Secure Sparse Matrix Multiplications and their Applications to Privacy-Preserving Machine Learning

Marc Damie
University of Twente
Enschede, The Netherlands
Inria
Villeneuve-d'Ascq, France

Florian Hahn
f.w.hahn@utwente.nl
University of Twente
Enschede, The Netherlands

Andreas Peter
Carl von Ossietzky Universität Oldenburg
Oldenburg, Germany

Jan Ramon
Inria
Villeneuve-d'Ascq, France

Abstract

To preserve data privacy, multi-party computation (MPC) enables executing Machine Learning (ML) algorithms on private data. However, MPC frameworks do not include optimized operations on sparse data. This absence makes them unsuitable for ML applications involving sparse data; e.g., recommender systems or genomics. Even in plaintext, such applications involve high-dimensional sparse data, that cannot be processed without sparsity-related optimizations due to prohibitively large memory requirements.

Since matrix multiplication is a central building block of ML algorithms, our work proposes dedicated MPC algorithms to multiply secret-shared sparse matrices. Our sparse algorithms have several advantages over secure dense matrix multiplications (i.e., the classic multiplication). On the one hand, they avoid the memory issues caused by the "dense" data representation of dense multiplications. On the other hand, our algorithms can significantly reduce communication costs (up to ×1000) for realistic problem sizes. We validate our algorithms in two machine learning applications where dense matrix multiplications are impractical. Finally, we take inspiration from real-world sparse data properties to build 3 techniques minimizing the public knowledge necessary to secure sparse algorithms.

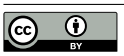
CCS Concepts

• **Security and privacy → Privacy protections**; **Cryptography**;
• **Computing methodologies → Machine learning**.

Keywords

Sparse data, Matrix multiplication, Cryptography, Multi-Party Computation, Privacy-Preserving Machine Learning

ACM Reference Format:
Marc Damie, Florian Hahn, Andreas Peter, and Jan Ramon. 2026. Secure Sparse Matrix Multiplications and their Applications to Privacy-Preserving Machine Learning. In *Proceedings of the Sixteenth ACM Conference on Data and Application Security and Privacy (CODASPY '26), June 23–25, 2026, Frankfurt am Main, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3800506.3803493



This work is licensed under a Creative Commons Attribution 4.0 International License.
CODASPY '26, Frankfurt am Main, Germany
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2562-3/2026/06
https://doi.org/10.1145/3800506.3803493

1 Introduction

MPC protocols are cryptographic primitives allowing to compute functions on secret inputs. They have been applied in many contexts involving privacy concerns such as privacy-preserving ML (PPML) [31] or telemetry [3]. However, existing protocols remain impractical for many real-world applications, especially when the applications involve high-dimensional *sparse data*, i.e., data with a large majority of zero values.

Sparse data arises naturally in many applications including recommender systems, genomics, and natural language processing. For instance, in a video recommender system, each user watches only a tiny fraction of the catalog. We can encode a user's history as a binary vector, with 1 indicating a watched video and 0 otherwise, yielding very long vectors that are mostly zeros. Real-world datasets can be extremely sparse [27]: about 99% zeros in Netflix dataset, 99.9% in Tox21, 99.99% in Bookcrossing, and 99.999% in Flickr. At such sparsity rates, specialized algorithms are essential; otherwise, computations become inefficient, or even infeasible if the dataset cannot fit in memory using dense storage.

Storing large sparse datasets in a dense format (one value per cell) demands excessive memory. On real-world sparse data, this "dense storage" often becomes prohibitive; making computation infeasible due to memory constraints. Moreover, dense representations lead to inefficient linear algebra operations, as most of the computational effort is wasted processing zeros. Over the last 50 years, algorithms to process plaintext sparse data efficiently have been proposed [5, 11]. Since 2002, dedicated software libraries [12] popularized them. Sparse data is present in many ML applications but also in other fields, including graph algorithms. Our paper focuses on PPML applications, but our results apply to any application involving private sparse data.

Several MPC frameworks enable training ML models on secret-shared data [24, 29–31, 34, 40, 41]. However, **none of these frameworks provide algorithms optimized for sparse data**.

The scalability limitations (especially memory issues) of dense plaintext algorithms transfer naturally to dense MPC algorithms. Thus, **MPC frameworks need dedicated sparse algorithms** for ML applications like recommender systems. These frameworks are not fundamentally incompatible with sparse data; there is simply no suitable secure sparse algorithm in the literature.

Related works. There exist protocols to sum sparse vectors securely, including sorting-based [3, 4], and shuffling-based [43].

A few papers [6, 9, 36] studied secure sparse multiplications in a two-party setting. However, these protocols require the data owners to actively participate because one of the computation parties must *know the plaintext sparse data*. Hence, they do not support more than two data owners simultaneously, which is too limiting, as modern ML applications involve thousands of data owners.

Typical ML applications require a more generic MPC setup: outsourced training. In this setting, the data owners share their secret data with a group of computation servers and disconnect. This distinction between input parties and computation parties enables supporting an arbitrary number of data owners, contrary to existing works [6, 9, 36]. Appendix C details why the adaptation of these works either results in an inefficient construction or is not straightforward. The secure multiplication of sparse matrices in the outsourced setting is then **a significant open problem**. *Our work will focus on this outsourced setting; providing sparse multiplications compatible with any MPC framework.*

Some information-theory works [15, 42] studied the multiplication of a secret-shared matrix with a public matrix. Due to the public input, their setting differs from classic MPC where all inputs are secret. Moreover, Yu et al. [44] has recently proposed a sparse matrix-vector multiplication based on fully homomorphic encryption (FHE). While they highlight the importance of secure sparse multiplications, we cannot compared directly to them as their FHE threat model (single dishonest server) differs from our MPC-based one (group of non-colluding servers). Finally, Nayak et al. [33] presented secure graph computations (a problem closely related to sparse operations due to the sparse adjacency matrices), but they do not present multiplication protocols.

Our contributions. We present **two secure algorithms** to compute matrix multiplications on secret-shared sparse data. Our algorithms **avoid the memory issues** of dense matrix multiplications on sparse data (e.g., in an experiment from Sec. 5: 19TB using dense multiplications vs. 60GB using ours). Moreover, our algorithms **reduce the communication costs** up to a factor 1000, and we implement two ML applications that are impractical with existing secure algorithms. Third, as we highlight that any efficient MPC protocol on sparse data (also acknowledged in [36]) requires public knowledge about the sparsity, we leverage the properties of real-world sparse data to **minimize this mandatory public knowledge** and obtain it in a privacy-preserving manner.

Notations. Let $\mathrm{nnz}(x)$ (resp. $\mathrm{nnz}(X)$) refer to the number of non-zero values in a vector x (resp. matrix X). Let $[\![a]\!]$ refer to a share of a secret value a. In our algorithms, any operation involving shared values corresponds to a secure variant; e.g., $[\![a]\!] + [\![b]\!]$ is the secure addition of a and b and $[\![\mathrm{sort}(l)]\!]$ is the oblivious sort of l. Secure algorithms are executed jointly by all shareholders.

2 Preliminaries

2.1 Threat model

Threat modeling describes the expected behavior of the agents involved in an MPC protocol: an *honest* agent follows the protocol, an *honest-but-curious* agent follows the protocol and attempts to infer private information passively, and a *malicious* agent does not necessarily follow the protocol and can actively disrupt the protocol to obtain private information. MPC works distinguish security against a majority of honest agents from security against a majority of dishonest agents. "Dishonest agent" is an umbrella term encompassing malicious and honest-but-curious agents (depending on the security setup considered).

We analyze our algorithms under honest and dishonest majority. We focus on honest-but-curious security, but our algorithms rely on oblivious shuffling and sorting with known maliciously secure variants [19, 25]. We leave for future work the extension to malicious security and the subsequent security proofs.

In line with related works, our threat model excludes poisoning attacks where malicious data owners forge input data so the protocol reveal private information. We assume the data owners provide well-formed secret-shared inputs.

2.2 Outsourced setting

Our contribution focuses on an "*outsourced*" computation setup (popular in recent PPML works [21, 31, 40, 41]): each data owner shares their secret data with a group of computation servers. These servers build a secret-shared matrix $[\![X]\!]$ containing all data owners' inputs; e.g., one row per data owner – referred to as "horizontal" data partitioning in PPML literature. Then, they can perform secure computations on this matrix. For example, the servers can execute a protocol to train an ML model $[\![\theta]\!]$ on the dataset X. In this setting, the data owners share their secret data and disconnect. Outsourced protocols support an arbitrary number of data owners because they separate input parties (i.e., data owners) from computation parties.

Contrary to many PPML works that support a two-party [21, 31] or three-party [40, 41] outsourced setups, our work support an unbounded number of computation parties. Nevertheless, secret-sharing protocols induce a quadratic communication cost in the number of parties. This quadratic cost makes outsourcing necessary to support thousands of data owners like in ML applications.

All existing secure sparse multiplications [6, 9, 36] support a non-outsourced setting. They presented *two-party protocols* assuming *the computation parties are also data owners* ($\Rightarrow$ two data owners maximum) because they require one of the computation parties to know the plaintext sparse data. Such a setting is too limiting for ML applications as they typically involve many data owners.

Adapting existing protocols to avoid designing new ones? The adaptation is not easy because a plaintext knowledge is often essential to these protocols. **Appendix C** details why the adaptation of [6, 9, 36] to the outsourced setting either results in an inefficient construction or is not straightforward.

2.3 MPC primitives

Secret sharing [37] enables sharing a secret value x between multiple parties. Each shareholder j holds a share $[\![x]\!]_j$, revealing *nothing* about x. The shareholders can reconstruct the secret value by gathering their shares. Several secret sharing schemes exist, e.g., additive shares are finite field elements drawn randomly, s.t. $\sum_j [\![x]\!]_j = x$. Many protocols [16] enable processing secret-shared values; from arithmetic operations to complex operations (e.g., sorting).

Our algorithms rely notably on **oblivious shuffling and sorting**. "Oblivious sorting" designates MPC protocols for sorting secret-shared values without revealing information about the values. Sorting networks [19] are straightforward oblivious sorting solutions, as their execution is data-independent. Their cost is $O(m \log^2 m)$ for a list of size m. The oblivious radix sort [19] is the state-of-the-art honest-majority protocol, with its optimal complexity of $O(m \log m)$ (and low constant factors). Recent papers optimized this protocol in a three-party setting [2, 3].

"Oblivious shuffling" protocols randomly permute lists of secret values without revealing the permutation. Laur et al. [25] proposed an honest-majority protocol with linear costs, used in many recent protocols [2, 3, 19]. A recent work [28] also described dishonest-majority protocols, but with supra-linear costs.

