple and, if the data is shuffled offline, asymptotically do not add overhead as the algorithm has to perform prediction for every value in the sample. Otherwise, the overhead of sorting is amortized as T is usually set to at least one.

Theorem 2. *The* SVM *algorithm described above runs in time $O(n(\log n)^2)$ and is data-oblivious, as there exists a simulator for* SVM *that depends only on T, n, d, λ and l, where d is the number of features in each input instance.*

The simulator can be constructed by composing a simulator for oblivious sorting and one that follows the steps of Algorithm 1.

We note that the oblivious computation of a label in the training algorithm ($\langle w, x\rangle$ in Line 6 in Algorithm 1) can be used also for the prediction phase of SVM.

4.4 Neural Networks

Feedforward neural networks are classic models for pattern recognition that process an observation using a sequence of learned non-linear transformations [10]. Recently, deep neural networks made significant progress on difficult pattern recognition applications in speech, vision, and natural language understanding and the collective set of methods and models is known as *deep learning* [26].

Formally, a feedforward neural network is a sequence of transformations $f(x) = f_t(\dots f_2(f_1(x)))$, where each transformation f_i is described by a fixed family of transformations and a parameter w_i to identify one particular element in that family. *Learning* a neural network means to find suitable parameters by minimization of the learning objective (1).

To minimize (1) efficiently in the context of neural networks, we use stochastic gradient methods (SGD) on small subsets of training data [26]. In particular, for $l \ll n$, say $l = 32$, we compute a *parameter gradient* on a subset $S \subset \{1, 2, \dots, n\}$, $|S| = l$ of the data as

$$\nabla_w \Omega(w) + \frac{1}{l} \sum_{i \in S} \nabla_w L(y_i, f(x_i)). \qquad (2)$$

The expression above is an unbiased estimate of the gradient of (1) and we can use it to update the parameters via gradient descent. By repeating this update for many subsets S we can find parameters that approximately minimize the objective. Instead, as for SVM, we use disjoint subsets that are contiguous within the set of all (obliviously or privately) shuffled instances and iterate T times.

Because most neural networks densely process each input instance, memory access patterns during training and testing do not depend on the particular data instance. There are two exceptions. First, the initialization of a vector with $|\mathscr{Y}|$ ground truth labels depends on the true label y_i of the instance (recall that $\mathscr{Y}$ is the set of possible prediction classes or labels). In particular, the y_ith entry is set, for example, to 1 and all other entries to 0. We initialize the label vector and hide the true label of the instance by using our oblivious comparison and move operations. The second exception is due to special functions that occur in certain f_i, for example in tanh-activation layers. Since special functions are relatively expensive, they are usually evaluated using piecewise approximations. Such conditional computation may leak parameter values and, in our adapted algorithm, we instead compute the approximation obliviously using a sequence of oblivious move operations. Neither of these changes affects the complexity of the algorithm.

The prediction counterpart of NN, similar to the k-means and SVM algorithms, is a subroutine of the training algorithm. Hence, our changes can be also used to make an oblivious prediction given a trained network.

4.5 Decision Tree Evaluation

Decision trees are common machine learning models for classification and regression tasks [15, 51, 52]. In these models, a tree is traversed from root to leaf node by performing a simple test on a given instance, at each interior node of the tree. Once a leaf node is reached, a simple model stored at this node, for example a constant value, is used as prediction.

Decision trees remain popular because they are *non-parametric*: the size of the decision tree can grow with more training data, providing increasingly accurate models. Ensembles of decision trees, for example in the form of *random forests* [14] offer improved predictive performance by averaging the predictions of many individual tree models [16]. Decision trees illustrate a class of data structures whose usage is highly instance-specific: when evaluating the model, the path traversed from root to leaf node reveals a large amount of information on both the instance and the tree itself. To enable the evaluation of decision tress without leaking side information, we adapt the evaluation algorithm of an existing library for random forests to make it data oblivious. We keep modifications of the existing implementation at a minimum, relying on the primitives of Section 3 wherever possible. Our target tree evaluation algorithm operates on one instance $x \in \mathbb{R}^d$ at a time. In particular, the trees are such that, at each interior node, a simple *decision stump* is performed:

$$\phi(x; j, t) = \begin{cases} \text{left}, & \text{if } x(j) \leq t, \\ \text{right}, & \text{otherwise}, \end{cases} \tag{3}$$

where $j \in \{1, \ldots, d\}$ and $t \in \mathbb{R}$ are learned parameters stored at the interior tree node.

In order to conceal the path taken through a tree, we modify the algorithm such that each tree layer is stored as an array of nodes. During evaluation of an instance x, the tree is traversed by making exactly one oblivious lookup in each of these arrays. At each node, the lookup $x(j)$ and corresponding floating point comparison are done using our oblivious primitives. In case a leaf is found early, that is before the last layer of the tree was reached, its ID is stored obliviously and the algorithm proceeds with dummy accesses for the remaining layers of the tree. The predictions of all trees in a random forest are accumulated obliviously in an array; the final output is the prediction with the largest weight.

Together, the described modifications guarantee data-obliviousness both for instances and for trees (of the same size, up to padding). The algorithmic overhead is linear in the number of nodes n in a tree, i.e., $O(n)$ for a fixed d; we omit the corresponding formal development.

4.6 Matrix Factorization

Matrix factorization methods [55] are a popular set of techniques for constructing recommender systems [32]. Given *users* and *items* to be rated, we take as input the observed ratings for a fraction of user-item pairs, either as explicit scores (“five stars”) or implicit user feedback. As a running example, we consider a system to recommend movies to viewers based on their experience.

Matrix factorization embeds users and items into a latent vector space, such that the inner product of a user vector with an item vector produces an estimate of the rating a user would assign to the item. We can then use this expected rating to propose novel items to the user. While the individual preference dimensions in the user and item vectors are not assigned fixed meanings, empirically they often correspond to interpretable properties of the items. For example, a latent dimension may correspond to the level of action the movie contains.

Matrix factorization methods are remarkably effective [9] because they learn to transfer preference information across users and items by discovering dimensions of preferences shared by all users and items.

Let n be the number of users and m the number of items in the system. We may represent all (known and unknown) ratings as a matrix $R \in \mathbb{R}^{n \times m}$. The input consists of M ratings $r_{i,j}$ with $i \in 1..n$ and $j \in 1..m$, given by users to the items they have seen (using the movies analogy). The output consists of $U \in \mathbb{R}^{n \times d}$ and $V \in \mathbb{R}^{m \times d}$ such that $R \approx U V^\top$; these two matrices may then be used to predict unknown ratings $r_{i,j}$ as inner products $\langle u_i, v_j \rangle$. Following [48], we refer to u_i and v_j as user and item profiles, respectively.

The computation of U and V is performed by minimizing regularized least squares on the known ratings:

$$\min \frac{1}{M} \sum (r_{i,j} - \langle u_i, v_j \rangle)^2 + \lambda \sum \|u_i\|_2^2 + \mu \sum \|v_j\|_2^2 \tag{4}$$

where λ and μ determine the extent of regularization. The function above is not jointly convex in U and V, but becomes strictly convex in U for a fixed V, and strictly convex in V for a fixed U.

We implement matrix factorization using a gradient descent, as in prior work on oblivious methods [47, 48]. More efficient methods are now available to solve (4), such as the so-called *damped Wiberg method*, as shown in an extensive empirical evaluation [30], but they all involve more advanced linear algebra, so we leave their privacy-preserving implementation for future work.

