# Secure Computation Protocols for Privacy-Preserving Machine Learning

Dissertation
zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von
Phillipp Schoppmann

Präsidentin der Humboldt-Universität zu Berlin
Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät
Prof. Dr. Elmar Kulke

Gutachter:
1. Prof. Dr. Björn Scheuermann
2. Dr. Adrià Gascón
3. Prof. Dr. Peter Scholl

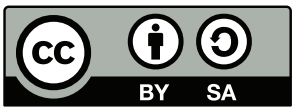
Tag der mündlichen Prüfung: 27. August 2021

# SECURE COMPUTATION PROTOCOLS FOR PRIVACY-PRESERVING MACHINE LEARNING

PHILLIPP SCHOPPMANN

Phillipp Schoppmann: *Secure Computation Protocols for Privacy-Preserving Machine Learning*





# ABSTRACT

Machine learning (ML) greatly benefits from the availability of large amounts of training data, both in terms of the number of samples, and the number of features per sample. However, aggregating more data under centralized control is not always possible, nor desirable, due to security and privacy concerns, regulation, or competition. Secure multi-party computation (MPC) protocols promise a solution to this dilemma, allowing multiple parties to train ML models on their joint datasets while provably preserving the confidentiality of the inputs. However, generic approaches to MPC result in large computation and communication overheads, which limits the applicability in practice.

The goal of this thesis is to make privacy-preserving machine learning with secure computation practical. First, we focus on two high-level applications, linear regression and document classification. We show that communication and computation overhead can be greatly reduced by identifying the costliest parts of the computation, and replacing them with sub-protocols that are tailored to the number and arrangement of parties, the data distribution, and the number representation used. One of our main findings is that exploiting sparsity in the data representation enables considerable efficiency improvements. We go on to generalize this observation, and implement a low-level data structure for sparse data, with corresponding secure access protocols. On top of this data structure, we develop several linear algebra algorithms that can be used in a wide range of applications. Finally, we turn to improving a cryptographic primitive named vector-OLE, for which we propose a novel protocol that helps speed up a wide range of secure computation tasks, within private machine learning and beyond.

Overall, our work shows that MPC indeed offers a promising avenue towards practical privacy-preserving machine learning, and the protocols we developed constitute a substantial step in that direction.

# ZUSAMMENFASSUNG

Machine Learning (ML) profitiert erheblich von der Verfügbarkeit großer Mengen an Trainingsdaten, sowohl im Bezug auf die Anzahl an Datenpunkten, als auch auf die Anzahl an Features pro Datenpunkt. Es ist allerdings oft weder möglich, noch gewollt, mehr Daten unter zentraler Kontrolle zu aggregieren. Gründe dafür können Bedenken bezüglich Sicherheit, Datenschutz, Privatsphäre, oder Konkurrenz sein, sowie gesetzliche Einschränkungen. Multi-Party-Computation (MPC)-Protokolle stellen eine Lösung dieses Dilemmas in Aussicht, indem sie es mehreren Parteien erlauben, ML-Modelle auf der Gesamtheit ihrer Daten zu trainieren, ohne die Eingabedaten preiszugeben. Generische MPC-Ansätze bringen allerdings erheblichen Mehraufwand in der Kommunikations- und Laufzeitkomplexität mit sich, wodurch sie sich nur beschränkt für den Einsatz in der Praxis eignen.

Das Ziel dieser Arbeit ist es, Privatsphäreerhaltendes Machine Learning mittels MPC praxistauglich zu machen. Zuerst fokussieren wir uns auf zwei Anwendungen, lineare Regression und Klassifikation von Dokumenten. Hier zeigen wir, dass sich der Kommunikations- und Rechenaufwand erheblich reduzieren lässt, indem die aufwändigsten Teile der Berechnung durch Sub-Protokolle ersetzt werden, welche auf die Zusammensetzung der Parteien, die Verteilung der Daten, und die Zahlendarstellung zugeschnitten sind. Insbesondere das Ausnutzen dünnbesetzter Datenrepräsentationen kann die Effizienz der Protokolle deutlich verbessern. Diese Beobachtung verallgemeinern wir anschließend durch die Entwicklung einer Datenstruktur für solch dünnbesetzte Daten, sowie dazugehöriger Zugriffsprotokolle. Aufbauend auf dieser Datenstruktur implementieren wir verschiedene Operationen der Linearen Algebra, welche in einer Vielzahl von Anwendungen genutzt werden. Zuletzt wenden wir uns einem kryptographischen Primitiven namens Vector-OLE zu. Hierfür entwickeln wir ein neues Protokoll, welches es ermöglicht, viele andere MPC-Protokolle für sicheres Machine Learning und darüber hinaus zu beschleunigen.

Insgesamt zeigt die vorliegende Arbeit, dass MPC ein vielversprechendes Werkzeug auf dem Weg zu Privatsphäre-erhaltendem Machine Learning ist, und die von uns entwickelten Protokolle stellen einen wesentlichen Schritt in diese Richtung dar.

# ACKNOWLEDGMENTS

This thesis would not have been possible if not for the many people who have helped, supported, and positively influenced me for several years. First, I want to thank my advisor Björn Scheuermann for providing the support I needed to successfully pursue my PhD, while at the same time giving me the freedom to follow my research interests wherever they would lead me. This allowed me to collaborate with researchers around the world, while spending most of my time in the beautiful city of Berlin.

My thanks also go to Adrià Gascón, my main coauthor and former MSc advisor. His curiosity and contagious enthusiasm for open problems—interdisciplinary ones in particular—have inspired me throughout my research career, and will certainly continue to do so. Adrià not only introduced me to the field of multi-party computation, but also to most of my collaborators, who rank among the best in their respective fields.

I would like to thank several of my coauthors in particular: Mariana Raykova for her help in understanding even the most complex cryptographic protocols; Borja Balle for his deep knowledge of numerics, probability theory, and machine learning, for his help with setting up experiments, and for adding a mathematician's perspective to our joint works; Benny Pinkas for his never-ending supply of great ideas and his clear way of communicating them; Kevin Yeo for his thorough and ruthless code reviews, which improved the quality of my code by orders of magnitude; and Peter Rindal for rightfully questioning my beliefs about compiler optimization and improving the running times of my code by orders of magnitude.

At HU Berlin, I had the pleasure to work in a great team with several talented colleagues over the years: Sabine Becker, Samuel Brack, Stefan Dietzel, Holger Doebler, Martin Florian, Wladislaw Gusew, Sven Hager, Sebastian Henningsen, Olga Kondrateva, Roman Naumann, Leonie Reichert, Siegmar Sommer, Hagen Sparka, Kashyap Thimmaraju, Steffen Tschirpke, Florian Tschorsch, and Frank Winkler. Thank you for your feedback in several practice talk sessions, for great discussions during lunch and coffee breaks, for first-level Arch Linux support, for maintaining our state-of-the-art computing infrastructure, and for general administrative support. I would additionally like to thank my two student assistants, Hendrik Borchert and Lennart Vogelsang, for their help preparing publications and posters, running experiments, reviewing code, and fruitful discussions. Special thanks also go to Samuel Brack, Jan Brehmer, and Jonas Marasus for giving me lots of valuable feedback on earlier versions of this dissertation.

Something that I have come to really value over the years is the academic community around privacy, security, and cryptography. Venues such as PETS, TPMPC, and CCS weren't just ways to publish and publicize my work. They also proved essential for the exchange of ideas, for identifying opportunities to collaborate, for last-minute paper sprints, but also for having a great time together outside of work, whether on a hike in Minnesota or on a pub crawl in Tel Aviv. Thank you all for reviewing my papers, listening to my talks, asking questions, inviting me to program committees, and overall for being a great community.

Finally, I want to thank my family for always being there for me, even as my visits became rarer and shorter: my parents Stefan and Katharina, my siblings Florian and Felicia, and especially my wife Lilly, who continued to support me through stressful deadline times, while at home or abroad, during multiple public lockdowns, and through life in general for the past seven years and more.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# 1 INTRODUCTION

The availability of vast amounts of data has been a major factor to the success of machine learning (ML) in the past two decades, and has enabled several advances in fields such as artificial intelligence, natural language processing, and computer vision. A common observation is that the quality of machine learning models largely depends on the amount of training data available—both in terms of the number of samples and the number of features present in the dataset. At the same time, traditional machine learning algorithms rely on centralized access to the training data; computation can be distributed to several machines in practice, but these are still controlled by a single entity. As a result, both private companies and public sector organizations have been focusing on collecting large amounts of user data, in the hope that it might be useful in the future.

Collecting more data to improve machine learning is not always possible, or desirable, though. Around the world, several laws have been put in place to limit or regulate how organizations can collect and store user data. Examples include the EU's General Data Protection Regulation (GDPR), or the California Consumer Privacy Act (CCPA). Furthermore, privacy and data protection concerns aside, organizations can be hesitant to share data with each other, out of fear of disclosing trade secrets or giving up on a competitive advantage. These observations lead to the following dilemma: On one hand, machine learners would like to train their models on more data to improve accuracy. On the other hand, aggregating more data under centralized control is often undesirable or outright infeasible.

A possible solution to this dilemma is provided by the cryptographic field of *multi-party computation* (MPC). Originally introduced in the 1980s by Yao [Yao86] and Goldreich, Micali, and Wigderson [GMW87], MPC allows two or more parties to jointly compute a function on private inputs, revealing only the result, but nothing beyond that. General-purpose MPC compilers [MNPS04; DPSZ12; DSZ15; ZE15; Büs+18] allow developers to describe functionalities as source code in a centralized fashion, which then automatically get compiled to provably secure MPC protocols.

Like machine learning, MPC has made significant progress since its inception, bringing it closer to practicality. For example, the amortized computation time of the fundamental cryptographic primitive oblivious transfer (OT) was considerably reduced by the introduction of OT extension [IKNP03], which replaces expensive public-key operations with cheap, hardware-assisted private-key operations. Im-

provements of other cryptographic primitives such as homomorphic encryption (HE) [Gen09; FV12; BGV14] likewise translated into improvements of MPC protocols relying on these. Generic MPC protocols similarly have seen several improvements. Notable example include the *free XOR* [KS08] and *half-gates* [ZRE15] optimizations to Yao's two-party garbled circuit protocol, new efficient instantiations [BLO16] of its multi-party variant [BMR90], as well as the multi-party SPDZ protocol [DPSZ12] and its enhancements [KOS16; KPR18]. Finally, optimizing MPC compilers have been proposed [Dem+15; Büs+18], which aim to minimize the communication and computation costs when compiling high-level functionality descriptions to generated MPC protocols.

Even with all these improvements in place, however, it seems unlikely that general-purpose MPC alone leads to practical protocols for private machine learning. A major obstacle is the communication overhead introduced by generic MPC protocols. For example, computing a single 32-bit integer multiplication of secret-shared numbers using the ABY framework [DSZ15] requires over 1 KiB of communication. Clearly, this does not scale to the billions of multiplications needed in many machine learning settings.

An alternative approach to generic MPC protocols are application-specific, hand-written protocols. For some concrete MPC applications, such as private set intersection (PSI), such custom protocols have consistently outperformed generic approaches [KKRT16; PSZ18; Ion+20; PRTY20; CM20]. However, unlike PSI, machine learning consists of many different functionalities with a variety of settings and data distributions. Developing, optimizing, and proving security of separate MPC protocols for each of these is a tedious task that requires expert knowledge. Therefore, smaller organizations might not be able to afford developing custom protocols for their private machine learning needs.

In this thesis, we propose a hybrid approach that conceptually lies between the two extremes outlined above. Concretely, we observe that many ML algorithms can be split up into multiple distinct phases. If we can ensure that intermediate results do not reveal any additional information, we can split up secure computation protocols in a similar way. Secret sharing and other cryptographic tools can be used to achieve such secure protocol compositions, which allows us to develop custom MPC protocols for the most expensive parts of the computation, while using generic MPC for the rest. This way, we can tailor our protocols to the *setting* (e. g., number and arrangement of parties) and the *data* (e. g., its distribution and encoding). Finally, this modular approach allows us to reuse protocols for common functionalities, such as inner products and other linear algebra operations, across different ML tasks.

We start in Chapters 3 and 4 by taking a look at two prominent ML use cases, namely *regression* and *classification*. One of our main observations here is that significant speedups can be obtained by exploiting sparsity inherent in the data representation. In Chapter 5, we generalize this observation, developing data structures and corresponding secure access protocols for sparse data. On top of these, we then develop multiple sparse linear algebra protocols, and show that these can speed up various ML applications on sparse data. Finally, in Chapter 6, we design and implement a protocol for vector oblivious linear evaluation (vector OLE or VOLE), a low-level cryptographic primitive that is equivalent to a secure scalar–vector multiplication and that can be used as a building block in several MPC applications. The implementations of all of the protocols developed in this thesis have been published under open-source licenses.

## 1.1 CONTRIBUTIONS

### 1.1.1 Linear Regression

In Chapter 3, we look at the problem of securely computing a linear regression model on *vertically partitioned* data. Linear regression tries to approximate the relationship between a set of independent variables and a dependent variable by a linear function. More formally, if we express our training dataset as a matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, where rows correspond to $n$ data points (i. e., samples), and columns correspond to $d$ independent variables (i. e., features), then linear regression aims to find a vector $\boldsymbol{\theta}$ such that $\boldsymbol{X\theta}$ approximates the vector for the dependent variable $\boldsymbol{y}$. One advantage of linear regression is the fact that it can be reduced to solving a system of linear equations $\boldsymbol{A\theta} = \boldsymbol{b}$, where $\boldsymbol{A} = \frac{1}{n}\boldsymbol{X}^\intercal\boldsymbol{X} + \lambda\boldsymbol{I}$ and $\boldsymbol{b} = \frac{1}{n}\boldsymbol{X}^\intercal\boldsymbol{y}$. This approach is also called *ridge regression* with regularization parameter $\lambda$. In prior work, Nikolaenko et al. [Nik+13b] observed that the computation of the solution $\boldsymbol{\theta}$ in MPC can be split up into two phases, by first computing secret-shares of $\boldsymbol{A}$ and $\boldsymbol{b}$, and then solving the secret-shared linear system in a second phase.

In the horizontally-partitioned setting of Nikolaenko et al. [Nik+13b], additive shares of of $\boldsymbol{X}^\intercal\boldsymbol{X}$ can be computed *locally*, since $\boldsymbol{X}^\intercal\boldsymbol{X} = \sum_{i=0}^{n} \boldsymbol{x}_i^\intercal\boldsymbol{x}_i$, and each row $\boldsymbol{x}_i$ is owned by a single party. For vertically partitioned input data, i.e., when the *columns* of $\boldsymbol{X}$ are owned by different parties, this does not work any more. However, we observe that each element of $\boldsymbol{X}^\intercal\boldsymbol{X}$ can be computed as the inner product of two columns of $\boldsymbol{X}$. Thus, we can reduce the multi-party computation of $\boldsymbol{A}$ to multiple parallel two-party computations.

In the second phase, Nikolaenko et al. [Nik+13b] solve the linear system using *Cholesky decomposition* in a two-party computation be-

[1] *This is commonly referred to as the* two-server *model.*

tween two additional, non-colluding servers[1]. While this approach theoretically returns the exact solution in time $O(d^3)$, the accuracy is limited in practice by the errors introduced by the fixed-point encoding used for real numbers. Our approach instead relies on the iterative *conjugate gradient descent* (CGD). While this algorithm also needs time $O(d^3)$ to converge in theory, we observe that the precision limit due to fixed-point encoding is reached far earlier. Thus, we can save running time by stopping CGD after fewer iterations.

Overall, our work in Chapter 3 shows that exploiting the partitioning scheme of the data, as well as its fixed-point encoding, leads to substantial speedups in practice, while maintaining the accuracy of previous work.

### 1.1.2 Document Similarity and Classification

Parametric models such as linear regression require the training data to be structured as a fixed set of independent variables, or features. However, real-world data (such as text documents) is often unstructured and doesn't come with an immediate feature representation. Furthermore, parametric models need to be re-trained whenever the dataset changes, limiting their flexibility in some settings.

In Chapter 4, we develop solutions to these issues in a secure computation setting. Concretely, we design and implement MPC protocols for securely extracting numerical features from text documents, and for computing the similarities between document vectors. We then apply these to the *k-nearest neighbors* ($k$-NN) classification algorithm. As a similarity-based, non-parametric algorithm, $k$-NN doesn't involve a training phase, but instead treats the entire dataset as the model, which allows it to handle cases where the database often changes.

For feature extraction, we use *term frequency–inverse document frequency* (TF–IDF) features, one of the most common representations used for text data. Here, a first challenge is the fact that the inverse document frequency (IDF) depends on the entire dataset. Our solution involves pre-computing *differentially private* IDF values using an MPC protocol, and releasing these private IDF values to all parties. Intuitively, differential privacy (DP) [DMNS16; DR14] guarantees that these values don't contain much information about any individual document in the dataset. Revealing private IDF values has two advantages: First, it allows parties to compute their document vectors locally, thus allowing them to extend the database with new documents. Second, TF–IDF vectors are very sparse, so they only have few non-zero values.

We exploit this sparsity in our similarity computation, where our goal is to compute additive shares of the cosine similarity of two TF–IDF vectors. By first securely removing zero values from the input vectors, we considerably reduce the time needed for a secure similarity

computation. Furthermore, our protocol allows batching multiple similarity computations, which works particularly well on natural language texts due to the distribution of words encountered there.

Composing our feature extraction and scoring protocols with a generic MPC for top-$k$ selection yields an end-to-end $k$-NN classification protocol that provides a notion of security that is a hybrid of differential privacy and MPC: Our protocol first reveals differentially private IDF values, which are subsequently used together with the raw data as inputs to an MPC protocol that reveals the classification result. We formalize this notion of *security with differentially private leakage*, and prove security of our protocol in this model.

### 1.1.3 Sparse Linear Algebra

In Chapter 5, we generalize the ideas from our sparse inner product protocol introduced in chapter 4, taking a more principled approach to secure linear algebra on sparse data. Our motivation is the fact that sparsity patterns can vary a lot across different applications. For example, only one of the vectors might be sparse, while the other is dense, and different protocols might be optimal in each case. In the more general case of matrix–vector multiplication, different sparsity patterns of the matrix can result in even more possible protocol variants.

To address these different flavors of sparsity, we first define a *read-only oblivious map* (ROOM) data structure and corresponding secure computation protocols for storing and accessing sparse data. We present multiple instantiations of this primitive with different trade-offs. Then, using ROOM as a building block, we propose protocols for basic linear algebra operations such as Gather, Scatter, and multiple variants of sparse matrix multiplication. Finally, we use these to build secure protocols for logistic regression, naive Bayes, and $k$-NN classification.

The resulting framework is flexible enough to allow users to freely combine ROOM instantiations and linear algebra protocols into high-level applications. The architecture is inspired by the sparse BLAS standard [DHP02], which similarly allows users to manipulate sparse matrices and vectors in the centralized setting, independent of the underlying storage format. As a result, future improvements or new constructions of our ROOM primitive will directly benefit the higher-level linear algebra and application protocols.

### 1.1.4 Distributed Vector OLE

As our fourth and final contribution, we take a look at a low-level cryptographic primitive, namely *oblivious linear evaluation* (OLE) over vectors, or vector OLE (VOLE). The vector OLE functionality allows

two parties to compute additive shares of a vector–scalar product over some ring $\mathcal{R}$. More formally, given two vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{R}^n$ from Party 1, and a scalar $x \in \mathcal{R}$ from Party 2, it returns the vector $\boldsymbol{w} = \boldsymbol{u}x + \boldsymbol{v}$ to Party 2. Apart from secure linear algebra with long vectors, VOLE has numerous applications in cryptography, including oblivious polynomial evaluation, private set intersection, and oblivious RAM.

Boyle et al. [BCGI18] observe that a VOLE protocol can be constructed from its random variant, where $\boldsymbol{u}, \boldsymbol{v}$, and $x$ are pseudorandom. They further present a theoretical construction for such a pseudorandom VOLE generation protocol that has *sub-linear communication* in the size $n$ of the output vectors. On a high level, their protocol works as follows: Given a *short* random correlation $\boldsymbol{c} = \boldsymbol{a}x + \boldsymbol{b} \in \mathcal{R}^k$ and a public random matrix $\boldsymbol{C} \in \mathcal{R}^{k \times n}$, they compute $\boldsymbol{u} = \boldsymbol{a}\boldsymbol{C} + \boldsymbol{\mu}$. Here, $\boldsymbol{\mu}$ is a sparse error vector with a small number of non-zero elements at random indexes. Under the *learning parity with noise* (LPN) assumption, $\boldsymbol{u}$ computed this way is pseudorandom as long as $\boldsymbol{a}$ and $\boldsymbol{\mu}$ stay secret. Now $\boldsymbol{v}$ and $\boldsymbol{w}$ can be computed as $\boldsymbol{v} = \boldsymbol{b}\boldsymbol{C} - \boldsymbol{\nu}_1$ and $\boldsymbol{w} = \boldsymbol{c}\boldsymbol{C} + \boldsymbol{\nu}_2$, where $\boldsymbol{\nu}_1, \boldsymbol{\nu}_2$ are secret-shares of $\boldsymbol{\mu}$. The remaining question is how to secret-share $\boldsymbol{\mu}$ with communication sub-linear in $n$. Here, Boyle et al. propose to use a multi-point function secret sharing (MPFSS) scheme that combines distributed point functions [GI14; DS17] and combinatorial batch codes [PSW09].

In Chapter 6, we improve their construction in several ways. First, we observe that a weaker variant of MPFSS is sufficient to compute shares of $\boldsymbol{\mu}$, since one of the parties knows the non-zero positions of $\boldsymbol{\mu}$ in the clear. We call this variant *known-index* MPFSS, and present an efficient construction for it. The fact that one party knows the non-zero indices of $\boldsymbol{\mu}$ also allows us to use cheaper *probabilistic batching* based on cuckoo hashing [PR04], which has been used in a similar fashion in the context of private information retrieval [ACLS18]. Overall, our improvements enable the first implementation of a distributed VOLE generator over general fields with sub-linear communication, and we show that it outperforms the state-of-the-art approach for $n > 2^{13}$. Our known-index MPFSS construction can further be used to reduce the communication overhead of certain types of sparse matrix-vector multiplication presented in Chapter 5.

# 2 BACKGROUND

In this chapter we begin by specifying our setting and privacy goals. We then go on to introduce notation and terminology common to all of our contributions, the most important being the definition of security used in multi-party computation protocols. We also review established MPC techniques such as secret sharing, oblivious transfer, and garbled circuits, which we will use in our protocols.

## 2.1 PRIVACY-PRESERVING MACHINE LEARNING

Given the popularity of machine learning as the method of choice for solving various challenges, it is unsurprising that a lot of work has been done towards understanding and improving the privacy guarantees of machine learning algorithms. However, as there are several possible settings[1], threat models, and formalizations of privacy, not all of these approaches are comparable.

In this thesis, we focus on the setting where multiple mutually distrustful parties want to train or evaluate a machine learning model on their joint datasets via a distributed protocol. This setting assumes that all parties know how to connect to each other, what computation should be performed, and general facts about the input data (e. g., its partitioning across the parties). Unlike cross-device federated learning [Kai+19], we do not assume a central server coordinating communication between the computing parties, and we don't handle dropouts. While important for any practical deployment of privacy-preserving ML, we also don't cover preprocessing steps such as secure record linkage [Laz+18; Sta+20].

In terms of privacy, one can distinguish between what can be learned about the inputs by each subset of parties through the distributed computation itself (input privacy), and what can be inferred from the computation result (output privacy) [Cra+19]. We focus on the former, and we therefore see attacks such as model inversion or re-identification as beyond the scope of this thesis. However, we note that input and output privacy are not mutually exclusive[2]. For example, differential privacy (DP) [DMNS16; DR14], a standard approach to output privacy, can be generalized to distributed settings [Dwo+06; BNO08; Che+19; BBGN19], and combining these results with ours can yield protocols that provide both privacy goals simultaneously. In the next section, we formalize our notion of privacy via multi-party computation.

[1] *For an overview of possible settings in distributed ML, see Craddock et al. [Cra+19] and Kairouz et al. [Kai+19].*

[2] *One example of a protocol that provides both input privacy (via MPC) and output privacy (via DP) is given in Section 4.5.2.*

## 2.2 SECURE MULTI-PARTY COMPUTATION

The goal of multi-party computation (MPC) is to enable two or more data holders to compute a function on the union of their data sets, without giving any party access to the joint dataset. All the parties should learn is their respective part of the output of the computed function, but nothing beyond that. While this goal may be easy to understand intuitively, its formalization is a bit more involved, since it is not immediately clear how to measure the amount of information that can be learned *beyond* a certain function output, in particular if we assume a computationally-bounded adversary. This is what the *simulation paradigm* [Gol04; Lin17] aims to solve.

The idea is as follows: Assume there is an *ideal functionality* that takes the inputs from all parties and returns to them their respective outputs, without revealing anything else. Clearly, if such an ideal functionality existed, it would be secure by definition. In the real protocol execution, however, the parties receive more than only their outputs, since they have to exchange messages with each other, and these messages might contain more information than the final output. So in order to prove that a certain protocol is secure, we need to prove that this *view* of each party on the protocol execution does not reveal anything that cannot already be learned from the output of the ideal functionality. This can be done by constructing a *simulator* that computes a simulated view for each party from the inputs and outputs of the ideal functionality alone. If this simulated view in the ideal, secure-by-definition setting can't be distinguished from the real view of the protocol execution, then anything that an adversary could learn by analyzing its view on the real protocol execution, could also be learned in the ideal world by using the simulator. It follows that the real protocol doesn't reveal any information beyond what is learned from the ideal functionality. The following definition formalizes this notion of simulation-based security. It was adapted from Goldreich [Gol04, Definition 7.5.1], and has previously appeared in the same form in [SVGB20].

**Definition 2.1** (Security against semi-honest adversaries)**.** *For any number of parties $n \geq 2$, a probabilistic $n$-*party functionality *is a function $\mathcal{F} : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$. Let $\mathcal{F}_i(x_1, \ldots, x_n)$ denote the $i$-th element of $\mathcal{F}(x_1, \ldots, x_n)$. For each party $i \in [n]$, $\mathcal{F}_i$ takes all parties' inputs $\bar{x} := (x_1, \ldots, x_n)$ and returns the output $\mathcal{F}_i(\bar{x})$ to Party $i$. Let $\Pi$ be a protocol for computing $\mathcal{F}$. Let $\mathsf{output}^\Pi(\bar{x})$ denote the combined output of $\Pi$. Additionally, each party has a view on the protocol execution that is denoted by $\mathsf{view}_i^\Pi(\bar{x})$ and contains Party $i$'s inputs, internal random state, and all received messages. For any subset of parties $I = \{i_1, \ldots, i_t\} \subseteq [n]$, let $x_I = (x_{i_1}, \ldots, x_{i_t})$, $\mathcal{F}_I(\bar{x}) := \big(\mathcal{F}_{i_1}(\bar{x}), \ldots, \mathcal{F}_{i_t}(\bar{x})\big)$, and $\mathsf{view}_I^\Pi(\bar{x}) := \big(I, \mathsf{view}_{i_1}^\Pi(\bar{x}), \ldots, \mathsf{view}_{i_t}^\Pi(\bar{x})\big)$.*

*Now, a* simulator $\mathcal{S}$ *is a probabilistic polynomial-time algorithm that takes as arguments the set* $I$*, the inputs* $x_{i_1}, \ldots, x_{i_t}$ *of all parties in* $I$*, and their outputs from the functionality, i.e.* $\mathcal{F}_I(\bar{x})$*. Using these,* $\mathcal{S}$ *simulates a view for the parties in* $I$*. If such a simulator exists, and for each* $I$ *satisfies*

$$\left\{\left(\mathcal{S}\left(I, x_I, \mathcal{F}_I(\bar{x})\right), \mathcal{F}(\bar{x})\right)\right\}_{\bar{x}\in(\{0,1\}^*)^n} \overset{c}{\equiv} \left\{\left(\text{view}_I^{\Pi}(\bar{x}), \text{output}^{\Pi}(\bar{x})\right)\right\}_{\bar{x}\in(\{0,1\}^*)^n} \tag{2.1}$$

*then we say that* $\Pi$ privately computes $\mathcal{F}$*, or* $\Pi$ computes $\mathcal{F}$ with security against semi-honest adversaries.

Here, $\overset{c}{\equiv}$ denotes *computational indistinguishability* as defined by Goldreich [Golo4] and Lindell [Lin17]. In the two-party case, we write $\mathcal{S}_i(x_i, \mathcal{F}(x_1, x_2))$ instead of $\mathcal{S}(\{i\}, (x_i), \mathcal{F}(x_1, x_2))$ to denote the simulator for the view of Party $i \in \{1, 2\}$.

Most of the protocols we present throughout this thesis provide security according to the above simulation-based definition[3], and we provide the ideal functionalities in addition to formal protocol descriptions. We will focus on *semi-honest* or *passive adversaries*, meaning that all parties are assumed to follow the protocol faithfully, but at the same time they try to learn as much as possible from the protocol execution. A stronger adversary model, featuring *malicious* or *active adversaries*, also allows the parties to deviate from the protocol description in arbitrary ways. While there exist generic approaches to transform passively secure protocols to actively secure ones [IPSo8; LOP11], as well as general-purpose MPC frameworks with active security [DPSZ12; WMK16; Kel20], these still come with a considerable performance penalty. Still, in some cases we extend our protocols to offer some additional protection against malicious adversaries.

[3] *A notable exception is our k-NN protocol in Section 4.6, where we allow the ideal functionality to have an additional, but quantifiable, privacy leakage.*

## 2.3 COMPOSITION AND SECRET SHARING

A desirable property of secure computation protocols is that they can be composed into higher-level protocols, while maintaining their security properties. In particular, we would like the higher-level protocol to be secure independently of how the sub-protocol is implemented, as long as it securely computes the required functionality. This can be achieved using *composition theorems*, as for example given by Canetti [Canoo]. To that end, a *hybrid model* is introduced that lies between the *ideal model* and the *real world* that we described in the previous section. This hybrid model is the same as the real world execution of the composed protocol, except that each call to a sub-protocol is replaced by an evaluation of its ideal functionality instead. Modular composition now ensures that if we can prove security of the composed protocol in this hybrid model, and if each of the sub-protocols

securely implements its functionality, then the composed protocol is secure in the real world as well.

We formally state this composition property in the following theorem. It was adapted[4] from Canetti [Canoo, Theorem 5 and Corrollary 7], where we refer the reader for a full description of the model and the proof.

[4] *The main difference between Theorem 2.2 and Canetti's Corollary 7 is that we only care about the full-threshold case, i. e., we require protocols to be secure even when only a single party is honest.*

**Theorem 2.2** (Modular composition). *Let $m, n \in \mathbb{N}$, and let $\mathcal{F}_1, \ldots, \mathcal{F}_m$ and $\mathcal{G}$ be $n$-party functionalities. Let $\Pi$ be an $n$-party protocol that privately computes $\mathcal{G}$ in the $(\mathcal{F}_1, \ldots, \mathcal{F}_m)$-hybrid model where no more than one ideal evaluation call is made at each round. Let $\rho_1, \ldots, \rho_m$ be $n$-party protocols such that $\rho_i$ privately computes $\mathcal{F}_i$, and let $\Pi^{\rho_1, \cdots, \rho_m}$ denote protocol $\Pi$ with each call to $\mathcal{F}_i$ replaced by an execution of $\rho_i$. Then $\Pi^{\rho_1, \cdots, \rho_m}$ privately computes $\mathcal{G}$.*

Composition simplifies the development of secure computation protocols by allowing the protocol and proofs to take place in the hybrid model. However, to prove security in the hybrid model, we still have to make sure that the outputs of the ideal functionalities of sub-protocols don't reveal too much information. *Secret sharing* solves this issue by making sure that outputs of sub-protocols appear random to each proper subset of parties. Only by aggregating all shares, the secret value can be recovered. Thus, a common approach to MPC is to divide the computation into multiple parts, and design secure protocols with secret-shared inputs and outputs for each of these parts. Our protocols will also follow this approach, using additive secret sharing. For a value $x$ and any number of shares $n \in \mathbb{N}$, we write $[\![x]\!]$ to indicate that a value $x$ is additively secret-shared, and $[\![x]\!]_i$ to denote the $i$-th share, i. e., $\sum_{i=1}^{n} [\![x]\!]_i = x$. In some cases, when there is exactly one share given to each party, we alternatively write $[\![x]\!]_{\mathsf{P}}$ to denote the share given to party $\mathsf{P}$.

## 2.4 OBLIVIOUS TRANSFER

As with many secure computation protocols, oblivious transfer (OT) is a fundamental building block underlying most of the constructions presented in this thesis. Introduced by Rabin [Rab81], it has been shown to be sufficient to instantiate general-purpose MPC [GMW87], and many general-purpose MPC protocols [Yao86; IPS08; KOS16] rely on it. Figure 2.1 depicts the ideal functionality for OT. The sender inputs two messages, $m_0$ and $m_1$. The receiver learns exactly one of those messages, depending on a choice bit $b$, while the sender learns nothing.

While there exist instantiations of OT with both passive and active security [NP01; CO15], they require expensive public-key operations, limiting their direct applicability in practice. Fortunately, the computational overhead can be reduced with OT extension protocols [IKNP03;

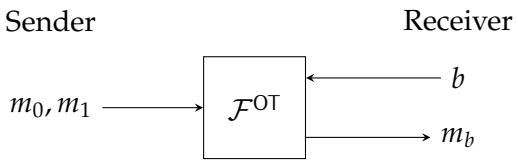



**Figure 2.1:** The ideal functionality for oblivious transfer (OT). The sender inputs two messages $m_0, m_1$, while the receiver inputs a bit $b \in \{0,1\}$ and receives the message that corresponds to $b$, $m_b$.

KOS15; ALSZ17], which allow extending a small number of base OTs using cheap private-key operations. Other optimizations such as *correlated* and *random* OT extension [ALSZ17; Yan+20] provide further speedups when the messages are correlated, or when inputs are randomly generated by the functionality instead of chosen by the parties. These improvements have made OT widespread throughout MPC implementations. Our protocols in Chapters 3–6 make use of OT extension both directly, and indirectly through general-purpose MPC such as garbled circuits.