2.4 Matrix multiplication

Let Z be the result of the multiplication between the $n \times m$ matrix X and the $m \times p$ matrix Y. Z is an $n \times p$ matrix defined as follows:

$$Z_{ij} = \sum_{k=1}^{m} x_{ik} \cdot y_{kj}, \forall i \in \{1 \dots n\}, j \in \{1 \dots p\} \quad (1)$$

This formula holds for all matrices. However, the implementation may differ depending on the type of matrices (e.g., vectors, diagonal matrices, or triangular matrices). Our work focuses on optimized algorithms for sparse matrices. Sparse algorithms avoid "useless" multiplications and only multiply the non-zero values. Indeed, if the matrix multiplication is implemented naively (i.e., triple nested loop), the algorithm would compute many scalar multiplications with a null result. Hence, sparse algorithms leverage the matrix sparsity to have a cost inverse proportional to the sparsity.

Software libraries usually implement dense matrix multiplication via a triple nested loop, resulting in an $O(nmp)$ cost. While this cost has long been considered the lower bound for dense matrix multiplication, several works described algorithms with a sub-cubic cost [8]. Dumas et al. [13] described an MPC protocol to execute a matrix multiplication with sub-cubic cost. These algorithms have constant factors limiting their practical interest (*even in plaintext*), so software libraries favor algorithms with a cubic cost.

2.5 Dense matrix multiplication on secret-shared data

The cubic-cost matrix multiplication (i.e., the naive implementation of Equation (1)) can easily be imported to MPC. However, recent works [7, 30–32, 34, 41] described more efficient protocols under honest or dishonest majority assumptions. These works reduce the communication costs, but the computation cost remains asymptotically cubic. Moreover, the storage costs remain asymptotically equivalent to the plaintext algorithm, so they cannot support larger matrices than the plaintext dense algorithm.

Dishonest majority. Beaver's multiplication triplets are a popular building block to speed up secure multiplications via preprocessing steps [16]. Mohassel and Zhang [31] generalized this concept to matrices: matrix multiplication triplets. These triplets make the communication costs proportional to the input and output size. For example, the multiplication of an $n \times m$ matrix with an $m \times p$ matrix requires $O(nm+mp+np)$ communications (instead of $O(nmp)$ with the naive algorithm). This approach reduces the asymptotic cost for matrix-matrix multiplications, but the communication costs of the vector-vector and matrix-vector multiplications remains asymptotically equal to the naive costs. Matrix multiplication triplets with malicious security were later described [7, 32].

Honest majority. Using the honest majority assumption, several recent works [24, 29, 30, 34, 41] designed protocols with a communication cost proportional to the output size. Hence, a square matrix multiplication only requires $O(n^2)$ communications, and a vector multiplication $O(1)$ communications. Before these recent protocols, De Hoogh [22] had already described a matrix multiplication with equivalent costs based on Shamir's secret sharing (SSS). The recent protocols [24, 29, 30, 41] are more complex to understand than the SSS-based algorithm of [22] because they rely on share conversion protocols. Moreover, these protocols only support 3 parties, while the SSS-based solution supports an arbitrary number of parties. Nevertheless, the SSS-based solution only has honest-but-curious security, while [24, 29, 34, 41] support malicious security.

To understand the gain provided by the honest-majority assumption, let us detail the optimized vector multiplication under SSS [22]. In SSS, each secret-shared value is represented using a polynomial of degree t. To multiply two secret-shared scalars in SSS, the parties locally multiply their local shares (obtaining a polynomial of degree $2t$) and then execute a degree reduction protocol (via a "resharing" protocol). For dense vector multiplication, we can delay the degree reduction and add all local multiplication results together. The addition is possible because all local multiplication results are shares of equal polynomial degree (i.e., $2t$). Then, a single degree reduction is done on the sum result.

3 Sparse data in PPML

3.1 Sparse data representation

The term "sparse data" refers to data containing a large proportion of zero values. If stored using the default matrix format, the zeros occupy a significant memory space. When the data contains a sufficient number of zeros, sparse data representations can significantly reduce the storage space.

Several sparse data representations exist [5], each with specific algorithms to perform operations on the data. Depending on the data representations, some operations are more efficient than others. In other words, there is no perfect sparse data representation.

We use the **tuple representation** [5]: each sparse vector is a list of non-zero tuples: $t = (i, v_i)$ with i the coordinate in the vector and v_i the non-zero value. Let $\text{coord}(t) = i$ denote the coordinate of the non-zero and $\text{val}(t) = v_i$ its value. This format is already used in plaintext operations, like in the SciPy library in which it is called COO format (i.e., COOrdinate format).

Other sparse data formats like the Compressed Sparse Row format are also used in plaintext. However, they require look-up operations which are particularly expensive in MPC. Thus, our work focused on the tuple representation.

We refer to algorithms supporting sparse data as "sparse algorithm", a common abuse of terminology. We use the term "dense algorithms" to refer to classic algorithms.

3.2 Larger datasets are sparser

Optimizing dense algorithms, a dead end. To process sparse data, one may consider to use dense matrix multiplication, with more memory. Unfortunately, it would require tremendous amount of memory. Secure dense algorithms have the same asymptotic memory footprint and hence run into the same memory issue. There is a limit to the scalability of dense algorithms. Even if extreme amounts of memory would be available, dense secure multiplications would still fail to scale because their communication cost is typically at least linear in the size of the dense input.

A workaround to avoid this memory issue could be to partition the matrix in sub-matrices and perform dense multiplications on these sub-matrices. This approach is effective in classic linear algebra to handle large dense datasets and could (to some extent) reduce memory consumption. However, it would still require storing the product as a dense matrix; the memory benefit will not be significant. Moreover, it adds significant costs for repeated sparse-to-dense conversions. Finally, this partitioning approach would not reduce the communication costs of MPC protocols that would remain correlated to the full matrix size.

To sum up, sparse algorithms are popular in plaintext because there is no other efficient solutions to process high-dimensional sparse datasets. In plaintext, dense matrix multiplications usually have a sparse equivalent (e.g., NumPy for dense and SciPy for sparse in Python). To cover all ML applications, MPC frameworks should similarly have sparse algorithms. The first motivation of sparse algorithms is to avoid the memory issues caused by dense algorithms. Furthermore, they often provide significant performance gains.

Sparsity-size correlation. Real-world sparse datasets show an interesting phenomenon: the sparsity is correlated to the matrix size. Indeed, larger sparse datasets have in average a larger sparsity. Thus, **the larger datasets are, the more beneficial the sparse algorithms become.**

To understand this phenomenon, let us take the example of recommender systems on a marketplace (e.g., Amazon): the matrix quantifies the interaction of each consumer with each product. The dataset is sparse because each consumer only interacts with a small subset of all possible products. If the marketplace increases its number of products by a factor 1000, the consumer will not consume 1000 times more products. Even if the consumption increases a bit, existing datasets show that it does not follow the matrix growth; i.e., the sparsity increases. The real-world datasets that we arbitrarily selected from [27] satisfy the same trend: MovieLens 1M has 95% sparsity for 1.7K columns, Yahoo Movie has 99.7% sparsity for 12K columns, Bookcrossing has 99.99% sparsity for 340K columns, and flicker has 99.999% sparsity for 500K columns.

3.3 Public knowledge in secure sparse multiplications

Other secure sparse algorithms [6, 9, 36] assume some public knowledge. For example, Schoppmann et al. [36] assumes "that the sparsity metric of the input is public." The definition of this "sparsity metric" varies in each of their algorithm: the total number of non-zeros, or the number of non-zero rows or columns. In our paper, this public "sparsity metric" is the number of non-zeros per row.

Contrary to [36], papers like [44] did not identify their need for public knowledge, but it is always present. For example, in their FHE-based sparse multiplication, Yu et al. [44] split a large sparse matrix into several smaller matrices; the number and structure of the smaller matrices indirectly reveal the number of non-zeros.

Public knowledge could be interpreted as a limiting assumption, but **public knowledge is mandatory to design efficient and secure sparse algorithms**. In sparse algorithms, the performance gains (in memory, communication, and computation) are correlated to the sparsity. Thus, it is necessary to know the sparsity to expect any performance gain.

Anyway, **sparsity knowledge is often necessary to ML practitioners**. ML applications can be seen as a pipeline: exploratory data analysis (to choose an appropriate ML model), ML training, and model deployment. Data analysis on sparse dataset necessarily includes a sparsity estimation; often essential to choose an appropriate model. Hence, the exploratory data analysis would provide this public knowledge, before the sparse matrix multiplications.

Finally, Sec. 6 proposes some **protocols to minimize the public knowledge** and Sec. 7 proposes protocols to obtain the minimized knowledge in a privacy-preserving manner.

4 Secure sparse multiplications

This section presents two secure sparse matrix multiplications: matrix-vector and matrix-matrix. Existing secure sparse multiplications are incompatible with the outsourced setting, so we can only compare to the dense baselines. We assume the bit length and number of computation parties are $O(1)$, so our asymptotic complexities are only in function of the matrix size.

Our algorithms rely on secure additions, multiplications, comparisons, oblivious shuffling, and oblivious sorting [19]. On the one hand, sorting-network-based oblivious sorting only requires secure additions, multiplications, and comparisons. On the other hand, we can implement oblivious shuffling using oblivious sorting: generate a random secret-shared value for each element in the input list and obliviously sort the list based on these random values. Our algorithms are then **compatible with all secret-sharing schemes** supporting basic arithmetic and comparison operations. However, for optimal performances, one should implement our algorithms using honest-majority sorting/shuffling [19, 25].

Appendix A sketches a security proof for our algorithms.

4.1 Toy protocol: vector-vector

We propose to start with a simple vector multiplication. Since dense vector multiplications have an $O(1)$ communication cost (under honest majority), we know in advance that it is unbeatable. Modern ML applications essentially involve matrix-vector and matrix-matrix multiplications, so vector multiplication is not a relevant focus. However, this *toy protocol* provides simple intuitions to understand how to build sorting-based sparse multiplications.

We want to multiply two n-dimensional sparse vectors x and y, represented as tuple lists. Alg. 1 presents our solution. First, we concatenate the two tuple lists. Note that this concatenation is trivial (and) secure because the sparse vectors x and y are lists of shares. Thus, the concatenation requires no communication; the shareholders simply concatenate locally their list of shares. Then,

Algorithm 1 Secure sparse vector multiplication

Input: $[[x]]$ and $[[y]]$ are two secret-shared sparse vectors
Public knowledge: Number of non-zero elements per vector

1: **function** SparseVectorMult($[[x]]$, $[[y]]$)
2: $[[z]] \leftarrow$ Concatenate($[[x]]$, $[[y]]$)
3: $[[z]] \leftarrow [[\text{sort}(z)]]$ ▷ Asc. sort on the non-zero coordinate.
4: $[[s]] \leftarrow 0$
5: **for** $i \leftarrow 1$ to $\text{nnz}(x) + \text{nnz}(y) - 1$ **do**
6: **[[obliv. if]]** $[[\text{coord}(z_i)]] = [[\text{coord}(z_{i+1})]]$
7: $[[s]] \leftarrow [[s]] + [[\text{val}(z_i)]] \times [[\text{val}(z_{i+1})]]$
8: **end [[obliv. if]]**
9: **end for**
10: **return** $[[s]]$ ▷ a scalar
11: **end function**

we sort (*obliviously*) the resulting list by coordinate. If two consecutive tuples have equal coordinates, we multiply their values and sum all the multiplication results to build the vector multiplication value.