Gradient descent This method iteratively updates U and V based on the current prediction error on the input ratings. The error is computed as $e_{i,j} = r_{i,j} - \langle u_i, v_j \rangle$, and u_i and v_j are updated in the opposite direction of the gradient as follows:

$$u_i^{(t+1)} \leftarrow u_i^{(t)} + \gamma \left[\textstyle\sum_j e_{i,j} v_j^{(t)} - \lambda u_i^t \right] \tag{5}$$

$$v_j^{(t+1)} \leftarrow v_j^{(t)} + \gamma \left[\textstyle\sum_i e_{i,j} u_i^{(t)} - \mu v_j^t \right] \tag{6}$$

The descent continues as long as the error (4) decreases, or for a fixed number of iterations (T). Each iteration can be efficiently computed by updating U and V sequentially. To update each user profile u_i, we may for instance use an auxiliary linked list of user ratings and pointers to the corresponding movie profiles in V.

The gradient descent above runs in time $\Theta(TM)$ in the RAM model, since all ratings are used at each iteration. For a fixed number of iterations, the access pattern of the algorithm does not depend on the actual values of the input ratings. However, it still reveals much sensitive information about which user-item pairs appear in the input ratings. For example, assuming they indicate which users have seen which movies, it trivially reveals the popularity of each movie, and the intersection of movie profiles between users, during the gradient update (see [46] for privacy implications of leaking movie ratings).

Our data-oblivious algorithm We design an algorithm whose observable behaviour depends only on public parameters n, m, M and T, and, hence, it can be simulated and does not reveal R. (We assume that d, λ, μ, and γ are public and do not depend on the input data.)

The high level idea is to use data structures that interleave user and movie profiles. This interleaving allows us to perform an update by sequentially reading and updating these profiles in-place. Once all profiles have been updated, some additional processing is required to interleave them for the next iteration but, with some care, this can also be implemented by sequential traversals of our data structures. (An illustration of the algorithm can be found in the Appendix.)

Our algorithm preserves the symmetry between users and items. It maintains data structures $\mathbf{U}$ and $\mathbf{V}$ that correspond to expanded versions of the matrices U and V. Intuitively, every user profile in $\mathbf{U}$ is followed by the movie profiles required to update it (that is, the profiles for all movies rated by this user), and symmetrically every movie profile in $\mathbf{V}$ is followed by its user profiles. We use superscript notation $\mathbf{U}^{(t)}$ and $\mathbf{V}^{(t)}$ to distinguish these data structures between iterations.

$\mathbf{U}$ stores n *user tuples* that embed the user profiles of the original U, and M *rating tuples* that contain both movie profiles and their ratings. All tuples have the same size; they each include a *user id*, a *movie id*, a *rating*, and a vector of d values. User tuples are of the form $(i,0,0,u_i)$ with $i \in 1..n$; Rating tuples are of the form $(i,j,r_{i,j},v_j)$. Hence, for each rating for j, we have a copy of v_j in a rating tuple. $\mathbf{V}$ symmetrically stores m *item tuples*, of the form $(0,j,0,v_j)$, and M *rating tuples*, of the form $(i,j,r_{i,j},u_i)$.

The precise ordering of tuples within $\mathbf{U}$ (and $\mathbf{V}$) is explained shortly but, as long as the tuples of $\mathbf{U}$ are grouped by user ids (i), and the user tuple precedes its rating tuples, we can compute each $u_i^{(t+1)}$ according to Equation (5) by traversing $\mathbf{U}^{(t)}$ once, in order. After an initial *Setup*, each iteration actually consists of three data-oblivious phases:

- *Update* the user profiles u_i within $\mathbf{U}^{(t)}$ using Equation (5); let $\tilde{\mathbf{U}}$ be the updated data structure;
- *Extract* $U^{(t+1)}$ from $\tilde{\mathbf{U}}$;
- *Copy* $U^{(t+1)}$ into the rating tuples of $\tilde{\mathbf{V}}$ to obtain $\mathbf{V}^{(t+1)}$ for the next iteration.

(We omit symmetric steps producing $\tilde{\mathbf{V}}$, $V^{(t+1)}$, and $\mathbf{U}^{(t+1)}$.) The extraction step is necessary to compute the prediction error and prepare $\mathbf{U}$ and $\mathbf{V}$ for the next iteration. Without tweaking the tuple ordering, the only efficient way of doing so would be to sort $\tilde{\mathbf{U}}$ lexicographically (by j, then i) so that the updated user profiles appear in the first n tuples (the approach taken in [48]). Oblivious sorting at each iteration is expensive, however, and would take $O((M+n)(\log(M+n))^2)$ oblivious compare-and-swap of pairs of $d+3$ elements.

Instead, we carefully place the user tuples in $\tilde{\mathbf{U}}$ so that they can be extracted in a single scan, outputting profiles *at a fixed rate*: one user profile every $(M+n)/n$ tuples, on average. Intuitively, this is achieved by interleaving the tuples of users with many ratings with those of users with few ratings—Section 4.7 explains how we efficiently compute such a tuple ordering.

Setup phase: We first initialize the user and vector profiles, and fill $\mathbf{U}$ and $\mathbf{V}$ using the input ratings.

(1) We build a sequence $L_{\mathbf{U}}$ (and symmetrically $L_{\mathbf{V}}$) that, for every user, contains a pair of the user id i and the count w_i of the movies he has rated. To this end, we extract the user ids from the input ratings (discarding the other fields); we sort them; we rewrite them sequentially, so that each entry is extended with a partial count and a flag indicating whether the next user id changes; and we sort them again, this time by flag then user id, to obtain $L_{\mathbf{U}}$ as the top n entries. (Directly outputting $L_{\mathbf{U}}$ during the sequential scan would reveal the counts.) For instance, after the first sorting and rewriting, the entries may be of the form $(1,1,\perp),(1,2,\perp),(1,3,\top),(2,1,\perp),\ldots$.

(2) We expand $L_{\mathbf{U}}$ (and symmetrically $L_{\mathbf{V}}$) into a sequence $I_{\mathbf{U}}$ of size $M+n$ that includes, for every user i with w_i ratings, one tuple $(i,0,\perp,k,\ell)$ for each $k=0..w_i$, such that the values ℓ are ordered by the interleaving explained in Section 4.7.

(3) We construct $\mathbf{U}$ with empty user and rating profiles, as follows. Our goal is to order the input ratings according to $L_{\mathbf{U}}$. To this end, we extend each input rating with a user-rating sequence number $k=1..w_i$, thereby producing M tuples $(i,j,r_{i,j},k,\perp)$, and we append those to $I_{\mathbf{U}}$. We sort those tuples by i then k then $r_{i,j}$, so that $(i,0,\perp,k,\ell)$ is directly followed by $(i,j,r_{i,j},k,\perp)$ for $k=1..w_i$; we sequentially rewrite those tuples so that they become $(\_,\_,\_,\_,\_)$ directly followed by $(i,j,r_{i,j},k,\ell)$; we sort again by ℓ; and we discard the last M dummy tuples $(\_,\_,\_,\_,\_)$.