## 2.5 GARBLED CIRCUITS

As a general-purpose MPC protocol, garbled circuits provide a way to transform any polynomial-time functionality—described by a boolean circuit—into a secure two-party computation protocol. Introduced in the 1980s by Yao [Yao86], they have seen several improvements over the years, including *free XOR* [KS08], *half-gates* [ZRE15], and, very recently, *stacked garbling* [HK20]. Compared to MPC protocols based on secret-sharing, such as GMW [GMW87] and SPDZ [DPSZ12], garbled circuits have the advantage that the number of communication rounds is constant, i. e., it does not scale with the circuit depth. This has made garbled circuits a popular basis for various MPC frameworks over the years [MNPS04; ZE15; DSZ15; WMK16]. In the following, we will describe the high-level view of the basic protocol, without any optimizations. For a full formal treatment and a security proof, see Lindell and Pinkas [LP09].

The garbled circuit protocol is executed between two parties, a *Garbler* and an *Evaluator*. The Garbler encrypts (garbles) the input circuit gate by gate, before sending it over to the Evaluator. The Evaluator then evaluates the garbled circuit using a set of *input wire keys* obtained from the Garbler via oblivious transfer. Garbling ensures that the circuit cannot be evaluated on any other inputs, and at the same time hides the Garbler's input values from the Evaluator.

| $w_1$ | $w_2$ | $w_3$ | Garbled truth table |
|---|---|---|---|
| 0 | 0 | 0 | $E_{k_1^0}(E_{k_2^0}(k_3^0))$ |
| 0 | 1 | 1 | $E_{k_1^0}(E_{k_2^1}(k_3^1))$ |
| 1 | 0 | 1 | $E_{k_1^1}(E_{k_2^0}(k_3^1))$ |
| 1 | 1 | 1 | $E_{k_1^1}(E_{k_2^1}(k_3^1))$ |

**Table 2.2:** Garbled truth table of an OR gate. Each row corresponds to a valid combination of input and output values. The fourth column is computed by the Garbler and sent to the Evaluator in randomly shuffled order.

**CIRCUIT GARBLING.** Recall that the public functionality is described by a boolean circuit, i. e., a collection of boolean *gates* and *wires* connecting them. For each wire $w_i$ of a gate, the Garbler generates two random *wire keys* $k_i^0, k_i^1$, where one represents 0, and the other represents 1. Now the goal is for the Evaluator to learn exactly one key at each gate, which is the key representing the gate's output bit. For example, given input wire keys $k_1^1$ and $k_2^0$ of an OR gate, the Evaluator should learn $k_3^1$, which corresponds to a 1-bit on the output wire $w_3$. This is achieved by means of a *garbled truth table*, where each valid combination of input-output keys forms a row of the table.

Let $(G, E, D)$ be a private-key encryption scheme with key generation $k \leftarrow G(1^\lambda)$, encryption $ct \leftarrow E_k(pt)$, and decryption $pt \leftarrow D_k(ct)$. We require that $D_k(ct)$ only returns a valid plaintext if $ct$ was in fact encrypted using $k$. See Lindell and Pinkas [LP09, Section 3.1] for formal security requirements and constructions of suitable encryption schemes. The Garbler now encrypts the output wire keys of each wire with a combination of input wire keys that is valid under the gate's truth table. In the example above, output wire key $k_3^1$ would be encrypted with input wire keys $k_1^1$ and $k_2^0$. The full garbled truth table together with the corresponding wire values is given in Table 2.2. For each gate in the input circuit, a garbled truth table is computed in the above fashion, and then shuffled randomly to ensure that no information can be learned from the order of the rows. The resulting tables are then sent to the Evaluator.

**CIRCUIT EVALUATION.** Evaluation of the garbled circuit is again performed gate-by-gate. Given the wire keys of the input wires of a gate, the Evaluator can decrypt exactly one row of the gate's garbled truth table. The decryption will yield the gate's output wire key, which serves as input to the next gate.

Before evaluation, the Evaluator needs to obtain the keys corresponding to the circuit's input wires. There are two cases. If the value on an input wire is known to the Garbler (i. e., it is part of the Garbler's input, or public), then the corresponding wire key can be just sent to

the Evaluator along with the garbled circuit. The more challenging case is when the input value is only known to the Evaluator. In this case, only one of the two possible keys should be obtained by the Evaluator, but the Garbler shouldn't know which one. This is solved using oblivious transfer, where the Garbler acts as the sender and inputs the two wire keys, and the Evaluator acts as the receiver and inputs the wire value.

Revealing the outputs is again straight-forward. For values that should be learned by the Evaluator, the Garbler reveals the mapping of output wire keys to values. For values that should be learned by the Garbler, the Evaluator sends over the output keys.

Our protocols in Chapters 3–5 all have generic MPC phases that are implemented using garbled circuits. The frameworks we use [ZE15; WMK16] abstract away from raw circuit inputs, and provide higher-level interfaces such as integer and floating point computation, and oblivious data structures such as ORAM [Zah+16; DS17]. We refer the reader to these publications for details on the implementations and the optimizations used.

# 3 SECURE LINEAR REGRESSION ON HIGH-DIMENSIONAL DATA

*The contents of this chapter have previously appeared in* Proceedings on Privacy Enhancing Technologies 2017.4 *[Gas+17].*

## 3.1 OVERVIEW

*Linear regression* is a fundamental machine learning task that fits a linear curve over a set of high-dimensional data points. It is most commonly used to predict a *dependent variable* $y$ given a set of *independent variables* (or features) $x_1, \ldots, x_d$, using a training dataset $\left(x^{(i)}, y^{(i)}\right)_{i \in [n]}$. An obstacle arises when the training data is *vertically partitioned*, i. e., when different features of the training dataset are owned by different parties. As an example, consider a study to predict medical conditions given data related to socioeconomic background. While databases holding medical, judicial, and tax records linkable by common unique identifiers (e. g., social security numbers) exist, they are often held by different institutions. Aggregating these to build a higher-dimensional dataset might be useful from a medical perspective, but is undesirable for regulatory and privacy reasons. Our goal is to enable such analyses of vertically partitioned data without the need for a trusted third party or any exposure of sensitive data, using special-purpose MPC protocols.

An important property of linear regression is that it can be cast as an optimization problem whose solution admits a closed-form expression. Formally, linear regression can be reduced to solving a system of linear equations of the form $\boldsymbol{A\theta} = \boldsymbol{b}$. Interestingly, solving systems of linear equations is a basic building block of many other machine learning algorithms [Mur12]. Thus, secure MPC solutions for solving linear systems can also be used to develop other privacy-preserving data analysis algorithms.

### 3.1.1 Chapter Contributions

In this chapter, we propose a solution for securely computing a *linear regression* model from a vertically-partitioned dataset distributed among an arbitrary number of parties. Our protocols are the result of securely composing a protocol for privately computing the coefficients of the system of equations $\boldsymbol{A\theta} = \boldsymbol{b}$ from distributed datasets (*aggregation phase*), and a protocol for securely solving linear systems (*solving phase*). Our main contributions are as follows:

- Scalable MPC protocols for linear regression on vertically partitioned datasets (Section 3.4). We design, analyze, and evaluate two hybrid MPC protocols allowing for different trade-offs be-

tween running time and the availability of external parties that facilitate the computation.

- A fast solver for high-dimensional linear systems suitable for MPC (Section 3.6). Our solver relies on a new Conjugate Gradient Descent (CGD) algorithm that scales better, both in terms of running time and accuracy, than the best previous MPC-based alternative.
- We conduct an extensive evaluation of our system on real data from the UCI repository [DG17] (Section 3.7.4). The results show that our system can produce solutions with accuracy comparable to those obtained by standard tools without privacy constraints.

Our work is also the first to undertake a formal analysis of the effect different number representations have on the accuracy of privacy-preserving regression protocols. We further propose a thorough data pre-processing pipeline to mitigate accuracy loss (Section 3.5).

As previous work we build upon [Nik+13b], we work in the two-server model that assumes the existence of two non-colluding parties to facilitate the computation. While the basic versions of our protocols are secure against semi-honest adversaries in this model, in Section 3.8 we discuss extensions beyond semi-honest security. More precisely, we extend our solving phase with a *verification phase* that provides stronger security guarantees even if one of the additional parties is *actively* corrupted, and observe that it introduces minimal overhead over the semi-honest solution.

## 3.2 RELATED WORK

Questions of privacy-preserving data mining and private computation of machine learning algorithms have been considered in several works [LP02; DA01a; KLSR04; YVJ06], providing theoretical protocols without implementations or practical evaluations of their efficiency. Early implementations of privacy-preserving linear regression protocols [DHC04; SKLR04] either don't use formal threat models, or leak additional information beyond the result of the computation.

Hall et al. [HFN11] were the first to propose a protocol for linear regression with formally defined security guarantees. However, due to the dependence on expensive homomorphic encryption, the resulting system does not scale well to large datasets.

Another implementation of linear regression is given by Bogdanov et al. [BKLS18]. They compare multiple methods for solving regression problems, including standard Conjugate Gradient Descent. However, they work in a threat model with three *computing parties* and a *honest majority* among those. Although they allow for more than three *input parties*, this means that they cannot handle the two-party case.

Furthermore, their evaluation is limited to problems with at most 10 features.

Nikolaenko et al.'s work [Nik+13b] on secure computation for linear regression is the one most similar to ours. In particular, their approach also considers a protocol split into aggregation and solving phases implemented using multiple MPC primitives. However, Nikolaenko et al. only consider the horizontally-partitioned setting while we focus on the more challenging case of vertically-partitioned datasets. Our implementation of the solving phase using CGD improves the results obtained using Nikolaenko et al.'s secure Cholesky decomposition in terms of both speed and accuracy for datasets with more than 100 features (experiments in [Nik+13b] were limited to datasets with at most 20 features). Furthermore, our CGD algorithm can also be used directly to improve other MPC protocols requiring a secure linear system solver [HFN11; GLN12].

Vertically-partitioned databases have been also studied from the perspective of privacy-preserving querying of statistical databases [DN04]. This work falls under the paradigm of differential privacy [DR14; DMNS16], where the goal is to mitigate the privacy risks incurred by revealing to an attacker the model computed by a machine learning algorithm. Our work in this chapter addresses an orthogonal concern, namely that of preserving privacy during the computation the model. Combining MPC and differential privacy is an interesting problem, however, and we will present one possible approach in Chapter 4.

## 3.3 BACKGROUND ON LINEAR REGRESSION

Linear regression is a fundamental building block of many machine learning algorithms. It produces a model given a training set of sample data points by fitting a linear curve through them. Formally, a linear model is a real-valued function on $d$-dimensional vectors of the form $\boldsymbol{x} \mapsto \langle \boldsymbol{\theta}, \boldsymbol{x} \rangle$, where $\boldsymbol{x} \in \mathbb{R}^d$ is the input vector, $\boldsymbol{\theta} \in \mathbb{R}^d$ is the vector of parameters specifying the model, and $\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle = \sum_{j=1}^{d} \theta_j x_j$ is the inner product between $\boldsymbol{\theta}$ and $\boldsymbol{x}$. The term *linear* comes from the fact that the output predicted by the function $\langle \boldsymbol{\theta}, \boldsymbol{x} \rangle$ is a linear combination of the features represented in $x$.

The learning algorithm has access to a *training set* of samples representing the input-output relation the model should try to replicate. These are denoted as $(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(n)}, y^{(n)})$ with $\boldsymbol{x}^{(i)} = (x_1^{(i)}, \ldots, x_d^{(i)}) \in \mathbb{R}^d$ and $y^{(i)} \in \mathbb{R}$ for $i \in [n]$.

A standard approach to learn the parameters of a linear model is to solve the *ridge regression* optimization:

$$\operatorname*{argmin}_{\theta} \frac{1}{n} \sum_{i=1}^{n} (y^{(i)} - \langle \boldsymbol{\theta}, \boldsymbol{x}^{(i)} \rangle)^2 + \lambda \|\boldsymbol{\theta}\|^2 \tag{3.1}$$

Here, $\lambda\|\boldsymbol{\theta}\|^2$ is known as a ridge (or Tikhonov) regularization term weighted by the regularization parameter $\lambda > 0$. When $\lambda = 0$, this optimization finds the parameter vector $\boldsymbol{\theta}$ minimizing the *mean squared error* between the predictions made by the model on the training examples $\boldsymbol{x}^{(i)}$ and the desired outputs $y^{(i)}$. The ridge penalty is used to prevent the model from overfitting when the amount of training data is too small.

The optimization problem (3.1) is convex and its solution admits a closed-form expression. A way to derive this solution is to reformulate the optimization problem by writing $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ for the matrix with $n$ rows corresponding to the different $d$-dimensional row vectors $\boldsymbol{x}^{(i)\intercal}$ so that $\boldsymbol{X}(i,j) = x_j^{(i)}$, and $\boldsymbol{y} \in \mathbb{R}^n$ for a column vector with $n$ entries corresponding to the training labels $y^{(i)}$. With this notation, (3.1) is equivalent to

$$\operatorname*{argmin}_{\boldsymbol{\theta}} \frac{1}{n}\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}\|^2 + \lambda\|\boldsymbol{\theta}\|^2 \ .$$

Now, by taking the gradient of this objective function and equating it to zero one can see that the optimization (3.1) reduces to solving this system of linear equations:

$$\left(\frac{1}{n}\boldsymbol{X}^\intercal\boldsymbol{X} + \lambda\boldsymbol{I}\right)\boldsymbol{\theta} = \frac{1}{n}\boldsymbol{X}^\intercal\boldsymbol{y} \tag{3.2}$$

where $\boldsymbol{I}$ represents the $d \times d$ identity matrix.

Therefore, ridge regression reduces to solving a system of linear equations of the form $\boldsymbol{A}\theta = \boldsymbol{b}$, where the coefficients $\boldsymbol{A} = \frac{1}{n}\boldsymbol{X}^\intercal\boldsymbol{X} + \lambda\boldsymbol{I}$ and $\boldsymbol{b} = \frac{1}{n}\boldsymbol{X}^\intercal\boldsymbol{y}$ only depend on the training data $(\boldsymbol{X}, \boldsymbol{y})$. Since the coefficient matrix $\boldsymbol{A}$ is positive definite by construction ridge regression can be solved using one of the many known algorithms for solving positive definite linear systems.

## 3.4 PROTOCOL DESCRIPTION

We consider the problem of solving a ridge regression problem when the training data $(\boldsymbol{X} \in \mathbb{R}^{n\times d}, \boldsymbol{y} \in \mathbb{R}^n)$ is distributed vertically (by columns) among several parties. As in previous work on privacy-preserving ridge regression [HFN11; Nik+13b], we assume that the regularization parameter $\lambda$ is public to all the parties, and hence the implementation of a secure data analysis pipeline including hyper-parameter tuning (e.g., cross-validation) is beyond the scope of this chapter. Furthermore, we also make the assumption in the vertically partitioned case that the correspondence between the rows in the datasets owned by different parties is known a priori. This assumption is easy to enforce in databases with unique identifiers for each record, given that efficient private set intersection protocols exist. The more

complex task of privacy-preserving entity resolution is beyond the scope of this chapter.

As discussed in Section 3.3, solving a ridge regression problem reduces to the solution of a positive definite linear system. A superficial inspection of Equation (3.2) shows that this can be seen as two sequential tasks: (i) compute the matrix $\boldsymbol{A} = \frac{1}{n}\boldsymbol{X}^\intercal\boldsymbol{X} + \lambda\boldsymbol{I}$ and the vector $\boldsymbol{b} = \frac{1}{n}\boldsymbol{X}^\intercal\boldsymbol{y}$, and (ii) solve the system $\boldsymbol{A}\boldsymbol{\theta} = \boldsymbol{b}$. In the distributed privacy-preserving setting, task (i) above corresponds to an *aggregation phase* where data held by different parties has to be combined. In contrast, we refer to (ii) as the *solving phase*, where the system of linear equations resulting from the aggregation is solved and all the parties obtain the solution in some form. It is important to note that in practice, ridge regression is used on problems with $n \gg d$, and that, while the input of the aggregation phase is of the order of $n \times d$, the solving phase has input size independent of $n$, i.e., of the order of $d^2$. This demonstrates the importance of having an efficient aggregation phase.

As mentioned in the introduction, the scenario where $(\boldsymbol{X}, \boldsymbol{y})$ is partitioned horizontally among the parties was studied by Nikolaenko et al. [Nik+13b]. With this partitioning, the computation of $A$ and $b$ can be cast as an addition of matrices computed *locally* by the parties. More concretely, in that case one can write $\boldsymbol{A} = \sum_{i=1}^{n} \boldsymbol{A}_i + \lambda\boldsymbol{I}$ with $\boldsymbol{A}_i = \boldsymbol{x}^{(i)}{\boldsymbol{x}^{(i)}}^\intercal/n$, and similarly $\boldsymbol{b} = \sum_{i=1}^{n} = \boldsymbol{b}_i$ with $\boldsymbol{b}_i = \boldsymbol{x}^{(i)}y^{(i)}/n$. The absence of the need for secure multiplication is crucial from a secure computation perspective, and guided the design of the protocol of Nikolaenko et al. [Nik+13b], as non-linear operations are expensive in secure computation. In contrast, the setting where the data is vertically partitioned requires secure computation of inner products on vectors owned by different parties. In the following section, we analyze the exact requirements for this setting, and propose secure computation protocols to overcome this challenge.

### 3.4.1 Aggregation Phase

For clarity of presentation, we first describe the two-party case, where $\boldsymbol{X}$ is vertically partitioned among two parties $\mathsf{P}_1$ and $\mathsf{P}_2$ as $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$, and $\boldsymbol{y}$ is held by one of the parties, say $\mathsf{P}_2$. As described above, the goal of this phase is to have the parties hold additive shares of $(\boldsymbol{A}, \boldsymbol{b})$. With this notation, the functionality $f$ of the aggregation phase can be described as follows:

$$f(\boldsymbol{X}_1, \boldsymbol{X}_2, \boldsymbol{y}) = \big((\boldsymbol{A} - \boldsymbol{A}_2, \boldsymbol{b} - \boldsymbol{b}_2), (\boldsymbol{A}_2, \boldsymbol{b}_2)\big),$$

where $(\boldsymbol{A} - \boldsymbol{A}_2, \boldsymbol{b} - \boldsymbol{b}_2)$ and $(\boldsymbol{A}_2, \boldsymbol{b}_2)$ are the outputs received by the first and the second party respectively.

Our protocol implements $f$ in a secure way. Let us now have a closer look at what needs to be computed. For simplicity, assume that

each party holds a block of contiguous features. Then, the following equations show how the content of the output matrix depends on the inputs of the two parties.

$$\boldsymbol{X}^\intercal = \begin{bmatrix} \boldsymbol{X}_1^\intercal \\ \boldsymbol{X}_2^\intercal \end{bmatrix} \quad \boldsymbol{X} = [\boldsymbol{X}_1 \; \boldsymbol{X}_2]$$

$$\boldsymbol{X}^\intercal \boldsymbol{X} = \begin{bmatrix} \boldsymbol{X}_1^\intercal \boldsymbol{X}_1 & \boldsymbol{X}_1^\intercal \boldsymbol{X}_2 \\ \boldsymbol{X}_2^\intercal \boldsymbol{X}_1 & \boldsymbol{X}_2^\intercal \boldsymbol{X}_2 \end{bmatrix}$$

Observe that the upper left part of the matrix $\boldsymbol{M} = \boldsymbol{X}^\intercal \boldsymbol{X}$ depends only on the input of party $\mathsf{P}_1$ and the lower right part depends only on the input of party $\mathsf{P}_2$. Hence, the corresponding entries of $\boldsymbol{M}$ can be computed locally by $\mathsf{P}_1$, while $\mathsf{P}_2$ simply has to set her shares of those entries to 0. On the other hand, for entries $m_{ij}$ of $\boldsymbol{M}$ such that features $i$ and $j$ are held by distinct parties, the parties need to compute an inner product between a column vector from $\boldsymbol{X}_1$ and a column vector from $\boldsymbol{X}_2$. To do so, we rely on a secure inner product protocol which we present next (Section 3.4.1.1). We note also that because the two off-diagonal blocks of $\boldsymbol{M}$ are transpositions of each other, computing only one of these blocks is enough to get an additive share of $\boldsymbol{M}$.

In the multi-party case, similarly to the two-party case, the vertical partitioning of the database implies that each party holds a subset of the columns of $\boldsymbol{X}$ and a corresponding subset of the rows of $\boldsymbol{X}^\intercal$. Since each value in $\boldsymbol{A}$ is an inner product between a row vector of $\boldsymbol{X}^\intercal$ and a column vector of $\boldsymbol{X}$, this means that each value in $\boldsymbol{A}$ depends on the inputs of at most two parties. Thus, also in the multi-party case, the aggregation phase for vertically-partitioned datasets can be reduced to secure two-party computation of inner products. At the end of the aggregation phase, the parties will obtain $\boldsymbol{A}$ in a form where each entry is either known to one party or is shared between some pair of input parties.

#### 3.4.1.1 Secure Inner Product

In this section, we present protocols for two parties $\mathsf{P}_1$ and $\mathsf{P}_2$, holding vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively, to securely compute the inner product of their vectors and obtain additive shares of the result. As mentioned before, we use a fixed-point encoding for real numbers (see Section 3.5 for details). Thus, our numbers can be represented as elements of a finite field $\mathbb{Z}_q$, and hence the functionality we need can be described as

$$f(\boldsymbol{x}, \boldsymbol{y}) = (r, \langle \boldsymbol{x}, \boldsymbol{y} \rangle - r) \tag{3.3}$$

where $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{Z}_q^n$, $r$ is a random element of $\mathbb{Z}_q$, and each party gets one of the components of the output. There are several techniques

**Protocol 1:** Secure inner products from OT.

**Parties:** $\mathsf{P}_1$, $\mathsf{P}_2$.
**Inputs:** $\mathsf{P}_1 : \boldsymbol{x} \in \mathbb{Z}_q^n$; $\mathsf{P}_2 : \boldsymbol{y} \in \mathbb{Z}_q^n$, where $q = 2^b$.
**Outputs:** $\mathsf{P}_1 : s_1 \in \mathbb{Z}_q$; $\mathsf{P}_2 : s_2 \in \mathbb{Z}_q$ such that $s_1 + s_2 = \langle \boldsymbol{x}, \boldsymbol{y} \rangle$.

(1) **foreach** $i \in [n]$ **do**
  Let $y_{i,j}$ be the $j$-th bit of the binary decomposition of $y_i$.
  **foreach** $j \in [b]$ **do**
    The parties run a 1-out-of-2-correlated OT protocol, where the $\mathsf{P}_1$ acts as the sender and obtains $m_{i,j}^0, m_{i,j}^1 = m_{i,j}^0 + x_i \cdot 2^j$, and $\mathsf{P}_2$ acts as the receiver and obtains $m_{i,j}^{y_{i,j}}$.

(2) $\mathsf{P}_1$ computes its share as $s_1 = \sum_{i \in [n], j \in [b]} -m_{i,j}^0$.

(3) $\mathsf{P}_2$ computes its share as $s_2 = \sum_{i \in [n], j \in [b]} m_{i,j}^{y_{i,j}}$.

that can be used for this task as, essentially, it corresponds to secure multiplication.

Generally, MPC techniques represent the implemented functionality either as arithmetic circuits (e.g., SPDZ [DPSZ12], TinyOT [NNOB12]) or as Boolean circuits (e.g., Yao's protocol [Yao86], GMW [GMW87]). While the former makes it possible to represent arithmetic operations very efficiently, the latter is better for performing bit-level operations. A second important property of MPC protocols is round complexity. Most notably, some techniques such as Yao's garbled circuits have a *constant* number of rounds, while for others (e.g., SPDZ, TinyOT, GMW), the number of rounds increases with the multiplicative depth of the circuit being evaluated (for Boolean circuits, this is the AND-depth). Note that the functionality in Equation (3.3) only consists of additions and multiplications. Furthermore, all multiplications can be done in parallel. Thus, a representation as an arithmetic circuit with constant multiplicative depth is the natural choice.

We propose two protocols for the functionality in Equation (3.3). The first (Protocol 1) is based on oblivious transer (OT, see Section 2.4). It was originally proposed by Gilboa [Gil99], and is used in different MPC contexts [KOS16; DSZ15]. The protocol uses OT to compute additive shares of the product of two numbers in a binary field, which can be trivially extended to securely compute the inner product of two vectors. Correctness of Protocol 1 can be verified by observing that

$$
\begin{aligned}
s_1 + s_2 &= \sum_{i\in[n],j\in[b]} -m^0_{i,j} + \sum_{i\in[n],j\in[b]} m^{y_{i,j}}_{i,j} \\
&= \sum_{i\in[n],j\in[b]} -m^0_{i,j} + m^0_{i,j} + (x_i \cdot 2^j) \cdot y_{i,j} \\
&= \sum_{i\in[n]} x_i \cdot \sum_{j\in[b]} 2^j \cdot y_{i,j} \\
&= \sum_{i\in[n]} x_i \cdot y_i = \langle \boldsymbol{x}, \boldsymbol{y} \rangle.
\end{aligned}
$$

Security follows from the fact that all received messages are generated randomly by the OT functionality, and so the shares as sums of pseudorandom values are also pseudorandom. From an efficiency perspective, observe that all OTs in Step (1) of Protocol 1 can be done in parallel, using a single correlated OT extension [ALSZ17].

Our second secure inner product protocol (Protocol 2) uses only symmetric key operations and is much more efficient than the previous one. It builds upon techniques based on precomputation of multiplicative triples [Bea91], as the secret sharing techniques mentioned above, and relies on an initializer that aids in the computation. Our protocol can also be seen as a modification of the construction presented by Du and Attalah [DA01b] where the trusted initializer does not keep a share of the result.

In both our protocols, we exploit the facts that, in our case, (i) the input vectors are each completely owned by one of the parties, and not shared among them, (ii) only two-party computations are needed, (ii) we are concerned about semi-honest security at this point. These observations lead to simpler protocols than general MPC protocols based on secret sharing.

In Section 3.7.3 we present an evaluation of our two protocols for secure inner product, in the context of our implementation of the aggregation phase for secure linear regression.

### 3.4.2 Solving Phase

After the aggregation phase, the parties hold additive shares of a square, symmetric, positive definite matrix $\boldsymbol{A}$ and a vector $\boldsymbol{b}$, and the task is to securely solve $\boldsymbol{A\theta} = \boldsymbol{b}$.

In line with previous work [Nik+13b; HFN11], and in order to achieve constant round-complexity, we choose Yao's garbled circuits for our solving phase, and introduce two additional non-colluding parties. These aid in the computation without learning neither the result, nor anything about the parties' input. In fact, our architecture relying on two external parties can be seen as a way of reducing a multi-party problem to a two-party problem, and had been used before by Nikolaenko et al. [Nik+13b] and Naor, Pinkas, and Sumner

**Protocol 2:** Secure inner product with a trusted initializer.

**Parties:** $P_1$, $P_2$, and trusted initializer TI.
**Inputs:** $P_1 : \boldsymbol{x} \in \mathbb{Z}_q^n$; $P_2 : \boldsymbol{y} \in \mathbb{Z}_q^n$.
**Outputs:** $P_1 : s_1 \in \mathbb{Z}_q$; $P_2 : s_2 \in \mathbb{Z}_q$ such that $s_1 + s_2 = \langle \boldsymbol{x}, \boldsymbol{y} \rangle$.

(1) TI generates random vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{Z}_q^n$ and a random number $r \in \mathbb{Z}_q$, and sets $z = \langle \boldsymbol{a}, \boldsymbol{b} \rangle - r$. It sends $(\boldsymbol{a}, r)$ to $P_1$, and $(\boldsymbol{b}, z)$ to $P_2$.

(2) $P_1$ sends $\boldsymbol{x} + \boldsymbol{a}$ to $P_2$.

(3) $P_2$ sends $\boldsymbol{y} - \boldsymbol{b}$ to $P_1$.

(4) $P_1$ computes its output share as $s_1 = \langle \boldsymbol{x}, \boldsymbol{y} - \boldsymbol{b} \rangle - r$.

(5) $P_2$ computes its output share as $s_2 = \langle \boldsymbol{x} + \boldsymbol{a}, \boldsymbol{b} \rangle - z$.

[NPS99]. Analogously to the presentation in [Nik+13b], we call our additional roles Crypto Service Provider (CSP) and evaluator. For clarity, from now on, we will call our parties *data providers*. The CSP is in charge of generating a garbled circuit for solving $\boldsymbol{A\theta} = \boldsymbol{b}$, while the evaluator will do the evaluation. In contrast to Nikolaenko et al. [Nik+13b], the evaluator does not necessarily learn the solution of the system in our protocol. The solving phase is described in Protocol 3.

Note that the CSP and the evaluator are only required to be non-colluding and hence, for Protocol 3, their roles could be taken by some of the data providers. In particular, for the case with two data providers, no additional parties are needed. We discuss security considerations of the solving phase protocol in the next section, where we present our complete protocol for secure linear regression.

Finally, let us note that we have not specified the details of the circuit $\mathcal{C}$ yet. It is important to remark that, after having designed a fast aggregation phase, the bottleneck of our protocol in terms of running time will now be in the solving phase. Hence, the concrete algorithm for solving $\boldsymbol{A\theta} = \boldsymbol{b}$ is a crucial element for the scalability of our protocol for secure linear regression. In Sections 3.5 and 3.6 we discuss desirable properties of a good algorithm for solving systems of linear equations tailored for MPC, and propose a Conjugate Gradient Descent algorithm. Finally, let us mention that, as we will see in Section 3.6, the circuit $\mathcal{C}$ has high degree. As mentioned in the previous section this motivates the choice of GCs as underlying MPC technique. An extensive evaluation of MPC techniques for the task of linear system solving is left for further work.

### 3.4.3 Secure Linear Regression

Our main protocol is depicted in Figure 3.1. In this case, the composition between the aggregation and solving phases is implemented in

**Protocol 3:** Solving phase of our linear regression protocol.

**Parties:** Data providers $P_1 \ldots, P_n$, CSP, and evaluator.
**Inputs:** $P_i$ : share $(\boldsymbol{A}_i, \boldsymbol{b}_i)$ of an equation $\boldsymbol{A\theta} = \boldsymbol{b}$.
**Output:** The data providers learn a solution of $\boldsymbol{A\theta} = \boldsymbol{b}$.

(1) The CSP generates a garbled circuit $\mathcal{C}$ for a functionality $f((\boldsymbol{A}_1, \boldsymbol{b}_1), \ldots, (\boldsymbol{A}_n, \boldsymbol{b}_n))$ that reconstructs and solves $\boldsymbol{A\theta} = \boldsymbol{b}$ and sends it to the evaluator.

(2) Each data provider $P_i$ runs oblivious transfers with the CSP to obtain garbled values $(\hat{\boldsymbol{A}}_i, \hat{\boldsymbol{b}}_i)$ for $(\boldsymbol{A}_i, \boldsymbol{b}_i)$ in $\mathcal{C}$ and forwards them to the evaluator.

(3) The evaluator evaluates $\mathcal{C}$ and shares $\hat{\boldsymbol{\theta}}$ with the data providers.

(4) The CSP sends the decryption mappings for the data providers to obtain $\boldsymbol{\theta}$ from $\hat{\boldsymbol{\theta}}$.

steps (d) and (e) by means of oblivious transfers between the CSP and the data providers. Here, the roles of trusted initializer and CSP in the aggregation and solving phases are taken by a single additional non-colluding party, while the role of the evaluator could be taken by one of the data providers.

In the two-party case, the roles of the CSP and the evaluator can be taken by the data providers, and the composition of the two phases is straightforward, while the role of trusted initializer in the aggregation phase is taken by an additional non-colluding party that aids in the computation.

A variant of our protocol described above implements the aggregation phase using the protocol based on OT (Section 3.4.1.1). In this way, we remove the need for an external non-colluding party. We will consider the performance of this variant in the experimental evaluation, and revisit it when we consider extensions of our protocol to withstand malicious adversaries.

Up to this point we have focused on design and correctness aspects of our protocols, without discussing their security guarantees. We undertake that task in the following section.

#### 3.4.3.1 Security Against Semi-Honest Adversaries

Our work is in the semi-honest – also known as *honest-but-curious* – threat model. Security in this model does not provide privacy, correctness, or even fairness guarantees if a party deviates from the protocol. However, the semi-honest setting is appropriate for distributed machine learning applications where parties are generally trusted, but should be prevented from sharing their data in the clear for legal reasons, or the computation is run in a controlled environment. These conditions are reasonably easy to meet in scenarions where the parties

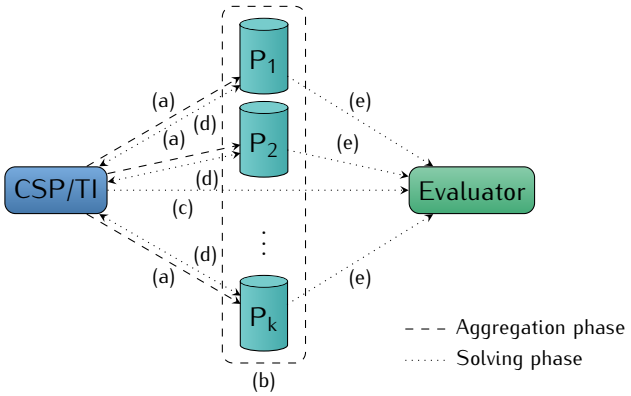



**Figure 3.1:** The multi-party protocol: (a, b) The data providers compute additive shares of $\boldsymbol{A\theta} = \boldsymbol{b}$, by doing local precomputations and executing several instances of Protocol 2. (c) The CSP generates a garbled circuit $\mathcal{C}$ for a functionality that reconstructs and solves $\boldsymbol{A\theta} = \boldsymbol{b}$ and sends it to the evaluator. (d) Each data provider $P_i$ obtains garbled values for their shares using OT and forwards them to the evaluator. (e) The evaluator evaluates $\mathcal{C}$ and shares the garbling of the result $\hat{\boldsymbol{\theta}}$ with the data providers. Finally, The CSP sends the decryption mappings for the data providers to obtain $\boldsymbol{\theta}$ from $\hat{\boldsymbol{\theta}}$.

are corporations or government agencies. In this section we formally argue the correctness of our protocol in the semi-honest threat model, and in Section 3.8 we discuss an extension to guarantee correctness of the result.

Similarly to how we presented our protocols, we argue security of our linear regression protocol with a modular approach, by proving the security of the protocol based on the security of its subprotocols.

Let us therefore first argue security of our protocols for the inner product functionality. For the OT-based protocol (Protocol 1) we refer the reader to Gilboa [Gil99] and Demmler, Schneider, and Zohner [DSZ15]. We state the correctness and security of the protocol based on a trusted initializer in the following lemma.