We use the notation **[[obliv. if]]** to describe conditional structures on secret-shared values. The presence of conditional branches may seem contrary to the obliviousness goal. Indeed, revealing the executed branch would leak information about the secret-shared value. However, conditional structures on secret-shared values are implemented so that the executed branch remains secret. All the **[[obliv. if]]** instructions are implemented via a combination of arithmetic and comparison operations. For example, we implement the lines 6 to 8 of Alg. 1 using the following expression: $[[s]] \leftarrow [[s]] + ([[\text{coord}(z_i)]] = [[\text{coord}(z_{i+1})]]) \times [[\text{val}(z_i)]] \times [[\text{val}(z_{i+1})]]$; with the secure comparison $[[\text{coord}(z_i)]] = [[\text{coord}(z_{i+1})]]$ outputting $[[1]]$ or $[[0]]$. Such instructions are less readable, so we keep conditional structures in our algorithms with a dedicated notation. Such conditional statements are recurrent in the literature: e.g., [3] used conditional statements in their heavy hitters protocol.

Alg. 1 costs $O((\text{nnz}(x)+\text{nnz}(y))\log(\text{nnz}(x)+\text{nnz}(y)))$ communications and computations in $O(\log(\text{nnz}(x)+\text{nnz}(y)))$ rounds. The sorting step induces the superlinear complexity. The multiplication loop is parallelizable, so it only costs $O(\text{nnz}(x)+\text{nnz}(y))$ in $O(1)$ rounds. In comparison, the dense multiplication has $O(1)$ communication costs under honest majority, and an $O(n \log n)$ cost under dishonest majority. Moreover, our sparse algorithm significantly reduces the memory footprint under both threat models.

4.2 Matrix-vector

We want to multiply an $n \times m$ matrix X to an m-dimensional vector y. Our work assumes the public knowledge motivated in Sec. 3.3: per-row sparsity. Thus, we can adapt the sparse matrix representation and group the non-zero elements by rows: a sparse matrix would be a list of sparse vectors, one per row.

An intuitive solution is to compute a vector-vector multiplication on each matrix row. This approach is adequate for dense multiplications, but inefficient for sparse matrices. Indeed, this naive extension induces a linear dependency on the number of rows: $O((\text{nnz}(X)+n\cdot\text{nnz}(y))\log(\text{nnz}(X)+n\cdot\text{nnz}(y)))$ communications and computations. The term $n \cdot \text{nnz}(y)$ comes from the "replication" of y for each row of X. The vector y is sorted independently for each row, which induces this inefficiency. To sum up, we need a dedicated sparse matrix-vector multiplication because the naive extension is inefficient.

Algorithm 2 Secure sparse matrix-vector multiplication

Input: $[[X]]$ a secret-shared sparse matrix, $[[y]]$ a secret-shared sparse vector
Public knowledge: Number of non-zeros per matrix/vector
Notation: the function RowCoord() (resp. ColCoord()) returns the first/row (resp. second/column) coordinate of a non-zero tuple.

1: **function** SparseMatVectMult($[[X]]$, $[[y]]$)
2: $[[z]] \leftarrow [[X]]$
3: **for** $([[j]], [[v]])$ in $[[y]]$ **do**
4: Append tuple $(\text{coord} : ([[\perp]], [[j]]), \text{val} : [[v]])$ to $[[z]]$
5: **end for**
6: $[[z]] \leftarrow [[\text{sort}(z)]]$ ▷ Asc. sort on the 2nd and then 1st coordinates (with $\perp < 1$)
7: $\text{prev}Y \leftarrow (\text{coord} : ([[\perp]], [[\perp]]), \text{val} : [[\perp]])$
8: **for** $k \leftarrow 1$ to $(\text{nnz}(X) + \text{nnz}(y))$ **do**
9: **[[obliv. if]]** $[[\text{ColCoord}(\text{prev}Y)]] = [[\text{ColCoord}(z_k)]]$
10: $[[\text{val}(z_k)]] \leftarrow [[\text{val}(z_k)]] \times [[\text{val}(\text{prev}Y)]]$
11: **else**
12: $[[\text{val}(z_k)]] \leftarrow [[0]]$
13: **end [[obliv. if]]**
14: **[[obliv. if]]** $[[\text{RowCoord}(z_k)]] = [[\perp]]$
15: $[[\text{prev}Y]] \leftarrow [[z_k]]$
16: $[[\text{val}(z_k)]] \leftarrow [[\perp]]$
17: **end [[obliv. if]]**
18: **end for**
19: Drop the second coordinate from all tuples in $[[z]]$
20: $[[z]] \leftarrow [[\text{sort}(z)]]$ ▷ Asc. sort on the remaining coord.
21: AggEqualCoord($[[z]]$)
22: PlaceholderRemoval($[[z]]$)
23: **return** $[[z]]$ ▷ sparse vector stored as a list of non-zeros
24: **end function**

An intuitive solution is to compute a vector-vector multiplication on each matrix row. This approach is adequate for dense multiplications, but inefficient for sparse matrices. Indeed, this naive extension induces a linear dependency on the number of rows: $O((\text{nnz}(X)+n\cdot\text{nnz}(y))\log(\text{nnz}(X)+n\cdot\text{nnz}(y)))$ communications and computations. The term $n \cdot \text{nnz}(y)$ comes from the "replication" of y for each row of X. The vector y is sorted independently for each row, which induces this inefficiency. To sum up, we need a dedicated sparse matrix-vector multiplication because the naive extension is inefficient.

Alg. 2 presents a more optimized solution avoiding the linear dependency on n. The intuition behind this optimization is to build a tuple list sorted as follows $[y_j, x_{i_1,j}, x_{i_2,j}, \ldots y_{j'}, x_{i_3,j'} \ldots]$. In this list, we group the elements related to the same column, starting with the element from vector y. We then fix y_j, iterate over all $x_{*,j}$, and multiply it with y_j until we reach the next $y_{j'}$.

Once we finish the vector component multiplications, we sort the multiplication results according to the first coordinate to aggregate all the results with equal coordinates and build the result. Every time two values are aggregated, the first value is replaced by a placeholder. Alg. 3 details this aggregation step.

After the aggregation (Line 21 of Alg. 2), we remove the placeholders using Alg. 4. To avoid revealing information about the non-zero coordinates, Alg. 4 uses the "shuffle-and-reveal" trick frequently used in secure sparse sum [3, 4].

Alg. 2 costs $O((\text{nnz}(X) + \text{nnz}(y))\log(\text{nnz}(X) + \text{nnz}(y)))$ communications and computations, but it has a linear round complexity

Algorithm 3 Aggregation of tuples with equal coordinates

Input: $[\![Z]\!]$, a sorted list of secret shared tuples

1: **procedure** AggEqualCoord($[\![Z]\!]$)
2: **for** $k \leftarrow 1$ to length($[\![Z]\!]$) $- 1$ **do**
3: $[\![$**obliv. if**$]\!]$ $[\![\text{coord}(Z_k)]\!] = [\![\text{coord}(Z_{k+1})]\!]$
4: $[\![\text{coord}(Z_k)]\!] \leftarrow [\![\perp]\!]$
5: $[\![\text{val}(Z_{k+1})]\!] \leftarrow [\![\text{val}(Z_{k+1})]\!] + [\![\text{val}(Z_k)]\!]$
6: **end** $[\![$**obliv. if**$]\!]$
7: **end for**
8: **end procedure**

Algorithm 4 Placeholder removal

Input: $[\![Z]\!]$, a list of secret shared tuples

1: **procedure** PlaceholderRemoval($[\![Z]\!]$)
2: $[\![Z]\!] \leftarrow [\![\text{shuffle}(Z)]\!]$
3: **for** $k \leftarrow 1$ to length($[\![Z]\!]$) **do**
4: is_placeholder $\leftarrow$ reveal($[\![\text{coord}(Z_k)]\!] = \perp$)
5: **if** is_placeholder
6: Remove the k-th tuple from $[\![Z]\!]$
7: **end if**
8: **end for**
9: **end procedure**

Majority	Algorithm	Comm.	Comp.	Storage
Dishonest	Dense [7, 31, 32]	$O(nm)$		$O(nm)$
	Sparse (**ours**: Alg. 2)	$O(\text{nnz}_* \cdot \log(\text{nnz}_*))$		$O(\text{nnz}_*)$
Honest	Dense [22, 24, 29, 30, 41]	$O(n)$	$O(nm)$	$O(nm)$
	Sparse (**ours**: Alg. 2)	$O(\text{nnz}_* \cdot \log(\text{nnz}_*))$		$O(\text{nnz}_*)$

Table 1: Complexities of the sparse and dense matrix-vector multiplications. Notations: $\text{nnz}_* = \text{nnz}(X) + \text{nnz}(y), \text{nnz}(X) \ll n \times m, \text{nnz}(y) \ll m$**.**

due to the multiplication loop (lines 8 to 18 of Alg. 2) and the aggregation step (Alg. 3). The full version of this paper [10] also presents recursive algorithms taking inspiration from the logarithmic-round secure maximum (notably implemented in MPyC [35]) to build a multiplication loop and an aggregation step with logarithmic round complexity.

Table 1 compares the complexities of our sparse algorithm to the dense baselines. On the one hand, honest-majority dense multiplication is no longer constant in communications, so our algorithm is advantageous for highly sparse matrices. On the other hand, the memory footprint is more critical in this operation. Several real-world sparse matrices do not fit in memory using a dense format (see example application in Sec. 5), even in plaintext! **Our sparse algorithm is then mandatory when memory overflowing prevents from using dense algorithms.**

4.3 Matrix-matrix

We want to multiply two sparse matrices. Such multiplications are used in ML algorithms for specific computations, especially correlation matrices (i.e., $X^\top X$). Other ML algorithms also require sparse-dense matrix multiplication (e.g., to multiply a sparse input

Algorithm 5 Secure sparse matrix multiplication

Input: $[\![X]\!]$ and $[\![Y]\!]$ are two secret-shared sparse matrices
Public knowledge: Number of non-zeros per column in X (with $X^{(k)}$ a column), number of non-zeros per row in Y (with $Y^{\{k\}}$ a row)

1: **function** SparseMatMult($[\![X]\!]$, $[\![Y]\!]$)
2: $[\![Z]\!] \leftarrow \emptyset$
3: **for** $k \leftarrow 1$ to m **do**
4: **for** $i \leftarrow 1$ to length($[\![X^{(k)}]\!]$) **do**
5: **for** $j \leftarrow 1$ to length($[\![Y^{\{k\}}]\!]$) **do**
6: $[\![r_i]\!] \leftarrow [\![\text{coord}(X_i^{(k)})]\!]$ ▷ Row coordinate
7: $[\![r_j]\!] \leftarrow [\![\text{coord}(Y_j^{\{k\}})]\!]$ ▷ Column coordinate
8: $[\![r_v]\!] \leftarrow [\![\text{val}(X_i^{(k)})]\!] \times [\![\text{val}(Y_j^{\{k\}})]\!]$
9: Append a tuple $(([\![r_i]\!], [\![r_j]\!]), [\![r_v]\!])$ to $[\![Z]\!]$
10: // Non-zero tuple: coordinates then value
11: **end for**
12: **end for**
13: **end for**
14: $[\![Z]\!] \leftarrow [\![\text{sort}(Z)]\!]$ ▷ Asc. sort on the coordinates.
15: AggEqualCoord($[\![Z]\!]$)
16: PlaceholderRemoval($[\![Z]\!]$)
17: **return** $[\![Z]\!]$ ▷ sparse matrix stored as a list of non-zeros
18: **end function**

matrix with a dense parameter matrix). Even in plaintext, sparse-dense multiplications require specific algorithms differing from those of sparse-sparse multiplications (i.e., the focus of our paper). Thus, we leave sparse-dense multiplications for future works. This subsection then emphasizes the computation $X^\top X$, which is a key sparse-sparse operation in ML.