(4) We generate initial values for the user and item profiles by scanning $\mathbf{U}$ and filling in u_i and v_j using two pseudo-random functions (PRFs): one for u_is and one

for v_js. For each, user tuple $(i,0,0,u_i)$, we use the first PRF on inputs $(i-1)d+1,..,id$ to generate d random numbers that we normalize and write to u_i. For each, rating tuple $(i,j,r_{i,j},v_j)$, we use the second PRF on inputs $(j-1)d+1,..,jd$ to generate d random numbers that we also normalize and write to v_j. We then use the same two PRFs for $\mathbf{V}$: the first one for rating tuples and the second one for item tuples.

Update phase: We compute updated user profiles (and symmetrically item profiles) in a single scan, reading each tuple of $\mathbf{U}$ (and symmetrically $\mathbf{V}$) and (always) rewriting its vector—that is, its last d values, storing u_i for user tuples and v_j for rating tuples.

We use 4 loop variables u, δ, u°, and δ° each holding a $\mathbb{R}^d$ vector, to record partially-updated profiles for the current user and for user i°. We first explain how u and δ are updated for the current user (i). During the scan, upon reading a user tuple $(i,0,0,u_i)$, as is always the case for the first tuple, we set u to u_i and δ to $u_i(1-\lambda\gamma)$ and we overwrite u_i (to hide the fact that we scanned a user tuple). Upon reading a rating tuple $(i,j,r_{i,j},v_j)$ for the current user i, we update δ to $\gamma v_j(r_{i,j}-\langle u,v_j\rangle)+\delta$ and overwrite v_j with δ. Hence, the last rating tuple (before the next user tuple) now stores the updated profile $u_i^{(t+1)}$ for the current user i.

We now bring our attention to i°, u°, and δ°. Recall that our interleaving of users in $\mathbf{U}$ splits the rating tuples for some users. In such cases, if there are ratings left to scan, the running value δ written to v_j (before scanning the next user) may not yet contain the updated user profile. Accordingly, we use i°, u°, and δ° to save the state of the 'split' user while we process the next user, and restore it later as we scan the next rating tuple of the form $(i^\circ,j,r_{i^\circ,j},v_j)$. In the full version of the paper we prove that a single state copy suffices during the expansion of $L_{\mathbf{U}}$ as we split at most one user at a time.

Extraction phase: The update leaves some of the values $u_i^{(t+1)}$ scattered within $\tilde{\mathbf{U}}$ (and similarly for $v_j^{(t+1)}$ within $\tilde{\mathbf{V}}$). Similar to the update phase we can extract all profiles by maintaining a state i° and u° for only one user. We extract U while scanning $\tilde{\mathbf{U}}$. In particular, after reading the last tuple of every chunk of size $(M+n)/n$ in $\tilde{\mathbf{U}}$ we always append an entry to U. This entry is either i° and u° or the content of the last tuple i and u. Meanwhile, after reading every tuple of $\tilde{\mathbf{U}}$ we write back either the same entry or the profile that was written to U last. This step ensures that user tuples contain the updated $u^{(t+1)}$. We also update i° and u° on every tuple: either performing a dummy update or changing the state to the next (split) user.

This step relies on a preliminary re-ordering and interleaving of users, such that the ith chunk of tuples always contains (a copy of) a user profile, and all n user profiles can be collected (details of the expansion properties that are used here are described in the following section and in the full version of the paper).

Copying phase: We finally propagate the updated user profiles $U^{(t+1)}$ to the rating tuples in $\tilde{\mathbf{V}}$, which still carry (multiple copies of) the user profiles $U^{(t)}$. We update $\tilde{\mathbf{V}}$ sequentially in chunks of size n, that is, we first update the first n rows of $\mathbf{V}$, then rows $n+1$ to $2n$ and so on until all $\mathbf{V}$ is updated, each time copying from the same n user profiles of $U^{(t+1)}$, as follows. (The exact chunk size is irrelevant, but n is asymptotically optimal.)

Recall that each rating tuple of $\tilde{\mathbf{V}}$ is of the form $(i,j,r_{i,j},u_i^t,\ell)$ where $i\neq 0$ and ℓ indicates the interleaved position of the tuple in $\mathbf{V}$. To each chunk of $\tilde{\mathbf{V}}$, we append the profiles of $U^{(t+1)}$ extended with dummy values, of the form $(i,0,\_,u_i^{(t+1)},\_)$; we sort those $2n$ tuples by i then j, so that each tuple from $U^{(t+1)}$ immediately precedes tuples from (the chunk of) $\tilde{\mathbf{V}}$ whose user profile must be updated by $u_i^{(t+1)}$; we perform all updates by a linear rewriting; we sort again by ℓ; and we keep the first n tuples. Finally, $\mathbf{V}^{(t+1)}$ is just the concatenation of those updated chunks.

Theorem 3. *Our matrix factorization algorithm runs in time* $O\left((M+\tilde{n})(\log(M+\tilde{n}))^2+T(M+\tilde{n})(\log\tilde{n})^2\right)$ *where* $\tilde{n}=\max(n,m)$*. It is data-oblivious, as there exists a simulator that depends only on* T*,* M*,* n*,* m*, and* d *and produces the same trace.*

Proof Outline. The Setup phase is the most expensive, as it involves oblivious sorting on all the input ratings at once, with a $O((M+\tilde{n})(\log(M+\tilde{n}))^2)$ run time. The update phase runs in time $O(M+n+m)$ since it requires a single scan of $\mathbf{U}$ and $\mathbf{V}$. The extraction phase similarly runs in time $O(M+n+m)$. The copying phase runs in time $O((M+m)(\log n)^2+(M+n)(\log m)^2)$ due to $(M+m)/n$ sorts of U of size n and $(M+n)/m$ sorts of V of size m. Since all phases except Setup run T times, the total run time is

$$O\left((M+\tilde{n})\log^2(M+\tilde{n})+T(M+\tilde{n})\log^2\tilde{n}\right).$$

A simulator can be built from the public parameters mentioned in the beginning of the algorithm. It executes every step of the algorithm: it creates the interleaving data structures that depend only on n, m, M and d and updates them using the steps of the Setup once and runs the Update, Extraction and Copy phases for T iterations. As part of Setup, it invokes the simulator of the sequence expansion algorithm described in the full version of the paper. □

4.7 Equally-Interleaved Expansion

We finally present our method for arranging tuples of $\mathbf{U}$ and $\mathbf{V}$ in Matrix Factorization. We believe this method is applicable to other data processing scenarios. For example, Arasu and Kaushik [4] use a similar, careful arrangement of tuples to obliviously answer database queries.

Definition 1. *A* weighted list *L is a sequence of pairs (i, w_i) with n elements i and integer weights $w_i \geq 1$.*

An expansion *I of L is a sequence of elements of length $\sum_1^n w_i$ such that every element i occurs exactly w_i times.*

Definition 2. *Let $\alpha = \sum w_i/n$ be the average weight of L and the jth* chunk *of I be the sub-sequence of $\lceil \alpha \rceil$ elements $I_{\lfloor (j-1)\alpha+1 \rfloor}, \ldots, I_{\lceil j\alpha \rceil}$.*

I equally interleaves *L when all its elements can be collected by selecting one element from each chunk.*

For example, for $L = (a,4),(b,1),(c,1)$, every chunk has $\alpha = 2$ elements. The expansion $I = a,b,a,c,a,a$ equally interleaves L, as its elements a, b, and c can be chosen from its third, first, and second chunks, respectively. The expansion $I' = a,a,a,a,b,c$ does not.