**Lemma 3.1.** *Assuming the trusted initializer (TI) does not collude with either* $\mathsf{P}_1$ *or* $\mathsf{P}_2$*, Protocol 2 computes the functionality* $f(\boldsymbol{x}, \boldsymbol{y}) = (r, \langle \boldsymbol{x}, \boldsymbol{y} \rangle - r)$ *with security against semi-honest adversaries, where* $(\boldsymbol{x}, \boldsymbol{y})$ *and* $(r, \langle \boldsymbol{x}, \boldsymbol{y} \rangle - r)$ *are the inputs and outputs of the parties.*

*Proof.* Correctness can be easily verified from the definition of the resulting shares. To prove security, we construct simulators $\mathcal{S}_1$ and $\mathcal{S}_2$, which simulate the views of the two parties.
$\mathcal{S}_1$ generates a view for $\mathsf{P}_1$ given $\boldsymbol{x}$ and $\mathsf{P}_1$'s output in the ideal functionality, denoted $s_1^*$. The message from the TI is simulated as $(\boldsymbol{a'}, \langle \boldsymbol{x}, \boldsymbol{b'} \rangle - s_1^*)$, while the message from $\mathsf{P}_2$ is simulated as $\boldsymbol{b'}$, where $\boldsymbol{a'}, \boldsymbol{b'} \in U(\mathbb{Z}_q^n)$. The resulting view has the same distribution as $\mathsf{P}_1$'s view in the real protocol execution.

Similarly, $\mathcal{S}_2$ generates a view for $P_2$ given $\boldsymbol{y}$ and $s_2^*$ as follows: the message from the TI is simulated as $(\boldsymbol{b'}, \langle \boldsymbol{a'}, \boldsymbol{b'} \rangle - s_2^*)$, while the message from $P_1$ is simulated as $\boldsymbol{a'}$, where $\boldsymbol{a'}, \boldsymbol{b'} \in U(\mathbb{Z}_q^n)$. These messages have the same distribution as $P_2$'s view in the real execution, which completes the proof. □

The security of our main protocol depicted in Figure 3.1 is stated in the following theorem. Informally, the theorem states that, as long as the participants do not deviate from the protocol, and the external parties CSP/TI and evaluator) are non-colluding, none of the parties learn anything from the inputs of the data providers that cannot be deduced from the result of the computation, namely the regression parameter $\theta$. Moreover, recall that the external parties do not obtain $\theta$.

**Theorem 3.2** (Security)**.** *Let $\Pi_1$ be a secure two-party computation protocol for the inner product functionality defined in Section 3.4.1.1 with security against semi-honest adversaries, and $\Pi_2$ be a two-party computation protocol based on garbled circuits with semi-honest security. The construction in Figure 3.1 is a secure computation protocol that provides security against semi-honest adversaries, in the setting where the CSP/TI and the evaluator collude with neither the data providers nor each other.*

*Proof.* We need to show simulation security against a semi-honest non-colluding adversary that controls the CSP, a semi-honest non-colluding adversary that controls the evaluator, and a semi-honest adversary that controls a subset of the input parties.

The security of the inner product protocol $\Pi_1$ implies that there exist simulators $\mathcal{S}_{P_1}$ and $\mathcal{S}_{P_2}$ which can simulate the view of each of the two parties in the execution of the protocol. The security of the two party computation protocol based on garbled circuits $\Pi_2$ implies the existence of simulators $\mathsf{Sim}_{\mathsf{garb}}$ and $\mathsf{Sim}_{\mathsf{eval}}$ which work as follows. $\mathsf{Sim}_{\mathsf{garb}}$ simulates the view of the garbling party, which consists of the execution of the OT protocols as a sender. $\mathsf{Sim}_{\mathsf{eval}}$ simulated the view of the evaluator which consists of the garbled circuit itself as well at the view in the execution of the OT protocols as a receiver. We will use these simulators in our proof.

*Semi-honest CSP.* The view of the CSP in the secure computation protocol consists only of the messages exchanged in the garbled circuit computation where it has no input, and the messages exchanged in the OT executions with the data providers. Both can be simulated using $\mathsf{Sim}_{\mathsf{garb}}$.

*Semi-honest evaluator.* Similarly to the case of the CSP, the view of the evaluator in the execution of the protocol consists of the messages exchanged in the execution of the garbled circuit evaluation where it has no inputs. Therefore, we can use the simulator $\mathsf{Sim}_{\mathsf{eval}}$ that can simulate its view.

*Semi-honest Input Parties.* The view of an adversary who controls a subset of the input parties in Protocol 3 consists of the messages exchanged in the executions of the inner product protocol in the aggregation phase with other input parties not controlled by the adversary, the messages in the OT executions with the CSP in the solving phase together with mappings for the output garbled labels as well as the garbled output the parties receive from the Evaluator. We can use the simulators $\mathcal{S}_{P_1}$ and $\mathcal{S}_{P_2}$ to obtain the view of the adversary in the aggregation phase. Additionally, we can use $\mathsf{Sim}_{\mathsf{garb}}$ for $\Pi_2$ to simulate the view of the adversary in the OT executions. Given the value $\theta$ of the output solution, which the parties will receive, the simulator $\mathsf{Sim}_{\mathsf{garb}}$ can also generate mappings for the output garbled labels to real values and a set of garbled labels for the output that open to $\theta$. This completes the proof. □

The above security argument can be easily adapted to the case where the evaluator is one of the data providers, as long as CSP/TI does not collude with any other party.

We can further extend the security guarantees of our protocol, to the setting where the roles of the CSP and the evaluator are executed by two of the parties, as long as these two parties are semi-honest and non-colluding with each other. In this scenario, we use the OT-based protocol for the aggregation phase, as it does not require a trusted initializer (which was instantiated by the CSP). The security guarantees for this variant in which *no external parties are required*, are stated in the next theorem.

**Theorem 3.3** (Security without external parties)**.** *Let* $\Pi$ *be an instantiation of the protocol described in Figure 3.1 where the roles of CSP and the evaluator are implemented by two distinct input providers, and the aggregation phase is instantiated with the OT-based inner product protocol. Then,* $\Pi$ *provides security against semi-honest adversaries in the setting where the CSP and the evaluator do not collude.*

In Section 3.7 we provide an experimental evaluation of the protocols of the two theorems above where we quantify the speed-up obtained by relying on an offline phase performed by a non-colluding external party acting as trusted initializer.

## 3.5 NUMBER REPRESENTATION

*The accuracy analysis in this section was performed by co-author Borja Balle.*

Implementing secure MPC protocols to work with numerical data poses a number of challenges. In our case, the most significant of those challenges is choosing an encoding for numerical data providing a good trade-off between efficiency and accuracy. The IEEE 754 floating-point representation is the standard choice used by virtually every numerical computation software because of its high accuracy

and numerical stability when working with numbers across a wide range of orders of magnitude. However, efficient implementations of IEEE 754-compliant secure MPC protocols are a subject of on-going research [Dem+15; PS15], with the current state-of-the-art yielding 32 bit floating-point multiplication circuits running in time comparable to computers from the 1960's [Wei61]. Unfortunately, these implementations do not yet scale to the throughput required for data analysis tasks involving large datasets, and forces us to rely on fixed-point encodings like previous work in this area [HFN11; Nik+13b; CDNN15]. The rest of this section presents the details of the particular fixed-point encoding used by our system, with a particular emphasis on its effect on the accuracy of the protocol. We further introduce data normalization and scaling steps that play an important role in the context of linear regression.

### 3.5.1 Fixed-Point Arithmetic

Our use of fixed-point encodings follows closely previous works on secure linear regression [HFN11; Nik+13b; CDNN15]. However, unlike these, we provide a formal analysis of the errors such encodings introduce when used to simulate operations on real numbers.

In a fixed-point signed binary encoding scheme each number is represented using a fixed number of bits $b$. These bits are subsequently split into three parts: one sign bit, $b_f$ bits for the fractional part, and $b_i$ bits for the integral part, with the obvious constraint $b = b_i + b_f + 1$. For each possible value of $b_f$ and $b_i$ one gets an encoding capable of representing all the numbers in the interval $[-2^{b_i} + 2^{-b_f-1}, 2^{b_i+1} - 2^{-b_f-1}]$ with accuracy $2^{-b_f}$. In general, arithmetic operations with fixed-point numbers can be implemented using signed integer arithmetic, though special care is required to deal with operations that might result in overflow. We point the reader to [EL04] for an in-depth discussion of fixed-point arithmetic.

In order to implement Protocol 2 we need to simulate fixed-point arithmetic over a finite ring $\mathbb{Z}_q$ whose elements can be represented using $b$ bits. Thus, the encoding used by our protocols involves two steps: mapping reals into integers, and mapping integers to integers modulo $q$ with $q = 2^b$. We introduce encoding and decoding maps for each of these two steps, as summarized in the following diagram:

$$\mathbb{R} \underset{\tilde{\phi}_\delta}{\overset{\phi_\delta}{\rightleftarrows}} \mathbb{Z} \underset{\tilde{\varphi}_q}{\overset{\varphi_q}{\rightleftarrows}} \mathbb{Z}_q \ .$$

The map encoding reals into integers is given by $\phi_\delta(r) = \lfloor r/\delta \rceil$, where the parameter $\delta = 2^{-b_f}$ controls the encoding precision and $\lfloor \cdot \rceil$ returns the rounding of a real number to the closest integer (with ties favouring the largest integer). In particular, $\delta$ is the smallest number that can be represented by our encoding $\phi_\delta$. The integer decoding

mapping is given by $\tilde{\phi}_\delta(z) = z\delta$. Next we recall several well-known facts [EL04] about the error introduced by operating under a fixed-point encoding with finite precision $\delta$. For any reals $r, r' \in \mathbb{R}$ we have the following:

1. $|r - \tilde{\phi}_\delta(\phi_\delta(r))| \leq \delta$,
2. $|(r + r') - \tilde{\phi}_\delta(\phi_\delta(r) + \phi_\delta(r'))| \leq 2\delta$,
3. $|(rr') - \tilde{\phi}_{\delta^2}(\phi_\delta(r)\phi_\delta(r'))| \leq (|r| + |r'|)\delta + \delta^2$.

Note the last fact involves a decoding with precision $\delta^2$ to account for the increase in the number of fractional bits required to represent the product of two fixed-point numbers.

Encoding reals in a bounded interval $[-M, M]$ with $\phi_\delta$ yields integers in the range $[-M/\delta] \leq \phi_\delta(r) \leq [M/\delta]$. This interval contains $K = 2M/\delta + 1$ integers. Thus, it is possible to map this interval injectively into any ring $\mathbb{Z}_q$ of integers modulo $q$ with $q \geq K$. We obtain such injection by defining the encoding map $\varphi_q(z) = z \mod q$, which for integers in the range $-q/2 \leq z \leq q/2$ yields $\varphi_q(z) = z$ if $z \geq 0$ and $\varphi_q(z) = q + z$ for $z < 0$. The corresponding decoding map is given by $\tilde{\varphi}_q(u) = u$ if $0 \leq u \leq q/2$ and $\tilde{\varphi}_q(u) = u - q$ for $q/2 < u \leq q - 1$. Although $\varphi_q$ is a ring homomorphism mapping operations in $\mathbb{Z}$ into operations in $\mathbb{Z}_q$, decoding from $\mathbb{Z}_q$ to $\mathbb{Z}$ after operating on encoded integers might not yield the desired result due to overflows. To avoid such overflows one must check that the result falls in the interval where the coding $\varphi_q$ is the inverse of $\tilde{\varphi}_q$. In particular, the following properties only hold for integers $z, z'$ such that $|z|, |z'| \leq q/2$:

1. $\tilde{\varphi}_q(\varphi_q(z)) = z$,
2. $|z + z'| \leq q/2$ implies $z + z' = \tilde{\varphi}_q(\varphi_q(z) + \varphi_q(z'))$,
3. $|z \cdot z'| \leq q/2$ implies $z \cdot z' = \tilde{\varphi}_q(\varphi_q(z) \cdot \varphi_q(z'))$.

### 3.5.2 Accuracy of Inner Product

The properties of the encodings described in the previous section can be used to provide a bound on the accuracy of the result of Protocol 2. At the end of the protocol, both parties have an additive share of of $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ for some integer vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{Z}_q^n$. Therefore, we want to show that when the input vectors $\boldsymbol{x}$, $\boldsymbol{y}$ provided by the parties are fixed-point encodings of some real vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^n$, then the result of the inner product over $\mathbb{Z}_q$ can be decoded into a real number approximating $\langle \boldsymbol{u}, \boldsymbol{v} \rangle$. We note that the comparison here is against real arithmetic, while in the experimental section we will be comparing against floating-point arithmetic, which can represent real arithmetic to high degrees of accuracy across a much larger range of numbers [Knu97].

According to the previous section, there are two sources of error when working with fixed-point encoded real numbers: the systematic error induced by working with finite precision (which can be mitigated by increasing $b_f$), and the occasional error that occurs when the numbers in $\mathbb{Z}_q$ overflow (which can be mitigated by increasing $b_i$). To control the overflow error we assume that $\boldsymbol{u}$ and $\boldsymbol{v}$ are $n$-dimensional real vectors with entries bounded by $R$, $|u_i|, |v_i| \leq R$ for $i \in [n]$, and $\boldsymbol{x} = \varphi_q(\phi_\delta(\boldsymbol{u}))$ and $\boldsymbol{y} = \varphi_q(\phi_\delta(\boldsymbol{v}))$ are encodings of those vectors with precision $\delta$.

**Theorem 3.4** ([Wil88, Ch. 3]). *If $q > 2nR^2/\delta^2$, then:*

$$|\langle u, v\rangle - \tilde{\phi}_{\delta^2}(\tilde{\varphi}_q(\langle \boldsymbol{x}, \boldsymbol{y}\rangle))| \leq 2nR\delta + n\delta^2 \ .$$

*Proof.* In the first place we observe that any number occurring in the computation of the inner product $\langle \phi_\delta(\boldsymbol{u}), \phi_\delta(\boldsymbol{v})\rangle$ of the integer encodings of $u$ and $v$ is bounded by $nR^2/\delta^2$. Therefore, the condition on $q$ ensures there are no overflows in the computation in $\mathbb{Z}_q$ and we have $\langle \phi_\delta(u), \phi_\delta(v)\rangle = \tilde{\varphi}_q(\langle \boldsymbol{x}, \boldsymbol{y}\rangle)$. Now we use the formulas for the error of sum and product of integer encodings from previous section to show that

$$\begin{aligned}
&|\langle \boldsymbol{u}, \boldsymbol{v}\rangle - \tilde{\phi}_{\delta^2}\left(\langle \phi_\delta(\boldsymbol{u}), \phi_\delta(\boldsymbol{v})\rangle\right)| \\
&\quad = \left|\sum_{i=1}^{n} u_i v_i - \tilde{\phi}_{\delta^2}\left(\sum_{i=1}^{n} \phi_\delta(u_i)\phi_\delta(v_i)\right)\right| \\
&\quad \leq \sum_{i=1}^{n} |u_i v_i - \tilde{\phi}_{\delta^2}(\phi_\delta(u_i)\phi_\delta(v_i))| \\
&\quad \leq n(2R\delta + \delta^2) \ ,
\end{aligned}$$

where the first inequality follows by the triangle inequality and linearity of the decoding map $\tilde{\phi}_{\delta^2}$. □

We note that the assumption of a bound $R$ on the entries of the vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ is not very stringent: if both parties agree on a common bound $R$, or there is a publicly known $R$, then the vectors can be normalized locally by each party before the execution of the protocol. Thus, in practice it is convenient to normalize the real vectors $u$ and $v$ such that their entries are bounded by $C/\sqrt{n}$ for some constant $C > 0$. In this case, the bounds above can be used to show that if one requires the final inner product to have error at most $\varepsilon$, this can be achieved by taking $\delta = \varepsilon/2C\sqrt{n}$ and $q = 8C^2n/\varepsilon^2$. Using these expressions we see that an encoding with $b = O(\log(n/\varepsilon))$ bits is enough to achieve accuracy $\varepsilon$ when performing $n$-dimensional inner products with fixed-point encoded reals.

### 3.5.3 Data Standardization and Scaling

The statistical theory behind ridge regression generally assumes that the covariates in the columns of $\boldsymbol{X}$ are normally distributed with zero

mean and unit variance. In practical applications this is not always satisfied. However, it is a common practice to standardize each column of the training data to have zero mean and unit variance. This column-wise procedure is implemented as part of the local pre-processing done by each party in our linear regression protocol. We use $\bar{\boldsymbol{X}}$ to denote the data matrix obtained after standardization.

The importance of proper data scaling to prevent overflows in the aggregation phase is the key message from Theorem 3.4 (Section 3.5.2). Additionally, a critical phenomenon occurring in the solving phase is the degradation in the quality of the solution when the encoding is fixed and the dimension of the linear system grows. To alleviate the effect of the dimension on the accuracy of the solving phase, our pre-processing re-scales every entry in $\bar{\boldsymbol{X}}$ and $\boldsymbol{y}$ by $\frac{1}{\sqrt{d}}$. We note that these scalings can be performed locally by the parties holding each respective column.

The standardization and scaling steps described above imply that our protocols effectively solve the linear system

$$\left(\frac{1}{nd}\bar{\boldsymbol{X}}^{\intercal}\bar{\boldsymbol{X}} + \lambda \boldsymbol{I}\right)\theta = \frac{1}{nd}\bar{\boldsymbol{X}}^{\intercal}\boldsymbol{y} \ .$$

This scaling has no effect on the linear regression problem being solved beyond changing the scale of the regularization parameter. On the other hand, the standardization step needs to be reversed if one wants to make predictions on test data following the same distribution as the original training data. This task corresponds to a simple linear computation that can be efficiently implemented in MPC, both in the case where the model $\boldsymbol{\theta}$ is disclosed to the parties and the case where it is kept shared among the parties and the prediction task using $\boldsymbol{\theta}$ is itself implemented in a secure way using MPC.

## 3.6 SOLVING LINEAR SYSTEMS

As discussed in Section 3.4.2, the solving phase of our protocol involves solving positive definite linear systems of the form $\boldsymbol{A\theta} = \boldsymbol{b}$ in a garbled circuits protocol. A wide variety of solutions to this problem can be found in the numerical analysis literature, though not all of them are suitable for MPC. For example, any variant of Gaussian elimination involving data-dependent pivoting can be immediately ruled out because implementing non-data-oblivious algorithms inside garbled circuits produces unnecessary blow-ups in the circuit size. This limitation has already been recognized by previous works on private linear regression using garbled circuits and other MPC techniques [HFN11; Nik+13b]. In particular, these works consider two main alternatives: computing the full inverse of $\boldsymbol{A}$, or solving the linear system directly using the Cholesky decomposition of $\boldsymbol{A}$

**Functionality 4:** Our variant of the Conjugate Gradient Descent algorithm, optimized for fixed-point encoded numbers.

**Inputs:** positive definite system $(\boldsymbol{A}, \boldsymbol{b})$, number of iterations $T$.
**Output:** approximate solution $\boldsymbol{\theta}_T$ with $\boldsymbol{A}\boldsymbol{\theta}_T \approx \boldsymbol{b}$.

(1) $\boldsymbol{\theta}_0 \leftarrow 0$, $\boldsymbol{g}_0 \leftarrow \boldsymbol{A}\boldsymbol{\theta}_0 - \boldsymbol{b}$, $\tilde{\boldsymbol{g}}_0 \leftarrow \frac{\boldsymbol{g}_0}{\|\boldsymbol{g}_0\|_\infty}$, $\boldsymbol{p}_0 \leftarrow \tilde{\boldsymbol{g}}_0$

(2) For $t = 0 \dots T-1$ repeat:

(a) $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \left(\frac{\boldsymbol{p}_t^\intercal \boldsymbol{g}_t}{\boldsymbol{p}_t^\intercal \boldsymbol{A}\boldsymbol{p}_t}\right)\boldsymbol{p}_t$.

(b) $\boldsymbol{g}_{t+1} \leftarrow \boldsymbol{g}_t - \left(\frac{\boldsymbol{p}_t^\intercal \boldsymbol{g}_t}{\boldsymbol{p}_t^\intercal \boldsymbol{A}\boldsymbol{p}_t}\right)\boldsymbol{A}\boldsymbol{p}_t$.

(c) $\tilde{\boldsymbol{g}}_{t+1} \leftarrow \frac{\boldsymbol{g}_{t+1}}{\|\boldsymbol{g}_{t+1}\|_\infty}$.

(d) $\boldsymbol{p}_{t+1} \leftarrow \tilde{\boldsymbol{g}}_{t+1} - \left(\frac{\boldsymbol{p}_t^\intercal \tilde{\boldsymbol{g}}_{t+1}}{\boldsymbol{p}_t^\intercal \boldsymbol{A}\boldsymbol{p}_t}\right)\boldsymbol{p}_t$.

as an intermediate step. Although both of these methods have an asymptotic computational cost of $O(d^3)$, in practice Cholesky is faster and numerically more stable. This justifies the choice of Cholesky as a procedure for solving linear systems in garbled circuits made in [Nik+13b]. However, as we show in Section 3.7.2, Cholesky's cubic cost in the dimension can become prohibitive when working with high-dimensional data, in which case one must resort to iterative approximate algorithms like Conjugate Gradient Descent (CGD).

In the following Section we present a novel CGD algorithm tailored for working with fixed-point arithmetic and compare it to the classical CGD algorithm.

### 3.6.1 Conjugate Gradient Descent

Factorizing the coefficient matrix $\boldsymbol{A}$ in order to make the solution easier to compute, as in Cholesky's algorithm, is not the only efficient way to solve a system of linear equations. An entirely different approach relies on iterative algorithms which construct a monotonically improving sequence of approximate solutions $\boldsymbol{\theta}_t$ converging to the desired solution. The main virtue of iterative algorithms is the possibility to reduce the cost of solving the system below the cubic complexity required by exact algorithms at the expense of providing only an approximate solution. In particular, for linear systems arising from ridge regression the practical importance of iterative algorithms is twofold: since intensive computations inside an MPC framework can be expensive, iterative methods provide a natural way to spend a fixed computational budget on finding an approximation to the desired solution by stopping after a fixed number of iterations; and, since the coefficients of the linear system occurring in ridge regression are inherently noisy because they are computed from a finite amount

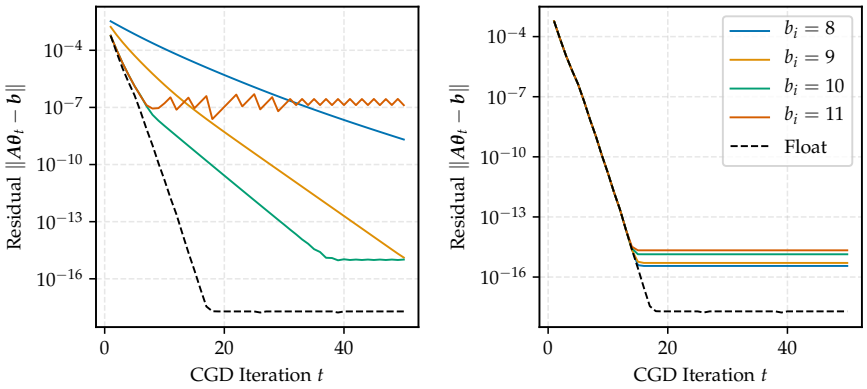



**Figure 3.2:** Fixed-point CGD Implementations (with 64 bits on a system of dimension 50). (Left) Textbook CGD [NW99]. (Right) Our Fixed-point CGD.

of data, finding an approximate solution whose accuracy is on the same order of magnitude as the noise in the coefficients is generally sufficient for optimizing the predictive performance of the model.

When the coefficient matrix of the linear system $\boldsymbol{A\theta} = \boldsymbol{b}$ is positive definite, a popular iterative method is the *conjugate gradient descent* (CGD) algorithm. The CGD algorithm can be interpreted as solving the system by iteratively minimizing the objective $\|\boldsymbol{A\theta} - \boldsymbol{b}\|^2$ with respect to the parameter vector $\boldsymbol{\theta}$ using the method of conjugate gradients [NW99]. However, the numerical stability of CGD is in general worse than that of Cholesky's algorithm, making it very sensitive to the choice of fixed-point encoding. This can be observed in Figure 3.2 (left), where we plot the behavior of the residual $\|\boldsymbol{A\theta}_t - \boldsymbol{b}\|$ as a function of $t$ for several settings of the number of number of integer bits $b_i$ when solving a system with $d = 50$ using 64-bit fixed-point encodings. This plot shows that CGD is very sensitive to the choice of $b_i$, and even in the optimal setting ($b_i = 10$) the convergence rate is much slower than the one achieved by the corresponding floating-point implementation (dashed black curve).

After a thorough investigation of the behavior of a fixed-point implementation of CGD we concluded that the main problem occurs when the norm of the conjugate gradients decreases too fast and consequently the step sizes grow at each iteration. This motivates a variant of CGD, which we call *fixed-point CGD* (FP-CGD), and whose pseudocode is given in Functionality 4. The only difference between standard CGD and FP-CGD is the use of normalized gradients $\tilde{\boldsymbol{g}}_t = \boldsymbol{g}_t / \|\boldsymbol{g}_t\|_\infty$ in the computation of the conjugate search directions $p_t$; in particular, by taking $\tilde{\boldsymbol{g}}_t = \boldsymbol{g}_t$ one recovers the classical textbook algorithm. While CGD has been used in the context of secure computation in previous work [BKLS18], to the best of our knowledge, we are the first to describe this modification of CGD.

It is easy to show that CGD and FP-CGD produce exactly the same sequence of approximate solutions $\theta_t$ when working in exact arithmetic. However, normalizing the search directions in FP-CGD makes the method much more stable with respect to the errors introduced by fixed-point arithmetic. This is illustrated by Figure 3.2 (right), where we show the evolution of the residuals for the same system and fixed-point encodings that we used to test standard CGD. Here, we observe that we recover the same converge rate as CGD with floating-point arithmetic while only suffering a loss between 2 and 3 digits of accuracy. We will see later in our experiments with real data (Section 3.7.4) that this loss in accuracy is negligible in real applications, and it is comparable (or better) than the accuracy provided by a fixed-point implementation of Cholesky's method. In terms of computation, we note that each iteration of FP-CGD is slightly more expensive than an iteration of standard CGD due to the normalization step, but the asymptotic cost per iteration is in $\Theta(d^2)$ in both cases.

Further justification for the use of FP-CGD instead of other iterative methods is provided by its favorable theoretical properties. In exact arithmetic, FP-CGD inherits two important properties from CGD: (i) it converges to the *exact solution* of $\boldsymbol{A\theta} = \boldsymbol{b}$ in exactly $d$ iterations; and (ii) it achieves the *optimal convergence rate* among all first-order methods, yielding a solution with error at most $\varepsilon$ after $O\big(\sqrt{\kappa(\boldsymbol{A})}\log(1/\varepsilon)\big)$ iterations. Property (i) is important because not all iterative algorithms for solving linear systems are guaranteed to produce an exact solution after a finite number of iterations, including standard gradient descent methods like the ones implemented in [GLN12] using leveled homomorphic encryption. Furthermore, property (ii) says essentially that it is not possible to find a first-order method producing more accurate approximate solutions than CGD, and that only a few iterations of CGD are required to accurately solve linear systems with small condition number $\kappa(\boldsymbol{A})$. In principle, these properties might not be preserved by the fixed-point implementations of CGD and FP-CGD due to the numerical errors introduced by the finite-precision arithmetic. In the case of CGD, [Meu06] shows that when working with a finite number of bits for the fractional part and an infinite number of bits for the integer part, several important properties of the exact version, including the convergence rate, are preserved up to small order modifications. However, the assumption of infinite bits for the integer part is not realistic in practice, and the experiments in Figure 3.2 show that when a fixed number of bits needs to be split between the integer and fractional part of the representation a crucial tension arises. FP-CGD is designed to alleviate such tension by making sure that a minimal number of bits in the integer part suffice to solve a properly scaled system accurately. We demonstrate this through our experimental evaluations in Section 3.7, where we also compare the accuracy and running time of FP-CGD against the method based in

Cholesky decomposition used in [Nik+13b], and study the effect of the condition number $\kappa(\boldsymbol{A})$ on both methods.

## 3.7 EXPERIMENTAL RESULTS

This section presents an extensive empirical evaluation of our protocols. We start by describing the details about the concrete implementation of our system and the experimental setup. Then we present experiments for different implementations of the solving phase using FP-CGD and Cholesky, comparing them in terms of accuracy and running time across a wide range of settings. The second set of experiments evaluates two implementations for the aggregation phase: one based on the inner product protocol relying on a trusted initializer, and the one based on OT. Finally, we present an evaluation of the complete system on nine real datasets from the UCI repository with different algorithms in the solving phase and compare the resulting predictive performance of the learned model against the performance obtained in the non-private setting. Overall, our experiments highlight the importance of using FP-CGD for high-dimensional data, and provide a guide on how to choose a good fixed-point encoding depending on the characteristics of the problem.

### 3.7.1 Implementation and Setup

We implemented our MPC protocols using Obliv-C [ZE15], an extension of the C programming language for secure computation. Obliv-C includes an implementation of Yao's garbled circuit protocol that incorporates recent optimizations including free XOR [KS08], fixed key block ciphers [BHKR13], and half gates [ZRE15]. Further, it features efficient implementations of OT extension, including variants such as correlated OT [ALSZ17].

*Our source code is available at* `https://github.com/schoppmp/linreg-mpc`.

To support arbitrary precision arithmetic, we rely on a big integer library [Doe] for multi-party computation as an extension of Obliv-C. This library composes an arbitrary number of garbled integers to represent larger integer values in two's complement binary. It features standard operations such as comparisons, bit manipulation and shifts, and arithmetic functions including addition, multiplication, integer division and modulo computation. All the arithmetic operations are implemented using common, efficient algorithms for extended precision arithmetic, such as the Karatsuba-Comba [KO62] method for multiplication, and Knuth's algorithm D [Knu97] for division.

*The big integer library was developed by co-author Jack Doerner.*

For fixed-point arithmetic, we implement two variants of our protocols. One uses Obliv-C's native integer types to represent fixed-point numbers. Since for fixed-point multiplication, the bit width of intermediate values can be large as the sum of the arguments' bit widths,

this version is limited to 32-bit numbers. Our second fixed-point implementation uses the big integer library described above, allowing to represent numbers with arbitrary precision. This arrangement incurs some overhead relative to using Obliv-C's native types. However, this overhead is justified in cases where 32-bit arithmetic introduces large errors. Although in principle, our second implementation allows for an arbitrary number of bits, we only evaluate it with 64 bit numbers. As the results in Section 3.7.4 show, this is enough to get accurate results on real datasets.

For the solving phase, we implemented the Cholesky and FP-CGD methods described by Nikolaenko et al. [Nik+13b] and Section 3.6, respectively. Cholesky descomposition requires square root computation. Nikolaenko et al. use an iterative algorithm for computing square roots in garbled circuits based on Newton's method. Our implementation of Cholesky's method follows closely their approach and is based on the pseudo-code given in [Nik+13b].

For the experiments in Sections 3.7.2 and 3.7.3, where each phase of the protocol is evaluated separately, we use synthetically generated data. For each setting of $d$ and $n$, we sample $n$ data points $\boldsymbol{x}^{(i)}$ from a standard $d$-dimensional Gaussian distribution and a $d$-dimensional vector of parameters $\boldsymbol{\theta}^*$ with independent coordinates sampled uniformly in the interval $[0, 1]$. The training labels are obtained as $y^{(i)} = \langle \boldsymbol{\theta}^*, \boldsymbol{x}^{(i)} \rangle + \epsilon^{(i)}$, where $\epsilon^{(i)}$ is a noise term sampled from a Gaussian distribution with zero mean and variance $\sigma^2 = 0.1$. The regularization parameter in the solving phase is set to $\lambda = \sigma^2 d / n \|\boldsymbol{\theta}^*\|^2$, which is the optimal choice suggested by the statistical theory of ridge regression [SL03].

For the experiments with synthetic data using 64 and 32 bit fixed-point encodings we used 60 and 28 bits for the fractional part, respectively. In the experiments with real data, the split between the fractional and integer part of the fixed-point encoding was optimized individually for each problem to obtain the best predictive performance. All experiments were executed using Amazon EC2 C4 instances, each having 4 CPU cores, 7.5 GiB of RAM and 1 Gbps bandwidth.

### 3.7.2 Solving Phase

Figure 3.3 (left) compares the running times of the solving phase using Cholesky and FP-CGD for dimensions ranging from 10 to 500. For FP-CGD, we give the running time for 5, 10, 15, and 20 iterations. The top plot corresponds to the 64-bit implementation and the bottom plot to the 32-bit implementation. While Cholesky is faster than CGD for lower values of $d$, its cubic dependence on the dimension makes running a fixed number of iterations of FP-CGD faster as $d$ increases. Thus, for high-dimensional data the iterative FP-CGD method is preferable in terms of computation time. The time spent running

oblivious transfers is also shown, and accounts for a small fraction of the running time. For example, for the 64-bit implementations with $d = 200$, FP-CGD with 15 iterations runs in less than 45 minutes, while Cholesky takes more than an hour and a half. For $d = 500$, Cholesky takes more than 24 hours while FP-CGD with 15 iterations takes less than 4.5 hours. Similar results reporting circuit size instead of running time are presented Figure 3.7 in Appendix 3.A.

Next, we compare both algorithms in terms of accuracy of the results. Here, the error is measured using the Euclidean distance to the optimal solution, obtained via floating-point computation. Figure 3.3 (middle) shows how the accuracy of the result is affected by the condition number of the input matrix $\boldsymbol{A}$, comparing Cholesky with FP-CGD for varying numbers of iterations on problems with $d = 20$. We observed that FP-CGD achieves the same accuracy as Cholesky in the 64-bit version, and has even lower error when using only 32 bits. Moreover, the influence of the input condition number decreases with the number of iterations. After convergence, the error remains the same for all tested condition numbers. This advantage over Cholesky increases with higher $d$, as is shown in Figure 3.3 (right). For both the 32-bit and 64-bit versions, FP-CGD is more accurate than Cholesky as soon as $d > 50$.

### 3.7.3 Aggregation Phase

Table 3.4 shows a comparison between the running times of the 64-bit versions of our two aggregation protocols: *OT* is the protocol using the OT-based inner product protocol, and *TI* is the protocol that leverages a trusted initializer for the inner product protocol. We vary the number of records, $n$, from 50,000 up to one million, and the number of features, $d$, from 20 to 200. As expected, the protocol that takes advantage of the Trusted Initializer performs significantly better in all cases. However, since the TI's resources are the bottleneck in this setting, the protocol does not scale well to multiple data providers, as the fraction of the coefficient matrix $\boldsymbol{A}$ that can be computed locally shrinks as the number of parties increases. The OT-based version, on the other hand, handles many parties very well, due to the fact that OTs between different pairs of parties can be performed in parallel. Timing results our protocols for the aggregation phase for a more extensive set of configurations can be found in Tables 3.8 and 3.9 of Appendix 3.A.