To compute $X^\top X$, we multiply an $m \times n$ sparse matrix to an $n \times m$ matrix. With our public knowledge, we know the per-column sparsity of the first matrix (because X is transposed), and per-row sparsity of the second. Alg. 5 generalizes this problem: multiplying an $n \times m$ sparse matrix X to an $m \times p$ sparse matrix Y, with per-column sparsity knowledge for X and a per-row knowledge for Y. We use the notation $X^{(k)}$ (resp. $Y^{\{k\}}$) to refer to the k-th column (resp. row) of X (resp. Y) stored as a sparse vector.

When implementing a matrix-matrix multiplication naively, it is common to design a triple nested loop multiplying: each row i of X is multiplied to each column j of Y. The third loop is the vector multiplication on each k-th element of the vectors. Linear algebra libraries usually leverage a simple optimization to speed up the dense matrix multiplication: swapping the loops on i and k. This optimization significantly speeds up the implementation by reducing the memory access costs [26].

Our sparse algorithm reuses a similar intuition to design a fast algorithm and avoid performing a series of vector multiplications. Our algorithms have two main steps: (1) compute all the individual scalar multiplications, (2) aggregate them using oblivious sorting. In matrix-matrix multiplications, all values in the k-th column of X must be multiplied by all values in the k-th row of Y. Our public

Majority	Algorithm	Comm.	Comp.	Storage
Honest	Dense [7, 31, 32]	$O(nm+mp+np)$	$O(nmp)$	$O(nm+mp+np)$
	Sparse (**ours**: Alg. 5)	$O(\text{MinMult} \cdot \log(\text{MinMult}))$		$O(\text{MinMult})$
Dishonest	Dense [22, 29, 30, 41]	$O(np)$	$O(nmp)$	$O(nm+mp+np)$
	Sparse (**ours**: Alg. 5)	$O(\text{MinMult} \cdot \log(\text{MinMult}))$		$O(\text{MinMult})$

Table 2: Complexities of the sparse and dense matrix-matrix multiplications. Notations: MinMult $= \sum_{k=0}^{m} \text{nnz}(X^{(k)}) \cdot \text{nnz}(Y^{\{k\}})$, $\text{nnz}(X) \ll n \times m, \text{nnz}(Y) \ll m \times p$.

knowledge provides direct access to the k-th column of X and the k-th row of Y, so we can compute the individual scalar multiplications by iterating on the non-zeros of $X^{(k)}$ and $Y^{\{k\}}$. Alg. 5 reuses Alg. 3 and 4 for aggregation and placeholder removal.

Table 2 compares our sparse algorithm to the dense algorithms. The notation MinMult $= \sum_k \text{nnz}(X^{(k)}) \cdot \text{nnz}(Y^{\{k\}})$ denotes the number of non-zero scalar multiplications required by the matrix multiplication. It represents an intuitive lower bound: the best plaintext algorithm is $O(\text{MinMult})$. Our algorithm *only adds a logarithmic factor* compared to the plaintext sparse algorithm [5].

5 Experimental analysis

This section compares the dense algorithms to our sparse algorithms under three sparsity levels: 99%, 99.9%, 99.99% of zero values. These thresholds are recurrent in sparse data applications (see real-world datasets mentioned in Sec. 3.2). The dense algorithm is, by definition, insensitive to the sparsity level, so we provide one curve for the dense algorithm and one per sparsity level for our sparse algorithm.

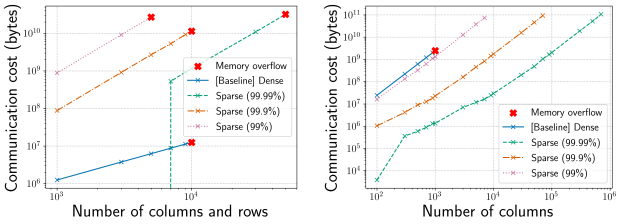
Our comparison focuses on communication costs and memory usage. The communication costs corresponds to the total communication exchanged between the servers during the protocol. The memory usage is represented thanks to the memory overflows: we scale up our experiments until the algorithms cannot be executed due to a lack of memory. We use red crosses to represent these memory overflows on the plots.

For each multiplication type (i.e., matrix-vector and matrix-matrix), we also implement an example use case built upon it. Each example use case relies on a real-world dataset to demonstrate the impracticality of dense algorithms on simple ML applications involving sparse data.

5.1 Experimental setup

We execute our experiments on a server with 188GB of RAM and an Intel Xeon CPU. We use the MPyC framework [35] to simulate 3-party protocols.

Our experiments simulate an honest majority because this setting provides the **best dense baselines** (see Tables 1 and 2). More specifically, we chose the *SSS-based matrix multiplication [22] as dense baseline* instead of the other honest-majority protocols [24, 29, 30, 41] for two reasons. First, the SSS-based matrix multiplication has smaller constant factors; e.g., it only requires the communication of a single ring element, while [24, 29, 30, 41] require multiple communication rounds. Second, the SSS-based protocol supports an arbitrary number of parties (like our sparse multiplications), while [24, 29, 30, 41] are 3-party protocols.



(a) Matrix-vector. **(b) Matrix-matrix.**

Figure 1: Comparison of the secure matrix multiplications under honest majority: dense algorithm vs. *our sparse algorithm* with three sparsity levels.

We use 64-bit fixed-point numbers, with a 32-bit fractional part. Floating-point arithmetic is still poorly scalable in MPC protocols. MPC works usually rely on fixed-point arithmetic [23] to avoid performance bottlenecks. The minor precision loss due to fixed-point arithmetic is often acceptable for ML applications [23].

We use the protocol by Laur et al. [25] for oblivious shuffling and the oblivious radix sort [19] for sorting. The first step of the oblivious radix sort is a secure bit decomposition. As this operation is expensive, we accelerate the process using pre-computations. Our algorithms assume the data owners pre-compute and share a bit decomposition of their non-zero coordinates. Hence, they share a bit array instead of a single integer per non-zero coordinate. This pre-computed bit decomposition creates a storage overhead proportional to the bit-length of the indices. As large sparse matrices hardly scale above 10^8 columns, so this factor would not exceed 30, and is compensated (as shown in our experiments) by the gains provided by sparse data representations.

5.2 Matrix-vector

Fig. 1a compares the dense and sparse matrix-vector multiplications for a square matrix. Note that our algorithm with 99.99% sparsity has a negligible cost when the number of rows is small because the high sparsity leads to empty sparse vectors. Overall, the dense algorithm is less expensive than our algorithm.

The main takeaway is the memory issue. Indeed, while the dense algorithm is less expensive, our algorithm supports larger matrices (i.e., the first motivation of sparse algorithms). For example, the dense algorithm (contrary to our sparse algorithm with 99.99% sparsity) triggers memory overflows for matrices with more than 10K columns. As developed in Sec. 3.2, there is no (dense) solution to avoid memory issues (19TB of memory would be needed).

One may argue that our algorithm also has some memory overflows. To analyze these results, we must remind the results from Sec. 3.2: larger datasets become sparser. Understanding this phenomenon is essential to interpret Fig. 1a: the memory overflows illustrated in this figure should not impact most real-world datasets because of their correlation between sparsity and matrix sizes.

Our figures represent the costs for fixed sparsity rates, but large matrices are unlikely to have a low sparsity rate. For example, real-world datasets with 10K columns [27] can have much more than 99% sparsity; e.g., with 10K columns, Yahoo Movie dataset

already has a 99.7% sparsity! Thus, using real-world sparse data, our algorithm should not be affected by such memory issues, because larger datasets are sparser.

More generally, any algorithm (even sparse) has memory issues when the data is large enough. Sparse algorithms simply support larger sparse matrices than dense algorithms. In practice, the correlation between sparsity and matrix size even amplifies the gains provided by sparse algorithms.

Example use case: Recommender system. Matrix-vector is a core component of some recommender systems. This example use case builds a simple recommender system in which a client sends a (secret-shared) item identifier and receives a list of similar items. We use the **Bookcrossing dataset**: contains ratings from 279K users on 340K books, resulting in 99.998% zeros in the dataset (1K users as training dataset). Each row corresponds to a user, and each column to a book.

Our recommender system relies on a **nearest neighbors** algorithm [1]. This model has no training step: the inference is computed on the training dataset, and requires two matrix-vector multiplications. This algorithm computes the similarity between the book the client chose and all the other books in the database. Then, the system returns the identifiers of the k most similar books.

The dense algorithm triggers a memory overflow because it cannot store the training data using a dense matrix. The sparse algorithm takes, in average, 48 minutes. **The dense algorithm is impractical while our sparse algorithms support this application**. Our runtime could be further reduced using MPC optimizations (e.g., precomputed comparisons [38]).

5.3 Matrix-matrix

Fig. 1b shows the communication costs for the multiplication $X^\top X$. This experiment considers a varying number of columns and a fixed number of rows (100 rows). The sparse algorithm brings a significant communication reductions compared to the dense multiplication: ×100 gain for 99.9% sparsity and a ×1000 gain for 99.99% sparsity. The cost reduction is much higher for matrix-matrix multiplication than for matrix-vector because the matrix-matrix multiplication complexity has a quadratic dependency on sparsity.

Besides the performance gain, **our algorithm supports much larger matrices**: the dense algorithm triggers a memory overflow for matrices with more than 1K, while our sparse algorithm scales up to 1M columns. Fig. 1b shows no memory overflow for our algorithm because we stopped experiments exceeding 10 hours.

Example use case: Access control. This multiplication can be used to build an ML-based access control system. This example use case trains an ML model deciding whether an access request is suspicious. Access data can be particularly privacy-sensitive, e.g., the access log of medical databases can indirectly leak patient information. As medical access logs are sensitive, there is no publicly available dataset, so we use use the **Amazon access control dataset**[1]; that should have similar properties as more sensitive datasets.