We propose an efficient method for generating equal interleavings. Since it is used as an oblivious building block, we ensure that it accesses L, I and intermediate data structures in a manner that depends only on n and $M = \sum w_i$, not on the individual weights. (In matrix factorization, M is the total number of input ratings.) We adopt the terminology of Arasu and Kaushik [4], even if our definitions and algorithm are different. (In comparison, our expansions do not involve padding, as we do not require that copies of the same element are adjacent).

Given a weighted list L, we say that element i is heavy when $w_i \geq \alpha$, and light otherwise. The main idea is to put at most one light element in every chunk, filling the rest with heavy elements. We proceed in two steps: (1) we reorder L so that each heavy element is followed by light elements that compensate for it; (2) we sequentially produce chunks containing copies of one or two elements.

Step 1: Assume L is sorted by decreasing weights ($w_i \geq w_{i+1}$ for $i \in [1, n-1]$), and b is its last heavy element ($w_b \geq \alpha > w_{b+1}$). Let δ_i be the sum of differences defined as $\sum_{j \in [1,i]} (w_j - \alpha)$ for heavy elements and $\sum_{j \in [b+1,i]} (\alpha - w_j)$ for light elements. Let S be L (obliviously) sorted by δ_j, breaking ties in favor of higher element indices. This does not yet guarantee that light elements appear after the heavy element they compensate for. To this end, we scan S starting from its last element (which is always the lightest), swapping any light element followed by a heavy element (so that, eventually, the first element is the heaviest).

Step 2: We produce I sequentially, using two loop variables: k, the latest heavy element read so far; and w, the remaining number of copies of k to place in I. We repeatedly read an element from the re-ordered list and produce a chunk of elements. For the first element k_1, we produce α copies of k_1, and we set $k = k_1$ and $w = w_{k_1} - \alpha$. For each light element k_i, we produce w_{k_i} copies of k_i and $\alpha - w_{k_i}$ copies of k, and we decrement w by $\alpha - w_{k_i}$. For each heavy element k_i, we produce w copies of k and $\alpha - w$ copies of k_i, and we set $k = k_i$ and $w = w_{k_i} - (\alpha - w)$.

Continuing with our example sequence L above, a is heavy, b and c are light, and we have $\delta_a = 2$, $\delta_b = 1$, and $\delta_c = 2$. Sorting L by δ yields $(b,1),(a,4),(c,1)$. Swapping heavy and light elements yields $(a,4),(b,1),(c,1)$ and we produce the expansion $I = a,a,b,a,c,a$.

In the full version of the paper we prove that the algorithm is oblivious, always succeeds and runs in time $O(n(\log n)^2 + \sum w)$.

5 Protocols

For completeness, we give an overview of the protocols we use for running multi-party machine learning algorithms in a cloud equipped with SGX processors. Our protocols are standard, and similar to those used in prior work for outsourcing computations [29,57]. For simplicity, we describe protocols involving a single enclave.

We assume that each party agrees on the machine-learning code, its public parameters, and the identities of all other parties (based, for example, on their public keys for signature). One of the parties sends this collection of code and static data to the cloud data center, where an (untrusted) code-loader allocates resources and creates an enclave with that code and data.

Each party independently establishes a secure channel with the enclave, authenticating themselves (e.g., using signatures) and using remote attestation [2] to check the integrity of the code and static data loaded into the enclave. They may independently interact with the cloud provider to confirm that this SGX processor is part of that data center. Each party securely uploads its private data to the enclave, using for instance AES-GCM for confidentiality and integrity protection. Each party uses a separate, locally-generated secret key to encrypt its own input data set, and uses its secure channel to share that key with the enclave. The agreed-upon machine learning code may also be optionally encrypted but we expect that in the common case this code will be public.

After communicating with all parties, and getting the keys for all the data sets, the enclave code runs the target algorithm on the whole data set, and outputs a machine learning model encrypted and integrity protected with a fresh symmetric key. We note that denial-of-service attacks are outside the threat model for this paper—the parties or the data centre may cause the computation to fail before completion. Conversely, any attempt to tamper with the enclave memory (including its code and data) would be caught as it is read by the SGX processor, and hence the job completion guarantees the integrity of the whole run. Finally, the system needs to guarantee that all parties get access to the output. To achieve this, the enclave sends the encrypted output to every party over their secure authenticated channel, and waits for each of them to acknowledge its receipt and integrity. It then publishes the output key, sending it to all parties, as well as to any reliable third-party (to ensure its fair availability).

6 Evaluation

This section describes our experiments to evaluate the overhead of running our machine learning algorithms with privacy guarantees. We ran oblivious and non-oblivious versions of the algorithms that decrypt and process the data inside SGX enclaves, using off-the-shelf Intel Skylake processors. Our results show that, in all cases, the overhead of encryption and SGX protection was low. The oblivious version of algorithms with irregular data structures, such as matrix factorization and decision trees, adds substantial overhead, but we find that it is still several orders of magnitude better than previous work based on advanced cryptography.

6.1 Datasets

We use standard machine learning datasets from the *UCI Machine Learning Repository*.[4] We evaluate matrix factorization on the MovieLens dataset [28]. Table 1 summarizes our datasets and configuration parameters.

The *Nursery* dataset describes the outcomes of the Slovenian nursery admission process for 12,960 applications in the 1980s. Given eight socio-economic attributes about the child and parents, the task is to classify the record into one out of five possible classes. We use the 0/1 encoding of the attributes as we evaluate the records on binary decision trees. Hence, each record in the dataset is represented using 27 features.

The *MNIST* dataset is a set of 70,000 digitized grayscale images of 28-by-28 pixels recording handwritten digits written by 500 different writers. The task is to classify each image into one of ten possible classes.

The *SUSY* dataset comprises 5,000,000 instances produced by Monte Carlo simulations of particle physics processes. The task is to classify, based on 18 observed features, whether the particles originate from a process producing supersymmetric (SUSY) particles or not.

The *MovieLens* datasets contain movie ratings (1–5): 100K ratings given by 943 users to 1682 movies.

The datasets were chosen either because they were used in prior work on secure ML (e.g., Nursery in [13], MovieLens in [47, 48]) or because they are one of the largest in the UCI repository (e.g., SUSY), or because they represent common benchmarks for particular algorithms (e.g., MNIST for neural networks).

Our learning algorithms are iterative—the accuracy (and execution time) of the model depends on the number of iterations. In our experiments, we fixed the number of iterations a priori to a value that typically results in convergence: 10 for k-means, 5 for neural network, 10 for SVM, and 20 for matrix factorization.

6.2 Setup

The experiments were conducted on a single machine with quad-core Intel Skylake processor, 8GB RAM, and 256GB solid state drive running Windows 10 enterprise. This processor limits the amount of platform memory that can be reserved for enclaves to 94MB (out of a total of 128MB of protected memory). Each benchmark is compiled using the Microsoft C/C++ compiler version 17.00 with the O2 flag (optimize for speed) and linked against the Intel SGX SDK for Windows version 1.1.30214.81. We encrypted and integrity protected the input datasets with AES-GCM; we used a hardware-accelerated implementation of AES-GCM based on the Intel AES-NI instructions. We ran non-oblivious and oblivious versions of our algorithms that decrypt and process the binary data inside SGX enclaves. We compare the run times with a baseline that processes the data in plaintext and does not use SGX protection. Table 1 summarizes the relative run time for all the algorithms (we report averages over five runs). Next we analyze the results for each algorithm.