As a baseline, we implemented a garbled circuit protocol for performing a *single* inner product in Obliv-C. The running time for $n = 10^6$ was over 90 minutes. This implies that our protocol outperforms this naive approach by several orders of magnitude.

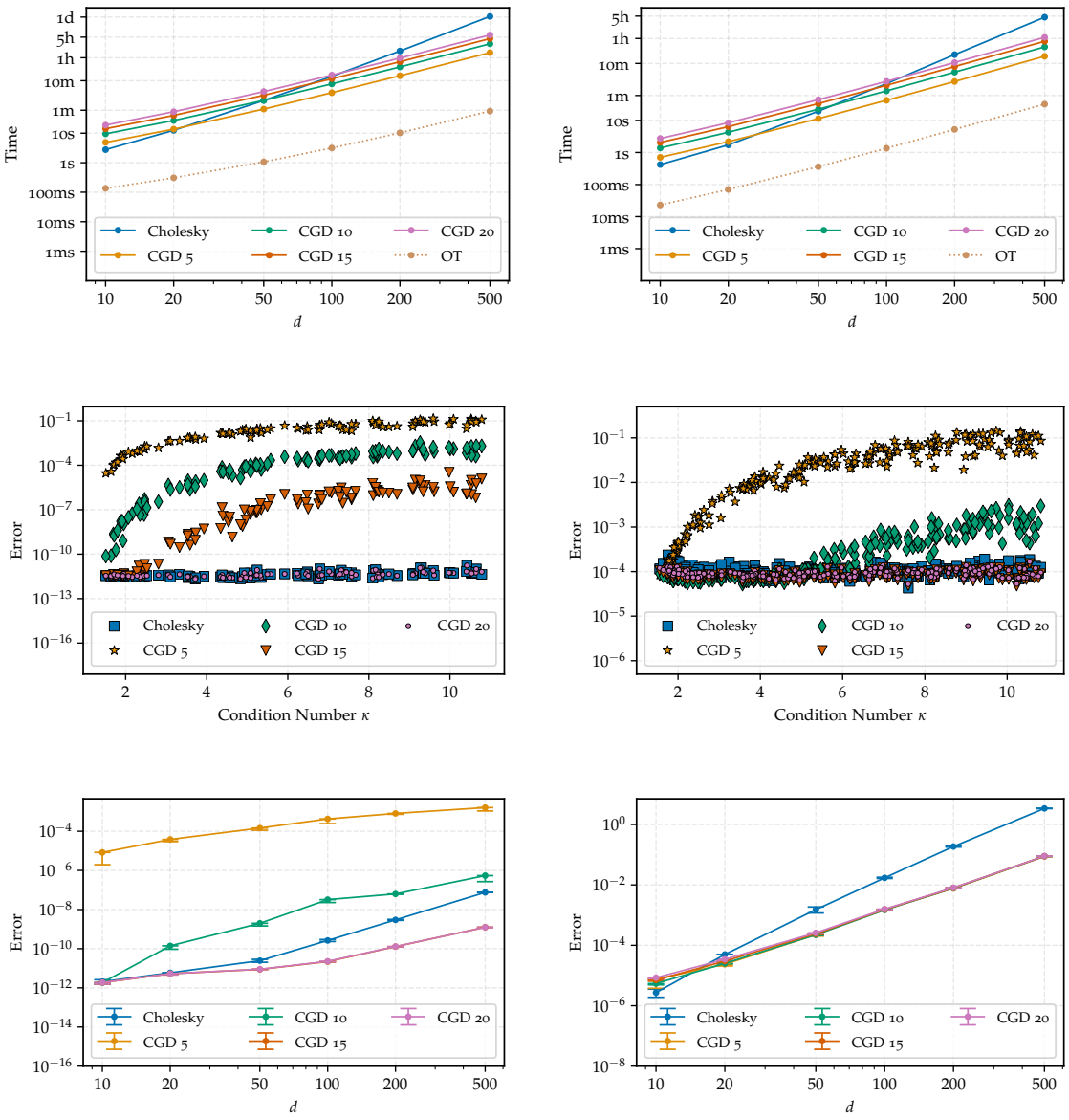



**Figure 3.3:** Comparison between different methods for solving linear systems. (Top) Running time (seconds) of Cholesky and FP-CGD (with 5, 10, 15, and 20 iterations) as a function of input dimension. (Middle) Accuracy of Cholesky and CGD depending on the condition number of the input matrix $\boldsymbol{A}$ with $d = 20$. (Bottom) Accuracy of Cholesky and CGD, as a function of the input dimensionality $d$. (Left) Fixed-point numbers with $b = 64$ bits, with $b_f = 60$ in the fractional part. (Right) $b = 32, b_f = 28$.

| $n$ | $d$ | Number of parties | | | | | |
|---|---|---|---|---|---|---|---|
| | | 2 | | 3 | | 5 | |
| | | OT | TI | OT | TI | OT | TI |
| 50000 | 20 | 1m 50s | 1s | 1m 32s | 2s | 1m 07s | 2s |
| 50000 | 100 | 42m 12s | 25s | 34m 39s | 32s | 24m 58s | 37s |
| 500000 | 20 | 18m 18s | 15s | 14m 29s | 18s | 12m 10s | 21s |
| 500000 | 100 | 7h 03m 56s | 4m 47s | 5h 20m 52s | 6m 01s | 4h 17m 08s | 6m 58s |
| 1000000 | 100 | - | 10m 01s | - | 12m 42s | - | 14m 48s |
| 1000000 | 200 | - | 39m 16s | - | 49m 56s | - | 59m 22s |

**Table 3.4:** Comparison of running times between OT-based (left) and TI-based (right) aggregation protocols using 64 bit numbers. The running time of the trusted initializer, which is an offline preprocessing phase, is included. The complete results with additional parameter values can be found in Appendix 3.A.

### 3.7.4 Experiments on Real Datasets

*The results in this section are based on experiments performed by co-author Borja Balle.*

Although we have discussed the accuracy of our aggregation and solving protocols independently, we still have to evaluate our protocol in the task of building a ridge regression model. We evaluate our secure multi-party ridge regression system on 9 different regression problems selected from the UCI repository [DG17]. Each problem comes with a set of $n$ examples of some dimension $d$ which are randomly split into training (70%) and test sets (30%). Details about the names, original references where the dataset appeared, dimensions and number of examples in each task are given in Table 3.5. The dimensions of the problems range from 7 to 384, and the number of training examples ranges from over 200 to almost 3 million.

Examples in the test set are used to evaluate the predictive accuracy of the learned models in terms of their root mean squared error (RMSE). The predictive accuracy of the model will depend on the particular regularization parameter $\lambda$ used in the regression: larger values prevent overfitting and should be used on small datasets, while smaller values reduce the bias in the regression and should be used on large datasets. Hyper-parameter tuning algorithms can use a set of test examples to evaluate the predictive power obtained with different regularization parameters and find a near-optimal setting. In principle, any hyper-parameter tuning algorithm based on solving the regression multiple times with different regularization parameters could be run on top of our system at the cost of executing the protocol repeatedly. This would involve many repeated computations but it is also possible to design a variation of our protocol to avoid most of these repetitions, in particular the computation of $\boldsymbol{X}^\intercal \boldsymbol{X}$. We leave this optimization for future work and concentrate on evaluating our ridge regression protocol with a fixed regularization $\lambda$ for each problem selected a

priori using the given train and test split. Table 3.5 gives the values selected for each task in addition to the condition number of the resulting linear system.

In each case, the features were split evenly between two parties. The scalings and normalizations described in Section 3.5.3 are performed locally by each party during the aggregation phase. We use Protocol 2 for the aggregation phase, using a bit width of either 32 or 64 bits. We compare four implementations for the solving phase: Cholesky and Fixed-Point CGD, both in their 32 and 64 bits versions. In the case of CGD, we fixed the number of iterations to 20, regardless of the dimension or any other feature of the problem. For each task, we select the number of bits $b_f$ for the fractional part in order to make sure we have enough bits $b_i$ in the integer part for representing the largest coefficient appearing in the linear system $(\boldsymbol{A}, \boldsymbol{b})$. The particular settings for each problem are also given in Table 3.5.

The results are presented in Table 3.6. We give the total running time of the protocol and RMSE on the test set for each of the four different implementations of the solving phase. The RMSE of each algorithm is compared to the one obtained using an insecure ridge regression algorithm implemented in Matlab that uses 64-bit floating-point representations (given in the column headed *Optimal*). The percentages in parentheses next to each RMSE give the relative increase – or in very few cases, decrease – on the error incurred using the private algorithm. Remarkably, the differences in accuracy are almost negligible for both 64 bit implementations, while in the 32 bit implementations, CGD is better than Cholesky except in one task.

In terms of running time, we note that both bit settings observe the same pattern: Cholesky is faster for small dimensions $d \leq 100$, and CGD is faster for large dimensions $d > 100$. However, even in the cases where Cholesky is faster, CGD is not far behind (a bit over 3 minutes in the worst case). On the other hand, in the cases where CGD is faster Cholseky has prohibitive running times, e.g., CGD takes ~4h on the largest problem ($d = 384$) while Cholesky takes practically half a day.

Overall, we observe that for except for the last task, Fixed-point CGD with 32 bits always obtains relative errors below 1% and runs in less than 45 minutes, thus offering a very competitive solution. For the last dataset, that has large number of training examples, 64 bits are required to achieve good accuracies, mainly due to the errors introduced in the aggregation phase when the number of examples is large and there are not enough bits to represent the intermediate values of the computation of $\boldsymbol{X}^\intercal \boldsymbol{X}$ with sufficient precision. A possible workaround to this problem would require having a different number of bits in the aggregation and solving phases; we will consider this optimization in future work.

| id | Name | Reference | $d$ | $n$ | $\lambda$ | $\kappa(\boldsymbol{A})$ | $b_f(b=32)$ | $b_f(b=64)$ |
|---|---|---|---|---|---|---|---|---|
| 1 | Student Performance | [Cor14; CS08] | 30 | 395 | $1.4 \cdot 10^{-2}$ | $5.5 \cdot 10^{0}$ | 30 | 62 |
| 2 | Auto MPG | [Qui93a; Qui93b] | 7 | 398 | $2.2 \cdot 10^{-3}$ | $9.0 \cdot 10^{1}$ | 28 | 61 |
| 3 | Communities and Crime | [Red09; RB02] | 122 | 1994 | $2.0 \cdot 10^{-4}$ | $1.1 \cdot 10^{3}$ | 24 | 53 |
| 4 | Wine Quality | [Cor+09b; Cor+09a] | 11 | 4898 | $2.9 \cdot 10^{-3}$ | $6.8 \cdot 10^{1}$ | 29 | 60 |
| 5 | Bike Sharing Dataset | [FG13; FG14] | 12 | 17 379 | $8.2 \cdot 10^{-7}$ | $2.2 \cdot 10^{2}$ | 28 | 60 |
| 6 | Blog Feedback | [Buz14; Buz12] | 280 | 52 397 | $1.1 \cdot 10^{-5}$ | $1.3 \cdot 10^{4}$ | 25 | 54 |
| 7 | CT slices | [Gra+11b; Gra+11a] | 384 | 53 500 | $9.3 \cdot 10^{-6}$ | $1.6 \cdot 10^{4}$ | 25 | 51 |
| 8 | Year Prediction MSD | [Ber11; BEWL11] | 90 | 515 345 | $1.1 \cdot 10^{-5}$ | $1.7 \cdot 10^{2}$ | 26 | 58 |
| 9 | Gas sensor array | [FH15; FSHM15] | 16 | 4 208 261 | $4.1 \cdot 10^{-7}$ | $1.2 \cdot 10^{5}$ | 26 | 53 |

**Table 3.5:** Specifications of UCI datasets considered in our evaluation. The number of samples $n$ is split randomly into training (70%) and test sets (30%). For each dataset, the regularization parameter $\lambda$ and the number of fractional bits $b_f$ were chosen as described in Section 3.7.1. The condition number $\kappa$ was computed after data standardization and scaling (Section 3.5.3).

| id | Optimal | FP-CGD (32 bits) | | Cholesky (32 bits) | | FP-CGD (64 bits) | | Cholesky (64 bits) | |
|---|---|---|---|---|---|---|---|---|---|
| | RMSE | time | RMSE | time | RMSE | time | RMSE | time | RMSE |
| 1 | 4.65 | 19s | 4.65 (-0.0%) | 5s | 4.65 (-0.0%) | 1m 53s | 4.65 (-0.0%) | 35s | 4.65 (-0.0%) |
| 2 | 3.45 | 2s | 3.45 (-0.0%) | 0s | 3.45 (-0.0%) | 13s | 3.45 (0.0%) | 1s | 3.45 (0.0%) |
| 3 | 0.14 | 4m 27s | 0.14 (0.3%) | 4m 35s | 0.14 (-0.0%) | 24m 24s | 0.14 (0.2%) | 26m 31s | 0.14 (-0.0%) |
| 4 | 0.76 | 3s | 0.76 (-0.0%) | 0s | 0.80 (4.2%) | 23s | 0.76 (-0.0%) | 4s | 0.76 (-0.0%) |
| 5 | 145.06 | 4s | 145.07 (0.0%) | 1s | 145.07 (0.0%) | 26s | 145.06 (0.0%) | 4s | 145.06 (0.0%) |
| 6 | 31.89 | 24m 05s | 31.90 (0.0%) | 53m 24s | 32.19 (0.9%) | 2h 03m 39s | 31.90 (0.0%) | 4h 40m 23s | 31.89 (-0.0%) |
| 7 | 8.31 | 44m 46s | 8.34 (0.4%) | 2h 13m 31s | 8.87 (6.7%) | 3h 51m 51s | 8.32 (0.1%) | 11h 49m 40s | 8.31 (-0.0%) |
| 8 | 9.56 | 4m 16s | 9.56 (0.0%) | 3m 50s | 9.56 (0.0%) | 16m 43s | 9.56 (0.0%) | 13m 28s | 9.56 (0.0%) |
| 9 | 90.33 | 48s | 95.05 (5.2%) | 42s | 95.06 (5.2%) | 1m 41s | 90.35 (0.0%) | 1m 9s | 90.35 (0.0%) |

**Table 3.6:** Results of the evaluation of our system on UCI datasets. For each choice of algorithm and bit width, running time is reported, and the root mean squared error (RMSE) of the solution obtained by our system and an insecure implementation of ridge regression using floating point are compared.

## 3.8 BEYOND SEMI-HONEST SECURITY

The protocols that we discussed in Section 3.4 guarantee security in an adversarial setting where all parties, including the data providers, the CSP, and the evaluator, are semi-honest, and the CSP and the evaluator do not collude. While security against semi-honest adversaries may be enough in a federated learning setting as argued in Section 3.4.3.1, some applications require stronger guarantees.

In this section, we propose an extension of our protocol that allows the data providers to check the correctness of the result of the second phase, which is computed by CSP and evaluator. This correctness check works even if one of these two is malicious, or *actively* corrupted, i. e., they may deviate arbitrarily from the protocol description. We stress, however, that this correctness check is *not* enough to satisfy the

definition of security against malicious adversaries as given by Goldreich [Golo4]. In particular, a malicious CSP can cause the correctness check to succeed or fail depending only on one of the parties' private inputs. This *selective failure* attack has been discussed thoroughly in the context of garbled circuits [MFo6; Hua+11].

Our correctness check comes in the form of a third phase, which we call the *verification phase*. It is implemented using another small garbled circuit, which is secure against malicious adversaries. The general idea of using a tailored, lightweight verification procedure with high security guarantees to ensure correctness of a protocol with semi-honest security has been used by Hoogh, Schoenmakers, and Veeningen [HSV16] for linear programming, Laud and Pettai [LP16] for sorting, and Nikolaenko et al. [Nik+13b] for ridge regression.

### 3.8.1 The Verification Phase

First, note that for the aggregation phase, we need to use the inner product protocol based on OT (Section 3.4.1.1), since we can no longer assume that the CSP can act as a trusted initializer. Since neither the CSP nor evaluator take part in this aggregation phase, it needs no further analysis.

Nikolaenko et al. [Nik+13b] made a perceptive observation that in the setting of linear regression, the correctness of the result can be verified simply by checking that the solution $\boldsymbol{\theta}$ of the system $\boldsymbol{A\theta} = \boldsymbol{b}$ indeed minimizes the least squares expression of equation (3.1). This is done by verifying that $\boldsymbol{\theta}$ evaluates to 0 in the derivative of the least squares function in equation (3.2), which is much cheaper than computing $\boldsymbol{\theta}$ with malicious security. Concretely, this means checking that $\|2(\boldsymbol{A\theta} - \boldsymbol{b})\| = 0$. As we are working with finite-precision arithmetic, the equality check must be approximated as $\|2(\boldsymbol{A\theta} - \boldsymbol{b})\| \in [-u, u]$, for some $u$ chosen by the parties. We assume that the election of $u$ is done correctly, in the sense that, if the CSP does not deviate from the protocol, then $\|2(\boldsymbol{A\theta} - \boldsymbol{b})\| \in [-u, u]$ holds. If we consider the infinity norm, then the verification check of a solution $\boldsymbol{\theta}$, corresponds to checking

$$\forall i \in [d] : v_i \in [-u, u] \tag{3.4}$$

where $\boldsymbol{v} = \boldsymbol{A\theta} - \boldsymbol{b}$, and $\boldsymbol{A}, \boldsymbol{b}$ are additively shared among the data providers as $(\boldsymbol{A}_1, \boldsymbol{b}_1), \ldots, (\boldsymbol{A}_k, \boldsymbol{b}_k)$ as a result of the aggregation phase. We call this check the *verification phase*, which is run after the solving phase. Hence, as mentioned above, we use a semi-honest protocol for the solving phase, and then run a verification protocol with malicious security.

In our protocol for the semi-honest case of Figure 3.1, $\boldsymbol{\theta}$ is revealed to the data providers as a result of the solving phase. Consequently, an additive share of $\boldsymbol{v}$ in (3.4) can be precomputed locally by the

parties, and hence securely evaluating (3.4) only requires additions and comparisons. Using this, we implemented the verification phase as follows.

1. The data providers generate a uniformly pseudorandom vector $\boldsymbol{v}_{i,1} \in \mathbb{Z}_q^d$ and locally compute $\boldsymbol{v}_{i,2} = \boldsymbol{A}_i\boldsymbol{\theta} - \boldsymbol{b}_i - \boldsymbol{v}_{i,1}$. Then, they send $\boldsymbol{v}_{i,1}$ to the CSP and $\boldsymbol{v}_{i,2}$ to the evaluator.

2. CSP and evaluator add up their received shares and obtain $\boldsymbol{v}_1 = \sum_{i=1}^k \boldsymbol{v}_{i,1}$ and $\boldsymbol{v}_2 = \sum_{i=1}^k \boldsymbol{v}_{i,2}$, and then run a two-party garbled circuit protocol with malicious security. In the circuit, $\boldsymbol{v} = \boldsymbol{v}_1 + \boldsymbol{v}_2$ is recovered, and a single bit is returned, indicating whether (3.4) holds.

3. CSP and evaluator send the result of the verification circuit to all data providers.

Note that since not both CSP and evaluator cannot be malicious at the same time, the parties only need to check if both bits they received are equal to 1.

We implemented and evaluated the garbled circuit with malicious security for this verification phase, as an extension for our solving phase, using the EMP framework [WMK16]. As expected, it runs extremely fast: in our experiments, garbling and execution took less than 3 seconds for $d \leq 500$. This is in contrast with the time needed for the solving phase for $d = 500$: 30 minutes, for 10 iterations of CGD and a bit width of 32 bits (see Figure 3.3, left).

## 3.9 DISCUSSION

In this chapter, we provided a first example of a secure machine learning model trained on distributed data. We focused on linear regression as a task that is widely used in practical applications. Beyond the settings described in this chapter, our implementation of secure conjugate gradient descent with fixed-point arithmetic and early stopping can also be used to deal with non-linear regression problems based on kernel ridge regression, given MPC protocols for evaluating kernel functions typically used in machine learning. Moreover, as mentioned in Section 3.4, an extensive evaluation of MPC techniques for the task of linear system solving, including our conjugate gradient descent algorithm, is an interesting continuation of the work presented here. From a more theoretical perspective, the problem of providing security guarantees against malicious adversaries for approximate MPC functionalities poses interesting open challenges both in general and from the perspective of concrete machine learning tasks.

One assumption implicitly made in this chapter is that the training dataset is given as a matrix of real numbers. However, in practice,

many datasets will not have this clear structure, but instead come as unstructured documents such as text or images. In this case, the inputs first need to be converted into a numerical representation by an appropriate feature extraction phase. In the next chapter, we will shift our focus to this problem in the context of similarity computations on text documents.

# CHAPTER APPENDIX

## 3.A FURTHER EXPERIMENTAL RESULTS

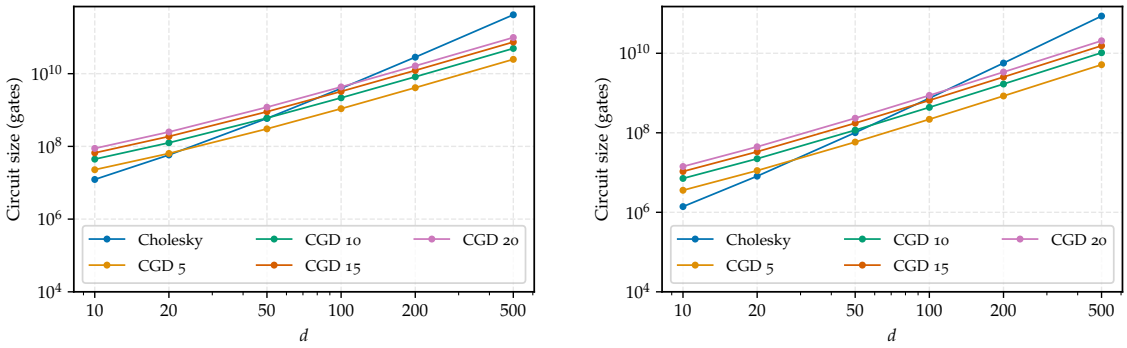



**Figure 3.7:** Circuit sizes for 64 bits (left), 32 bits (right) as a function of the input dimensionality $d$. One can see a clear correlation between the circuit size and the execution time shown in Figure 3.3 (left).

| $n$ | $d$ | Number of parties (data providers) | | | | | |
|---|---|---|---|---|---|---|---|
| | | 2 | | 3 | | 5 | |
| | | $b = 64$ | $b = 32$ | $b = 64$ | $b = 32$ | $b = 64$ | $b = 32$ |
| 10000 | 20 | 24s | 13s | 20s | 11s | 14s | 8s |
| | 50 | 2m 20s | 1m 16s | 1m 55s | 1m 02s | 1m 22s | 43s |
| | 100 | 9m 06s | 4m 58s | 7m 18s | 3m 55s | 5m 13s | 2m 47s |
| 20000 | 20 | 45s | 24s | 38s | 20s | 29s | 15s |
| | 50 | 4m 26s | 2m 18s | 3m 39s | 1m 56s | 2m 39s | 1m 21s |
| | 100 | 17m 25s | 9m 58s | 14m 09s | 7m 24s | 10m 10s | 5m 14s |
| 50000 | 20 | 1m 50s | 56s | 0m 32s | 47s | 1m 07s | 35s |
| | 50 | 10m 47s | 5m 27s | 8m 58s | 4m 37s | 6m 31s | 3m 12s |
| | 100 | 42m 12s | 21m 25s | 34m 39s | 17m 41s | 24m 58s | 12m 32s |
| 100000 | 20 | 3m 40s | 1m 50s | 3m 01s | 1m 33s | 2m 18s | 1m 07s |
| | 50 | 21m 40s | 10m 47s | 17m 25s | 9m 06s | 13m 05s | 6m 40s |
| | 100 | 1h 25m 14s | 42m 18s | 1h 07m 30s | 35m 03s | 50m 16s | 24m 12s |
| 200000 | 20 | 7m 25s | 3m 40s | 5m 57s | 3m 04s | 4m 46s | 2m 18s |
| | 50 | 43m 47s | 21m 43s | 34m 14s | 18m 26s | 26m 47s | 12m 40s |
| | 100 | 2h 52m 08s | 1h 25m 06s | 2h 11m 35s | 1h 10m 39s | 1h 43m 12s | 49m 23s |
| 500000 | 20 | 18m 18s | 9m 10s | 14m 29s | 7m 27s | 12m 10s | 5m 52s |
| | 50 | 1h 47m 59s | 54m 13s | 1h 23m 49s | 45m 00s | 1h 08m 08s | 32m 14s |
| | 100 | 7h 03m 56s | 3h 32m 37s | 5h 20m 52s | 2h 53m 54s | 4h 17m 08s | 2h 04m 53s |

**Table 3.8:** Computation time of the aggregation phase using using the OT-based inner product protocol, for 2, 3, and 5 data providers and different values of $n$ (number of records) and $d$ (number of features). Note that unlike the TI-based protocol in Table 3.9, this algorithm scales well with the number of parties.

| $n$ | $d$ | Number of parties (data providers) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 2 | | | 3 | | | 5 | | |
| | | CSP | DP | Total | CSP | DP | Total | CSP | DP | Total |
| 100000 | 20 | 1s | 2s | 3s | 2s | 2s | 3s | 2s | 1s | 4s |
| | 50 | 7s | 9s | 14s | 9s | 8s | 17s | 11s | 6s | 20s |
| | 100 | 27s | 31s | 50s | 36s | 28s | 1m 04s | 43s | 21s | 1m 15s |
| | 200 | 1m 46s | 1m 57s | 3m 12s | 2m 22s | 1m 44s | 4m 10s | 2m 50s | 1m 17s | 4m 54s |
| | 500 | 10m 52s | 11m 53s | 19m 14s | 14m 40s | 10m 29s | 25m 16s | 17m 29s | 7m 37s | 30m 02s |
| 200000 | 20 | 2s | 4s | 6s | 3s | 4s | 7s | 4s | 3s | 8s |
| | 50 | 14s | 19s | 30s | 18s | 17s | 37s | 22s | 13s | 42s |
| | 100 | 55s | 1m 06s | 1m 47s | 1m 11s | 59s | 2m 16s | 1m 26s | 44s | 2m 39s |
| | 200 | 3m 37s | 4m 13s | 7m 00s | 4m 44s | 3m 43s | 9m 00s | 5m 38s | 2m 43s | 10m 32s |
| | 500 | 21m 46s | 25m 33s | 41m 47s | 29m 11s | 22m 23s | 54m 21s | 34m 56s | 16m 06s | 1h 04m 00s |
| 500000 | 20 | 6s | 10s | 15s | 8s | 9s | 18s | 10s | 8s | 21s |
| | 50 | 36s | 48s | 1m 17s | 47s | 42s | 1m 35s | 57s | 32s | 1m 51s |
| | 100 | 2m 17s | 2m 51s | 4m 47s | 3m 04s | 2m 29s | 6m 01s | 3m 42s | 1m 51s | 6m 58s |
| | 200 | 9m 00s | 10m 55s | 18m 52s | 12m 15s | 9m 33s | 24m 08s | 14m 47s | 6m 52s | 27m 43s |
| | 500 | 56m 22s | 1h 06m 41s | 1h 53m 52s | 1h 14m 37s | 57m 31s | 2h 25m 55s | 1h 30m 41s | 40m 56s | 2h 50m 53s |
| 1000000 | 20 | 12s | 20s | 31s | 16s | 18s | 37s | 20s | 15s | 44s |
| | 50 | 1m 12s | 1m 34s | 2m 40s | 1m 37s | 1m 24s | 3m 21s | 1m 58s | 1m04s | 3m 53s |
| | 100 | 4m 45s | 5m 40s | 10m 01s | 6m 16s | 4m 58s | 12m 42s | 7m 31s | 3m 41s | 14m 48s |
| | 200 | 18m 49s | 21m 44s | 39m 16s | 25m 07s | 18m 48s | 49m 56s | 30m 28s | 13m 48s | 59m 22s |

**Table 3.9:** Computation time of the aggregation phase using the TI-based inner product protocol with 64 bits, for 2, 3, and 5 data providers and different values of $n$ (number of records) and $d$ (number of features). For each number of data providers, computation time of the CSP (left) and the data providers (middle, averaged) is reported, as well as total running time (right).

# 4 SECURE AND SCALABLE DOCUMENT SIMILARITY

*The contents of this chapter have previously appeared in* Proceedings on Privacy Enhancing Technologies 2020. *[SVGB20].*

## 4.1 OVERVIEW

In this chapter, we focus on secure similarity computations on text documents. This is a challenging task for two reasons. First, unlike in Chapter 3, we cannot assume any more that our data comes in a "nice" structured form. Instead, we have to perform *feature extraction* on the data, in order to turn unstructured text documents into feature vectors over the real numbers. The second challenge comes from the fact that common feature representations result in very high-dimensional vectors. If we performed feature extraction using standard MPC security with secret-shared outputs, this would result in extremely poor scalability for any subsequent computation on the computed features.

Instead we're going to use a relaxed security notion for MPC, where the parties are allowed to obtain some information beyond the final computation result, while *provably quantifying and limiting* the incurred privacy leakage. For this, differential privacy (DP) [DR14; DMNS16] is a natural candidate, and multiple recent works explore variants of *MPC with differentially private leakage* [MG18; GRR19; HMFS17]. Protocols under such relaxed definitions may reveal additional information, as long as that information leakage adheres to DP.

We follow a similar approach, applying MPC with DP leakage to secure classification of text documents with the $k$-nearest neighbors ($k$-NN) algorithm on a standard *term frequency–inverse document frequency* (TF-IDF) feature representation. In $k$-NN, a query document is assigned a class by taking a majority vote among the $k$ most similar documents in the training database. Despite its simplicity, $k$-NN enjoys remarkable theoretical properties [CD14] and provides competitive accuracies in a wide range of applications [Efr17]. This power comes at the price of scalability: due to its non-parametric nature $k$-NN requires similarity computations against the whole dataset at prediction time. It is therefore crucial to reduce the time for each similarity computation as much as possible. Our main observation is that if we allow a one-time precomputation of differentially private statistics about the distributed dataset, we can significantly speed up classification time by using a novel sparse inner product protocol. We review our contributions and how they compose to a full $k$-NN protocol in the following subsection.

### 4.1.1 Chapter Contributions

The core of this chapter consists of two contributions: (i) a secure two-party protocol for sparse inner products, and (ii) a mechanism for extracting differentially private IDF coefficients from text documents, and a corresponding two-party implementation. Both protocols are tailored to exploit inherent sparsity and word frequency properties commonly found in text data. These are composed to obtain a three-party protocol for distributed $k$-NN classification capable of withstanding arbitrary collusions. All the protocols presented in this chapter are formally secure in the semi-honest model (Definition 2.1).

**SECURE DOCUMENT SIMILARITY FROM SPARSE INNER PRODUCTS (SECTION 4.4).** Our first contribution is a novel protocol for secure sparse inner product. It allows two parties holding private sparse vectors to compute additive shares of their inner product, while revealing nothing except an upper bound on the number of non-zero entries. As described in Section 4.4, this can be used to compute similarities between sparse representations of text documents. We also propose a batched version of our protocol to improve the scalability of computing many inner products in parallel. In Section 4.7.1.1, we experimentally evaluate the running time of our protocol for a wide range of parameters, and show that it outperforms its state-of-the-art dense counterparts by at least one order of magnitude.

**DIFFERENTIALLY PRIVATE IDF COEFFICIENTS (SECTION 4.5).** Secondly, we develop a mechanism for extracting inverse document frequency (IDF) coefficients from a distributed database of text documents, while guaranteeing differential privacy (DP). We show why a standard approach based on adding Laplace noise to each coefficient fails at this task, and formally prove privacy and accuracy guarantees for our custom mechanism. While our proposal is already of interest in the centralized setting, we further show how to instantiate it for generic circuit-based MPC, thus achieving multi-Party computational differential privacy (MPC-DP) [BNO08]. To that end, we rely on a method for oblivious sampling without replacement that has, to the best of our knowledge, not been reported in the academic literature before. Our experiments (Section 4.7.2.4) showcase the advantage of having access to privatized data-dependent IDFs over a vanilla data-independent term frequency (TF) representation, and demonstrate that the noise introduced by our DP protocol incurs only a small accuracy loss. Finally, we implement our protocol for the special case of two parties holding a databases of secret documents and show that it scales to real-world vocabulary sizes (Section 4.7.1.2).

**APPLICATION TO SECURE $k$-NN (SECTION 4.6).** While both of the above are of independent interest, we show how in combination

they allow us to implement an efficient $k$-Nearest Neighbors protocol in a three-party setting with two servers and one client. Here, the two servers each hold a collection of labeled documents, and the client would like to classify a document against the union of the servers' datasets. Our protocol achieves this by combining the two sub-protocols for IDF precomputation and secure inner products with a generic MPC phase for top-$k$ selection. Apart from the final classification, our full protocol releases differentially private statistics about the dataset (i. e., IDF coefficients) as a one-time precomputation. We formalize this as *differentially private leakage* similar to previous work [GRR19; MG18] and prove security of our protocol in that model. We emphasize that these differentially private IDF coefficients need to be computed only once, and can then be reused in any subsequent classifications.

We implement our $k$-NN protocol and show that it scales to real-world dataset sizes. For example, the time needed for a classification of a query document against a database of 28K documents is less than 40 minutes.

## 4.2 RELATED WORK

Several recent works have proposed MPC protocols for machine learning tasks such as linear and logistic regression training [Nik+13b; MZ17], neural network training [MZ17; ASKG19] and evaluation [BNO08; LJLA17; JVC18], matrix factorization [Nik+13a], principal component analysis [AWCK17], as well as evaluation of decision trees and naive Bayes classifiers [BPTG15]. However, none of them exploit the input distribution for efficiency. This is in contrast with computation in the clear, where dedicated algorithms and data structures have been developed for different kinds of sparsity patterns. Moreover, all of these protocols assume that feature extraction has been already performed. This is reasonable for settings where that step can be computed locally by each party. However, as in the case of TF-IDF, several powerful feature extraction techniques and normalization steps may require data held by different parties.