The Amazon dataset contains 32.7K samples, each containing the user's properties, the service properties, and the access decision (granted or not). Each property is a categorical value that we transform into binary arrays using the one-hot encoding. This encoded dataset has 15K features with 99.95% sparsity.

Our access control relies on **Linear Discriminant Analysis**. The training requires computing: (1) the proportion of each class (i.e., access decision), (2) the mean vector of each class, and (3) the covariance matrix of the whole dataset. The most expensive operation is, by far, the covariance matrix estimation.

The covariance matrix is necessarily dense, so we use a trick to store it efficiently. As $Cov(X) = \frac{1}{n}X^\top X - \bar{x}^\top \bar{x}$, we store the covariance in two parts: the sparse product $X^\top X$ and the dense mean vector $\bar{x}$. Covariance matrices are used at inference time via a matrix-vector multiplication (e.g., $Cov(X) \cdot y = \frac{1}{n}X^\top X \cdot y - \bar{x}^\top \bar{x} \cdot y$). Thus, this compact representation does not change the model inference algorithm and avoids a potential memory overflow due to the high-dimensional dense matrix.

We implement this application using the dense and sparse matrix multiplications. We use the first 10K samples as training dataset. As expected from our previous benchmark, the dense algorithm triggers a memory overflow during the covariance computation. The sparse algorithm completes the training algorithm in 5 hours. Hence, *our sparse algorithm provides a satisfying solution* to this problem, while the **dense algorithm is impractical**.

6 Minimizing the public knowledge

Sec. 3.3 highlighted the need for sparsity-related public knowledge in secure sparse algorithms. While necessary, this information disclosure might be problematic in some specific use cases in which revealing the exact per-row sparsity is too sensitive. For such cases, we propose three techniques to minimize the public knowledge necessary to our algorithms.

6.1 Row anonymization

By default, our algorithms assume the per-row sparsity to be public. This information means that the number of non-zeros from each data owner is public (but the position and value of the non-zeros remain secret). To enhance their privacy, the data owner can submit their shares via an anonymization layer (e.g., Tor network). Machine learning algorithms typically assume that training instances are independently and identically drawn and don't extract information from the order of the instances (rows). Permuting the rows of the input matrix hence does not substantially change the input.

Thanks to the anonymization layer, the servers would be unable to link a data owner to a number of non-zeros. This privacy enhancement also reduces the public knowledge: instead of the per-row sparsity, the distribution of the per-row sparsity is public.

This difference is subtle, but it is important for privacy: instead of individual information, a "collective" information is public. Indeed, this public knowledge provides information about the whole population, but not about an individual.

ML practitioners usually need this sparsity distribution information, before performing any ML training. Sparsity is essential information to choose an appropriate ML model. Thus, it is realistic to assume that the distribution of the per-row sparsity is public.

[1] https://github.com/pyduan/amazonaccess/tree/master/data

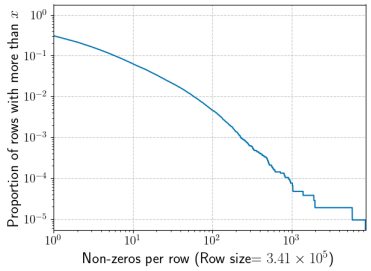


Figure 2: Per-row sparsity distribution in Bookcrossing [45]

6.2 Max-row padding

As motivated above, the per-row sparsity distribution is acceptable in practical ML applications. However, if needed, we can further reduce the public knowledge and only reveal an upper bound on the per-row sparsity.

To only assume this upper bound as public knowledge, the input sparse matrix must be padded to this maximum per-row sparsity. To pad a row, the data owner represents some of the zero elements as non-zeros anyway (i.e., dummy non-zeros), such that all rows have the pre-specified number of non-zeros. Computing the maximum per-row sparsity over all rows via MPC requires $O(n \log n)$ communications (in $O(\log n)$ rounds) for a matrix of n rows.

6.3 Matrix templating

The main drawback of max-row padding is its sensitivity to the maximum number of non-zeros per row. Sparse datasets usually contain only a few rows with many non-zeros. Most rows contain much fewer non-zeros than the maximum. For example, Fig. 2 shows the per-row sparsity distribution on the Bookcrossing dataset. While the maximum number of non-zeros per row is nearly 10^4, 99% of the rows have less than 100 non-zeros and 90% of the rows have less than 10! Thus, padding the rows until a global maximum number of non-zeros would add many unnecessary dummy non-zeros to the dataset.

Up until now, all our techniques and algorithms were oblivious to the number of rows owned by each data owner: they support all possible settings from data owners owning a single row to settings with data owners owning a sub-matrix. Note that the input size (i.e., number of rows) of each data owner is usually public in MPC. If each data owner owns a sub-matrix, we can propose a more subtle padding technique avoiding the shortcomings of max-row padding.

Instead of padding each row to a fixed maximum, we propose to build a "matrix template", a safe upper bound for the distribution of number of non-zeros per row, for which we are sure the real data matrix will fit in it. Let us call $\hat{F}(d) = \sum_{i=d}^{\infty} \hat{f}(i)$ the number of rows in the real dataset having d non-zeros or more. Let us describe what we call a template for a matrix Y: the template divides Y (horizontally) into K sub-matrices: $Y_1, \ldots, Y_K$, with respectively $n_1, \ldots, n_K$ rows. Each row in Y_k must have $\overline{\text{nnz}_k}$ non-zero elements, and we have $\overline{\text{nnz}_1} \leq \overline{\text{nnz}_2} \leq \ldots \overline{\text{nnz}_K}$. A matrix template is then defined by $T = \{(n_1, \overline{\text{nnz}_1}), \ldots, (n_K, \overline{\text{nnz}_K})\}$. A real dataset will fit in the template (after a permutation of the rows, see Sec. 6.2) if $\forall i \in [K] : \hat{F}(\overline{\text{nnz}_i} + 1) \leq \sum_{j=i+1}^{K} n_j$.

To set the thresholds $\{\overline{\text{nnz}_1}, \ldots, \overline{\text{nnz}_K}\}$, we propose to consider the quantiles: 0.25, 0.5, 0.75, 0.9, 0.99, 1, because they capture nicely the power-law distribution of sparse data. In other words, $\hat{F}(\overline{\text{nnz}_1}) = 0.25$, $\hat{F}(\overline{\text{nnz}_2}) = 0.5, \ldots, \hat{F}(\overline{\text{nnz}_K}) = 1$.

A data owner P owning a dataset Y can fit Y into a template T by padding Y according to T. First, they sort (increasingly) the rows of Y based on their number of non-zeros. Remember that the order of the rows does not matter in ML applications. Second, they pad the first n_1 rows of Y to have $\overline{\text{nnz}_1}$ non-zeros. They repeat this padding process for the next n_2 rows with $\overline{\text{nnz}_2}$ non-zeros, etc. until the last n_K rows with $\overline{\text{nnz}_K}$ non-zeros. Thanks to the sub-matrices, only Y_K is padded to have the maximum number of non-zeros, the rest of the rows can have much fewer non-zeros. Note that matrix templating with $K = 1$ has the same output as max-row padding. Similarly, the higher K is, the smaller the overhead because the matrix template becomes more accurate.

We envision four privacy-preserving approaches to obtain such matrix templates. First, one may know the population distribution, i.e., the probability $f(d)$ of drawing a row with d non-zeros in the process generating the data. Second, without such domain knowledge one may have seen a public dataset D from which one could draw statistics and approximate the population distribution by inferring $\hat{F}_D(d) \geq F(d)$, before one has to process the sensitive data. This typically leads to quite accurate results. Third, one could start from the real data provided as input and use differential privacy to release safe upper bounds $\hat{F}_{DP}(d) \geq \hat{F}(d)$. This strategy leading to looser bounds to protect privacy. Fourth, we can compute it using MPC-based quantile estimation.

Sec. 7 details these private estimations of the matrix template.

6.4 Overhead comparison

Row padding and matrix templating induce some overhead due to the dummy non-zeros they add. Fig. 3 compares the memory footprint of several alternatives: dense multiplication, no knowledge minimization, row anonymization, max-row padding, and matrix templating (using quantiles as in Sec. 6.3). The memory footprint is a simple proxy to understand the impact of knowledge minimization techniques on our algorithms' related communication and computation costs (see also our tables in Sec. 4). We base the comparison on four real-world datasets: SMS Spam [18], Amazon Access Control, Bookcrossing [45], and MovieLens [20]. For the matrix templating, our experiments simulate the MPC-based estimation with 20 data owners.

On the one hand, (as expected) row anonymization has no effect on the memory footprint. On the other hand, padding rows to a global maximum induces a small overhead on the Spam and the Access control datasets, while it induces significant overhead on the Bookcrossing and MovieLens datasets. The overhead on Bookcrossing and MovieLens makes the sparse matrix nearly as large as the dense matrix... in such case, the sparse algorithms would have no interest.

Our matrix templating significantly reduces this overhead. For example, on MovieLens, adding dummy non-zeros via matrix templating only doubles the memory footprint (vs. nearly ×100 for max-row padding). As mentioned earlier, the overhead could be even further reduced if we reveal more than 5 quantiles.

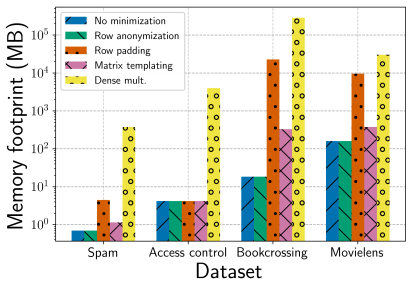


Figure 3: Matrix storage costs for different techniques of public knowledge minimization on four real-world datasets

Finally, on the Access Control dataset, all knowledge minimization techniques have an equal memory footprint. In this dataset, all rows have *by design* the same number of non-zeros (i.e., naturally padded), so max-row padding and matrix templating does not need to add any dummy non-zero.

7 Private estimation of the public knowledge

While matrix templates minimize the necessary prior knowledge, we can further enhance privacy by estimating it using privacy-preserving techniques. This section details how to estimate privately matrix templates based on MPC and on Differential Privacy. Moreover, Appendix B shows how to estimate them based on the population distribution.

7.1 MPC-based

First, each data owner i shares a value $[[\text{nnz}_j]]$ for each row j they own. Then, the servers compute securely the quantiles on the list $[[[\text{nnz}_1]], \ldots, [[\text{nnz}_n]]]$. To compute them, the servers simply need to sort obliviously the list and to obtain the elements on indices: $\lfloor 0.25n \rfloor$, $\lfloor 0.5n \rfloor$, $\lfloor 0.75n \rfloor$, $\lfloor 0.9n \rfloor$, $\lfloor 0.99n \rfloor$, n.