6.3 K-Means

We have implemented a streaming version of the k-means clustering algorithm to overcome space constraints of enclaves. Our implementation partitions the inputs into batches of a specified size, copies each batch into enclave memory, decrypts it and processes each point within the batch.

Table 1 shows the overheads for partitions of size 1MB. The non-oblivious and oblivious versions have overheads of 91% and 199% over baseline (6.8 seconds). The overhead for the non-oblivious version is due to the cost of copying encrypted data into the enclave and decrypting it.

We observe similar overheads for longer executions. The overheads decrease with the number of clusters (34% with 30 clusters and 11% with 50 clusters for non-oblivious version) and (154% for 30 clusters and 138% for 50 clusters for oblivious version) as the cost of input decryption is amortized over cluster computation. By comparison, recent work [38] based on cryptographic primitives reports 5 to 6 orders of magnitude slowdown for k-means.

6.4 Neural Networks

We have implemented a streaming version of the algorithm for training a convolution neural network (CNN) on top of an existing library [20]. Table 1 shows the overheads of training the network for the MNIST dataset.

[4] https://archive.ics.uci.edu/ml/

Algorithm	SGX+enc.	SGX+enc.+obl.	Dataset	Parameters	Input size	# Instances
K-Means	1.91	2.99	MNIST	k=10, d=784	128MB	70K
CNN	1.01	1.03				
SVM	1.07	1.08	SUSY	k=2, d=18	307MB	2.25M
Matrix fact.	1.07	115.00	MovieLens	n=943, m=1,682	2MB	100K
Decision trees	1.22	31.10	Nursery	k=5, d=27	358KB	6.4K

Table 1: Relative run times for all algorithms with SGX protection + encryption, and SGX protection + encryption + data obliviousness, compared with a baseline that processes the data in plaintext without SGX protection. Parameters of the datasets used for each algorithm are provided on the right, where d is the number of features, k is the number of classes, n is the number of users and m is the number of movies in the MovieLens dataset.

The low overheads ($< 0.3\%$) reflect the observation that the training algorithm is predominantly data oblivious, hence running obliviously does not increase execution time while achieving the same accuracy. We are aware that state-of-the-art implementations use data-dependent optimizations such as max pooling and adding noise; finding oblivious algorithms that support these optimizations with good performance remains an open problem.

6.5 SVM

As described in Section 4.2, the correctness of supervised learning methods requires that the input data instances be independent and identically distributed. Our oblivious SVM implementation achieves this by accessing a batch of l data instances uniformly at random during each iteration. We implement random access by copying the partition containing the instance into enclave memory, decrypting the partition and then accessing the data instance. In the experiments we set data partitions to be of size 2KB and $l = 20$. In addition, we use conditional move instructions to make data accesses within the training algorithm oblivious. These modifications allow us to process datasets much larger than enclave memory. Our evaluation (Table 1) shows that random access adds a 7% overhead to the non-oblivious SVM algorithm, whereas the additional overhead of the oblivious algorithm is marginal.

6.6 Decision Tree Evaluation

For the Nursery dataset, we use an offline-trained ensemble of 32 sparse decision trees (182KB) with 295–367 nodes/leaves each and depths ranging from 14 to 16 layers. For this dataset, as shown in Table 1, our oblivious classifier running inside an enclave has an average overhead of 31.1x over the baseline (255ms vs. 10ms). The oblivious implementation of the algorithm employs the `oget()` primitive (see Section 3) for all data-dependent array lookups. Without this optimization, using `omoveEx()` for the element-granular scanning of arrays instead, the overhead is much higher (142.27x on average). We observe that our oblivious implementation scales well to even very large trees. For example, for an ensemble of 32 decision trees (16,860KB) with 30,497–32,663 nodes/leaves and 35–45 layers each,[5] the average overhead is 63.16x over the baseline.

In comparison, prior work based on homomorphic encryption [13] uses much smaller decision trees (four nodes on four layers for the Nursery dataset), has higher overheads and communication costs, and scales poorly with increasing depth. Our experiments show that smaller depth trees have much lower accuracy (82% for depth 4 and 84% for depth 5). In contrast, our classifier for the Nursery dataset achieves an accuracy of 98.7%.

6.7 Matrix Factorization

We measure the performance of our gradient descent on the MovieLens dataset. We implemented both the baseline algorithm and the oblivious algorithm of Section 4.6. As for k-means, we stream the input data (once) into the enclave to initialize the data structures, then we operate on them in-place. We also implemented the oblivious method of Nikolaenko *et al.* [48] to compare its overhead with ours (see Section 7 for the asymptotic comparison). We did not use garbled circuits, and merely implemented their algorithm natively on Skylake CPUs.

In each experiment, we set the dimension of user and vector profiles to $d = 10$, following previous implementations in [47, 48]. We experimented with fixed numbers of iterations $T = 1, 10, 20$. With higher number of iterations the prediction rate of the model improves. For example, when using 90K instances for training, the mean squared error of prediction on 10K test dataset drops from 12.94 after 1 iteration, to 4.04 after 20 iterations, to 1.06 after 100 iterations (with $\lambda = \mu = \gamma = 0.0001$).

Table 1 reports the overheads for the MovieLens-100K dataset. The oblivious version takes 49s, versus 0.43s for the baseline, reflecting the cost of multiple oblivious sorting for each of the $T = 20$) iterations. With smaller number of iterations, the running times are 8.2s versus 0.03s for $T = 1$, and 27s versus 0.21s for $T = 10$. (As a sanity check, a naive oblivious algorithm that accesses

[5] The numbers correspond to a random forest trained on the standard *Covertype* dataset from the UCI repository.

T	This work	Previous work
1	8	14 (1.7x)
10	27	67 (2.4x)
20	49	123 (2.5x)

Table 2: Comparison of running times (in seconds) of oblivious matrix factorization methods on the MovieLens dataset: our work is the method in Section 4.6 and previous work is our implementation of an algorithm in [48] *without* garbled circuits. T is the number of algorithm iterations.

all entries in U and V to hide its random accesses runs in 1850s for $T = 10$.)

Table 2 compares the overheads of our oblivious algorithm and the one of [48]. As expected, our method outperforms theirs as the number of iterations grows, inasmuch as it sorts smaller data structures.

Comparison with cryptographic evaluations: We are aware of two prior evaluations of oblivious matrix factorization [47, 48]. Both solutions are based on garbled circuits, and exploit their parallelism. Both only perform one iteration ($T = 1$). Nikolaenko *et al.* (in 2013) report a run time of 2.9 hours for 15K ratings (extracted from 100K MovieLens dataset) using two machines with 16 cores each. Nayak *et al.* (in 2015) report a run time of 2048s for 32K ratings, using 32 processors.

6.8 Security Evaluation

We experimentally confirmed the data-obliviousness of all enclave code for each of our algorithms, as follows. We ran each algorithm in a simulated SGX environment[6] and used Intel's Pin framework [42] to collect runtime traces that record all memory accesses of enclave code, not only including our core algorithms, but also all standard libraries and SGX framework code. For each algorithm, we collected traces for a range of different inputs of the same size and compared code and data accesses at cache-line granularity, simulating the powerful attacker from Section 2. While we initially discovered deviations in the traces due to implementation errors in our oblivious primitives and algorithmic modifications, we can report that the final versions of all implementations produce uniform traces that depend only on the input size.