Regarding our more concrete contributions, several secure 2PC protocols for matrix multiplication have been recently proposed, including SecureML [MZ17], GAZELLE [JVC18], and our protocols from Section 3.4.1. These protocols operate over explicit matrix or vector representation, and thus are not tailored to exploit sparsity. On the other hand, combinations of MPC with DP have been proposed before in the context of limiting leakage of access patterns in secure computation [MG18], private set intersection [GRR19], and protocols for private record linkage (see [HMFS17] and references therein). He et al. [HMFS17] use an indistinguishability-based definition that is

limited to deterministic functionalities. In contrast, we define security with DP leakage in the simulation-based paradigm that is also used by [MG18; GRR19], but unlike these works we allow the output of the final computation to depend on the leakage. The advantage of a simulation-based definition is the fact that it allows for straightforward composition, which we use in our security proofs.

Regarding our application in private text analysis, related work can be found in the context of *similar document detection* [JMCS08; Mur+10; BCG14; BB13]. Another trend of related work is in *privacy-preserving nearest neighbors computation* [Ria+16; LSP15; RWLX16; Che+20]. However, a remarkable difference between these existing works and ours is in the threat model. In all the contributions mentioned above, either the computation is delegated to two *non-colluding* parties – sometimes referred to as the *two-server model* in MPC – or only involves two parties (for example, one server and one client). In contrast, our threat model allows the database to be distributed among multiple servers who might collude with each other or the client. We also stress that we are the first to even consider the feature extraction phase as part of a distributed $k$-NN protocol. Previous work starts directly with feature vectors as inputs and therefore implicitly assumes feature extraction can be done locally by the parties, which is not the case for TF-IDF. Thus, the fact that our protocol releases differentially private IDF values in fact strengthens the privacy guarantees compared to the previous works mentioned above.

## 4.3 BACKGROUND: TF–IDF FEATURES

Throughout this chapter, we work on *text document data*. Before being able to process this data it is necessary to construct a vectorial representation for each document. Here, we rely on the *term frequency–inverse document frequency* (TF-IDF) feature representation, which is one of the most common encodings for text data. For example, 83% of text-based document search and recommendation systems in digital libraries use TF-IDF [BGLB16].

The TF-IDF feature representation of a text document is defined with respect to a fixed vocabulary and a database of documents. Let $\mathcal{V}$ be a fixed vocabulary – e.g., $\mathcal{V}$ might be all the words in a given dictionary – and consider a database $Z$ of documents over the common vocabulary $\mathcal{V}$. Given an arbitrary document $x$ with words in $\mathcal{V}$, its TF-IDF representation is a $|\mathcal{V}|$-dimensional vector $\psi(x) \in \mathbb{R}^{\mathcal{V}}$ where each coordinate corresponds to a word $v \in \mathcal{V}$. The $v$th coordinate $\psi(x)(v)$ of this vector is the product of two terms: the *term frequency* (TF) $\phi_{\mathrm{tf}}(v, x) = |x|_v$ of $v$ in $x$ (i.e., the number of times $v$ occurs in $x$) and the *inverse document frequency* (IDF) $\phi_{\mathrm{idf}}(v, Z) = \log((|Z| + 1)/(|Z|_v + 1)) + 1$ of $v$ in $Z$ (where $|Z|_v$ counts the number of documents in the

database that contain the word $v$). By taking the product $\psi(x)(v) = \phi_{\text{tf}}(v, x) \cdot \phi_{\text{idf}}(v, Z)$, the TF-IDF representation ensures that coordinates corresponding to frequent words in the document are larger (TF term), while also down-weighting words that are frequent across the dataset (IDF term) and therefore not representative of a particular document.

Two properties of TF-IDF are particularly relevant to this work. First, since the TF component is zero for all words that do not appear in a document, TF-IDF vectors are very sparse. In Section 4.4, we use this fact to efficiently compute similarity scores between TF-IDF vectors of documents. Second, observe that the IDF component depends on the whole dataset, not only the document being encoded. It is therefore non-trivial to encode a document without access to the entire database. However, we will see in Section 4.5 that we can precompute differentially private IDF coefficients, which then can be safely released to allow parties to locally compute TF-IDF embeddings of their documents.

## 4.4 SPARSE INNER PRODUCTS AND DOCUMENT SIMILARITY

Similarity computation is a common task in the evaluation of non-parametric models such as $k$-NN, but also for example in information retrieval, clustering, and collaborative filtering. Common similarity metrics include the Euclidean distance, Pearson's correlation coefficient, or the cosine similarity. All three of these can be computed in two-party settings using only secure inner products. Here, we focus on cosine similarity, which is commonly used for text documents with TF-IDF features [MRS08]:

$$\text{sim}_{\cos}(\boldsymbol{a}, \boldsymbol{b}) = \frac{\langle \boldsymbol{a}, \boldsymbol{b} \rangle}{\|\boldsymbol{a}\| \|\boldsymbol{b}\|}.$$

When each of the feature vectors $\boldsymbol{a}, \boldsymbol{b}$ is held by one of the two parties, the denominator can be computed using a single secure multiplication [Bea91; Gil99], while the numerator requires a secure inner product. As described in Section 4.3, the feature vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ are sparse when encoding text documents using TF-IDF. Hence, in this section, we present a secure two-party protocol for inner products that is optimized for *sparse* inputs.

Our protocol takes as input private vectors $\boldsymbol{a}$ and $\boldsymbol{b}$—each provided by one party—and compute an *additive secret share* of their inner product $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$. Here, the elements of $\boldsymbol{a}$ and $\boldsymbol{b}$ are taken from $\mathbb{Z}_q$, where $q$ is usually chosen as $2^\sigma$ for some bit width $\sigma$. Rational numbers can be used by relying on an appropriate fixed-point encoding (See section 3.5 and Mohassel and Zhang [MZ17]). Since in practice, many similarity scores need to be computed at once, we generalize our protocol and

propose a *batched* version for computing multiple inner products simultaneously; in this case the goal is to compute $\boldsymbol{AB}$ given private matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ which are respectively row and column sparse. Since the single inner product case corresponds to the multiplication of a one-row by a one-column matrix, for simplicity we sometimes use matrix notation to denote properties that apply to both the single and batched inner product case.

As we have seen in Section 3.4.1.1, a secure inner product in the dense case can be realized in multiple ways. Further optimizations and alternative approaches based on homomorphic encryption are presented by Mohassel and Zhang [MZ17], and Juvekar, Vaikuntanathan, and Chandrakasan [JVC18]. For our generalization to sparse matrix multiplication (Section 4.4.5) we use the dense protocol of Mohassel and Zhang [MZ17], which is a variant of the protocol from Section 3.4.1.1 optimized for the batched setting. While our protocol neither reveals the non-zero indexes in each party's inputs, nor their values, we require an upper bound on the *number* of non-zeros to be known. In the next subsection, we will see that such a bound on the sparsity is naturally a available in many real-world scenarios, including text classification.

### 4.4.1 Sparsity in Real-World Data

In the context of data analysis, sparsity is frequently induced by the feature representation used. For example, the well-known bag-of-word representations of documents, where a document is represented as a vector of (possibly normalized) word counts, is sparse. The TF-IDF representation is a special case of this. Furthermore, in datasets of genomic variants, individuals are represented as a set of deviations (index-value pairs) from a common reference. Compared to the 3.2 billion base pairs in the human reference genome, these deviations only make up a small fraction (~5M sites [The+15]), and so this representation is also very sparse. A third example can be found in recommender systems, where users are represented by the vector of their rated items, which again amount to a tiny percentage of the total number of available items.

In all of the above applications, an upper bound on the sparsity is readily available. For genomics, upper bounds can be derived from public information about the distribution of variants [The+15]. In recommender systems, the input comes in the form of a list of (item, rating) pairs for each user. The length of a list directly translates to the sparsity of the corresponding vector. We note that existing recommendation protocols based on secure matrix factorization reveal the sparsity as the input size [Nik+13a; Nay+15]. Finally, for text documents, the input length does not directly give the exact sparsity

(words can be repeated in a document), but it provides an upper bound on it.

Our protocol can therefore be applied to all of the settings described here. We now formally describe secret sharing and introduce some notation, before we present our protocol for sparse inner product in Section 4.4.3.

### 4.4.2 Notation

For any sparse vector $\boldsymbol{v}$, we write $\mathcal{I}_{\boldsymbol{v}}$ to denote the set of indexes where $\boldsymbol{v}$ is non-zero, and call its cardinality $l_{\boldsymbol{v}} := |\mathcal{I}_{\boldsymbol{v}}|$. We further denote the $k$-th element of this set (using canonical ordering) by $(\mathcal{I}_{\boldsymbol{v}})_k$.

For a matrix $\boldsymbol{M}$, we write $\mathrm{Col}_i(\boldsymbol{M})$ to denote the $i$-th column vector of $\boldsymbol{M}$, and $\mathrm{Row}_j(\boldsymbol{M})$ for the $j$-th row. In analogy to vectors, we write $\mathcal{I}_{\boldsymbol{M}}^{\mathrm{Col}} := \{i \mid \mathrm{Col}_i(\boldsymbol{M}) \neq \boldsymbol{0}\}$ for the set of indexes corresponding to non-zero columns of $\boldsymbol{M}$, likewise for rows.

### 4.4.3 Secure Sparse Inner Products

We now present our protocol for computing a sparse inner product. This serves as a stepping stone towards our *batched* variant, which we introduce in Section 4.4.5 and which is the one that we use in practice.

Here, we consider two sparse vectors $\boldsymbol{a}, \boldsymbol{b}$, where $\boldsymbol{a}$ is owned by Party 1 and $\boldsymbol{b}$ by Party 2. The goal of our protocol is to compute additive shares $\boldsymbol{c}_1, \boldsymbol{c}_2$, such that $\boldsymbol{c}_1 + \boldsymbol{c}_2 = \langle \boldsymbol{a}, \boldsymbol{b} \rangle$. To develop intuition, let us first discuss an insecure solution where the parties reveal to each other their respective non-zero indexes $\mathcal{I}_{\boldsymbol{a}}$ and $\mathcal{I}_{\boldsymbol{b}}$. Then each party can simply compute locally the set of common non-zero indexes and construct vectors $\tilde{\boldsymbol{a}}, \tilde{\boldsymbol{b}}$ of shorter length $|\mathcal{I}_{\boldsymbol{a}} \cap \mathcal{I}_{\boldsymbol{b}}|$ containing only the values of common indexes in some canonical order, such that $\langle \tilde{\boldsymbol{a}}, \tilde{\boldsymbol{b}} \rangle = \langle \boldsymbol{a}, \boldsymbol{b} \rangle$. Then the inner product of $\tilde{\boldsymbol{a}}$ and $\tilde{\boldsymbol{b}}$ can then be computed using a standard non-sparse MPC protocol.

While this insecure approach would greatly increase efficiency by exploiting the sparsity of $\boldsymbol{a}$ and $\boldsymbol{b}$, it is also clear that it leaks too much, as $\mathcal{I}_{\boldsymbol{a}}$ and $\mathcal{I}_{\boldsymbol{b}}$ are private. Instead, the solution we propose avoids this leakage while retaining the efficiency of this insecure solution by assuming upper bounds on $|\mathcal{I}_{\boldsymbol{a}}|$ and $|\mathcal{I}_{\boldsymbol{b}}|$ are public.

For clarity, we now describe our solution assuming the availability of a trusted third party. The parties first create vectors $\hat{\boldsymbol{a}}, \hat{\boldsymbol{b}}$ containing the values at indexes in $\mathcal{I}_{\boldsymbol{a}}$ and $\mathcal{I}_{\boldsymbol{b}}$, respectively, padded with zeroes to length $l_{\boldsymbol{a}} + l_{\boldsymbol{b}}$. Then, the third party receives $\hat{\boldsymbol{a}}$ and $\hat{\boldsymbol{b}}$ from the parties, and returns permutations $\pi_1$ and $\pi_2$ to Party 1 and Party 2, respectively, such that $\langle \pi_1(\hat{\boldsymbol{a}}), \pi_2(\hat{\boldsymbol{b}}) \rangle = \langle \boldsymbol{a}, \boldsymbol{b} \rangle$. Note that padding with zeroes is crucial so that such permutations exist, as intuitively all they have to do is make indexes in $\mathcal{I}_{\boldsymbol{a}} \cap \mathcal{I}_{\boldsymbol{b}}$ end up in the same position in both permuted vectors, while all others get matched to a zero. Now, if

**Functionality 5:** Correlated permutations ($\mathcal{F}^{\text{Perm}}$)

**Parties:** 1, 2.
**Input:** Party 1: $\mathcal{I}_1$, Party 2: $\mathcal{I}_2$,
Public parameters: $l_1 = |\mathcal{I}_1|$, $l_2 = |\mathcal{I}_2|$.
**Output:** Permutations $\pi_1$ and $\pi_2$.

(1) Choose a permutation $\pi$ of $\{1, \ldots, l_1 + l_2\}$ uniformly at random.

(2) Compute the function

$$\rho : \{1, \ldots, l_1\} \to \{1, \ldots, l_1 + l_2\},$$
$$\rho(i) = \begin{cases} \pi(j) & \text{if } (\mathcal{I}_1)_i = (\mathcal{I}_2)_j, \\ \pi(l_2 + i) & \text{if no such } j \text{ exists.} \end{cases}$$

(3) Extend $\rho$ to a random permutation $\pi_1$ of $\{1, \ldots, l_1 + l_2\}$ by mapping all elements $i > l_1$ to uniformly random unmapped elements of its codomain.

(4) Output $\pi_1$ to Party 1 and $\pi_2 = \pi$ to Party 2.

the permutations $\pi_i$ are such that they do not reveal anything about $\hat{\boldsymbol{a}}$ or $\hat{\boldsymbol{b}}$ to the parties, this protocol would be secure.

While we relied on a trusted third party for explanatory purposes, such a third party is not available in practice. Instead, we will now present a 2PC-sub-protocol that replaces such a party, i. e., it generates *correlated* permutations $\pi_1, \pi_2$ with the required property. We call this functionality $\mathcal{F}^{\text{Perm}}$ and describe it formally in the next section.

### 4.4.4 Secure Correlated Permutations

Note that we only require the output to each party alone to be a uniformly random permutation. Thus, $\mathcal{F}^{\text{Perm}}$ can generate one of the permutations randomly, and derive the other from it. A detailed description of $\mathcal{F}^{\text{Perm}}$ is given in Functionality 5.

By the construction of $\rho$ in Step (2), it is clear that the condition from Figure 4.1, that matching indexes get mapped to the same position, holds for $\pi_1$ and $\pi_2$. We now prove that the outputs to both parties are indeed random permutations.

**Theorem 4.1.** *For any input sets $\mathcal{I}_1$ and $\mathcal{I}_2$ of size $l_1$ and $l_2$, and any party $i \in \{1, 2\}$, $\pi_i = \mathcal{F}_i^{\text{Perm}}(\mathcal{I}_1, \mathcal{I}_2)$ is a uniformly random permutation of $\{1, \ldots, l_1 + l_2\}$.*

*Proof.* If $i = 1$, then by the definition in Step (2) in Functionality 5, $\rho$ is constructed by selecting $l_1$ different mappings from a uniformly

random permutation. Clearly, each of the $(l_1 + l_2)! \,/\, l_2!$ possible functions is selected with equal probability. The extension of $\rho$ in Step (3) reduces to selecting a uniformly random permutation of $l_2$ elements. Thus, our functionality produces $(l_1 + l_2)! \,/\, l_2! \cdot l_2! = (l_1 + l_2)!$ different permutations, each with equal probability. Together with the observation that there are exactly $(l_1 + l_2)!$ *possible* permutations of $[l_1 + l_2]$, the claim follows. If $i = 2$, the claim follows immediately from Step (1) and $\pi_2 = \pi$. □

We use Yao's garbled circuit protocol (see Section 2.5) to implement $\mathcal{F}^{\text{Perm}}$. Our circuit design for this functionality is inspired by the private set intersection (PSI) protocol of Huang, Evans, and Katz [HEK12], also known as the Sort-Compare-Shuffle approach to PSI. Essentially, our circuit computes the set $\mathcal{I}_1 \cap \mathcal{I}_2$ in $O((l_1 + l_2)\log^2(l_1 + l_2))$—using Batcher's sorting network [Bat68]—as a Boolean array of length $l_1 + l_2$, and constructs $\rho$ and $\pi_1$ from it using a permutation network [Wak68] of size $O((l_1 + l_2)\log(l_1 + l_2))$. Thus, our protocol for the $\mathcal{F}^{\text{Perm}}$ functionality runs in $O((l_1 + l_2)\log^2(l_1 + l_2))$, and exploits efficient implementations of sorting/permutation networks.

*In Section 5.3.2, we will present alternative constructions that are more efficient in many cases.*

Since we operate in the semi-honest model, we can further employ the following optimizations:

1. Instead of choosing $\pi$ inside the Garbled Circuit, we let Party 2 choose it locally and use it as an input. Note that this is trivially simulatable since $\pi = \pi_2$ is Party 2's output.

2. Similarly, we reveal $\rho$ to Party 1 and let it perform Step (3) locally. Again, simulating $\rho$ it trivial by restricting $\pi_1$ to $\{1, \ldots, l_1\}$.

Together with the security of Yao's protocol in the semi-honest model [LP09], security of our implementation of $\mathcal{F}^{\text{Perm}}$ follows.

An important observation is that the cost of $\mathcal{F}^{\text{Perm}}$ is independent of the range of the values in the vectors $u, v$ in the overall inner product protocol. In the next section, we show how this, and previous observations, extend to matrix multiplications for batched inner products, and provide formal proofs of correctness and security for our two-party protocols.

### 4.4.5 From Inner Products to Sparse Matrix Multiplication

We will now consider computing additively shared matrix products $[\![\boldsymbol{C}]\!]_1 + [\![\boldsymbol{C}]\!]_2 = \boldsymbol{AB}$, where the matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ are owned by Party 1 and 2 respectively. We assume these matrices exhibit the sparsity patterns corresponding to batched sparse inner products, i. e., $\boldsymbol{A}$ has many zero columns, and $\boldsymbol{B}$ many zero rows. This is a common sparsity pattern in machine learning computations, as, for example, TensorFlow has a dedicated matrix representation for this case, called *IndexedSlices* [Aba+16].

**Protocol 6:** Dense matrix multiplication protocol.

**Parties:** 1, 2.
**Input:** Party 1: $\boldsymbol{A} \in \mathbb{Z}_q^{l \times m}$
Party 2: $\boldsymbol{B} \in \mathbb{Z}_q^{m \times n}$
**Output:** Party $i$ : $[\![\boldsymbol{C}]\!]_i \in \mathbb{Z}_q^{l \times n}$, s. t. $\boldsymbol{C} = \boldsymbol{AB}$

(1) Party 1 computes $\boldsymbol{U}, [\![\boldsymbol{UV}]\!]_1 \leftarrow \mathcal{F}^{\text{Off}}(l, m, n)$ and sends $\boldsymbol{E} = \boldsymbol{A} - \boldsymbol{U}$
(2) Party 2 computes $\boldsymbol{V}, [\![\boldsymbol{UV}]\!]_2 \leftarrow \mathcal{F}^{\text{Off}}(l, m, n)$ and sends $\boldsymbol{F} = \boldsymbol{B} - \boldsymbol{V}$
(3) Party 1 sets $[\![\boldsymbol{C}]\!]_1 = \boldsymbol{EF} + \boldsymbol{UF} + [\![\boldsymbol{UV}]\!]_1$
(4) Party 2 sets $[\![\boldsymbol{C}]\!]_2 = \boldsymbol{EV} + [\![\boldsymbol{UV}]\!]_2$;

#### 4.4.5.1 Baseline: Secure Dense Matrix Multiplication

As before, our goal is to improve on the baseline case where the sparsity is ignored. The state of the art in terms of (dense) secure matrix multiplication is presented in Mohassel and Zhang [MZ17]. Similar to our TI-based inner product protocol (Protocol 2), their protocol is split into two phases: an offline phase that provides correlated random matrices to the parties, and an online phase that uses said randomness to securely multiply secret-shared input matrices. For the offline phase, Mohassel and Zhang propose three different approaches: One based on a trusted initializer (like Protocol 2), one based on homomorphic encryption, and one based on oblivious transfer (like Protocol 1). In Protocol 6 we refer to this offline phase as $\mathcal{F}^{\text{Off}}$, and we use the OT-based variant in our experiments.

#### 4.4.5.2 Our Protocol

A naive solution to generalize our protocol from section 4.4.3 to sparse matrices is to run it $|\boldsymbol{AB}|$ times. In the rest of this section we focus on improving this by leveraging the sparsity patterns in the matrices to avoid the quadratic cost of the naive solution. The advantage of our solution is not only in asymptotic cost, but is also confirmed experimentally in Section 4.7, using both random and real-world sparsity patterns.

The entire sparse matrix multiplication protocol is depicted in Figure 4.1, and goes as follows. Party 1 locally computes $\mathcal{I}_{\boldsymbol{A}}^{\text{Col}} := \{i \mid \text{Col}_i(\boldsymbol{A}) \neq \boldsymbol{0}\}$, and Party 2 computes $\mathcal{I}_{\boldsymbol{B}}^{\text{Row}} := \{j \mid \text{Row}_j(\boldsymbol{B}) \neq \boldsymbol{0}\}$. These index sets are then used as inputs to the functionality for generating correlated permutations, $\mathcal{F}^{\text{Perm}}$, which generates two correlated permutations $\pi_1, \pi_2$ that map elements of $\mathcal{I}_{\boldsymbol{A}}^{\text{Col}} \cap \mathcal{I}_{\boldsymbol{B}}^{\text{Row}}$ to the same indexes. Note that, up to this point, the secure computation is independent of

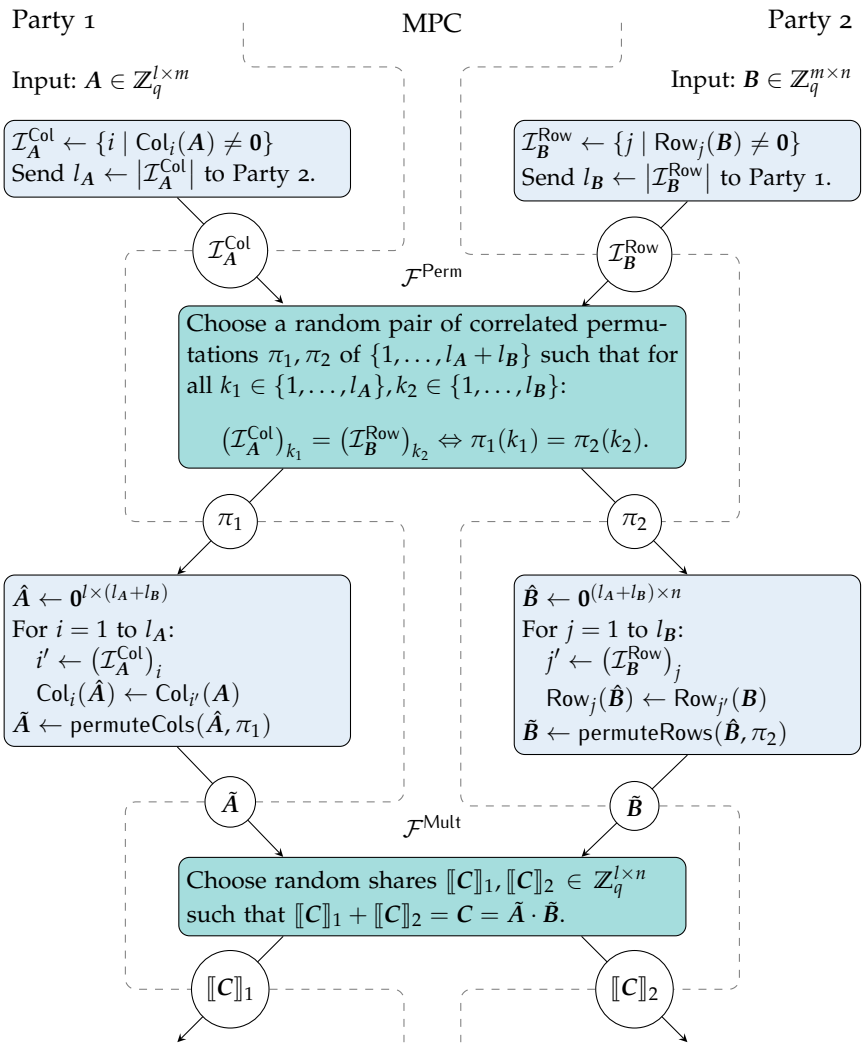



**Figure 4.1:** Secure sparse matrix multiplication. For details on the implementations of $\mathcal{F}^{\text{Perm}}$and $\mathcal{F}^{\text{Mult}}$, see Sections 4.4.4 and 4.4.5.1, respectively.

the inner (resp. outer) dimension of $\boldsymbol{A}$ (resp. $\boldsymbol{B}$). This has an important effect on efficiency, and follows from the remark above about $\mathcal{F}^{\text{Perm}}$ in the inner product protocol being independent of the domain size of the values in the vectors. In fact, intuitively we can think of multiplying $\boldsymbol{A}$ and $\boldsymbol{B}$ as computing sparse inner product where values are vectors, instead of scalars. Next, $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{B}}$ are computed by first padding non-zero columns/rows with zeroes, and then applying $\pi_1$ and $\pi_2$, just like $\tilde{\boldsymbol{a}}$ and $\tilde{\boldsymbol{b}}$ in Section 4.4.3. The result is again obtained by using a standard MPC protocol for secure matrix multiplication with $\tilde{\boldsymbol{A}}$ and $\tilde{\boldsymbol{B}}$ as inputs.

**Theorem 4.2** (Correctness). *For any $\boldsymbol{A} \in \mathbb{Z}_q^{l \times m}$, $\boldsymbol{B} \in \mathbb{Z}_q^{m \times n}$, let $\tilde{\boldsymbol{A}}, \tilde{\boldsymbol{B}}$ be constructed according to the protocol described in Figure 4.1. Then $\boldsymbol{AB} = \tilde{\boldsymbol{A}}\tilde{\boldsymbol{B}}$.*

*Proof.* By construction of $\tilde{\boldsymbol{A}}$, for all $i \in \{1,\dots,l\}$ and all $j \in \{1,\dots,n\}$,

$$(\tilde{\boldsymbol{A}}\tilde{\boldsymbol{B}})_{ij} = \sum_{k=1}^{l_A+l_B} \tilde{a}_{ik}\tilde{b}_{kj} = \sum_{k=1}^{l_A+l_B} \hat{a}_{i\pi_1^{-1}(k)}\hat{b}_{\pi_2^{-1}(k)j}.$$

From the definition of $\mathcal{F}^{\text{Perm}}$, one of the following cases holds for any pair $(k_1,k_2) := \left(\pi_1^{-1}(k), \pi_2^{-1}(k)\right), k \in \{1,\dots,l_A+l_B\}$:

**CASE 1.** $k_1 \leq l_A$ and $k_2 \leq l_B$. Then there is a unique $k' \in \{1,\dots,m\}$ such that $\left(\mathcal{I}_A^{\text{Col}}\right)_{k_1} = \left(\mathcal{I}_B^{\text{Row}}\right)_{k_2} = k'$ and $\hat{a}_{ik_1}\hat{b}_{k_2j} = a_{ik'}b_{k'j}$.

**CASE 2.** $k_1 > l_A$ or $k_2 > l_B$. Then $\hat{a}_{ik_1}$ or $\hat{b}_{k_2j}$ are zero, and thus $\hat{a}_{ik_1}\hat{b}_{k_2j} = 0$.

On the other hand, for any $k' \in \{1,\dots,m\}$ with $a_{ik'}b_{k'j} \neq 0$, there is a pair $(k_1,k_2)$ with $(\mathcal{I}_A^{\text{Col}})_{k_1} = (\mathcal{I}_B^{\text{Row}})_{k_2} = k'$ and thus a unique $k \in \{1,\dots,l_A+l_B\}$ s.t. $\pi_1(k_1) = \pi_2(k_2) = k$. Therefore,

$$(\tilde{\boldsymbol{A}}\tilde{\boldsymbol{B}})_{ij} = \sum_{k' \in \mathcal{I}_A^{\text{Col}} \cap \mathcal{I}_B^{\text{Row}}} a_{ik'}b_{k'j} = (\boldsymbol{A}\boldsymbol{B})_{ij}.$$

□

**Theorem 4.3** (Security). *Given public sparsity values $l_A, l_B$ and implementations of $\mathcal{F}^{Mult}$ and $\mathcal{F}^{Perm}$ that are secure against semi-honest adversaries, the protocol in Figure 4.1 implements $\mathcal{F}^{Mult}$ with security against semi-honest adversaries.*

*Proof.* We only give the proof for the view of Party 1. By symmetry, the proof for Party 2 follows analogously. For a functionality $\mathcal{F}$ and a protocol $\Pi$, we denote the output to player $i$ of an execution with inputs $x, y$ by $\mathcal{F}_i(x,y)$ and $\text{output}_i^{\Pi}(x,y)$, respectively.

We present our proof in the $(\mathcal{F}^{\text{Perm}}, \mathcal{F}^{\text{Mult}})$-hybrid model. That is, we construct a simulator $\mathcal{S}_1^{\Pi}$ for the view of party 1 assuming ideal functionalities for $\mathcal{F}^{\text{Perm}}$ and $\mathcal{F}^{\text{Mult}}$. Security in the standard model then follows immediately from the security of the corresponding protocols and Theorem 2.2.

Our simulator $\mathcal{S}_1^{\Pi}$ in the ideal model simulates the view of Party 1 on the Protocol in Figure 4.1, i.e.,

$$\text{view}_1^{\Pi}(\boldsymbol{A},\boldsymbol{B}) = \left(\boldsymbol{A}, \mathcal{F}_1^{\text{Perm}}(\mathcal{I}_A^{\text{Col}}, \mathcal{I}_B^{\text{Row}}), \mathcal{F}_1^{\text{Mult}}(\tilde{\boldsymbol{A}}, \tilde{\boldsymbol{B}})\right).$$

Upon receiving input $\boldsymbol{A}$ and output $\mathcal{F}_1^{\text{Mult}}(\boldsymbol{A},\boldsymbol{B})$, the simulator $\mathcal{S}_1^{\Pi}$:

1. samples a permutation $\pi_1'$ of $\{1,\dots,l_A+l_B\}$ uniformly at random,
2. outputs $\left(\boldsymbol{A}, \pi_1', \mathcal{F}_1^{\text{Mult}}(\boldsymbol{A},\boldsymbol{B})\right)$.

By Theorem 4.1, $\mathcal{F}_1^{\text{Perm}}(\mathcal{I}_A^{\text{Col}}, \mathcal{I}_B^{\text{Row}})$ outputs a uniformly random permuation of $l_A + l_B$. Similarly, $\mathcal{F}_1^{\text{Mult}}(\tilde{A}, \tilde{B})$ is a uniformly random matrix. Finally, by Theorem 4.2, $\mathcal{F}_1^{\text{Mult}}(\tilde{A}, \tilde{B})$ is identically distributed to $\mathcal{F}^{\text{Mult}}(A, B)$. Therefore

$$\left(\text{view}_1^{\Pi}(A, B), \text{output}^{\Pi}(A, B)\right) \stackrel{c}{\equiv} \left(S_1^{\Pi}(A, \mathcal{F}_1^{\text{Mult}}(A, B)), \mathcal{F}^{\text{Mult}}(A)\right).$$

□

In order to compute similarities between documents using the protocol described in this section, each data holder must know the feature representation of their documents. This becomes a challenge when trying to use data-dependent feature representations such as TF-IDF. We tackle the problem of private feature extraction for this particular case in the next section.

## 4.5 PRIVATE FEATURE EXTRACTION

Feature extraction is a fundamental pre-processing step in any data processing pipeline, including those used in machine learning, data mining, information retrieval, computer vision and natural language processing. The goal of feature extraction is to convert raw data (e. g., text documents, RGB images, etc.) into a vectorial format suitable for downstream applications. The same feature representation is often useful for many different applications, and the most effective feature representations are typically tailored to the dataset at hand. This poses a challenge in scenarios where datasets containing sensitive records are distributed among multiple parties, as obtaining an adequate feature representation requires a privacy-preserving distributed computation, and the resulting feature representation might leak information about the dataset on which it was computed.

In this section we address this challenge for the well-known *term frequency–inverse document frequency* (TF-IDF) feature representation for text documents (cf. Section 4.3). Our contribution is a two-party protocol for securely computing IDF coefficients on a distributed document database and releasing them under differential privacy. As in the previous section, we work in the standard simulation-based paradigm of security (see Section 2.2) and our computation is secure against semi-honest adversaries. At the same time, its output preserves differential privacy with respect to changing one document in the distributed dataset. Using the output of this protocol, multiple parties can *locally* compute a TF-IDF representation of their documents. This representations will be compatible across parties, and can then be used in any subsequent privacy-preserving computation. In Section 4.6.1, we formalize this as secure computation with differentially-private leakage. Before diving into the details of our protocol, we first review the formal privacy definition in the distributed setting we consider.

### 4.5.1 Multi-Party Computational Differential Privacy

Differential privacy (DP) is a technique for privacy-preserving disclosure [DR14; DMNS16; Dwo+06]. It prevents a potential adversary observing the output of a computation from recovering information about individual input data points, i.e., individual document in our case. This is made formal by saying that two datasets $Z$ and $Z'$ are neighbors if they only differ in one data point; this relation is denoted by $Z \simeq Z'$. We say that a randomized algorithm $\mathcal{A} : \mathcal{Z} \to \mathcal{W}$ is $(\varepsilon, \delta)$-DP if for any indicator function $\chi : \mathcal{W} \to \{0, 1\}$ we have

$$\forall Z \simeq Z' : \quad \mathbb{E}\big[\chi(\mathcal{A}(Z))\big] \leq e^{\varepsilon}\mathbb{E}\big[\chi(\mathcal{A}(Z'))\big] + \delta \ .$$

When $\delta = 0$ we also say that $\mathcal{A}$ is $\varepsilon$-DP. This definition models the setting where a curator owns the input $Z$, executes the computation $\mathcal{A}$, and discloses the output $\mathcal{A}(Z)$.

For the purpose of providing DP in a multi-party setting one needs to modify the above definition to account for the fact that $Z$ is distributed among several parties. Additionally, implementing DP inside an MPC protocol requires a further modification to account for the information that could be obtained by a coalition of adversarial parties involved in the computation who try to break the cryptography used in the MPC protocol. This leads to the definition of *multi-party computationally differential privacy* [Dwo+06; BNO08; MPRV09].