These quantiles provide an approximate template: some submatrices may not fit perfectly into it because each data owner has a slightly different row sparsity. For example, data owner i may have 50% of rows with less than 13 non-zeros while data owner j has 50% with less than 17 non-zeros. If the MPC protocol outputs 15 as quantile 0.5, data owner i can pad its rows, but data owner j has to remove some non-zeros to fit the template.

To finalize our template, we compute a scaling factor ensuring that all sub-matrices fit in it. Let $\{\widehat{\text{nnz}_1} \ldots \widehat{\text{nnz}_K}\}$ refer to the parameters from the previous approximate template. Based on the imperfect template, each data owner i computes a scaling factor $\alpha_i \geq 1$ such that: for any row X_j owned by data owner i, $\text{nnz}_j \leq \alpha_i \widehat{\text{nnz}_k}$ ($\widehat{\text{nnz}_k}$ being the padding parameter corresponding to X_j in the template). Then, the data owners share their respective $[[\alpha_i]]$ and the servers compute $\overline{\alpha}$ using a secure maximum protocol on these secret-shared values. We can now deduce the final templates: $\overline{\text{nnz}_k} = \overline{\alpha}\widehat{\text{nnz}_k}$, for $k \in \{1 \ldots K\}$.

Based the values $\{\overline{\text{nnz}_1}, \ldots, \overline{\text{nnz}_K}\}$ and n_i number of rows by data owner i (usually public in MPC), we can infer the dimensions of their padded matrix: $\overline{\text{nnz}_1}$ non-zeros in their first $\lfloor 0.25n_i \rfloor$ rows, $\overline{\text{nnz}_2}$ in their next $\lfloor (0.5 - 0.25)n_i \rfloor$ rows, ..., and $\overline{\text{nnz}_6}$ non-zeros in their last $\lfloor (1 - 0.99)n_i \rfloor$ rows.

Using matrix templating, the public knowledge is then reduced to a few quantiles from the per-row sparsity distribution.

7.2 Based on Differential Privacy

Let $[n]$ be the set of first n integers. Let $X = \{x_i\}_{i=1}^n$ be a set of numbers identically and independently drawn from an unknown but fixed probability distribution $\mathcal{D}_X$. Let $\hat{f}(X,t) = |\{i \in [n] \mid x_i = t\}|$ be the empirical probability distribution (histogram) of X and $\hat{F}(X,t) = |\{i \in [n] \mid x_i \geq t\}|$ be the empirical cumulative distribution function.

Consider the PPML setting where each data owner has exactly one subset $X^{(j)}$, with $\mathcal{X} = \cup_{j=1}^k X^{(j)}$. Then, the data owners can compute a secret share of $\hat{F}(X,t)$ by doing computations linear in n to first locally compute the partial counts $\hat{F}(X^{(j)},t)$ and then summing these partial counts by MPC.

If the numbers in X are sensitive, revealing $\hat{F}(X,t)$ for some values of t may be undesirable. A classic approach is to use differential privacy. In particular, two sets X and X' are adjacent if the sets differ in only one element. Next, a mechanism $\mathcal{M}$ is ϵ-differentially private (DP) if for all pairs of adjacent X and X', and for all possible subsets Y of the output space of $\mathcal{M}$ there holds that

$$\left|\log\left(\frac{P(\mathcal{M}(X) \subseteq Y)}{P(\mathcal{P}(X') \subseteq Y)}\right)\right| \leq \epsilon$$

In contrast to classic DP mechanisms which add zero-mean noise, we are here interested in finding a DP upper bound $U_{\hat{F}}$ of $\hat{F}$.

Let us first consider finding a DP upper bound to a single number $y = \hat{F}(X,t)$. We define $\mathcal{M}^+_{\epsilon,\delta}(y) = y - \frac{1}{\epsilon}\log(2\delta) + \text{Lap}\left(\frac{1}{\epsilon}\right)$, where $\text{Lap}(b)(y) = \exp(-|y|/b)/2b$ is the Laplace distribution.

Lemma 7.1. *$\mathcal{M}^+_{\epsilon,\delta}$ is ϵ-DP and with probability $1-\delta$ there holds $y \leq \mathcal{M}^+_{\epsilon,\delta}(y)$.*

Proof. For $\lambda > 0$, the probability that $Lap(1/\epsilon) \leq -\lambda$ is

$$\int_{-\infty}^{-\lambda} \exp(-t\epsilon)\epsilon/2 \, \mathrm{d}t = \frac{1}{2}\exp(-\lambda\epsilon)$$

Set $\lambda = -\log(2\delta)/\epsilon$, then the probability that $Lap(1/\epsilon) \leq -\lambda$ is δ. So, the probability that $\mathcal{M}^+_{\epsilon,\delta}(y) = y + \lambda + \text{Lap}(1/\epsilon) < y$ is δ. □

Next, let us consider the problem of finding a DP vector $(u_i)_{i=1}^l$ such that $\forall i \in [l] : \hat{F}(X,t_i) \leq u_i$ with high probability. A well-known result on differential privacy describing the relation between the privacy cost ϵ and the needed amount of noise for histograms and similar data structures was first introduced by Dwork [14]. The main idea is that for two adjacent datasets X and X', the sequence $(\hat{F}(X,t_i) - \hat{F}(X',t_i))_{i=1}^l$ starts with 0 or more zeros, followed by a series of 0 or more +1 or −1 values, followed by 0 or more zeros. This is a consequence of the assumption that X and X' differ in only one element, say x_T and x'_T, and hence this causes the sequence to only be different at the thresholds t_i between the smallest and the largest of x_T and x'_T. As a consequence, the series $(\hat{F}(X,t_i))_{i=1}^l$ and $(\hat{F}(X',t_i))_{i=1}^l$ correlate strongly and a strategy adding correlated noise gives favorable results (low variance and low ϵ). In particular, let us assume $l = 2^L$ is a power of 2 (else, we can just add a few

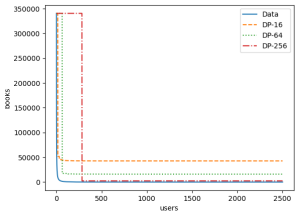


Figure 4: Distribution of row non-zero counts, and DP safe upper bounds for variable template block sizes

thresholds t_i and benefit from determining DP upper bounds to $\hat{F}(X, t_i)$ for these thresholds). For $i \in [l]$ and $j+1 \in [L]$, let $b_{i,j}$ give the bit decomposition of $i-1$ in the sense that $i-1 = \sum_{j=0}^{L-1} 2^j b_{i,j}$. Let $T = (t_i)_{i=1}^{l}$ be a set of thresholds. For $i \in [L]$ and $j+1 \in [2^{i-1}]$, let $\eta_{i,j} = Lap((L+1)/\epsilon)$ be Laplace random variable. Then, for $j+1 \in [L]$ and $k \in [2^j]$ we define

$$\mathcal{M}^{+l}_{\epsilon,\delta}(X, t_i) = \hat{F}(X, t_i) + \frac{L(L+1)}{\epsilon} \log\left(\frac{L(L+1)}{2\delta}\right) + \sum_{j=0}^{L-1} \eta_{j+1,\lfloor (i-1)/2^j \rfloor}$$

Lemma 7.2. *Let $\delta = \frac{\lambda^L e^{-\lambda}}{L!}$ The complete sequence $\left(\mathcal{M}^{+l}_{\epsilon,\delta}(X, t_i)\right)_{i=1}^{l}$ is ϵ-DP and with probability at least $1-\delta$ there holds $\forall i \in [l]$: $\hat{F}(X, t_i) \le \mathcal{M}^{+l}_{\epsilon,\delta}(X, t_i)$.*

Proof. That the sequence $\left(\mathcal{M}^{+l}_{\epsilon,\delta}(X, t_i)\right)_{i=1}^{l}$ is ϵ-DP follows directly from a result by Dwork [14, thm 4.1]. Similarly to Lemma 7.1. For the second part of the statement, we need to analyze the probability that $\sum_{j=0}^{L-1} \eta_{j+1,\lfloor (i-1)/2^j \rfloor} \ge -\frac{L(L+1)}{\epsilon} \log\left(\frac{L(L+1)}{2\delta}\right)$.

The lefthand side is a sum of L independent Laplace distributions $Lap((L+1)/\epsilon)$, we have $\Pr\left(Lap\left(\frac{L+1}{\epsilon}\right) \le -\frac{L+1}{\epsilon} \log\left(\frac{L(L+1)}{2\delta}\right)\right) \le \frac{\delta}{L}$. So, the probability that this equality holds for all L terms is δ. □

Fig. 4 plots for the Bookcrossing dataset the real cumulative distribution of non-zeros per row; gives how many books were rated by x or more users (for any x). As can be seen, a few books were rated by a large number of users and most books were only rated by a few users. The figure also shows several curves where DP statistics of the empirical cumulative distribution function were used to publish upper bounds for the number of non-zero slots needed per row. The several curves use a different number of rows per block in the template matrix (the quantiles discussed in Sec. 6). In reality it is likely for most datasets (and also this one) we could perform better by taking smaller blocks for small x and larger blocks for larger x.

We used 0.01-differential privacy in Fig. 4, and the upper bounds are guaranteed to give sufficient space in the template for the data with probability at least $1 - 10^6$. In case the data owners would in the end send rows not fitting in the template, the servers would need to abort the protocol and restart. In practice, machine learning algorithms usually use values for ϵ larger than 0.01.

Revealing in a DP way more quantiles requires adding more noise, which in turns results in a higher upper bound. Too many template partitions (and hence quantiles) will unnecessarily closely model the data and suffer from the differential privacy protection, while too few quantiles will insufficiently take into account the decrease of the function. In contrast to Fig. 2 for the same dataset, Fig. 4 uses a linear scale, making the memory footprint more visual. In particular, the memory footprint is the area under the curve of a certain strategy. It seems that in this case blocks of 64 rows is the best among the three options shown.

8 Conclusion

Our paper introduced algorithms based on oblivious sorting to multiply secret-shared sparse matrices. Our algorithms are **compatible with a more generic setting** (i.e., outsourced MPC) than existing MPC works on sparse multiplications. Our algorithms **avoid memory issues** present in dense secure multiplications by leveraging the data sparsity. Our experiments show **communication cost reductions** up to ×1000 compared to dense matrix multiplications. We also implemented two real-world ML applications highlighting that **our sparse algorithms support applications impractical with existing protocols**. Finally, we proposed methods to **minimize the public knowledge** (mandatory in any secure sparse algorithm) and obtain it in a privacy-preserving manner.

Our implementation is publicly available and open-source, so our algorithms can be transferred into MPC frameworks: https://github.com/MarcT0K/Secure-sparse-multiplications

Acknowledgments

This work was supported by the Netherlands Organization for Scientific Research (NWO) in the context of the SHARE project [CS.011], the ANR PMR project (ANR-20-CE23-0013), the ANR REDEEM project (ANR-23-PEIA-005), and the Horizon Europe project HE TRUMPET.