7 Related Work

Secure multi-party machine learning General cryptographic approaches to secure multi-party computation are based on garbled circuits, secret sharing and encryption with homomorphic properties. Lindell and Pinkas [36] survey these techniques with respect to data mining tasks including machine learning. It is worth noting that beside mathematical assumptions, some of the above approaches also rely on (a subset of) computing parties being honest when running the protocol as well as non-colluding.

Garbled circuits [71] provide a mechanism for multiple parties to compute any function on their joint inputs without having to reveal the inputs to each other. Solutions based on garbled circuits have been tailored for several specific machine learning tasks including matrix factorization [48], and training of a decision tree [6, 35]. GraphSC [47] and ObliVM [38] are two recent programming frameworks for secure computation using garbled circuits. The former framework offers a paradigm for parallel computation (e.g., MapReduce) and the latter uses a combination of ORAM and garbled circuits.

Training of an SVM kernel [34] and construction of a decision tree [18] have been proposed based on secret-sharing and oblivious transfer.

Homomorphic encryption lets multiple parties encrypt their data and request the server to compute a joint function by performing computations directly on the ciphertexts. Bost *et al.* [13] study classification over encrypted data in the model where the server performs classification by operating on semi-homomorphic encrypted data; whenever the server needs to perform operations not supported by the encryption, it engages in a protocol with a party that can decrypt the data and perform the necessary computation. Solutions based on fully-homomorphic encryption have been proposed for training several ML algorithms [27, 69] and for classifying decision trees [68].

Shokri and Shmatikov [59] describe a method for multiple parties to compute a deep neural network on joint inputs. This method does not rely on cryptographic primitives. It assumes that the parties train their own model and do not share the data with each other, but exchange intermediate parameters during training. Our model is different as parties in our solution do not perform any computation and do not learn anything about the training process; after the training they can either obtain the model, if they agreed to, or use it for querying in a black-box manner.

Privacy implications of revealing the output of a machine learning algorithm, i.e., the model, is orthogonal to the focus of this paper; we refer the reader to Fredrikson *et al.* [21, 22] on this topic. As a remedy, differential privacy guarantees for the output of several machine learning algorithms have been studied by Blum *et al.* [11].

Data-oblivious techniques Oblivious RAM (ORAM) [25] is a general protection technique against

[6] For debugging purposes, the Intel SGX SDK allows for the creation of simulated SGX enclaves. Those simulated enclaves have largely the same memory layout as regular SGX enclaves, but are not isolated from the rest of the system. In simulation mode, SGX instructions are emulated in software inside and outside the enclave with a high level of abstraction.

side-channels on memory accesses. Though recent advances in this space [62] have significantly decreased the ORAM overhead, there are cases where the default solution does not always meet system requirements. First, many ORAM solutions offer a tradeoff between the size of the private memory and the overhead they incur. In current CPUs, registers act as an equivalent of processor's private memory, however their number is limited, e.g., even for the latest x86 generations, less than 2KB can be stored in all general purpose and SIMD registers combined. Second, ORAM does not hide the number of accesses. That is, if this number depends on a sensitive input (e.g., number of movies rated by each user) then fake accesses need to be generated to hide the real number of accesses. Finally, ORAM is ideal for programs that make few accesses in a large dataset. For algorithms that process data multiple times, customized solutions often perform better (e.g., MapReduce [47, 49]). Machine learning algorithms fall in the latter category as they need all input instances to train and use the model.

Raccoon [53] and GhostRider [37] propose general compiler techniques for protecting memory accesses of a program. Some of the techniques they deploy are ORAM and execution of both branches of if-else statements. However, general techniques are less effective in cases where an algorithm accesses data in a structured way that can be exploited for greater efficiency. For example, compiling matrix factorization using these techniques is not trivial as the interleaving of the accesses between internal data structures has to be also protected. (The interleaving depends on sensitive information such as rating counts per user and per movie which have to be taken into account.)

Asymptotical comparison of individual algorithms We now compare the asymptotic performance of our data-oblivious algorithms to prior work. We evaluate the overhead of obtaining oblivious properties only. That is, we do not consider the cost of their secure implementation on SGX (our approach) or using garbled circuits in [38, 47, 48] (though the latter is known to add large run time overheads).

ObliVM [38] uses a streaming version of MapReduce to perform oblivious *k-means* which is then compiled to garbled circuits. The algorithm relies on oblivious sorting to update the centroids at each iteration, resulting in the running time of $O\left(T(nkd + dn(\log n)^2)\right)$ (ignoring conversion to garbled circuits). Since our algorithm takes $O(Tnkd)$ time, the asymptotical comparison between the two depends on the relation between values k and $O((\log n)^2)$. Moreover, oblivious sorting incurs high constants and our experiments confirmed that our simple method was more efficient.

The algorithmic changes required to make *SVM* and *Neural Networks* oblivious can be captured with automated tools for compiling code into its oblivious counterpart. Instead of an oblivious shuffle or sort between the iterations, these methods would place input instances into an ORAM and then sample them by accessing the ORAM. Since all n instances are accessed at each iteration, the asymptotical cost of the two solutions remains the same. However, such tools either use a backend that would require careful adaption for the constrained SGX environment (for example, GhostRider [37] defines its own source language and ObliVM [38] translates code into circuits) or they are not as optimized as our approach (for example, Raccoon [53] always executes both code paths of a secret-dependent conditional statement and its described array scanning technique is less fine-tuned for the x86 architecture than ours).

Our simple data-oblivious *decision tree* algorithm is adequate for ad hoc tree evaluations, and scales up to reasonably large forests. With larger irregular data structures, algorithms based instead on oblivious data structures [38, 66] may be more effective as they store data in elaborate randomized data structures that avoid streaming over all the tree leaves. Though their asymptotical performance dominates our approach — $O((\log n)^2)$ vs. $O(n)$, assuming height of the tree of $O(\log n)$ — as pointed out in [53] ORAM-based solutions improve over the plain scanning of arrays only for larger n due to the involved constants. Moreover, private memory of size $O(\log n)$ is assumed in [66] while as mentioned earlier, private memory for SGX is limited to registers. Oblivious tree can be implemented also via ORAM with constant private memory size [33], incurring $O((\log n)^3/\log\log n)$ overhead.

Finally, oblivious *matrix factorization* for garbled circuits, rather than SGX processors, was considered in [48] and [47]. Nikolaenko *et al.* [48] also rely on a data structure that combines both user and movie profiles: They maintain a global matrix of size $M+n+m$ with M rows for the ratings, n rows for the users, and m rows for the movies. Their updates are performed in several sequential passes, and synchronized using a sorting network on the whole data structure. Hence, their algorithm runs in time $O\left(T(M+n+m)(\log(M+n+m))^2\right)$, dominated by the cost of sorting the rows of the matrix at every iteration. GraphSC [47] implements matrix factorization using an oblivious parallel graph processing paradigm. However, this method also relies on oblivious sorting of $M+n+m$ profiles, hence, asymptotically it incurs the same cost. In comparison, our approach sorts on that scale only during Setup and, besides, those costly operations only sort user ids and ratings—not the larger profiles in $\mathbb{R}^d$. Then, asymptotically, our iterations are more efficient due to a smaller logarithmic factor as we sort fewer tuples at a time: $O\left(T(M+\tilde{n})(\log\tilde{n})^2\right)$ where $\tilde{n} = \max(n,m)$. As we showed in the evaluation section our method also outperforms [48] in practice.