Suppose the input dataset is distributed among $n$ parties $Z = (Z_1, \ldots, Z_n)$ and write $Z \simeq_i Z'$ if $Z_i \simeq Z'_i$ and $Z_j = Z'_j$ for $i \neq j$. Suppose $\mathcal{A} : \mathcal{Z}^n \to \mathcal{W}$ is an $n$-party protocol and let $\mathsf{view}_{-i}(\mathcal{A}(Z))$ denote the information observed by all parties except the $i$th one during the execution of $\mathcal{A}(Z)$. Then we say that $\mathcal{A}$ is $(\varepsilon, \delta)$-MPC-DP if for all $i$ and all $Z \simeq_i Z'$ we have

$$\mathbb{E}\big[\chi\big(\mathsf{view}_{-i}(\mathcal{A}(Z))\big)\big] \leq e^{\varepsilon}\mathbb{E}\big[\chi\big(\mathsf{view}_{-i}(\mathcal{A}(Z'))\big)\big] + \delta$$

for any $\{0, 1\}$-valued polynomial time algorithm $\chi$. In this work we focus on MPC based on computational security, as opposed to information-theoretic MPC, and hence resort to the variant of MPC-DP studied in [MPRV09]. The following key result states that implementing a DP algorithm inside an MPC protocol yields an MPC-DP protocol.

**Theorem 4.4** (informal)**.** *If $\mathcal{A}$ is $(\varepsilon, \delta_{\mathrm{DP}})$-DP with respect to $Z \simeq Z'$, then an MPC implementation of $\mathcal{A}$ where is $Z$ distributed among $n$ parties is $(\varepsilon, \delta_{\mathrm{DP}} + \delta_{\mathrm{MPC}})$-MPC-DP, where $\delta_{\mathrm{MPC}}$ is a negligible function of $|Z|$ obtained from standard cryptographic assumptions.*

### 4.5.2 Differentially Private IDF Computation

One standard approach for making the output of a computation private is the Laplace mechanism [DMNS16]. It works by adding noise

drawn from the Laplace distribution to the output of a deterministic function, where the amount of noise is proportional to the $l_1$-sensitivity of that function and the inverse of the privacy parameter $\varepsilon$. This poses a challenge when it comes to computing the vector of IDF coefficients $\phi_{\text{idf}}(\cdot, Z) \in \mathbb{R}^{\mathcal{V}}$: since we want to provide privacy to whole documents and there is no a-priori knowledge about which words may occur in an arbitrary document, replacing a document might result in a change in several entries of the vector of IDF coefficients. Thus, for a fixed $\varepsilon$, the amount of noise needed for each IDF value using the Laplace mechanism would need to be proportional to the size of the vocabulary, which can be very large.

To avoid this problem, instead of releasing every single IDF value using the Laplace mechanism, we design our mechanism to only release them where they matter most, and resort to outputting a default value everywhere else. The key observation to justify this approach is that, in a corpus of documents $Z$, the distribution of the values of $|Z|_v$ for all $v \in \mathcal{V}$ typically follows a power-law distribution [Pow98]. This means there are few very frequent words and lots of infrequent words. The blue bars in Figure 4.2 exemplify such a typical scenario. The red curve in Figure 4.2 shows how the IDF values quickly converge to the maximum as the document frequency decreases. We can therefore obtain a good approximation across IDFs over all words by releasing differentially private IDFs for the $L$ most frequent words, and assume a default value $c_0$ for all other words. In this way the noise added to the IDFs of the most frequent words will only be proportional to $L$, as opposed to $\mathcal{V}$.

What remains is to make sure the selection of the $L$ most frequent words is also private. To achieve this, we do not release the $L$ most frequent words exactly, but instead release a selection of words that with high probability has a large overlap with the top $L$. This sampling is done using the exponential mechanism [MT07], which is a standard construction for differentially private top-$L$ selection [KOV17].

The pseudo-code of our mechanism is given as Functionality 7. It takes as input the absolute frequencies of each word in each party's dataset $Z_i$. It then proceeds to aggregate these into frequencies across the whole dataset $Z$, yielding $c_v = |Z|_v$ for each $v \in \mathcal{V}$. The counts are used in a private top-$L$ selection step to find $L$ words with the largest frequencies. The mechanism then releases privatized counts $\tilde{c}_v$ for each of the selected words using the Laplace mechanism. For unselected words the mechanism outputs a default public value $\tilde{c}_v = c_0$ which is independent of the true word count.

**Theorem 4.5.** *$\mathcal{F}^{\text{DP-IDF}}$ (Functionality 7) is $\varepsilon$-DP.*

*Proof.* Let $\varepsilon_0 = \varepsilon/(2L)$. Note that for any pair of neighboring datasets $Z \simeq Z'$ and any word $v \in \mathcal{V}$ we have $|c_v - c'_v| \leq 1$. Thus, the analysis of the exponential mechanism [DR14, Theorem 3.6] implies that releasing each selected word $v$ is $\varepsilon_0$-DP. Furthermore, the analysis of the Laplace

**Functionality 7:** Differentially Private IDFs ($\mathcal{F}^{\text{DP-IDF}}$).

**Public Inputs:** $n$, $\mathcal{V}$, $c_0$, $L$, $\varepsilon$
**Private Inputs:** Counts $\{|Z_i|_v\}_{v\in\mathcal{V}}$ for $i \in [n]$
**Output:** Privatized values $\{\tilde{c}_v\}_{v\in\mathcal{V}}$

**foreach** $v \in \mathcal{V}$ **do**
  Compute $c_v = \sum_{i=1}^{n} |Z_i|_v$.
**for** $\ell = 1, \ldots, L$ **do**
  Sample $v \in \mathcal{V}$ with probability $\propto \exp\left(\frac{\varepsilon c_v}{2L}\right)$.
  Sample $\eta$ from $\mathsf{Lap}\left(\frac{2L}{\varepsilon}\right)$.
  Release $\tilde{c}_v = c_v + \eta$.
  Remove $v$ from $\mathcal{V}$.
**foreach** $v \in \mathcal{V}$ **do**
  Release $\tilde{c}_v = c_0$.

mechanism [DR14, Theorem 3.10] implies that releasing $\tilde{c}_v$ for each selected word is $\varepsilon_0$-DP. Note also that the values released for the words which are not selected are independent of the dataset $Z$. Thus, the result follows by applying the standard composition theorem of differential privacy $2L$ times [DR14, Theorem 3.14]. □

Note that using the advanced composition theorem [DR14, Theorem 3.20] one can also show that $\mathcal{F}^{\text{DP-IDF}}$ is $(O(\varepsilon\sqrt{\log(1/\delta)/L}), \delta)$-DP for any $\delta > 0$. However, we will stick to the $\varepsilon$-DP guarantee given above for the sake of simplicity.

### 4.5.3 Implementing Private IDFs in MPC

By Theorem 4.4, we can obtain an MPC-DP protocol from Functionality 7. We propose a circuit-based implementation of $\mathcal{F}^{\text{DP-IDF}}$ which can be ran using any generic circuit-based MPC framework. While our protocol in theory supports any number of parties, we limit our implementation (Section 4.7) and the description in the remainder of this section to two parties.

The main challenge lies in securely generating noise for the Laplace mechanism, and sampling words from the exponential mechanism. Next we describe how to implement both steps.

#### *4.5.3.1 Laplace Mechanism*

For the Laplace Mechanism, we use a standard inversion sampling approach. Given a uniform real number $x \in (0, 1)$, a Laplace sample with mean 0 and scale $b$ can be computed using

$$\mathsf{Lap}(b) = \begin{cases} b\log(2x) & \text{if } x \leq 1/2, \\ -b\log(2-2x) & \text{otherwise.} \end{cases}$$

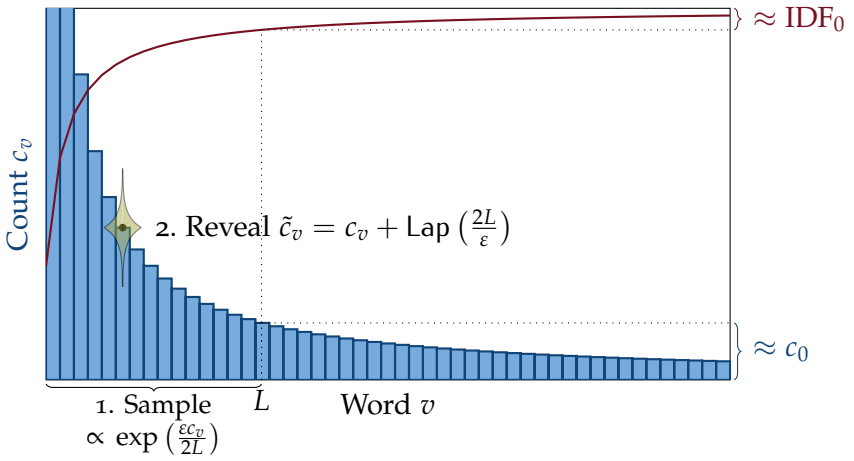



**Figure 4.2:** Graphical representation of our differentially private IDF computation functionality $\mathcal{F}^{\text{DP-IDF}}$. Term counts following a power law distribution are depicted in form of a histogram, and the corresponding IDF values are drawn as a solid line. It can be seen that as $c_v$ decreases, the IDF values quickly converge towards $\text{IDF}_0 = \log(|Z| + 1) + 1$. Steps 1 and 2 are repeated $L$ times in a loop (see Functionality 7).

The required uniform sample can be cheaply computed by adding up two such samples computed locally by each party inside the MPC, and subtracting 1 if the result is larger than 1. Note that this ensures that knowledge of one of the summands reveals nothing about the resulting uniform sample. For all the other operations, including the logarithm, there are circuits that give exact results up to the precision of floating-point numbers. Note that in general, floating-point computation in circuit-based MPC can be quite expensive. However, because we only need to sample $L$ times and $L \ll |\mathcal{V}|$, this only has a minor impact on the running time of our protocol.

#### 4.5.3.2 *Exponential Mechanism*

Implementing the exponential mechanism is more challenging since we need to sample words *without replacement*. However, we will see that this can be done in the same asymptotic time as sampling with replacement, using a Bernoulli tree that gets refreshed after each sample. First, we compute the sampling probability $p_v = \exp(\varepsilon_0 c_v) / \sum_v \exp(\varepsilon_0 c_v)$ of each word $v \in \mathcal{V}$ once and write them to the leaves of a balanced binary tree. Next, we traverse this tree bottom-up, labeling each inner node with the sum of the labels of its children. Now, to sample a word $v \in \mathcal{V}$, we traverse the tree starting from the root. At each node, we perform a Bernoulli trial and descend into each sub-tree with probability proportional to the label at the corresponding child (i.e., the sub-tree's root). Once we arrive at a leaf node, we return

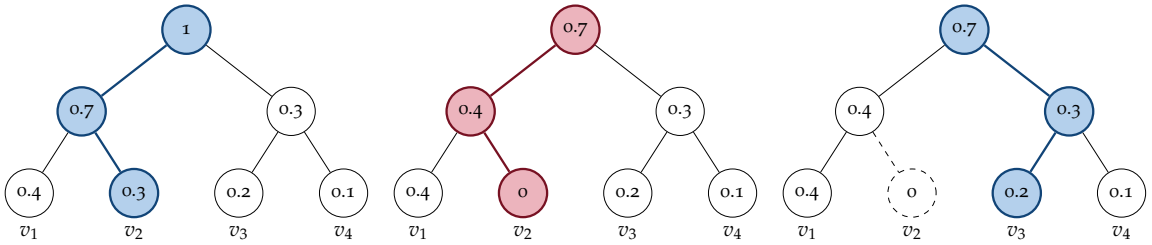



**Figure 4.3:** Example run of our MPC protocol for the exponential mechanism. (Left) The left node gets selected on the first level (probability $0.7/1$), and the right node on the second level (probability $0.3/0.7 \approx 0.43$). The sampled word is $v_2$. (Middle) Refresh step: $p_{v_2} = 0.3$ is subtracted from all nodes on the path to the root, then $p_{v_2}$ is set to zero. (Right) A second sample is drawn with updated probabilities. On the first level, the right node is selected (probability $0.3/0.7$), on the second level it is the left node (probability $0.2/0.3 \approx 0.67$). The result is $v_3$.

the word associated with it. Since the binary tree is balanced, it has depth $O(\log |\mathcal{V}|)$, so one sample can be computed using $O(\log |\mathcal{V}|)$ coin tosses and array accesses. The advantage of this approach is that we can refresh our Bernoulli tree using also just $O(\log |\mathcal{V}|)$ steps: after returning a leaf $v$, we subtract $p_v$ from all nodes on the path from the root and set $p_v$ to 0. This ensures that each leaf is reached at most once. Note that the updated labels do not need to be normalized as we take that into account when descending the tree and compute the probabilities accordingly. An example of our sampling method and the refresh step is shown in Figure 4.3.

The final piece is the implementation of oblivious reads and writes in the nodes of the Bernoulli tree as a circuit. This is needed in order to hide the order in which nodes are accessed, which could leak their associated probabilities and thus counts of individual words. Here, the asymptotically best choice is a generic ORAM construction, which has logarithmic overhead for each access [Ash+20]. However, in terms of concrete efficiency, the optimal choice depends on the level of the tree at which we are reading or writing. In particular for levels with few nodes, asymptotically sub-optimal solutions such as linear scans still outperform generic ORAMs. In practice (Section 4.7), we switch between linear scans, square-root ORAM [Zah+16], and FLORAM [DS17], depending on the level of the tree we are accessing. The cutoff points between those are informed by the measurements of Doerner and Shelat [DS17].

### 4.5.4 Utility Analysis

We now give a utility result about the mechanism in Functionality 7 for computing differentially private IDFs on a dataset of documents. While we motivated our mechanism $\mathcal{F}^{\text{DP-IDF}}$ using the observations that the distributions of words in a document corpus typically follows

a power law, we cannot assume this holds for any possible dataset. Thus, in the following utility analysis, we make the much weaker assumption that the documents in $Z$ are sampled i.i.d. from some unknown distribution. The following result bounds the relative error between the true vectors of IDFs $\phi_{\text{idf}}$ and the privatized vector $\tilde{\phi}_{\text{idf}}$ computed using the counts released by Functionality 7.

*The accuracy analysis underlying this section was performed by co-author Borja Balle. Its main result is repeated here for the sake of completeness. See [SVGB20, Appendix F] for the full result and the relevant proofs.*

**Theorem 4.6.** *For any large enough $m = |Z|$ there exists $c_0 = \Theta(\sqrt{m})$ such that with high probability*

$$\frac{\|\phi_{\text{idf}} - \tilde{\phi}_{\text{idf}}\|_1}{\|\phi_{\text{idf}}\|_1} \leq \tilde{O}\left(\frac{L^2}{\varepsilon m|\mathcal{V}|} + \left(1 - \frac{L}{|\mathcal{V}|}\right)\log\left(m - \frac{L}{\varepsilon}\right)\right) \quad .$$

Note how this result highlights an important trade-off in the choice of $L$ since the first term grows with $L$ while the second term becomes smaller for larger $L$. Additionally, the first terms decreases quickly with $m$, while the second term increases slowly with $m$. In our experiments we did not observe this growth of the error with $m$, which suggests that, for well-behaved datasets where the power-law assumption holds, the $O(\log(m))$ term could be removed. We leave this as an open problem for future work.

## 4.6 SECURE DOCUMENT CLASSIFICATION

In Section 4.4, we have introduced a protocol that can exploit sparse feature representations to compute similarities, and that is particularly efficient when computing many similarities at once. In Section 4.5, we have shown that such sparse features can be computed even if they depend on a whole database distributed among multiple parties, by revealing differentially private IDF coefficients. While each of these protocols is of independent interest, we will now show how they can be securely composed to form higher-level functionalities. We focus on $k$-Nearest Neighbors classification for the remainder of the chapter. However, we stress that our protocols can also be used to implement other functionalities, for example document rankings.

We assume a two-server setting, where each of the servers holds a database of labeled text documents. The labels are the target classes of the classification task. A third party, the client, holds a single unlabeled document $x$ she wants to classify. A $k$-Nearest Neighbors classification algorithm can be used to achieve this. In general, it consists of the following steps: (1) for each document $j$ in the full database, compute the similarity score $s_j(x)$ between $x$ and $j$; (2) compute the labels $\hat{y}_1, \ldots, \hat{y}_k$ corresponding to the documents with the top $k$ scores; and, (3) return the majority vote $\hat{y} = \text{majority}(\hat{y}_1, \ldots, \hat{y}_k)$. This process is formally described in Functionality 8.

**Functionality 8:** $k$-NN Classification

**Public Inputs:** $n$, $\mathcal{V}$, $k$, $\{\tilde{c}_v\}_{v\in\mathcal{V}}$
**Server** $i$ **Inputs:** Document database $Z_i$, and labels $l_x$ for each $x \in Z_i$
**Client Input:** Query document $q$

**foreach** $x \in \bigcup_{i\in[n]} Z_i \cup \{d\}$ **do**
  Compute $\phi_{\text{tf}}(x, v)$, the number of occurrences of $v$ in $x$.
**foreach** $v \in \mathcal{V}$ **do**
  Compute IDF coefficient $\phi_{\text{idf}}(v, Z) = \log((|Z|+1)/(\tilde{c}_v+1)) + 1$.
Compute query vector $\psi(q) = \left(\phi_{\text{tf}}(q, v)\phi_{\text{idf}}(v)\right)_{v\in\mathcal{V}}$.
**foreach** $x \in \bigcup_{i\in[n]} Z_i$ **do**
  Compute $\psi(x) = \left(\phi_{\text{tf}}(x, v)\phi_{\text{idf}}(v)\right)_{v\in\mathcal{V}}$, and similarity score $s_x = \text{sim}_{\cos}(\psi(x), \psi(q))$.
Compute $l_q$ as the label most common among the $k$ documents $x$ with the highest scores $s_x$.
Reveal $l_q$ to the client.

### 4.6.1 Security with Differentially Private Leakage

We define security of our combined protocol in a similar fashion as previous work [MG18; GRR19]. That is, in addition to the output of the ideal functionality, we allow for a randomized leakage $\mathcal{L}$ that depends on the input data. However, said leakage must be differentially private with respect to individual records. Unlike [MG18; GRR19], we additionally allow that the output of our functionality may depend on $\mathcal{L}$. This captures the fact that using the differentially private IDFs from Section 4.5, we do not compute the exact result, but instead an approximation that depends on the privatized IDFs. Note that this setting is also suitable for scenarios where one wants to transfer differentially private hyper-parameter tuning [CV13; FR18] to multi-party learning settings.

**Definition 4.7** (Security with Differentially Private Leakage). *Let $\mathcal{F}$ be an $n$-party functionality with inputs $\bar{x} \in (\{0,1\}^*)^n$, additional input $\ell \in \{0,1\}^*$, and outputs $\left(\mathcal{F}_1(\bar{x}, \ell), \ldots, \mathcal{F}_n(\bar{x}, \ell)\right)$. Let $\mathcal{L}$ denote a randomized leakage function with domain $(\{0,1\}^*)^n$. We write $\tilde{\mathcal{F}}(\bar{x}, \mathcal{L}) = \left(\mathcal{F}(\bar{x}, \mathcal{L}(\bar{x})), \mathcal{L}(\bar{x})\right)$ to denote the function $\mathcal{F}$* with leakage *$\mathcal{L}$, and for any $I \subseteq [n]$, we write $\tilde{\mathcal{F}}_I(\bar{x}, \mathcal{L}) := \left(\mathcal{F}_I(\bar{x}, \mathcal{L}(\bar{x})), \mathcal{L}(\bar{x})\right)$, where $\mathcal{F}_I$ is defined as in Definition 2.1. We say a protocol* $\Pi$ securely computes $\mathcal{F}$ with $(\varepsilon, \delta)$-differentially private leakage $\mathcal{L}$ *if $\mathcal{L}$ is $(\varepsilon, \delta)$-differentially private*

*with respect to individual records in each $x_i$, and there exists a PPT simulator $S$ such that for any $I \subseteq [n]$:*

$$\left\{\left(\mathcal{S}(I, x_I, \tilde{\mathcal{F}}_I(\bar{x}, \mathcal{L})), \tilde{\mathcal{F}}(\bar{x}, \mathcal{L})\right)\right\}_{\bar{x}\in(\{0,1\}^*)^n} \overset{c}{\equiv} \left\{\left(\text{view}_I^{\Pi}(\bar{x}), \text{output}^{\Pi}(\bar{x})\right)\right\}_{\bar{x}\in(\{0,1\}^*)^n} \tag{4.1}$$

Note that the additional argument $\ell$ to $\mathcal{F}$ captures the fact that the output may depend on the leakage. If $\mathcal{F}$ does not depend on $l$, i.e., $\mathcal{F}(\bar{x}, \ell) = \mathcal{F}(\bar{x}, \perp)$ for all $\ell$, and $n = 2$, we get back the definition from [MG18]. Also observe that in order to prove security with leakage of a protocol $\Pi$, it is enough to define $\tilde{\mathcal{F}}$ and $\mathcal{L}$, show that $\Pi$ securely computes $\tilde{\mathcal{F}}$ according to Definition 2.1, and show that $\mathcal{L}$ is $(\varepsilon, \delta)$-differentially private.

In the next section, we describe a 2-server implementation of the $k$-NN functionality $\mathcal{F}^{k\text{-NN}}$ (Functionality 8) that uses the protocols from Sections 4.4 and 4.5 and show that it satisfies this notion of secure computation with differentially private leakage.

### 4.6.2 Secure $k$-NN Classification

Figure 4.4 shows the protocol that we use to implement Functionality 8 in our distributed setting with two servers. There are four phases, two of which correspond to the preceding sections: In a precomputation phase (a), the two servers perform a two-party computation that implements $\mathcal{F}^{\text{DP-IDF}}$ from Section 4.5.2. Then, the client and each of the servers run the secure matrix multiplication protocol from Section 4.4 to obtain shares of the similarities of their respective documents. What remains is to select the top $k$ labels and perform a majority vote in a secure way. While so far we were able to split up the entire protocol into two-party components, we need a generic three-party computation for the top-$k$ selection. However, we can ensure its running time only depends on $k$ and not on the number of documents: observe that each document in the top $k$ overall must also be among the top $k$ of the server that owns this particular documents. Therefore, we can compute shares of the top $k$ on each server using cheap two-party computation (c), and then only need $2k$ inputs to the three-party phase.

**Theorem 4.8.** *The protocol $\Pi^{k\text{-NN}}$ described in Figure 4.4 securely computes $\mathcal{F}^{k\text{-NN}}$ with $\varepsilon$-differentially private leakage.*

*Proof.* Let $\mathcal{L} = \mathcal{F}^{\text{DP-IDF}}$, and let $\tilde{\mathcal{F}}^{k\text{-NN}}$ be defined as the functionality running $\mathcal{L}$ and then using the output $\{\tilde{c}_v\}_{v\in\mathcal{V}}$ as input to $\mathcal{F}^{k\text{-NN}}$. Note that $\tilde{\mathcal{F}}^{k\text{-NN}}$ has the structure required by eq. (4.1). By the definition from Section 4.6.1, it suffices to show that (i) $\mathcal{L}$ is $\varepsilon$-differentially private, and (ii) $\Pi^{k\text{-NN}}$ computes $\tilde{\mathcal{F}}^{k\text{-NN}}$ with security against

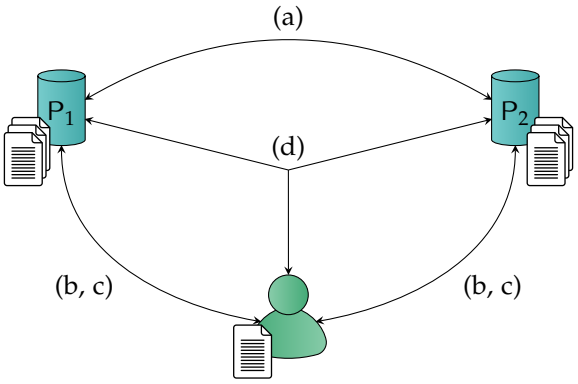



**Figure 4.4:** Diagram of our protocol for $k$-Nearest Neighbors classification of text documents. Each of the two servers holds a collection of labeled text documents, while the client holds an unlabeled document she wants to classify. The protocol has four stages. (a) The servers precompute and release private IDF values for their joint database (Section 4.5). Note that this is a one-time setup step. (b) With each of the servers, the client performs a secure batched similarity computation via a secure sparse matrix-vector multiplication (Section 4.4), where the server inputs a sparse matrix with rows corresponding to documents, while the client inputs a single sparse document vector. (c) Using the similarity shares from the previous step, the client computes with each server shares of the labels and similarities of the $k$ most similar documents to her query. (d) The shares from step (c) are used as inputs to a three-party computation that selects the top $k$ documents overall. The label for the query document is computed by a majority vote among those.

semi-honest adversaries. (i) follows directly from Theorem 4.5. As for (ii), observe that for any subset of the parties, intermediate outputs are either secret-shared (and can therefore be simulated by uniformly random values), or part of the final output. Thus, security of $\Pi^{k\text{-NN}}$ reduces to the security of the individual phases (Theorem 4.3 for Step (b), [LP09] for (a, c) and [DPSZ12] for (d)) and Theorem 2.2. □

Note that our security definition does not explicitly capture how $\mathcal{L}$ is implemented, and therefore does not require the notion of MPC-DP (Section 4.5.1). However, Theorem 4.4 states that any $\varepsilon$-DP functionality implemented as an MPC protocol yields a $(\varepsilon, \delta_{\text{MPC}})$-MPC-DP protocol. In our concrete case, this means that protocol $\Pi^{k\text{-NN}}$ securely computes $\mathcal{F}^{k\text{-NN}}$ with $\varepsilon$-DP leakage $\mathcal{F}^{\text{Perm}}$ (as in the definition from Section 4.6.1), while at the same time the output of the sub-protocol implementing $\mathcal{F}^{\text{Perm}}$ satisfies $(\varepsilon, \delta_{\text{MPC}})$-MPC-DP.

A distinctive feature of our $k$-NN application is the fact that our security definition allows for a *dishonest majority*, and thus the client's query remains secure even if both servers collude. This is in stark

contrast to previous work, as we discuss in Section 4.2. We also note that in principle, our protocol can be extended to more than two servers, by implementing Protocol 7 (Step (a) in Figure 4.4) using a generic semi-honest MPC protocol, as for example given by Ben-Efraim, Lindell, and Omri [BLO16].

## 4.7 EXPERIMENTS

We will now experimentally evaluate our protocols. We do so from two perspectives. First, we evaluate the running time for both of our main protocols, secure matrix multiplication (Section 4.4) and private IDF precomputation (Section 4.5), in a simple two-party setting. Then, we explore how our protocols scale when applied to a real-world classification task. To that end, we implement the $k$-nearest neighbors classification protocol from Section 4.6 and evaluate it on real data. That is, measure the running time taken for similarity computation and top-$k$ selection, taking into account characteristics of real-world data; and we measure the effect our privatized IDF values have on the classification accuracy.

**IMPLEMENTATION.** We implement our protocols in C++, using Obliv-C [ZE15] and MP-SPDZ [Kel20] for generic two-party and multi-party computation, respectively. We implement the dense matrix multiplication protocol from [MZ17] using Eigen [GJ+10] for the online phase, and the EMP toolkit [WMK16] for the offline phase. For the ORAMs needed for the private IDF precomputation, we rely on the implementations of square-root ORAM [Zah+16] and FLORAM [DS17] by Doerner [Doe]. For our accuracy experiments, we re-implement the private IDF protocol (Section 4.5) in Python, and evaluate it in the clear using Scikit-Learn [Ped+11].

*Our source code is available at* `https://github.com/schoppmp/private-knn`.

**EXPERIMENTAL SETUP.** Our timings were obtained on Azure DS14v2 instances, each having 32 vCPUs and 110 GB of RAM. For WAN experiments, we placed the instances in two different regions, East US and West Europe. For all of our experiments, we set the number of features (i. e., the inner dimension for matrix multiplication experiments, and the number of words for DP-IDF computation), to 150000. We chose that number because it is about the size of Aspell's `en_US-large` dictionary [Atk]. For our matrix multiplication experiments, we set the bit width to 64 bits. All the times we report are total running times, i. e., we do not distinguish between offline and online phase for dense matrix multiplication (cf. Section 4.4.5.1).

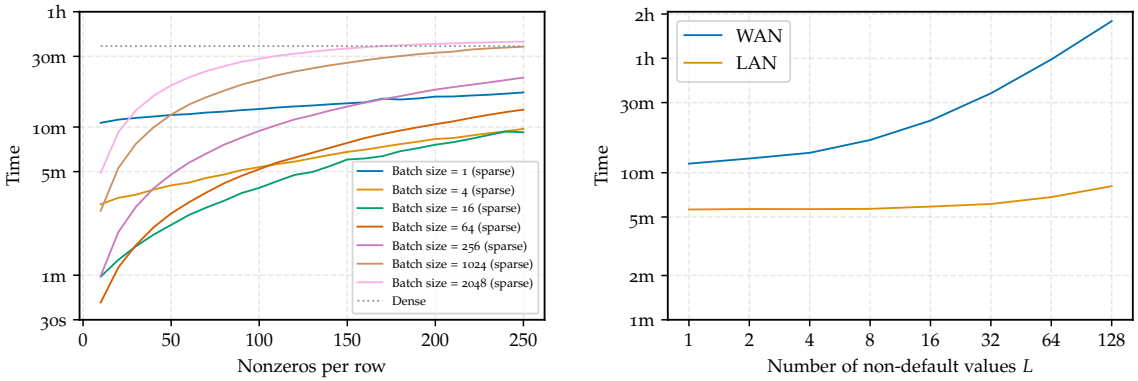



**Figure 4.5:** Evaluation of the running times of our protocols from Sections 4.4 and 4.5. (Left) Sparse matrix-vector multiplication in a LAN with varying sparsity and batch sizes. (Right) Private IDF precomputation with varying number of selected values $L$.

### 4.7.1 Running Time

*Running time experiments were mostly performed by co-author Lennart Vogelsang.*

#### 4.7.1.1 *Sparse Matrix Multiplication*

In this section, we want to measure the improvement of our sparse matrix multiplication protocol over the dense case, and explore trade-offs that occur by tuning different parameters. As we have seen in Section 4.4, our sparse matrix multiplication protocol is an extension of our inner product protocol. By processing multiple rows at once, we reduce the number of calls to $\mathcal{F}^{\text{Perm}}$ needed. On the other hand, we possibly increase the number of non-zeros in each such *batch* of rows, as we have to consider all columns that are non-zero in at least one row.

This results in an interesting trade-off between sparsity and batch size, which we explore here. To that end, we fix some of the parameters. In particular, we only evaluate matrix-vector products $\boldsymbol{Ab}$, and we fix the number of rows of $\boldsymbol{A}$ to 2048. Then, we measure the time taken for this sparse matrix-vector multiplication using different batch sizes and different sparsity levels. We also measure the time taken using only dense multiplication and use it as a baseline.

The results are shown in Figure 4.5 (left). We can see that for the range of parameters we tested, batches of size 16 or 64 give the best running times, depending on the sparsity level. It also becomes apparent that for a suitably chosen batch size, our protocol from Section 4.4 consistently outperforms the dense baseline by at least an order of magnitude. However, note that the running time of our sparse matrix-vector multiplication inherently depends on the assumed public upper bound on the sparsity (number of nonzeros per row). As the number of nonzeros approaches the total number of columns, the dense baseline will eventually become more efficient than our sparse protocol, due to the overhead of generating correlated permutations in the latter.

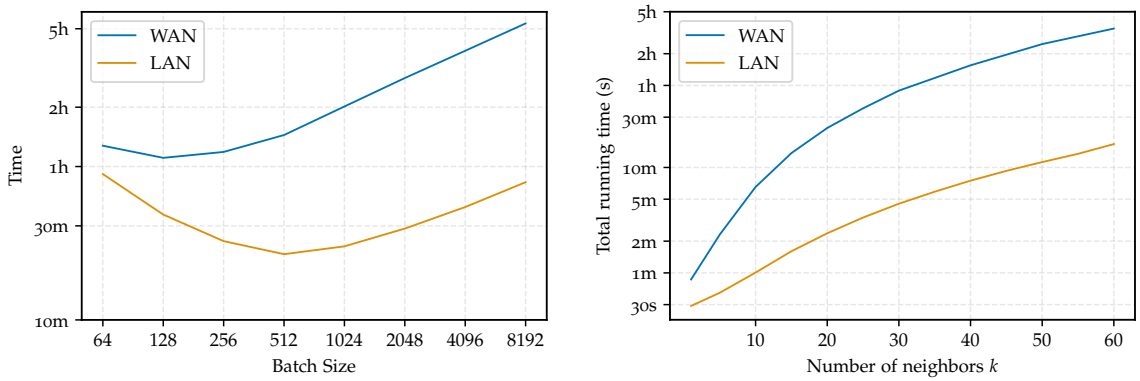



**Figure 4.6:** Running times of our private $k$-NN classification protocol from Section 4.6. The times correspond to a single classification run with 28000 documents consisting of Amazon product reviews [McA]. Overall, we can do a full classification run on this dataset in less than 40 minutes for any $k \in [1, 60]$. (Left) Sparse matrix-vector multiplication on real-world data, i. e., Step (b) from Figure 4.4. (Right) Top-$k$ selection phase, i. e., Steps (c, d) from Figure 4.4.

#### 4.7.1.2 *Private IDF Precomputation*

We also evaluate the time needed for our second protocol, the differentially private IDF generation from Section 4.5. Recall that this protocol is intended to be used as a one-time precomputation step. Once the private IDF are computed and released, all parties can use them to perform feature extraction locally (cf. Section 4.6).

For our evaluation, we fix the vocabulary size to 150000, which corresponds to a large english dictionary [Atk]. We then run the precomputation phase for different values of $L$, i.e., the number of non-default IDF values selected. The results are shown in Figure 4.5 (right). In the LAN setting, the running times stay below 10 minutes. This increases to up to two hours in the WAN setting. Still, as this protocol needs to be only run once per dataset, this is certainly practical in real-world settings.

### 4.7.2 Secure Document Classification

Until now, while we chose the dimensions of the inputs in the previous section to match the ones found in text data, we have not used any features of specific datasets in our evaluation. Here, we explore how our protocols scale when applied to real data. To that end, we evaluate our implementation of the $k$-NN application described in Section 4.6.

First, we evaluate the time needed for the top-$k$ selection phase (Steps (c) and (d) in Figure 4.4). We then explore how the sparsity characteristics of real-world data affects the running time of sparse matrix multiplication. And finally, we look at how our differentially private IDF values affect the accuracy of an end-to-end $k$-NN classification.