References

[1] Charu C. Aggarwal. 2016. *Recommender Systems.* Springer International Publishing. doi:10.1007/978-3-319-29659-3

[2] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security.* ACM. doi:10.1145/3460120.3484560

[3] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security.* doi:10.1145/3548606.3560691

[4] James Bell, Adrià Gascón, Badih Ghazi, Ravi Kumar, Pasin Manurangsi, Mariana Raykova, and Phillipp Schoppmann. 2022. Distributed, Private, Sparse Histograms in the Two-Server Model. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security.* doi:10.1145/3548606.3559383

[5] Aydın Buluç, John Gilbert, and Viral B. Shah. 2011. Implementing Sparse Matrices for Graph Algorithms. In *Graph Algorithms in the Language of Linear Algebra.* Society for Industrial and Applied Mathematics. doi:10.1137/1.9780898719918.ch13

[6] Chaochao Chen, Jun Zhou, Li Wang, Xibin Wu, Wenjing Fang, Jin Tan, Lei Wang, Alex X. Liu, Hao Wang, and Cheng Hong. 2021. When Homomorphic Encryption Marries Secret Sharing: Secure Large-Scale Sparse Logistic Regression and Applications in Risk Control. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining.* 2652–2662. doi:10.1145/3447548.3467210

[7] Hao Chen, Miran Kim, Ilya Razenshteyn, Dragos Rotaru, Yongsoo Song, and Sameer Wagh. 2020. Maliciously Secure Matrix Multiplication with Applications to Private Deep Learning. In *Advances in Cryptology – ASIACRYPT 2020.* Springer International Publishing. doi:10.1007/978-3-030-64840-4_2

[8] D. Coppersmith and S. Winograd. 1987. Matrix multiplication via arithmetic progressions. In *Proceedings of the nineteenth annual ACM symposium on Theory*

of computing. doi:10.1145/28395.28396
[9] Jinming Cui, Chaochao Chen, Lingjuan Lyu, Carl Yang, and Wang Li. 2021. Exploiting Data Sparsity in Secure Cross-Platform Social Recommendation. *Advances in Neural Information Processing Systems* 34 (2021).
[10] Marc Damie, Florian Hahn, Andreas Peter, and Jan Ramon. 2025. Secure Sparse Matrix Multiplications and their Applications to Privacy-Preserving Machine Learning. arXiv:2510.14894 [cs] doi:10.48550/arXiv.2510.14894
[11] I.S. Duff. 1977. A survey of sparse matrix research. *Proc. IEEE* 65, 4 (1977), 500–535. doi:10.1109/PROC.1977.10514
[12] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. 2002. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Trans. Math. Software* 28, 2 (2002), 239–267. doi:10.1145/567806.567810
[13] Jean-Guillaume Dumas, Pascal Lafourcade, Julio Lopez Fenner, David Lucas, Jean-Baptiste Orfila, Clément Pernet, and Maxime Puys. 2019. Secure Multiparty Matrix Multiplication Based on Strassen-Winograd Algorithm. In *Advances in Information and Computer Security*. doi:10.1007/978-3-030-26834-3_5
[14] Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N. Rothblum. 2010. Differential privacy under continual observation. In *Proceedings of the 42nd ACM symposium on Theory of computing - STOC '10*. ACM Press, 715. doi:10.1145/1806689.1806787
[15] Maximilian Egger, Marvin Xhemrishi, Antonia Wachter-Zeh, and Rawad Bitar. 2023. Sparse and Private Distributed Matrix Multiplication with Straggler Tolerance.
[16] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation. *Foundations and Trends® in Privacy and Security* 2, 2-3 (2018), 70–246. doi:10.1561/3300000019
[17] Oded Goldreich. 2009. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press.
[18] José María Gómez Hidalgo, Guillermo Cajigas Bringas, Enrique Puertas Sánz, and Francisco Carrero García. 2006. Content based SMS spam filtering. In *Proceedings of the 2006 ACM symposium on Document engineering*. doi:10.1145/1166160.1166191
[19] Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2014. Oblivious Radix Sort: An Efficient Sorting Algorithm for Practical Secure Multi-party Computation.
[20] F. Maxwell Harper and Joseph A. Konstan. 2016. The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems* 5, 4 (2016), 1–19. doi:10.1145/2827872
[21] Aditya Hegde, Helen Möllering, Thomas Schneider, and Hossein Yalame. 2021. SoK: Efficient Privacy-preserving Clustering. *Proceedings on Privacy Enhancing Technologies* 2021, 4 (2021), 225–248. doi:10.2478/popets-2021-0068
[22] de Hoogh, S.J.A. 2012. *Design of large scale applications of secure multiparty computation : secure linear programming*. PhD Thesis. Technische Universiteit Eindhoven. doi:10.6100/IR735328 ISBN: 9789038632032.
[23] Mahimna Kelkar, Mariana Raykova, and Karn Seth. 2022. Secure Poisson Regression. In *31st USENIX Security Symposium (USENIX Security 22)*.
[24] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. 2651–2668.
[25] Sven Laur, Jan Willemson, and Bingsheng Zhang. 2011. Round-Efficient Oblivious Database Manipulation. In *Information Security*. Springer. doi:10.1007/978-3-642-24861-0_18
[26] Charles E. Leiserson. 2018. Performance Engineering of Software Systems. Lecture 1: Introduction and Matrix Multiplication Leiserson.
[27] Xiang Li, Charles X. Ling, and Huaimin Wang. 2017. The Convergence Behavior of Naive Bayes on Large Sparse Datasets. *ACM Transactions on Knowledge Discovery from Data* 11, 1 (2017), 1–24. doi:10.1145/2948068
[28] Donghang Lu and Aniket Kate. 2023. RPM: Robust Anonymity at Scale. *Proceedings on Privacy Enhancing Technologies* 2023, 2 (2023). doi:10.56553/popets-2023-0057
[29] Tianpei Lu, Bingsheng Zhang, Lichun Li, and Kui Ren. 2023. Aegis: A Lightning Fast Privacy-preserving Machine Learning Platform against Malicious Adversaries.
[30] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 35–52. doi:10.1145/3243734.3243760
[31] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*. 19–38. doi:10.1109/SP.2017.12
[32] Johannes Mono and Tim Güneysu. 2023. Implementing and Optimizing Matrix Triples with Homomorphic Encryption. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. doi:10.1145/3579856.3590344
[33] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *2015 IEEE Symposium on Security and Privacy*. 377–394. doi:10.1109/SP.2015.30
[34] Arpita Patra and Ajith Suresh. 2020. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. In *Proceedings 2020 Network and Distributed System Security Symposium*. Internet Society. doi:10.14722/ndss.2020.24202
[35] Berry Schoenmakers. 2018. MPyC—Python package for secure multiparty computation. In *Workshop on the Theory and Practice of MPC*.
[36] Phillipp Schoppmann, Adrià Gascón, Mariana Raykova, and Benny Pinkas. 2019. Make Some ROOM for the Zeros: Data Sparsity in Secure Distributed Machine Learning. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1335–1350. doi:10.1145/3319535.3339816
[37] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613. doi:10.1145/359168.359176
[38] Shuang Sun and Eleftheria Makri. 2025. SoK: Multiparty Computation in the Preprocessing Model. https://eprint.iacr.org/2025/060 Publication info: Preprint..
[39] Tomas Toft. 2007. *Primitives and applications for multi-party computation*. Ph. D. Dissertation. University of Aarhus. Publisher: Citeseer.
[40] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 26–49.
[41] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. 2021. Falcon: Honest-Majority Maliciously Secure Framework for Private Deep Learning. *Proceedings on Privacy Enhancing Technologies* (2021). https://petsymposium.org/popets/2021/popets-2021-0011.php
[42] Marvin Xhemrishi, Rawad Bitar, and Antonia Wachter-Zeh. 2022. Distributed Matrix-Vector Multiplication with Sparsity and Privacy Guarantees. In *2022 IEEE International Symposium on Information Theory (ISIT)*. doi:10.1109/ISIT50566.2022.9834805
[43] Shuangqing Xu, Yifeng Zheng, and Zhongyun Hua. 2025. Harnessing Sparsification in Federated Learning: A Secure, Efficient, and Differentially Private Realization. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25)*.
[44] Jiping Yu, Kun Chen, Xiaoyu Fan, Yunyi Chen, Xiaowei Zhu, and Wenguang Chen. 2025. Lodia: Towards Optimal Sparse Matrix-Vector Multiplication for Batched Fully Homomorphic Encryption. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25)*.
[45] Cai-Nicolas Ziegler, Sean M. McNee, Joseph A. Konstan, and Georg Lausen. 2005. Improving recommendation lists through topic diversification. In *Proceedings of the 14th international conference on World Wide Web*. doi:10.1145/1060745.1060754

A Security proof sketch

This appendix sketches the security proof of our algorithms. We provide proof sketches in the honest-but-curious model using the real-ideal paradigm [16].

We can define the ideal functionality for the sparse vector multiplication $\mathcal{F}^{\eta}_{\mathsf{VectMult}}$ as follows:

- Input: $[\![x_a]\!] = \{([\![i_{1,a}]\!], [\![v_{1,a}]\!]), \ldots ([\![i_{\eta,a}]\!], [\![v_{\eta,a}]\!])\}$ and $[\![x_b]\!]$ (defined similarly)
- Output: a secret-shared scalar $[\![s]\!]$ s.t. $s = \sum_{j,k\in\{1\ldots\eta\}} v_{j,a} \cdot v_{k,b} \cdot \delta(i_{j,a}, i_{k,b})$ (with δ the Kroenecker delta).

This functionality has a property η. This property represents the protocol public knowledge in our functionality. As motivated in Sec. 3.3, public knowledge is necessary to have efficient secure sparse multiplications.

We sketch a proof showing that Alg. 1 securely realizes the ideal functionality $\mathcal{F}^{\eta}_{\mathsf{VectMult}}$. Our proof relies essentially on the composition theorem for the honest-but-curious model (Theorem 7.3.3 of [17]). Certain subprotocols we use have been proven to be secure and correct in the honest-but-curious model. By using the composition theorem, we can rely on their security and only prove the security of our overall protocol in which these subprotocols are embedded.

The correctness of Alg. 1 can be verified by an easy arithmetic exercise.

Security of the sub-protocols The first statement in Alg. 1 is a concatenation that is performed locally with no communication and hence does not need to be simulated. The second statement is an oblivious sort. The proof depends on the sub-protocol chosen for instantiation. This choice is highly influenced by the

threat model, so we consider the Batcher's sort for dishonest majority and the oblivious radix sort for honest majority (i.e., the two threat models studied in our paper). On the one hand, the Batcher's sort only requires comparisons and arithmetic operations. It is then secure because the arithmetic and comparison operations on secret-shared values are trivially secure in the honest-but-curious model [22]. On the other hand, the oblivious radix sort is proven secure in [19]. After the sorting statement, we have a loop. This loop statement reveals no information because it uses η, a property of $\mathcal{F}^{\eta}_{\text{VectMult}}$ (i.e., a public value). Finally, the loop contains a conditional statement. As highlighted in Sec. 4, we use the notation [[**obliv. if**]] to maximize the readability. This conditional statement would be implemented as follows: $[[s]] \leftarrow [[s]] + ([[\text{coord}(z_i)]] = [[\text{coord}(z_{i+1})]]) \times [[\text{val}(z_i)]] \times [[\text{val}(z_{i+1})]]$, with $[[\text{coord}(z_i)]] = [[\text{coord}(z_{i+1})]]$ outputting [[1]] or [[0]]. The arithmetic operations are trivially secure in the honest-but-curious model. For the secure equality, we can use the secure protocol proposed by Toft [39]. Hence, this oblivious conditional statement is secure because it relies on secure comparisons and arithmetic operations. Such oblivious conditional statements are recurrent in MPC protocols (e.g., in private heavy hitters [3]).