Similar to prior oblivious matrix factorization techniques [47,48], our method is easily parallelizable. First, an oblivious sort that runs in time $O(n(\log n)^2)$ sequentially can run in time $O((\log n)^2)$ with n parallel processes. Besides, each row in our data structures **U** and **V** can be processed independently, and aggregated in time $\log(M+\tilde{n})$, as in the method described in [47]. Even with parallel processing, our method is more efficient, because the depth of the computation stays logarithmic in $\tilde{n}$ for our method and M in theirs [47,48].

Secure hardware TrustedDB [5], Cipherbase [3], and Monomi [64] use different forms of trusted hardware to process database queries with privacy. Haven [8] runs unmodified Windows applications in SGX enclaves, and VC3 [57] proposes a cloud data analytics framework based on SGX. None of these systems provides protection from side-channel attacks. These systems were evaluated using SGX emulators. In contrast, we are the first to evaluate implementations of machine learning algorithms on real SGX processors.

8 Conclusions

We presented a new practical system for privacy-preserving multi-party machine learning. We propose data-oblivious algorithms for support vector machines, matrix factorization, decision trees, neural networks, and k-means. Our algorithms provide strong privacy guarantees: they prevent exploitation of side channels induced by memory, disk, and network accesses. Experiments with an efficient implementation based on Intel SGX Skylake processors show that our system provides good performance on realistic datasets.

References

[1] Ajtai, M., Komlós, J., and Szemerédi, E. An $O(n\log n)$ sorting network. In *ACM Symposium on Theory of Computing (STOC)* (1983).

[2] Anati, I., Gueron, S., Johnson, S., and Scarlata, V. Innovative technology for CPU based attestation and sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).

[3] Arasu, A., Blanas, S., Eguro, K., Kaushik, R., Kossmann, D., Ramamurthy, R., and Venkatesan, R. Orthogonal security with Cipherbase. In *Conference on Innovative Data Systems Research (CIDR)* (2013).

[4] Arasu, A., and Kaushik, R. Oblivious query processing. In *International Conference on Database Theory (ICDT)* (2014).

[5] Bajaj, S., and Sion, R. TrustedDB: A trusted hardware-based database with privacy and data confidentiality. In *IEEE Transactions on Knowledge and Data Engineering* (2014).

[6] Barni, M., Failla, P., Kolesnikov, V., Lazzeretti, R., Sadeghi, A., and Schneider, T. Secure evaluation of private linear branching programs with medical applications. In *European Symposium on Research in Computer Security (ESORICS)* (2009).

[7] Batcher, K. E. Sorting networks and their applications. In *Spring Joint Computer Conf.* (1968).

[8] Baumann, A., Peinado, M., and Hunt, G. Shielding applications from an untrusted cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[9] Bell, R. M., and Koren, Y. Lessons from the Netflix prize challenge. *ACM SIGKDD Explorations Newsletter 9*, 2 (2007).

[10] Bishop, C. M. *Neural networks for pattern recognition*. Oxford university press, 1995.

[11] Blum, A., Dwork, C., McSherry, F., and Nissim, K. Practical privacy: The SuLQ framework. In *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)* (2005).

[12] Boser, B. E., Guyon, I. M., and Vapnik, V. N. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory* (1992).

[13] Bost, R., Popa, R. A., Tu, S., and Goldwasser, S. Machine learning classification over encrypted data. In *Symposium on Network and Distributed System Security (NDSS)* (2015).

[14] Breiman, L. Random forests. *Machine Learning 45*, 1 (2001).

[15] Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. *Classification and Regression Trees*. Wadsworth, 1984.

[16] Criminisi, A., Shotton, J., and Konukoglu, E. Decision forests: A unified framework for classification, regression, density estimation, manifold learning and semi-supervised learning. *Foundations and Trends in Computer Graphics and Vision 7*, 2-3 (2012).

[17] C.Yao, A. Protocols for secure computations (extended abstract). In *IEEE Symposium on Foundations of Computer Science (FOCS)* (1982).

[18] de Hoogh, S., Schoenmakers, B., Chen, P., and op den Akker, H. Practical secure decision tree learning in a teletreatment application. In *Financial Cryptography and Data Security (FC)* (2014).

[19] Dwork, C., McSherry, F., Nissim, K., and Smith, A. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference (TCC)* (2006).

[20] Fast CNN library. http://fastcnn.codeplex.com/ (accessed 17/02/2016).

[21] Fredrikson, M., Jha, S., and Ristenpart, T. Model inversion attacks that exploit confidence information and basic countermeasures. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[22] Fredrikson, M., Lantz, E., Jha, S., Lin, S., Page, D., and Ristenpart, T. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *USENIX Security Symposium* (2014).

[23] Gentry, C. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC)* (2009).

[24] Goldreich, O., Micali, S., and Wigderson, A. How to play any mental game. In *ACM Symposium on Theory of Computing (STOC)* (1987).

[25] Goldreich, O., and Ostrovsky, R. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM) 43*, 3 (1996).

[26] Goodfellow, I., Bengio, Y., and Courville, A. Deep learning. Book in preparation for MIT Press, 2016.

[27] Graepel, T., Lauter, K., and Naehrig, M. ML confidential: Machine learning on encrypted data. In *International Conference on Information Security and Cryptology (ICISC)* (2013).

[28] Harper, F. M., and Konstan, J. A. The MovieLens datasets: History and context. In *ACM Transactions on Interactive Intelligent Systems (TiiS)* (2015).

[29] Hoekstra, M., Lal, R., Pappachan, P., Rozas, C., Phegade, V., and del Cuvillo, J. Using innovative instructions to create trustworthy software solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).

[30] Hyeong Hong, J., and Fitzgibbon, A. Secrets of matrix factorization: Approximations, numerics, manifold optimization and random restarts. In *Proceedings of the IEEE International Conference on Computer Vision* (2015).

[31] Intel Corp. Intel 64 and IA-32 architectures software developer's manual—combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c, 2013. No. 325462-048.

[32] Koren, Y., Bell, R., and Volinsky, C. Matrix factorization techniques for recommender systems. *Computer*, 8 (2009).

[33] Kushilevitz, E., Lu, S., and Ostrovsky, R. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2012).

[34] Laur, S., Lipmaa, H., and Mielikäinen, T. Cryptographically private support vector machines. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2006).

[35] Lindell, Y., and Pinkas, B. Privacy preserving data mining. *Journal of Cryptology* (2000).

[36] Lindell, Y., and Pinkas, B. Secure multiparty computation for privacy-preserving data mining. *IACR Cryptology ePrint Archive* (2008).

[37] Liu, C., Harris, A., Maas, M., Hicks, M. W., Tiwari, M., and Shi, E. Ghostrider: A hardware-software system for memory trace oblivious computation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015).

[38] Liu, C., Wang, X. S., Nayak, K., Huang, Y., and Shi, E. ObliVM: A programming framework for secure computation. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[39] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[40] Lloyd, S. P. Least squares quantization in PCM'S. *Bell Telephone Labs Memo* (1957).

[41] Lloyd, S. P. Least squares quantization in PCM. *IEEE Transactions on Information Theory 28*, 2 (1982).

[42] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2005).

[43] MacQueen, J. Some methods for classification and analysis of multivariate observations. In *Berkeley Symposium on Mathematics, Statistics and Probability, Vol. 1* (1967).