#### 4.7.2.1 Datasets

We used two publicly available datasets to set up a multi-class document classification tasks. The first dataset is a repository of Amazon product reviews spanning May 1996 to July 2014 [HM16; McA]. We used the *5-core* version of the dataset containing only products with at least five reviews. From the entire dataset we extracted reviews for products in four different categories: "Clothing, Shoes and Jewelry", "Toys and Games", "Tools and Home Improvement", and "Grocery and Gourmet Food". We use these product categorizations to set up a document classification problem with four classes. To construct the dataset we randomly selected 28k reviews from the four classes with a uniform class distribution. This resulted in a dataset with approximately 40K distinct words, where documents contain an average of 86 words and a maximum of 3300 words. The second dataset is the RCV1 corpus of Reuters newswire stories produced between August 1996 and August 1997 [LYRL04]. Documents in this dataset come annotated with multiple hierarchical labels related to topic, industry and geography. To set up a multi-class classification problem we selected all documents with a single label under topic "Government/Social (GCAT)" and removed any topic with less than five documents, resulting in a total of 19 classes. From the resulting corpus we randomly selected 28k via stratified sampling. Performing this process with the tokenized version of the dataset released with [LYRL04] resulted in a dataset with approximately 70k distinct words (tokens), where documents contain an average of 158 words and a maximum of 2746 words.

#### 4.7.2.2 Effect of Sparsity Distribution on Running Time

For the sparse matrix multiplication experiments in Section 4.7.1.1, we chose the locations of non-zero values in each row of the matrix uniformly and independently at random. This does not reflect the distribution of words in real-world texts, which usually follows a power-law distribution [Pow98]. Therefore, we re-run our matrix-vector multiplication experiment, this time fixing the sparsity for each batch size to the average sparsity of batches of that size in our first real-world dataset [McA]. We also set the number of rows in the matrix to the number of documents in our dataset, i. e., 28000. In Figure 4.6 (left), we show the results. While previously (Figure 4.5, left), the optimal batch size was 16 in most cases, it is 512 when considering the distribution of real data. This shows that our protocol is particularly well equipped to handle real-world inputs.

#### 4.7.2.3 Running Time of top-$k$ Selection

We implement steps (c) and (d) in Figure 4.4 using Obliv-C [ZE15] and MP-SPDZ [Kel20]. We then evaluate their running time on the

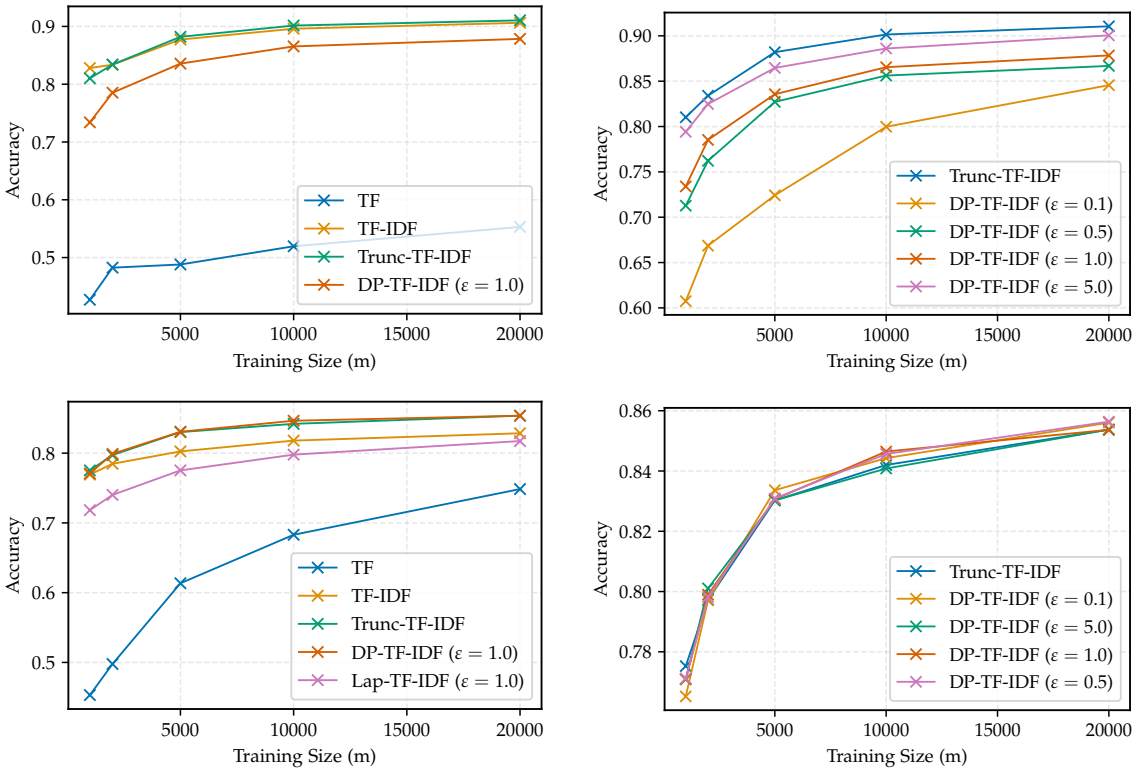



**Figure 4.7**: Results of accuracy vs. training size experiments for $k$-NN with differentially private TF-IDF. (Left) Comparison our DP-TF-IDF algorithm to different baselines. (Right) Effect of privacy budget $\varepsilon$ on the accuracy. (Top) Reviews dataset [McA]. (Bottom) RCV1 dataset [LYRL04].

Reviews dataset [McA]. The results are shown in Figure 4.6 (right). It can be seen that top-$k$ selection does impact the overall running time, while the majority of the computation time for a full classification is still spent on similarity computation.

Overall, for any $k \in [1, 60]$, a full classification run on the reviews dataset takes less than 40 minutes in total time. For comparison, the same computation using only dense matrix multiplication would take more than 8 hours, leading to an improvement of at least 12x.

#### 4.7.2.4 *Accuracy of Differentially Private IDFs*

To evaluate the effect of DP on the privacy-preserving IDF computation described in Section 4.5.2 we used the resulting feature representations in two document classification tasks using a $k$-NN classifier.

*Accuracy experiments were mostly performed by co-author Borja Balle.*

**BASELINES.** To quantify the effect on the accuracy of the resulting $k$-NN classifier of using a TF-IDF feature representation with differentially private IDFs (*DP-TF-IDF*), we compare our approach against the following baselines: *TF* where documents are only represented by their TF vector, which can be computed locally; *TF-IDF* where we

use true IDFs computed without DP; *Lap-TF-IDF* where differentially private IDFs are released using the naive application of the Laplace mechanism sketched at the beginning of Section 4.5.2; *Trunc-TF-IDF* where truncated IDFs computed as in Functionality 7 but without noise (i.e., the setting $\varepsilon = \infty$).

**HYPER-PARAMETER TUNING.** To assess the predictive performance of the different feature representations, we further split the data into 70% for training, 15% for validation, and 15% for testing while maintaining the class proportions in each of the subsets. When testing the effect of the amount of training data on the overall accuracy of the model we further subsample the ~20k training examples to obtain a smaller training set. For each training size and privacy parameter, we tune the hyper-parameters of each algorithm separately by optimizing the accuracy on the validation set, and then report the resulting accuracy on the test set. Since differential privacy introduces randomness in the computation, each accuracy measure is obtained by averaging over 20 independent runs. The hyper-parameter ranges over which the optimization is performed are as follows: number of neighbors $k \in [1, 60]$, number of non-default IDFs $L \in \{32, 64, 128\}$, and default values $c_0 \in \{16, 32, 64, 128\}$. These ranges were selected after an initial data exploration phase.

**RESULTS.** The results of these experiments are displayed in Figure 4.7. In the plots on the left side we compare the accuracy of $k$-NN classification as a function of the size of the training dataset using the TF-IDF representation obtained with our method (for $\varepsilon = 1$) and the baselines described above. We observe that IDFs are necessary to obtain good accuracies, as the TF baseline performs poorly on both datasets. Additionally we observe that Lap-TF-IDF is better than plain TF, but worse than our DP-TF-IDF. On the reviews dataset the Trunc-TF-IDF baseline has the same performance as standard TF-IDF, while in the RCV1 dataset the former is slightly better. Finally, our method is slightly worse than not using DP on the reviews dataset, but performs identically to the best baseline on the RCV1 dataset.

On the right side of Figure 4.7 we explore the effect of the privacy parameter $\varepsilon$ on DP-TF-IDF compared to the best baseline (Trunc-TF-IDF). On the reviews dataset we see that increasing $\varepsilon$ leads to a better feature representation, with $\varepsilon = 1$ incurring a 3% accuracy loss with respect to the best non-private feature representation. On the other hand, on the RCV1 dataset, DP-TF-IDF is quite insensitive to the choice of $\varepsilon$ and matches the behavior of the best baseline for all the values we tried ($\varepsilon \in \{0.1, 0.5, 1, 5\}$).

## 4.8 DISCUSSION

Our MPC protocol for $k$-NN classification achieves provable security in the distributed setting with possibly colluding servers, which has not been reported in academic literature before. At the same time, our evaluation shows that it scales to real-world dataset sizes and is viable in both LAN and WAN settings. We show that by precomputing differentially private statistics, performance can be improved by an order of magnitude, while providing a principled way to trade off between accuracy and privacy. This shows that hybrid solutions combining MPC and DP are a promising avenue for privacy-preserving data analysis on distributed data, as carefully designed DP mechanisms for approximated functionalities can enable efficient MPC protocols.

Apart from classification, our private $k$-NN algorithm can be easily adapted to support other types of queries on distributed datasets, for example private duplicate detection, or query answering. Additionally, other document similarity measures can be implemented atop our protocol for secure two-party sparse matrix multiplication. Moreover, our protocol for sparse matrix multiplication is general in that it works on arithmetic sharings, and hence can be directly used as a building block in other applications.

In the next chapter, we will generalize and improve our protocol from Section 4.4.3, obtaining a general framework for secure sparse linear algebra. We show that it can also be applied to other machine learning tasks beyond $k$-NN, such as logistic regression and naive Bayes classification.

# 5 THE ROOM FRAMEWORK FOR SPARSE LINEAR ALGEBRA

*The contents of this chapter have previously appeared in* Proceedings of the 2019 ACM Conference on Computer and Communications Security *[SGRP19].*

## 5.1 OVERVIEW

Today's popularity of machine learning is the result of a sequence of advances in several areas such as statistical modeling, mathematics, computer science, software engineering and hardware design, as well as successful standardization efforts. A notable example is the development of floating point arithmetic and software for numerical linear algebra, leading to standard interfaces such as BLAS (Basic Linear Algebra Subprograms): a specification that prescribes the basic low-level routines for linear algebra, including operations like inner product, matrix multiplication and inversion, and matrix-vector product. This interface is implemented by all common scientific computing frameworks such as Mathematica, MATLAB, NumPy/SciPy, R, Julia, uBLAS, Eigen, etc., enabling library users to develop applications in a way that is agnostic to the precise implementation of the BLAS primitives being used. The resulting programs are easily portable across architectures without suffering performance loss. The above libraries, and their variants optimized for concrete architectures, constitute the back-end of higher-level machine learning frameworks such as TensorFlow and PyTorch.

In this chapter, we present a framework for privacy-preserving machine learning that provides privacy preserving counterparts for several basic linear algebra routines. Similar to the sparse inner product protocol from Chapter 4, we exploit data sparsity for scalability, by tailoring the basic operations in our framework for that purpose. However, we aim to solve the problem in a much more general way, by providing low-level data structures and protocols for storing and accessing sparse data, and then buildings several sparse linear algebra on top of them. This is analogous to the *sparse* BLAS interface [DHP02], a subset of computational routines in BLAS with a focus on sparse matrices.

The constructions that we develop for the basic building blocks in our framework are cryptographic two-party computation protocols, which provide the formal security guarantees of Definition 2.1. We optimize our protocols for the setting where the sparsity level of the input data is not a sensitive feature of the input that needs to be kept secret. This is the case in many datasets where a bound on a sparsity metric is public information. For example, text data where the maximum length of the documents in the training dataset is public, or genomic data, as the number of mutations on a given individual is

known to be very small. Similarly to Sparse BLAS implementations, sparsity allows us to achieve substantial speed-ups in our secure protocols. These efficiency gains translate to the efficiency of the higher-level applications that we develop in our framework, as is described in Section 5.1.1.

Sparse linear algebra on clear data relies on appropriate data structures such as coordinate-wise or Compressed Sparse Row (CSR) representations for sparse matrices. For the MPC case, we develop a similar abstract representation, which we call *Read-Only Oblivious Map* (ROOM). A significant aspect of this modular approach is that our alternative back-end implementations of the ROOM functionality immediately lead to different trade-offs, and improvements, with regards to communication and computation. This also allows MPC experts to develop new efficient low-level secure computation instantiations for the ROOM primitive. These can then be seamlessly used by non-experts to develop higher level tools in a way that is agnostic to many of the details related to secure computation. Such usage of our framework will be similar to to how data scientists develop high-level statistical modeling techniques while benefiting from the high performance of back-ends of ML frameworks.

### 5.1.1 Chapter Contributions

We present a modular framework for secure computation on sparse data, and build three secure two-party applications on top of it. Our secure computation framework (depicted in Figure 5.1) emulates the components architecture of scientific computing frameworks. We define a basic functionality and then design and implement several secure instantiations for it in MPC; we build common linear algebra primitives on top of this functionality; and then we use these primitives in a black-box manner to build higher level machine learning applications. More concretely, we present the following contributions:

(1) A Read-Only Oblivious Map (ROOM) data structure to represent sparse datasets and manipulate them in a secure and composable way (Section 5.3).

(2) Three different ROOM protocol instantiations (Section 5.3.2) with different trade-offs in terms of communication and computation. These include a basic solution with higher communication and minimal secure computation (Basic-ROOM), a solution using sort-merge networks that trades reduced communication for additional secure computation (Circuit-ROOM), and a construction that leverages fast polynomial interpolation and evaluation (Poly-ROOM) to reduce the secure computation cost in trade-off for local computation, while preserving the low communication.

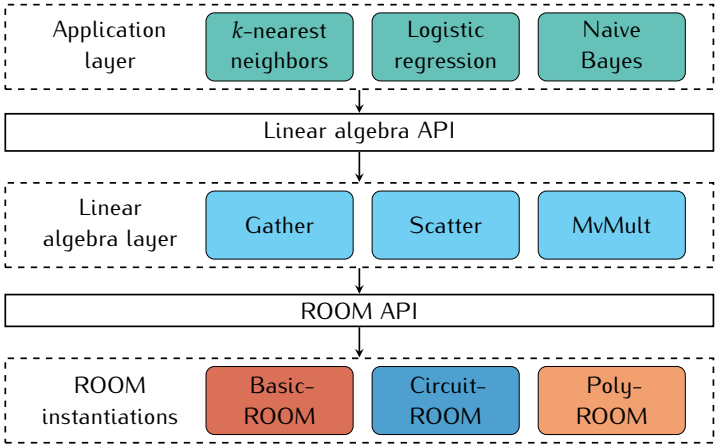



**Figure 5.1:** Components of our system.

(3) We leverage our ROOM primitive in several sparse matrix-vector multiplication protocols (Section 5.4.2), which are optimized for different sparsity settings. We also show how to implement sparse gather and scatter operations using our ROOM primitive.

(4) We build three end-to-end ML applications using our framework. The resulting protocols significantly improve the state of the art, as discussed below.

Our three chosen applications are $k$-nearest neighbors (Section 5.5.1), naive Bayes classification (Section 5.5.2), and stochastic gradient descent for logistic regression training (Section 5.5.3). We evaluate the performance of these applications on real-world datasets (Section 5.7) and show significant improvements over the state of the art:

- For $k$-NN, we have seen in Chapter 4 how to exploit sparsity using a hand-crafted sparse matrix multiplication protocol. We now show that using our new ROOM primitives can reduce the online running time by up to $5\times$.
- Our sparse stochastic gradient descent implementation improves upon the total runtime of the dense protocol by Mohassel and Zhang [MZ17] by factors of $2\times$–$94\times$, and improves communication by up to $215\times$.
- Our protocol for naive Bayes classification scales to datasets with tens of thousands of features, while the previous best construction by Bost et al. handled less than a hundred [BPTG15].

## 5.2 BACKGROUND AND SETUP

**HOW TO EXPLOIT SPARSITY, AND IMPLICATIONS FOR PRIVACY.** Two properties of real-world data representations used for automated anal-

ysis are (a) their high dimensionality and (b) their sparsity. For example, the Netflix dataset [BL07] contains ~480k users, ~17k movies, but only ~100 million out of ~8.5 billion potential ratings, less than 2%. In another common machine learning benchmark, the 20Newsgroups dataset [RL08], the training data consists of just over 9000 feature vectors with $10^5$ dimensions, but less than 100 (0.1%) non-zero values in each vector. Finally, in Genome-Wide Analysis Studies (GWAS), where the goal is to investigate the relationship between genetic mutations and specific diseases, the relevant mutations are limited to only about 5 million locations, while a full human genome contains ~3.2 billion base pairs [The+15].

To cope with memory limits, and speed up computations on sparse data in general, several data structures have been developed that exploit sparsity in the data by only storing relevant (i.e., non-zero) values. For a vector $v$, a straightforward approach is to store only pairs $\big((i, v_i)\big)_{v_i \neq 0}$. For sparse matrices, this generalizes to Compressed Sparse Row representation, where all rows are successively stored in the above fashion, and an additional row-index array stores pointers to the beginning of each row. Linear algebra libraries such as SciPy and Eigen provide implementations of these sparse vectors and matrices [JOP+; GJ+], and databases for genomic data use similar sparse storage formats [Dat+17].

Note that sparse data representation does not only reduce the storage overhead, but is also the basis for more efficient algorithms. For example, a matrix-vector product, where the matrix is stored as CSR, is independent of the number of columns in the original data and only depends on the number of rows and the number of non-zero values in the matrix. For the examples above, where columns correspond to hundreds of thousands of features, this saves large amounts of computation.

In this chapter, we show how to obtain the same benefits from sparsity in the secure distributed ML setting, revealing only the sparsity metric of the underlying data while hiding where in the input the sparsity occurs. There are many scenarios where the sparsity metric can be revealed safely without compromising privacy guarantees: that value might already be public (as with the GWAS example above), or a reasonable upper bound can be set in advance. The main challenge is hiding the *locations* of the non-zero values in the data, which are revealed in the plaintext algorithms for the above sparse data structures. Revealing those indices can leak private information. For example, in the common bag-of-words representation for text data, words in the input vocabulary correspond to columns of a sparse matrix. Revealing the columns where a particular row is non-zero would reveal the words contained in the training document corresponding to that row.

In the remainder of this section, we concretely state our privacy requirements and threat model, and introduce necessary notation and preliminaries.

**THREAT MODEL.** We consider a two-party computation setting with semi-honest parties, and the security of our protocols holds in the common simulation-based paradigm for secure computation (see Section 2.2).

Our computations are over matrices and vectors over a finite domain. In all cases we assume that the sparsity metric of the input set is public. For different protocols this metric will be either the total number of zeros, or the total number of zero rows or columns. In settings where we apply our protocols to collections of rows of a dataset, i.e. batches, the sparsity metric is revealed about the batch. In the context of our applications, the real-world interpretation of the sparsity metric is straightforward. For example, in our logistic regression application, the sparsity metric corresponds to revealing an upper bound on the number of different words in each batch of 128 documents.

**THE PREPROCESSING MODEL FOR MPC** Some secure computation protocols adopt the *online-offline* paradigm, which splits the computation work into an offline stage that can be executed before the inputs are available, and an online stage that depends on the concrete input values. We do not optimize our constructions for the online-offline setting but rather focus on minimizing the *total cost* of the protocol.

**RELATED WORK: CUSTOM MPC PROTOCOLS FOR SPARSE DATA.** Exploiting sparsity for efficiency has been explored before in some concrete privacy preserving applications. The work of Nikolaenko et al. [Nik+13a] develops a protocol for privacy preserving collaborative filtering, a typical application on the Netflix dataset described above. By disclosing a bound on the number of movies rated by a user and exploiting sorting networks, the proposed solution significantly improves on the naive approach that considers all user-movie pairs.

GraphSC [Nay+15] is a framework for secure computation that supports graph parallelization models. The design crucially relies on oblivious sorting, as it enables efficiently running computations expressed as sparse graphs while hiding communication patterns across processors. Another application of oblivious sorting to MPC on sparse graphs is given by Laud [Lau15], albeit in a different threat model (three parties with honest majority).

All of these works, as well as our protocols from Section 4.4, rely on oblivious sorting networks [HEK12], and task-specific optimizations. Our ROOM primitive (Section 5.3) abstracts away from the concrete application by providing a generic interface for secure sparse lookups. In

**Functionality 9:** Functionality of Shared Output ROOM.

**Inputs:** Server: Key-value pairs $\boldsymbol{d} \in (\mathcal{K} \times \mathcal{V})^n, \boldsymbol{\beta} \in \mathcal{V}^m$,
Client: Query $\boldsymbol{q} \in \mathcal{K}^m$.

**Output (shared):** $[\![\boldsymbol{r}]\!]$ such that $\boldsymbol{r} \in \mathcal{V}^m$ and $\forall j \in [m]$:

$$r_j = \begin{cases} v_{q_j} & \text{if } (q_j, v_{q_j}) \in \boldsymbol{d} \\ \beta_j & \text{otherwise.} \end{cases}$$

the case of $k$-NN, we show that this directly translates to a significant improvement in the online running time.

## 5.3 BASIC PRIMITIVE: ROOM

We define Read-Only Oblivious Maps (ROOMs) as a 2-party functionality between a server and a client. For fixed finite sets $\mathcal{K}$ and $\mathcal{V}$—which we call the domain of keys and values, respectively—the server holds a list of key-value pairs $\boldsymbol{d} = ((x_1, v_{x_1}), \ldots, (x_n, v_{x_n}))$, with *unique* keys $x_i \in \mathcal{K}$ and values $v_{x_i} \in \mathcal{V}$, and the client holds a query $(q_1, \ldots, q_m)$, with $q_j \in \mathcal{K}$.

The output of a ROOM is an array $(r_1, \ldots, r_m)$, where for each $q_j$, if $q_j$ is a key in $\boldsymbol{d}$ then $r_j$ is equal to the corresponding value, namely $r_j = v_{q_j}$. Otherwise $r_j$ gets a default value $\beta_j \in \mathcal{V}$ chosen by the server (and which might be different for each index $j$). This mirrors common implementations of a map data structure: for example, in Python `d.get(k, val)` returns the value associated with the key `k` in dictionary `d` if `k` is found, and a default value `val` otherwise. In Java, `d.getOrDefault(k, val)` does the same thing.

We formalize this in Functionality 9. Note that, as the output is secret-shared among the two parties, the question of whether the indexes in the client can be chosen adaptively by the client is not relevant. In some cases, when we want a single party to obtain the output, we write *designated output ROOM*. This variant can be trivially implemented by having one party send their shares to the other, or—as all our concrete implementations have a generic MPC phase at the end—omitting the secret-sharing step.

### 5.3.1 Existing Primitives

Before introducing our instantiations of ROOM, we overview what differentiates our ROOM functionality from existing primitives.

ROOM is related to Private Set Intersection (PSI) (see [PSZ18] and references therein). However, the ROOM functionality requires selecting data items based on key comparison and thus not every PSI

| Data Structure | Initialization | | Answer a query of length $m$ | |
|---|---|---|---|---|
| | Local | Comm. | MPC | Local |
| Basic-ROOM | $O(\lvert\mathcal{K}\rvert)$ (S) | $O(\lvert\mathcal{K}\rvert)$ | $O(m)$ | $O(m)$ (S and C) |
| Circuit-ROOM | - | - | $O((n+m)\log(n+m))$ | $O(n\log(n))$ (S), $O(m\log(m))$ (C) |
| Poly-ROOM | $O(n\log^2(n))$ (S) | $O(n)$ | $O(m)$ | $O((m+n)\log^2(n))$ (C) |

**Table 5.2:** Asymptotic costs of initializations and execution of ROOM instantiations, for a database $\boldsymbol{d} \in (\mathcal{K} \times \mathcal{V})^n$ held by the server (S), and a query $\boldsymbol{q}$ of length $m$ held by the client (C). Initialization is defined as preprocessing independent from the query $\boldsymbol{q}$. We assume the security parameter is constant, and we also do not show factors $\log(\mathcal{K})$ and $\log(\mathcal{V})$. The order of the communication for the online phase (answer length $m$ query) is the same as the MPC Runtime for that phase in all cases.

protocol will directly imply ROOM. In addition, PSI protocols leak the size of the intersection to the client, while it is crucial that a ROOM protocol does not reveal how many indexes in the query were found in the database. Still, extending recent developments on *labeled PSI* [CHLR18] and PSI with shared output [CO18; PSTY19] to the ROOM setting seems to be a promising approach for future improvements.

Functionalities similar to ROOM have recently been proposed as *private join and compute* [Ion+20; Lep+20] or *private matching* [Bud+20]. However, with the exception of [Lep+20][1], these protocols reveal the size of the intersection, whereas our ROOM functionality keeps it private.

[1] *Note that [Lep+20] appeared after our work was published, and in fact the authors experimentally compare their implementation to ours.*

ROOM can also be constructed using Oblivious RAM [GO96]. However, ROOM does not need support for writes, and thus the resulting solution will have much overhead that can be avoided.

Private Information Retrieval (PIR), in its *symmetric* variant [GIKM00], is another primitive relevant to ROOM. Keyword PIR [CGN98] considers the setting of a database that may not contain items at all indices, which is required for ROOM. Finally, while batching techniques that allow the execution of multiple queries have been developed for PIR [ACLS18], they do not directly apply to the keyword variant and also do not always have good concrete efficiency. Thus, from a PIR perspective, our ROOM techniques could be interpreted as improvements on batched symmetric keyword PIR with shared output.

### 5.3.2 Instantiations of ROOM

This section presents three instantiations of Functionality 9. As described in Section 5.3, they can be easily transformed into the designated output variant. The first two constructions are based on generic MPC techniques, while the third instantiation also leverages techniques for oblivious selection using polynomial interpolation.

A naive approach for constructing a ROOM protocol requires $mn$ comparisons since each of the client's queries may be present in

**Protocol 10:** Basic-ROOM.

Let $\boldsymbol{d} \in (\mathcal{K} \times \mathcal{V})^n$, $\boldsymbol{\beta} \in \mathcal{V}^m$, $\boldsymbol{q} \in \mathcal{K}^m$, and $\mathsf{K}$ be a PRF key.
**Inputs:** Server: $\boldsymbol{d}, \boldsymbol{\beta}, \mathsf{K}$; Client: $\boldsymbol{q}$.
**Output (shared):** $[\![\boldsymbol{r}]\!] \in \mathcal{V}^m$.

(1) For $i \in \mathcal{K}$, Server encrypts $c_i \leftarrow \left(v_i \oplus \mathsf{F}_{\mathsf{K}}(i)\right)$, where

$$v_i = \begin{cases} val & \text{if } (i, val) \in \boldsymbol{d} \\ \perp & \text{otherwise.} \end{cases}$$

Server sends $(c_i)_{i \in \mathcal{K}}$ to Client.

(2) For each $i \in [m]$, the parties run a secure two-party computation where Client inputs $c_{q_i}$ and $q_i$ and Server inputs $\mathsf{K}$ and $\beta_i$. The secure computation decrypts $c_{q_i}$ as $v = c_{q_i} \oplus \mathsf{F}_{\mathsf{K}}(q_i)$, and reveals shares $[\![\boldsymbol{r}]\!]$ to Client and Server where

$$r_i = \begin{cases} v, & \text{if } v \neq \perp, \\ \beta_i, & \text{otherwise} \end{cases}$$

the server's database. Our ROOM instantiations reduce this many-to-many comparison problem to one-to-one comparisons. The asymptotic behavior of our proposed instantiations of ROOM is presented in Table 5.2. The online cost distinguishes between local computation and generic MPC computation because the latter has a significantly higher overhead in practice, and hence this distinction is essential for the asymptotics to reflect concrete efficiency.

#### 5.3.2.1 *Basic-ROOM*

Our Basic-ROOM protocol (Protocol 10) is a baseline construction that does not exploit sparsity in the database $\boldsymbol{d}$, and instead expands the whole domain of keys $\mathcal{K}$. Namely, the server computes and sends an encrypted answer for each potential query in $\mathcal{K}$. However, as shown in Table 5.2, the linear dependency on $|\mathcal{K}|$ is limited to the local computation performed by the parties during initialization, whereas the more costly online MPC computation only depends on the length of the ROOM query.

**Lemma 5.1.** *Protocol 10 is a secure instantiation for the ROOM functionality with the following overhead: The initialization includes $O(|\mathcal{K}|)$ work for the server, and $O(|\mathcal{K}|)$ communication to send the encrypted database to the client. The online phase has an $O(m)$ overhead for the MPC protocol.*

SECURITY SKETCH. The security of the PRF implies that the client does not learn anything about database items due to the initialization.

The secure computation in the next step guarantees that both parties learn only shares of the result.

#### 5.3.2.2 Circuit-ROOM

Our second protocol for ROOM uses secure computation and leverages the following observation. We can compute the ROOM functionality by doing a join between the server's data and the query on their key attribute and then computing a sharing of the vector of the corresponding data items from the server's input. A common algorithm for performing equality joins in databases is the *sort-merge join* [Zho09], where elements of each relation are first sorted individually by the join attribute. Subsequently, the two sorted lists are merged (as in merge sort), yielding a combined list where elements from both tables with equal keys are adjacent. This combined list only needs to be scanned once in order to retrieve all elements of the joined table. In the ROOM setting, note that only the last two steps, merge and iteration, depend on data from both parties, as sorting can be performed locally. This makes this algorithm particularly useful for MPC, since merging of $n$ elements can be performed using a circuit of size $O(n \log n)$ [Bat68], and the circuit for comparing adjacent pairs is linear. A similar approach has been taken in previous work for private set intersection [HEK12], and it is also the basis of our sparse matrix multiplication protocol from Section 4.4. We call this ROOM instantiation Circuit-ROOM, and describe it in Protocol 11.

Note that we can assume without loss of generality that $\boldsymbol{d}$ and $\boldsymbol{q}$ are sorted. If they are not, we can extend the MPC protocol that we construct to first compute the sorting with a small $O(m \log(m))$ additive overhead.

The secure computation first arranges the inputs into vectors of triples $\boldsymbol{w}^{\mathsf{C}}$ and $\boldsymbol{w}^{\mathsf{S}}$, which consequently are merged by the first and then third component into a vector $\boldsymbol{v}$. Entries with matching keywords are adjacent in $\boldsymbol{v}$, and the third component of such entries indicates to which party they belong, i.e., indices greater than 0 indicate entries originally in the client's input.

In Step 2) the protocol computes vectors $\boldsymbol{b}$ and $\boldsymbol{c}$. Each entry of these vectors stores information about the result of comparing two adjacent entries of $\boldsymbol{v}$. In particular, $\boldsymbol{b}$ stores the selected value (i.e., answer), depending on whether keys of such entries matched. The vector $\boldsymbol{c}$ stores whether the $i$-th pair of compared adjacent entries involves a key from $\boldsymbol{q}$. In that case, $c_i > 0$, as it corresponds to the index of that key in $\boldsymbol{q}$. Otherwise $c_i = 0$. If $c_i > 0$ then the computation must return an answer in $\boldsymbol{b}$. The answer is the corresponding value from $\boldsymbol{d}$ if a match was found, or the corresponding value from $\boldsymbol{\beta}$ if no match was found.

**Protocol 11:** Circuit-ROOM.

Let $\boldsymbol{d} \in (\mathcal{K} \times \mathcal{V})^n$, and $\boldsymbol{\beta} \in \mathcal{V}^m$, $\boldsymbol{q} \in \mathcal{K}^m$. Assume $\boldsymbol{d}$ is sorted by the first component in each entry, and $\boldsymbol{q}$ is sorted.

**Inputs:** Server: $\boldsymbol{d}, \boldsymbol{\beta}$; Client: $\boldsymbol{q}$.

**Output (shared):** $[\![\boldsymbol{r}]\!] \in \mathcal{V}^m$.

Client and Server run the following steps in a secure two-party computation:

(1) Construct $\boldsymbol{w}^{\mathsf{C}} = \big((q_i, \beta_i, i)\big)_{i\in[m]}$ and $\boldsymbol{w}^{\mathsf{S}} = \big((d_{i,1}, d_{i,2}, 0)\big)_{i\in[n]}$ Merge $\boldsymbol{w}^{\mathsf{C}}$ and $\boldsymbol{w}^{\mathsf{S}}$ into a vector $\boldsymbol{v}$ of length $n+m$, sorted lexicographically by the first and then third component.

(2) Compute vectors $\boldsymbol{b}$ and $\boldsymbol{c}$ in $\mathcal{V}^{m+n}$ and $\{0,\dots,m\}^{m+n}$ by comparing adjacent entries from $\boldsymbol{v}$ (where $v_i = (a,b,c)$, $v_{i+1} = (a',b',c')$). In particular for $i \in [m+n-1]$:

$$(b_{i+1}, c_{i+1}) = \begin{cases} (b, c') & \text{if } a = a' \\ (b', c') & \text{otherwise.} \end{cases}$$

In addition, for $v_1 = (a,b,c)$ we set $(b_1, c_1) = (b, c)$.

(3) Shuffle $\boldsymbol{b}$ and $\boldsymbol{c}$ according to a random permutation $\pi$ unknown to either party, i. e., set $\tilde{\boldsymbol{b}} = \pi(\boldsymbol{b})$ and $\tilde{\boldsymbol{c}} = \pi(\boldsymbol{c})$.

(4) Iterate over $\tilde{\boldsymbol{b}}$ and $\tilde{\boldsymbol{c}}$ in parallel. Whenever $\tilde{c}_i \neq 0$, reveal $\tilde{c}_i$ and share $\tilde{b}_i$ between the parties, who both set $[\![r_{\tilde{c}_i}]\!] = [\![\tilde{b}_i]\!]$.

Next, in Step 3) $\boldsymbol{b}$ and $\boldsymbol{c}$ are obliviously shuffled to avoid leakage induced by relative positions of their entries. This is analog to the shuffle step in Sort-Compare-Shuffle PSI [HEK12].

Finally, in step 4), entries of $\boldsymbol{b}$ that correspond to comparisons with keys from the client are output in shares between the parties, along with the corresponding entries in $\boldsymbol{c}$. This allows the parties to map their output shares back to the order of the inputs. Note that the shuffling in step 3) makes sure that the indexes at which elements are revealed do not leak any information to either party.