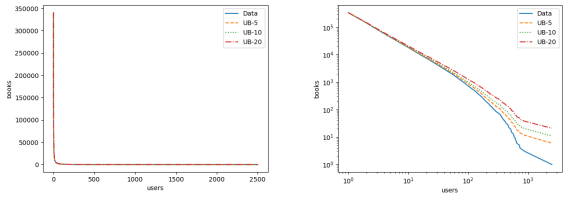
Security of the overall protocol. Since all the sub-protocols are individually secure, according to the Composition Theorem, Alg. 1 preserves the input secrecy in the honest-but-curious model.

Matrix-vector and matrix-matrix. We do not develop the proof sketch for our sparse matrix-vector (Alg. 2) and matrix-matrix (Alg. 5) multiplications, but the proof is very similar. As our sparse vector multiplication, these two algorithms use oblivious sorting, comparisons, arithmetic operations, and oblivious conditional structures. For all these operations, the security would be proven as sketched for the vector multiplication. However, they also use two additional sub-protocols: AggEqualCoord and PlaceholderRemoval.

The AggEqualCoord relies on an oblivious conditional structure and arithmetic operations. The proof is then trivial, because we can express the conditional structure using a combination of comparisons and arithmetic operators.

The PlaceholderRemoval uses a "shuffle-and-reveal" trick: shuffle a list of secret-shared values and reveal which of them are placeholders. Other security papers [19] used similar "shuffle-and-reveal" tricks in protocols with proven security. In our algorithms, revealing the number of placeholders indirectly reveals the number of non-zero elements in the output matrix. However, this information is public by definition because it is contained in the output.

Extension to malicious security. Our security proofs hold in the honest-but-curious model. Malicious security would require a verifiable secret-sharing scheme and dedicated proofs for the security in the malicious model. We remind that our main sub-protocols have known maliciously secure variant (especially oblivious sorting [3, 19] and oblivious shuffling [3, 25]) that would be the basis of a malicious security proof. We leave this extension for future work.



(a) Linear scale **(b) Logarithmic scale**

Figure 5: Distribution of row non-zero counts, and upper bounds based on knowledge of the population distribution

B Matrix template estimation based on the population distribution

This appendix considers the situation where either the population distribution of the non-zero counts of rows is known publicly, or one has a public sample from which one can compute statistics.

Assume that we know the population distribution f where $f(d)$ is the probability that a row has d non-zeros, and $F(d)$ is the probability that a row has d or more non-zeros. In that case, drawing rows identically and independently (as is needed for a sound statistical analysis) implies that we expect in a sample on average a fraction $F(d)$ of rows to require d or more non-zeros, and the standard deviation on this estimate is $\sqrt{F(d)(1-F(d))/n}$ where n is the total number of rows. We can from this compute the probability $\delta(\lambda)$ that the real fraction of rows with more than d non-zeros is larger than $F(d) + \lambda$ for $\lambda > 0$.

In case one doesn't know the population distribution but has a sample from which one can estimate these statistics, a similar approach can be followed applying the appropriate formulas, e.g., the standard deviation of a sample of size s is $\sqrt{p(1-p)/(s-1)}$ with p the measured probability.

Fig. 5a plots the number of non-zeros in the template matrix for every row in a way similar to Fig. 4 for the Bookcrossing dataset. As there is no differential privacy cost, we can take the size of a template block to be a single row. The several curves use 5, 10 and 20 standard deviations, i.e., going up to a high level of confidence that the real data will fit in the template matrix if the data is drawn according to the population distribution (or the same distributon as the data from which statistics were taken). In the plot, all curves coincide because much less margin is needed compared to the differential privacy based strategy. To see the difference better, the same plot is shown in Fig. 5b in logarithmic scale.

C Adapting existing secure sparse multiplication to the outsourced setting

Our work introduces new protocols to perform secure sparse multiplications in an outsourced setting. While we introduce the first outsourced secure sparse multiplications, existing works [6, 9, 36] had already described secure protocols, but in a non-outsourced setting. Instead of introducing new protocols, one may wonder whether the existing protocols can be adapted to the outsourced setting. This appendix details why such an adaptation of [6, 9, 36] is not straightforward and can result in inefficient constructions.

Adapting [6, 9]. These works described sparse-dense matrix multiplications where one party P_1 knows the plaintext sparse matrix X and the other party P_2 knows the plaintext dense matrix Y. In both protocols, the dense matrix Y is homomorphically encrypted and the first party P_1 performs a dense multiplication between their plaintext sparse matrix X and the encrypted dense matrix Y it receives from P_2. In [6], the encrypted matrix Y is entirely transmitted to the first party. In [9], the parties use a private information retrieval to avoid transmitting the whole matrix.

Despite their differences, these two protocols cannot be adapted to the outsourced setting for the same reasons. First, the homomorphic encryption step is complex in an outsourced setting because there is no evident key dealer (contrary to the two-data-owner setup). Second, these protocols are possible because P_1 knows the position of all non-zero values in X. Indeed, P_1 performs a multiplication between the plaintext sparse matrix and the encrypted dense matrix. This operation is possible because homomorphically encrypted values can be trivially multiplied by plaintext constants. If these non-zero positions are hidden, this simple multiplication is no longer possible. There is then no straightforward solution to adapt the approach of [6, 9] to the outsourced setting.

Third, these protocols only support sparse-dense multiplications: the matrix Y **must be dense**, or at least be converted into a dense representation before being homomorphically encrypted. Otherwise, P_1 may learn which values of Y are zero. As a result, the memory needed to store Y could blow up with several orders of magnitude, and so could the communication and computation costs.

Adapting Schoppmann et al. [36]. They adopted a different approach than [6, 9]: a low-level protocol called "ROOM" (Read-Only Oblivious Map) upon which they built their matrix multiplications. The ROOM protocol performs a secure look-up in a secret key-value store. This protocol guarantees that neither the key-value store, nor the look-up query are revealed. They introduce three instantiations of the ROOM protocol: BasicROOM, CircuitROOM, and PolyROOM

They presented all their functionality in a two-data-owner setup with the query issued by the first party and the key-value store owned by the second. However, we can easily write these functionalities for an outsourced setup.

Out of the three ROOM instanciations, only CircuitROOM is easily adaptabled to the outsourced setting because it relies on oblivious sorting and oblivious shuffling. Since our protocols also rely on sorting and shuffling, this observation may lead the reader to two questions: (1) What are the differences between the CircuitROOM-based sparse multiplications and our protocols? (2) What are the implications of these differences on efficiency?

They use the ROOM protocol to extract a dense sub-matrix from a sparse matrix. In contrast, we use sorting and shuffling for the overall sparse matrix multiplication. Besides this key algorithmic difference, *efficient* ROOM-based protocols in the outsourced setting is not straightforward. The following paragraphs study each matrix multiplication types to identify the algorithmic and efficiency differences.

To multiply two sparse vectors x and y, Schoppmann et al. [36] first use CircuitROOM to extract a dense vector of size $\mathrm{nnz}(x)$ from y (i.e., corresponding to the non-zero values of x). This operation costs $O((\mathrm{nnz}(x)+\mathrm{nnz}(y))\log(\mathrm{nnz}(x)+\mathrm{nnz}(y)))$ communications. Their protocol then performs a dense vector multiplication between the non-zero values of x and the extracted dense vector from y. This overall communication cost is $O((\mathrm{nnz}(x)+\mathrm{nnz}(y))\log(\mathrm{nnz}(x)+\mathrm{nnz}(y)))$, as in our sparse vector multiplication (Alg. 1). However, their protocol has larger constant factors: while our protocol only requires an oblivious sort, CircuitROOM performs both an oblivious sort and oblivious shuffle. Their protocol even requires a dense vector multiplication after CircuitROOM.

To multiply a sparse matrix X with a sparse vector y, Schoppmann et al. [36] call the ROOM protocol once for each data owner. In their protocol, each data owner shares a dense sub-matrix containing its non-zero columns (i.e., a column with at least one non-zero value). Hence, their protocol call the matrix-vector multiplication (and the underlying ROOM protocol) for each matrix of non-zero columns; in other words, for each data owner. On the contrary, our protocol can group the non-zeros from all data owners and process them all at the same time.

In their paper, this design choice is not an issue because there is only two data owners. However, in an outsourced setup, each matrix row can be owned by a different data owner. Hence, the matrix multiplication would perform one vector multiplication per matrix row. Their communication cost is then $O((\mathrm{nnz}(X)+n\cdot\mathrm{nnz}(y))\log(\mathrm{nnz}(X)+\mathrm{nnz}(y)))$, which is **worse than our cost, by a linear factor** n. Indeed, Sec. 4.2 presents a non-trivial matrix-vector multiplication avoiding this linear factor (i.e., Alg. 2). Since ML datasets can contain thousands or even millions of rows (e.g., 340K in the Bookcrossing dataset), the CircuitROOM-based matrix-vector multiplication would be inefficient compared to our Alg. 2 (and even compared to the dense multiplication).

To compute a correlation matrix $X^\top X$, Schoppmann et al. [36] require, in addition to ROOM, another sub-protocol called "Scatter". Unfortunately, their protocol described in the non-outsourced setting cannot be adapted easily to an outsourced setting because it relies on oblivious transfer, a purely two-party protocol with a party having access to the plaintext secret. Hence, we cannot adapt their sparse matrix-matrix multiplication to the outsourced setting.

To sum up, the adaptation of the CircuitROOM-based sparse multiplication is either inefficient (for vector-vector and matrix-vector multiplications) or not straightforward (for matrix-matrix multiplications).

Adapting Lodia [44]. Finally, Yu et al. [44] proposed a sparse matrix multiplication based on fully homomorphic encryption (FHE); assuming that a single party owns the sparse matrix. This data owner decomposes the large sparse matrix into a multiplication of smaller and denser matrices. These smaller matrices are then encrypted and can be multiplied by a third party securely. This matrix decomposition could be used also for MPC protocols, as a sparse matrix encoding. Unfortunately, it makes a strong assumption: having a single data owner; which differs from our outsourcing setting in which each row can be owned by a different party. The matrix decomposition being a complex non-linear operation, there is no straightforward way to implement it under MPC and avoid the single-owner assumption.