[44] Malkhi, D., Nisan, N., Pinkas, B., and Sella, Y. Fairplay: a secure two party computation system. In *USENIX Security Symposium* (2004).

[45] Mckeen, F., Alexandrovich, I., Berenzon, A., Rozas, C., Shafi, H., Shanbhogue, V., and Savagaonkar, U. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)* (2013).

[46] Narayanan, A., and Shmatikov, V. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy (S&P)* (2008).

[47] Nayak, K., Wang, X. S., Ioannidis, S., Weinsberg, U., Taft, N., and Shi, E. GraphSC: Parallel secure computation made easy. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[48] Nikolaenko, V., Ioannidis, S., Weinsberg, U., Joye, M., Taft, N., and Boneh, D. Privacy-preserving matrix factorization. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[49] Ohrimenko, O., Costa, M., Fournet, C., Gkantsidies, C., Kohlweiss, M., and Sharma, D. Observing and preventing leakage in MapReduce. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[50] Ohrimenko, O., Goodrich, M. T., Tamassia, R., and Upfal, E. The Melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages and Programming (ICALP)*, vol. 8573. Springer, 2014.

[51] Quinlan, J. R. Induction of decision trees. *Machine Learning 1*, 1 (1986).

[52] Quinlan, J. R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[53] Rane, A., Lin, C., and Tiwari, M. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security Symposium* (2015).

[54] Rastogi, A., Hammer, M. A., and Hicks, M. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *IEEE Symposium on Security and Privacy (S&P)* (2014).

[55] Sarwar, B., Karypis, G., Konstan, J., and Riedl, J. Application of dimensionality reduction in recommender system – A case study. Tech. rep., DTIC Document, 2000.

[56] Schölkopf, B., and Smola, A. J. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.

[57] Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., and Russinovich, M. *VC*3: Trustworthy data analytics in the cloud using sgx. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[58] Shalev-Shwartz, S., Singer, Y., Srebro, N., and Cotter, A. Pegasos: Primal estimated sub-gradient solver for SVM. *Mathematical programming 127*, 1 (2011).

[59] Shokri, R., and Shmatikov, V. Privacy-preserving deep learning. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[60] Sinha, R., Costa, M., Lal, A., Lopes, N., Seshia, S., Rajamani, S., and Vaswani, K. A design and verification methodology for secure isolated regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2016).

[61] Smola, A. J., and Schölkopf, B. A tutorial on support vector regression. *Statistics and computing 14*, 3 (2004).

[62] Stefanov, E., van Dijk, M., Shi, E., Fletcher, C. W., Ren, L., Yu, X., and Devadas, S. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[63] Tsochantaridis, I., Joachims, T., Hofmann, T., and Altun, Y. Large margin methods for structured and interdependent output variables. In *Journal of Machine Learning Research* (2005).

[64] Tu, S., Kaashoek, M. F., Madden, S., and Zeldovich, N. Processing analytical queries over encrypted data. In *International Conference on Very Large Data Bases (VLDB)* (2013).

[65] Vapnik, V. N., and Vapnik, V. *Statistical learning theory*, vol. 1. Wiley New York, 1998.

[66] Wang, X. S., Nayak, K., Liu, C., Chan, T., Shi, E., Stefanov, E., and Huang, Y. Oblivious data structures. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[67] Weston, J., and Watkins, C. Support vector machines for multi-class pattern recognition. In *ESANN* (1999).

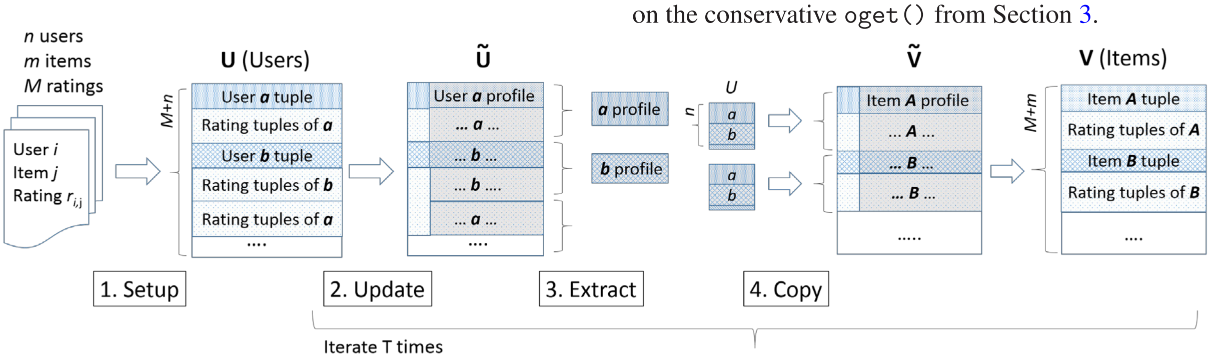
[68] Wu, D. J., Feng, T., Naehrig, M., and Lauter, K. Privately evaluating decision trees and random forests. *IACR Cryptology ePrint Archive* (2015).

[69] Xie, P., Bilenko, M., Finley, T., Gilad-Bachrach, R., Lauter, K. E., and Naehrig, M. Crypto-nets: Neural networks over encrypted data. *CoRR abs/1412.6181* (2014).

[70] Xu, Y., Cui, W., and Peinado, M. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[71] Yao, A. C. How to generate and exchange secrets (extended abstract). In *IEEE Symposium on Foundations of Computer Science (FOCS)* (1986).

A Illustration of Oblivious Matrix Factorization from Section 4.6



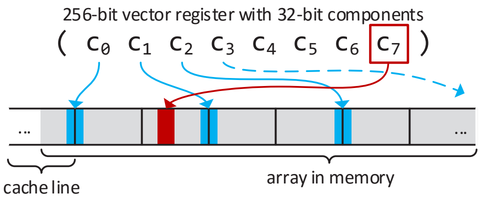

256-bit vector register with 32-bit components
(C0 C1 C2 C3 C4 C5 C6 C7)
...
...
cache line
array in memory

Figure 4: Optimized array scanning using the 256-bit vector instruction `vpgatherdd`

B Optimized Array Scanning

The `oget()` primitive can be further optimized using `vpgatherdd` as follows. We make sure that a certain number of components of the vector register load values that span *two* cache lines. (This can be done by loading two bytes from one cache line and two bytes from the next one, recall that each component loads 4 bytes.) Hence, up to 16 cache lines can potentially be touched with a single instruction.

We assign components to cache lines in a careful manner. The first few components request addresses within dummy cache lines or cache lines that contain the values of interest (whose addresses should be kept secret). The values of interest are loaded into the remaining components. The concept is depicted in Figure 4 where components C_0 and C_2-C_6 request dummy cache lines, C_1 requests the cache lines that contain the desired value which is loaded into C_7. In this configuration, four bytes are read obliviously from a memory region of size $7 \cdot 2 \cdot 64\ bytes = 896\ bytes$ with a single `vpgatherdd` instruction. The method easily generalizes when more bytes (e.g., 8 bytes using C_6 and C_7) are to be read.

This technique can significantly increase throughput (up to 2x in some micro-benchmarks outside enclaves on recent Intel Skylake processors). However, it requires that `vpgatherdd` appears as a truly atomic operation or, at least, that the hardware loads dummy components before secret ones (and those are then loaded from the cache). Though this may be true in a software-only attacker model, it is not the case in the powerful threat model in Section 2. Hence, our implementation relies on the conservative `oget()` from Section 3.