**Lemma 5.2.** *Protocol 11 is a secure instantiation for the ROOM functionality with the following overhead. The client and the server run a secure two-party computation protocol whose main bottleneck is computing $O((n+m)\log(n+m))$ comparisons. Additionally, local computations cost $O(n \log n)$ for the server and $O(m \log m)$ for the client.*

The security claim in the above lemma follows directly since our protocol is entirely done in MPC and any additional information revealed beyond the output shares is indistinguishable from random.

**Protocol 12:** Poly-ROOM.

Let $\boldsymbol{d} \in (\mathcal{K} \times \mathcal{V})^n$, $\boldsymbol{\beta} \in \mathcal{V}^m$, $\boldsymbol{q} \in \mathcal{K}^m$. Let $s \in \mathbb{N}$ be a statistical security parameter and $\mathsf{K}$ be a PRF key.
**Inputs:** Server: $\boldsymbol{d}, \boldsymbol{\beta}$, $\mathsf{K}$; Client: $\boldsymbol{q}$.
**Output (shared):** $[\![\boldsymbol{r}]\!] \in \mathcal{V}^m$.

(1) For each $t_j = (i, v) \in \boldsymbol{d}$, Server computes

$$c_i = \mathsf{F}_{\mathsf{K}}(i) \oplus (v \,||\, 0^{s+\log n}).$$

(2) Server interpolates a polynomial $\mathsf{P}$ of degree $n$, such that for each $(i, v) \in \boldsymbol{d}$, $\mathsf{P}(i) = c_i$.

(3) Server sends the coefficients of $\mathsf{P}$ to Client.

(4) For each $i \in [m]$, the parties run a two-party computation protocol where Server has input $\mathsf{K}$ and Client has inputs $q_i, \mathsf{P}(q_i)$. Both parties receive shares $[\![\boldsymbol{r}]\!]$ as output, where

$$r_i = \begin{cases} v & \text{if } \mathsf{P}(q_i) \oplus \mathsf{F}_{\mathsf{K}}(q_i) = (v \,||\, 0^{s+\log n}) \\ \beta_i, & \text{otherwise.} \end{cases}$$

#### 5.3.2.3 *Poly-ROOM*

Finally, as our main instantiation for ROOM, we present a protocol that has MPC runtime similar to Basic-ROOM (independent of $n$ and linear on $m$), but avoids the dependence on the key domain in initialization.

The main insight for our new construction (Protocol 12) is that the server can construct a polynomial which evaluates, for inputs which are keys of items in the server's database, to outputs which are encrypted versions of the corresponding values in the server's database. The encryptions are done with a key that is only known to the server. The resulting polynomial is of degree $n$ and is therefore a concise representation of the data. At the same time, the polynomial looks pseudorandom (since it is an interpolation over pseudorandom points), and therefore hides the points which have non-zero values.

The server sends this polynomial to the client. The client then evaluates the polynomial on its inputs and learns $m$ outputs. For each of the client's keys present in the database, the client obtains the encrypted version of the corresponding database value. The two parties then run a secure computation that decrypts each value that the client obtained, checks if it decrypts correctly (i.e., ends with a fixed string of zeros), and reveals to the client either a value in the database or a default value from $\boldsymbol{\beta}$ depending on the result of that check. Note that the check passes if the key in the client's query is in $\boldsymbol{d}$.

**Lemma 5.3.** *Protocol 12 is a secure instantiation for the ROOM functionality. The initialization cost includes $O(n \log^2 n)$ work for the server and communication of $O(n)$ to send the polynomial to the client. The online cost of the protocol has $O(m)$ cost for the MPC execution and then $O((m+n)\log^2 n)$ local computation for the client.*

The overhead of the local computation is based on running efficient algorithms for polynomial interpolation and multi-point evaluation, which interpolate a polynomial of degree $n$ in time $O(n \log^2 n)$, and evaluate such a polynomial on $n$ points also in time $O(n \log^2 n)$ [MB72] (we used an implementation of these protocols in our experiments).

SECURITY SKETCH. The security of the construction follows from the fact that the polynomial that the client receives is pseudorandom since the encryptions $c_i$ used in step 2) are pseudorandom. The rest of the computation is implemented in an MPC protocol.

## 5.4 ROOM FOR SECURE SPARSE LINEAR ALGEBRA

In this section we present efficient two-party protocols for several common sparse linear algebra operations, which leverage the ROOM functionality in different ways. Similar to how sparse BLAS operations are presented in [DHP02], we first focus on lower-level primitives (Gather and Scatter), and then use them to implement higher-level functionality, namely matrix-vector multiplication. However, we stress that our goal is not to provide implementations of each function described in [DHP02], but instead focus on the operations necessary for the applications described in Section 5.5.

### 5.4.1 Gather and Scatter

Intuitively, the Gather and Scatter operations correspond, respectively, to a sequence of indexed reads from a sparse array, and a sequence of indexed writes into a sparse array. More concretely, Gather takes a vector of indices $\boldsymbol{q} = (i_1, \ldots, i_n)$ and a (usually sparse) vector $\boldsymbol{v}$, and returns the vector $\boldsymbol{r} = (v_{i_1}, \ldots, v_{i_n})$ that results from *gathering* the values from $\boldsymbol{v}$ at the indices in $\boldsymbol{q}$. Scatter on the other hand takes a vector of values $\boldsymbol{v}$, a vector of indices $\boldsymbol{q} = (i_1, \ldots, i_n)$, and a vector $\boldsymbol{u}$, and updates $\boldsymbol{u}$ at each position $i_j$ with the new value $v_j$.

We transfer the two operations to the two-party setting as follows. Given a sparse vector $\boldsymbol{v}$ held by Party $\mathsf{P}_2$, and a set of query indices $\boldsymbol{q}$ held by Party $\mathsf{P}_1$, $\mathsf{Gather}(\boldsymbol{v}, \boldsymbol{q})$ returns additive shares of a dense vector $\boldsymbol{v}'$, with $\boldsymbol{v}'_i = \boldsymbol{v}_{\boldsymbol{q}_i}$. It is clear that this is equivalent to a ROOM query with $\mathsf{P}_2$ and $\mathsf{P}_1$ as the server and client, and inputs $\boldsymbol{q}$, $\boldsymbol{d} = \{(i, \boldsymbol{v}_i) \mid \boldsymbol{v}_i \neq 0\}$, and $\boldsymbol{\beta} = \boldsymbol{0}$.

**Functionality 13:** ScatterInit with shared inputs.

Let $\boldsymbol{v} \in \mathbb{Z}_{2^\sigma}^l$, and let $n \geq l$ be a public parameter.
**Inputs:** $\mathsf{P}_1$: Vector share $[\![\boldsymbol{v}]\!]_{\mathsf{P}_1}$, indices $i_1, \ldots, i_l$,
$\mathsf{P}_2$: Vector share $[\![\boldsymbol{v}]\!]_{\mathsf{P}_2}$.
**Output:** Vector shares $[\![\boldsymbol{v'}]\!]$, $\boldsymbol{v'} \in \mathbb{Z}_{2^\sigma}^n$, where

$$v'_i = \begin{cases} v_j, & \text{if there is a } j \in [l] \text{ s.t. } i_j = i, \\ 0 & \text{otherwise.} \end{cases}$$

For Scatter, we focus on a variant where $\boldsymbol{u}$ is zero. We call this functionality ScatterInit. Given a dense vector $\boldsymbol{v}$ and a set of indices $\boldsymbol{q}$, both of size $l$, and an integer $n \geq l$, ScatterInit$(\boldsymbol{v}, \boldsymbol{q}, n)$ returns a vector $\boldsymbol{v'}$ of length $n$ such that $\boldsymbol{v'}_{\boldsymbol{q}_i} = v_i$ for all $i \in [l]$, and $\boldsymbol{v'}_j = 0$ if $j \notin \boldsymbol{q}$. As in the case of gather, we are interested in secure protocols for ScatterInit that output $\boldsymbol{v'}$ additively shared. Regarding the inputs, we focus on the case where the input vector is also secret-shared between the parties, while $\boldsymbol{q}$ is held by one party, and $n$ is a public parameter. The reason for this setting will become apparent when we present our protocol for row-sparse matrix multiplication in Section 5.4.2.2, as it uses ScatterInit as a sub-protocol.

We formally define ScatterInit in Functionality 13. One direct way to implement this functionality is using generic MPC such as garbled circuits. The approach requires a circuit of size $O(nl\sigma)$ that for each $i \in [n]$ selects the $i$th output among the all possible values $(0, r_1, \ldots, r_{n-l})$. Hence a solution based on generic MPC constructions would require $O(nl\sigma)$ communication and computation. Alternatively, one can rely on additive homomorphic encryption to enable $\mathsf{P}_1$ to distribute the encrypted values of $\boldsymbol{r}$ into the right positions of encrypted $\boldsymbol{r'}$ and then execute a protocol with $\mathsf{P}_2$ to obtain shares of $\boldsymbol{r'}$ in the clear. This approach requires $O(n)$ computation and $O((n+l)L)$ communication where $L$ is the length of a ciphertext of additively homomorphic encryption, which adds considerable expansion to the length of the encrypted value. In the next paragraph, we describe a version that is *concretely efficient*, building on ROOM and oblivious transfer extensions.

*In Chapter 6 we will introduce a similar functionality termed* known-indices multi-point function secret sharing (MPFSS) *that can improve both asymptotic and concrete efficiency compared to the approaches described here.*

**SCATTERINIT FROM ROOM AND OT EXTENSION.** Our implementation of ScatterInit is described in Protocol 14. The idea is that $\mathsf{P}_2$ generates its random output share $[\![\boldsymbol{v'}]\!]_{\mathsf{P}_2}$ and then $\mathsf{P}_1$ and $\mathsf{P}_2$ execute an MPC protocol from which $\mathsf{P}_1$ obtains (a) all entries of $[\![\boldsymbol{v'}]\!]_{\mathsf{P}_2}$ at indices not in $\{i_1, \ldots, i_l\}$, and (b) its share of the output for the remaining indices, which is obtained securely from $\mathsf{P}_2$'s share of the output and the shared input vector $\boldsymbol{v}$. For (a) we use a well-known $(n-l)$-out-of-$n$ OT protocol that we describe in the next paragraph.

**Protocol 14:** ScatterInit based on ROOM.

Let $\boldsymbol{v} \in \mathbb{Z}_{2^\sigma}^l$, and let $n \geq l$ be a public parameter.
**Inputs:** $\mathsf{P}_1$: Vector share $[\![\boldsymbol{v}]\!]_{\mathsf{P}_1}$, indices $i_1, \ldots, i_l$,
$\mathsf{P}_2$: Vector share $[\![\boldsymbol{v}]\!]_{\mathsf{P}_2}$, key $\mathsf{K}$.
**Output:** $[\![\boldsymbol{v'}]\!]$, for $\boldsymbol{v'} \in \mathbb{Z}_{2^\sigma}^n$, as defined in Functionality 13.

(1) For each $i \in [n]$, $\mathsf{P}_2$ generates a random value $s_i$.

(2) The parties run a $(n-l)$-out-of-$n$ Oblivious Transfer protocol, with $\mathsf{P}_2$ acting as the sender, for $\mathsf{P}_1$ to obtain

$$\boldsymbol{u} = ((i, s_i) \mid i \notin \{i_1, \ldots, i_l\}).$$

(3) The parties run ROOM with $\mathsf{P}_2$ acting as Server and $\mathsf{P}_1$ as Client, and inputs $\boldsymbol{q} = (i_1, \ldots, i_l)$, $\boldsymbol{d} = ((i, s_i))_{i \in n}$, and $\boldsymbol{\beta} = \perp^n$. The parties obtain shares of the vector $\bar{\boldsymbol{u}} = (s_i \mid i \in \{i_1, \ldots, i_l\})$.

(4) The parties engage in a two-party computation with inputs $[\![\boldsymbol{v}]\!]_{\mathsf{P}_1}$, $[\![\boldsymbol{v}]\!]_{\mathsf{P}_2}$, $[\![\bar{\boldsymbol{u}}]\!]_{\mathsf{P}_1}$, and $[\![\bar{\boldsymbol{u}}]\!]_{\mathsf{P}_2}$, where $\boldsymbol{v}$ and $\bar{\boldsymbol{u}}$ are reconstructed, and $\bar{\boldsymbol{s}} = \boldsymbol{v} - \bar{\boldsymbol{u}}$ is revealed to $\mathsf{P}_1$.

(5) $\mathsf{P}_2$ sets $[\![\boldsymbol{v'}]\!]_{\mathsf{P}_2} = (s_i)_{i \in [n]}$ and $\mathsf{P}_1$ sets $[\![\boldsymbol{v'}]\!]_{\mathsf{P}_1} = \boldsymbol{s}$ with

$$s_i = \begin{cases} \bar{s}_j, & \text{if } i = i_j \in \{i_1, \ldots, i_l\}, \\ -s_i, & \text{where } (i, s_i) \in \boldsymbol{u}, \text{ otherwise.} \end{cases}$$

For (b) we use a ROOM query followed by a two-party computation where $\mathsf{P}_2$'s output share is reconstructed in the MPC (step (4) in Protocol 14) and used to mask $\boldsymbol{v}$ to produce $\mathsf{P}_1$'s output share. Note that Basic-ROOM is the natural instantiation to use in this setting.

The $(n-l)$-out-of-$n$ OT in step 2 is implemented using a folklore protocol that requires $n$ invocations of 1-out-of-2 OTs and an $l$-out-of-$n$ Shamir secret sharing of a PRF key. It works by the sender encrypting each of its $n$ inputs using a key $\mathsf{K}_{\mathsf{ot}}$, and then letting the receiver learn in each of the $n$ OTs either an encrypted input or a share (in $l$-out-of-$n$ secret sharing) of $\mathsf{K}_{\mathsf{ot}}$. This forces the receiver to learn at least $l$ shares of the key, and therefore at most $n-l$ values.

**SECURITY SKETCH** We argue that the view of each party after each step in the protocol includes only random values in addition to its inputs. This true after the first two steps since first $\mathsf{P}_2$ generates random shares and then, using the security of the oblivious transfer, $\mathsf{P}_1$ obtains a subset of them while $\mathsf{P}_2$ obtains nothing. The security properties of the ROOM protocol guarantee that the shares that each party obtains after the third step are also indistinguishable from random from that party's view. In the secure computation in step four, $\mathsf{P}_1$ obtains values that depend on the random masks $\bar{\boldsymbol{u}}$. The fact

that $P_1$ does not know $\bar{u}$, together with the guarantees of the secure computation, ensures the output is indistinguishable from random for $P_1$. The last step involves only local computation for each party and thus does not change their views.

We will now use the protocols from the previous section to build sparse matrix-vector multiplication protocols in the next section. Concretely, we will use Gather for column sparsity (Section 5.4.2.1), and ScatterInit for row sparsity (Section 5.4.2.2).

### 5.4.2 Sparse Matrix-Vector Multiplication

Throughout this subsection we consider party $P_1$ holding a private matrix $\boldsymbol{M} \in \mathbb{Z}_{2^\sigma}^{n \times m}$, with exactly $l$ nonzero columns or rows, depending on the context. Party $P_2$ holds a private vector $\boldsymbol{v} \in \mathbb{Z}_{2^\sigma}^{m}$ with $k$ nonzero entries. The value of $\sigma$ is at least 64 in standard ML applications, and potentially more in settings that require additional precision to represent real numbers using fixed point arithmetic, or secret shares. The goal of all protocols is to compute the vector $\boldsymbol{Mv}$ of length $n$, *additively shared* between $P_1$ and $P_2$. This allows us to easily integrate these protocols as part of higher-level secure protocols, such as the solutions to machine learning problems presented in Section 5.5. While we assume that $n, m, l, k$ and $\sigma$ are public, no additional information is revealed to the parties.

The underlying theme of our protocols is different private reductions of sparse matrix-vector multiplication to the dense case. The goal of such reductions is to avoid multiplications by zero, and hence have the cost of the dense multiplication be dependent only on $l$ and $k$, instead of the total size of the sparse dimension. Therefore, the last step in our protocols will be to use a sub-protocol for two-party dense matrix-vector multiplication. As discussed in Section 5.2, efficient dedicated protocols for this functionality have been recently presented. This includes solutions based on precomputed triples by Mohassel and Zhang [MZ17], as well as solutions based on homomorphic encryption [JVC18], and server-aided OT [LJLA17]. In our protocols in this section we will refer to a generic dense matrix multiplication protocol MvMult, as well as to a generic ROOM protocol ROOM. In our implementation, we use the OT-based protocol from [MZ17], which we presented in the previous chapter (Protocol 6).

#### *5.4.2.1 Column-Sparse Matrix*

We propose two protocols for the case where $\boldsymbol{M}$ is sparse in the second dimension (i.e., there is a small number of non-zero columns). These have different tradeoffs depending on the relationship between the sparsity of $\boldsymbol{M}$ and $\boldsymbol{v}$. Note that matrix-vector multiplication, where the matrix is sparse in its columns, can be viewed as a generalization

**Protocol 15:** Our first protocol for column-sparse matrix–vector multiplication.

**Inputs:** $\mathsf{P}_1$: Matrix $\boldsymbol{M} \in \mathbb{Z}_{2^\sigma}^{n \times m}$, with $l$ nonzero columns,
$\mathsf{P}_2$: Vector $\boldsymbol{v} \in \mathbb{Z}_{2^\sigma}^{m}$, with $k$ nonzero entries.
**Output:** $[\![\boldsymbol{Mv}]\!] = ([\![\boldsymbol{Mv}]\!]_{\mathsf{P}_1}, [\![\boldsymbol{Mv}]\!]_{\mathsf{P}_2})$.

(1) $\mathsf{P}_1$ sets $\boldsymbol{q} = (i_1, \ldots, i_l)$, the (sorted) list of indexes of non-zero columns in $\boldsymbol{M}$.

(2) $\mathsf{P}_1$ and $\mathsf{P}_2$ run Gather$(\boldsymbol{v}, \boldsymbol{q})$ to obtain shares $([\![\boldsymbol{v'}]\!]_{\mathsf{P}_1}, [\![\boldsymbol{v'}]\!]_{\mathsf{P}_2})$ of $\boldsymbol{v'}$, which is the restriction of $\boldsymbol{v}$ to the indexes in $\boldsymbol{q}$.

(3) $\mathsf{P}_1$ *locally* computes $\boldsymbol{M'}$ sub-matrix of $\boldsymbol{M}$ containing only the nonzero columns.

(4) $\mathsf{P}_1$ and $\mathsf{P}_2$ run MvMult with inputs $\boldsymbol{M'}$ and $[\![\boldsymbol{v'}]\!]_{\mathsf{P}_2}$ and obtain shares $([\![\boldsymbol{M'}[\![\boldsymbol{v'}]\!]_{\mathsf{P}_2}]\!]_{\mathsf{P}_1}, [\![\boldsymbol{M'}[\![\boldsymbol{v'}]\!]_{\mathsf{P}_2}]\!]_{\mathsf{P}_2})$ of $\boldsymbol{M'}[\![\boldsymbol{v'}]\!]_{\mathsf{P}_2}$.

(5) $\mathsf{P}_2$ sets the output $[\![\boldsymbol{Mv}]\!]_{\mathsf{P}_2}$ to $[\![\boldsymbol{M'}[\![\boldsymbol{v'}]\!]_{\mathsf{P}_2}]\!]_{\mathsf{P}_2}$ and $\mathsf{P}_1$ sets $[\![\boldsymbol{Mv}]\!]_{\mathsf{P}_1}$ to $[\![\boldsymbol{M'}[\![\boldsymbol{v'}]\!]_{\mathsf{P}_2}]\!]_{\mathsf{P}_1} + \boldsymbol{M'}[\![\boldsymbol{v'}]\!]_{\mathsf{P}_1}$.

of sparse vector inner product, and thus the following protocols can also be used for this functionality.

Our first protocol is shown in Protocol 15. Let $\boldsymbol{q} = (i_1, \ldots, i_l)$ be the indexes of the non-zero columns in $\boldsymbol{M}$. The goal of the sparse-to-dense reduction here is to replace the computation of $\boldsymbol{Mv}$ by the computation of $\boldsymbol{M'v'}$, where $\boldsymbol{M'}$ is the sub-matrix of $\boldsymbol{M}$ containing only the non-zero columns $i_1, \ldots, i_l$, and $\boldsymbol{v'}$ is the restriction of $\boldsymbol{v}$ to the indices in $\boldsymbol{q}$. Party $\mathsf{P}_1$ can compute $\boldsymbol{M'}$ locally. The two parties then call the Gather protocol to obtain shares $([\![\boldsymbol{v'}]\!]_{\mathsf{P}_1}, [\![\boldsymbol{v'}]\!]_{\mathsf{P}_2})$ of $\boldsymbol{v'}$. At this point the parties invoke the dense matrix multiplication protocol to compute $\boldsymbol{M'}[\![\boldsymbol{v'}]\!]_{\mathsf{P}_2}$. Further, $\mathsf{P}_1$ locally computes $\boldsymbol{M'}[\![\boldsymbol{v'}]\!]_{\mathsf{P}_1}$ and adds the result to its share of $\boldsymbol{M'}[\![\boldsymbol{v'}]\!]_{\mathsf{P}_2}$. As a result, both parties obtain shares of $\boldsymbol{M'v'}$. The security of the complete protocol follows directly from the security of the protocols for ROOM and dense multiplication.

There are two drawbacks of the above protocol: (a) the space of values of the ROOM sub-protocol coincides with the domain of the elements of $\mathsf{P}_2$'s input vector, $\mathbb{Z}_{2^\sigma}$. This is a problem in high-precision settings where $\sigma > 64$, which are not uncommon in ML applications where real numbers are encoded in fixed point. (b) the length of the ROOM query $q$ is $l$, which is the sparsity of the server's input matrix. In many settings, the vector $v$ has less non-zero values than $M$, so $k < l$. That is why we would like to have our ROOM query to only be of the smaller size $k$, which for two of our constructions directly translates into a speed-up in MPC time (see Table 5.2). However, if we simply have $\mathsf{P}_1$ act as the server and put the non-zero columns of $\boldsymbol{M}$ in the ROOM protocol as the database, while $\boldsymbol{v}$ becomes the query, the

**Protocol 16:** Column-sparse matrix and sparse vector multiplication protocol.

**Inputs:** $\mathsf{P}_1$: Matrix $\boldsymbol{M} \in \mathbb{Z}_{2^\sigma}^{n \times m}$, with $l$ nonzero columns.
$\mathsf{P}_2$: Vector $\boldsymbol{v} \in \mathbb{Z}_{2^\sigma}^{m}$, with $k$ nonzero entries.
**Output:** $[\![\boldsymbol{Mv}]\!] = ([\![\boldsymbol{Mv}]\!]_{\mathsf{P}_1}, [\![\boldsymbol{Mv}]\!]_{\mathsf{P}_2})$.

(1) $\mathsf{P}_1$ chooses a random permutation $\pi_1$ of $[l+k]$ and sets $\boldsymbol{d} = ((a_1, \pi_1(1)), \ldots, (a_k, \pi_1(l)))$, and $\boldsymbol{\beta} = (\pi_1(l+1), \ldots, \pi_1(l+k))$, where the $a_i$'s are the indices of the nonzero columns in $\boldsymbol{M}$.

(2) $\mathsf{P}_2$ sets $\boldsymbol{q} = (b_1, \ldots, b_k)$, where the $b_i$'s are the indices of the nonzero values in $\boldsymbol{v}$.

(3) $\mathsf{P}_1$ and $\mathsf{P}_2$ run a designated-output ROOM with inputs $\boldsymbol{d}, \boldsymbol{\beta}, \boldsymbol{q}$. $\mathsf{P}_2$ obtains $\boldsymbol{r} = (p_i)_{i \in [k]}$.

(4) Let $\hat{\boldsymbol{M}} \in \mathbb{Z}_{2^\sigma}^{n \times l}$ be $\boldsymbol{M}$ but with its zero columns removed. $\mathsf{P}_1$ defines $\bar{\boldsymbol{M}}$ as the result of appending $k$ zero columns to $\hat{\boldsymbol{M}}$, and computes $\boldsymbol{M'} = \pi_1(\bar{\boldsymbol{M}})$, where $\pi_1$ permutes the columns of $\bar{\boldsymbol{M}}$.

(5) Let $\hat{\boldsymbol{v}} \in \mathbb{Z}_{2^\sigma}^{k}$ be $\boldsymbol{v}$ but with its zero entries removed. $\mathsf{P}_2$ defines a permutation $\pi_2 : [k+l] \mapsto [k+l]$ such that $\pi_2(i) = p_i$ for $1 \leq i \leq k$. The values $\pi_2(i)$ for $k+1 \leq i \leq k+l$ are a random permutation of $\{1, \ldots, m\} \setminus \{p_1, \ldots, p_k\}$ (the set of unused indexes in $[m]$). $\mathsf{P}_2$ computes $\boldsymbol{v'} = \pi_2(\bar{\boldsymbol{v}})$, where $\bar{\boldsymbol{v}}$ is $\hat{\boldsymbol{v}}$ padded with zeros up to length $k+l$.

(6) $\mathsf{P}_1$ and $\mathsf{P}_2$ run MvMult with inputs $\boldsymbol{M'}$ and $\boldsymbol{v'}$ to obtain shares of $\boldsymbol{Mv}$.

values in the ROOM protocol become huge, as it would hold vectors of length $n$, namely the first dimension of $\boldsymbol{M}$.

Protocol 16 solves both issues (a) and (b), by relying on the correlated permutations introduced in Section 4.4. First, our protocol ensures that the server's input to the ROOM functionality are elements in $\mathcal{K} \times \mathcal{K}$, thus avoiding the dependence on $\sigma$. Second, it allows us to swap the roles of $\mathsf{P}_1$ and $\mathsf{P}_2$ in the ROOM protocol, allowing us to choose them depending on the relationship between $k$ and $l$, as well as other nonfunctional requirements induced by computation and communication limitations of $\mathsf{P}_1$ and $\mathsf{P}_2$.

These two optimizations come at the cost of replacing the input size to the dense multiplication sub-protocol from $2ln\sigma$ to $2(l+k)n\sigma$. Hence, in practice the actual values of $\min(l, k)$, $n$, and $\sigma$ determine a trade-off between Protocol 15 and Protocol 16.

The intuition behind Protocol 16 is as follows. Let $\hat{\boldsymbol{M}}$ and $\hat{\boldsymbol{v}}$ be the result of removing zero columns and entries of $\boldsymbol{M}$ and $\boldsymbol{v}$, as defined in Step (4), and let $\bar{\boldsymbol{M}}$ and $\bar{\boldsymbol{v}}$ be $\hat{\boldsymbol{M}}$ and $\hat{\boldsymbol{v}}$ padded with $k$ zero columns

and $l$ zeroes, respectively. Now consider a trusted third party that provides party $\mathsf{P}_i$ with a random permutation $\pi_i$ such that, after permuting columns of $\bar{\boldsymbol{M}}$ and $\bar{\boldsymbol{v}}$ according to $\pi_1$ and $\pi_2$ they are "well aligned", meaning $\pi_1(\bar{\boldsymbol{M}})\pi_2(\bar{\boldsymbol{v}}) = \boldsymbol{M}\boldsymbol{v}$. Note that it is crucial that $\pi_1$ and $\pi_2$ look random to $\mathsf{P}_1$ and $\mathsf{P}_2$ respectively. To achieve that, the third party generates *random* $\pi_1$ and $\pi_2$ subject to the constraint that $\forall i \in [l+k] : \pi_1(i) = \pi_2(i) \Leftrightarrow i \in (A \cap B)$, where $A$ and $B$ are the sets of indexes of nonzero columns and values in $\boldsymbol{M}$ and $\boldsymbol{v}$.

The idea in Section 4.4 was to implement the above third party functionality in MPC using garbled circuits. In Protocol 16, we implement this functionality in an different way using the ROOM primitive as follows. $\mathsf{P}_1$ acts as the ROOM's server with inputs $\boldsymbol{d} = ((a_1, \pi_1(1)), \ldots, (a_k, \pi_1(l)))$ and $\boldsymbol{\beta} = (\pi_1(k+1), \ldots, \pi_1(l+k))$, where $\pi_1$ is a random permutation of $[k+l]$ chosen by $\mathsf{P}_1$, and $\mathsf{P}_2$'s query is simply $\boldsymbol{q} = (b_1, \ldots, b_k)$ (see steps (1) and (2) in Protocol 16). The outputs for the two parties from the ROOM protocol are *secret shares* of the array $\boldsymbol{r} = (p_i)_{i\in[k]}$. Party $\mathsf{P}_1$ provides $\mathsf{P}_2$ with their share of the $p_i$'s so that $\mathsf{P}_2$ can reconstruct $\pi_2$. Note that this computation is independent of both $n$ and $\sigma$, in contrast to Protocol 15. Moreover, it provides flexibility to exchange the roles of the server and client in the ROOM protocol achieving the second goal defined above.

The security of this construction follows from Theorem 4.3 and the security of the ROOM protocol. The output of the ROOM allows party $\mathsf{P}_2$ to obtain the evaluation of a random permutation $\pi_1$ on its non-zero entries. The rest of the protocol involves local computations until the final secure computation for the dense multiplication.

#### 5.4.2.2 Row-Sparse Matrix

We now consider the case where $\boldsymbol{M}$ is sparse in its first dimension. In our solution to this variant $\mathsf{P}_1$ defines $\boldsymbol{M'}$ as the matrix resulting from removing all zero rows from $\boldsymbol{M}$. Then the parties run a protocol to compute shares of the vector $\boldsymbol{r} = \boldsymbol{M'}\boldsymbol{v}$ of length $l$. For this, we can either use Protocol 15, if $\boldsymbol{v}$ is sparse, or we can rely on dense multiplication. In any case, $\boldsymbol{r}$ now contains all non-zero values of the desired result, but its dimensions do not match $\boldsymbol{M}\boldsymbol{v}$. However, note that $\boldsymbol{M}\boldsymbol{v}$ can be recovered from $\boldsymbol{r}$ by inserting $n-l$ zeros between the values of $\boldsymbol{r}$ at positions corresponding to zero rows in the original matrix $\boldsymbol{M}$. This can directly be achieved by running $\mathsf{ScatterInit}(\boldsymbol{r}, \boldsymbol{i}, n)$, where $\boldsymbol{i}$ contains the indexes of non-zero rows in $\boldsymbol{M}$.

In our higher-level applications, which we describe next, we will use the matrix-vector multiplications described here in various ways. In particular, we use Protocol 16 for $k$-NN classification, and the column-sparse and row-sparse protocols from Protocol 15 and the previous section for logistic regression.

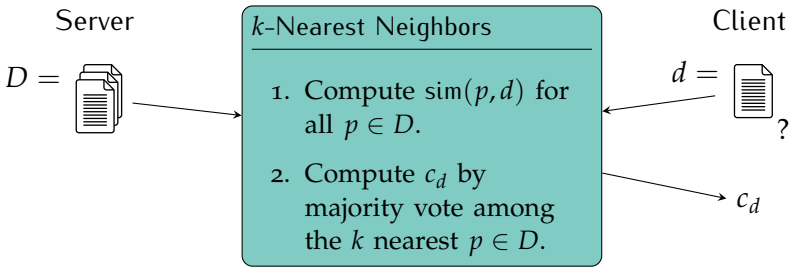



**Figure 5.3:** Setting for our secure $k$-NN application. We focus on a single server, but our protocol can naturally be generalized to multiple servers as described in Section 4.6.

## 5.5 APPLICATIONS

We consider three applications to exemplify the features of our framework. These include non-parametric data analysis tasks ($k$-nearest neighbors and naive Bayes classification) as well as parametric data analysis tasks (logistic regression trained by stochastic gradient descent (SGD)).

Our $k$-NN application is a simplified version of the one presented in Section 4.6, using only a single server. We choose the other two applications in order to show the flexibility of our framework, and to enable comparisons with previous works: secure naive Bayes classification was studied by Bost et al. [BPTG15], and the SecureML work by Mohassel and Zhang [MZ17] is the state of the art in secure two-party logistic regression learning with SGD.

### 5.5.1 Similarity Computation and $k$-Nearest Neighbors

For secure $k$-NN we use a simplified version of the protocol described in Section 4.6, using only a single database $D$ of documents held by a single server. A client wants to classify a document $d$ against $D$. The $k$-NN classification algorithm, which is parameterized by a constant $k$, achieves that goal by (a) computing $\mathsf{sim}(d, p)$, for each $p \in D$, and (b) assigning a class $c_d$ to $d$ as the result of a majority vote among the classes of the $k$ most similar documents according to the similarities computed in step (a). See Figure 5.3 for a schematic depiction of this setting.

Apart from using a single server, the main difference compared to Section 4.6 is our choice of multiplication algorithm. While the circuit-based approach described in Section 4.4.4 is essentially equivalent to Circuit-ROOM, we compare all three ROOM instantiations here. This allows us to achieve better efficiency in the online phase, while matching the speedups of Section 4.6 in the offline phase (see Section 5.2). In Section 5.7.3, we show that our Poly-ROOM indeed improves the online phase by up to 5×.

### 5.5.2 Naive Bayes Classification

A naive Bayes classifier is a non-parametric supervised classification algorithm that assigns to an item $d$ (for example a document) the class $c$ in a set of potential classes $C$ (for example {spam, no-spam}) that maximizes the expression $\text{score}(c) = P(c) \cdot \Pi_{t \in d} P(t|c)$, where $t \in d$ denotes the database features present in the feature representation of $d$. A common approach to keep underflows under control is to use logs of probabilities. This transforms the above expression into $\text{score}(c) = \log(P(c)) + \sum_{t \in d} \log(P(t|c))$.

In Naive Bayes, $P(c)$ is estimated as $P(c) = N_c/N$, namely the number of items $N_c$ of class $c$ in the dataset, divided by the dataset size $N$. $P(t|c)$ is estimated as $P(t|c) = T_{c,t}/N_c$, namely the number of occurrences (or score) $T_{c,t}$ of feature $t$ in items of class $c$, normalized by the total number of examples of class $c$ in the training dataset. Additionally, Laplace smoothing is often used to correct for terms not in the dataset, redefining $P(t|c)$ to be $P(t|c) = (T_{c,t} + 1)/(N_c + N)$

A secure two-party naive Bayes classification functionality is defined as follows: a server holds the dataset $D$ that consists of $n$ items with $k$ features. Each item in the dataset is labeled with its class from the set $C$ of potential classes. Hence, the server holds the values $P(t|c), P(c)$ defined above. A client wants to obtain a label for an item $d$. This needs to be done in a privacy preserving manner where only the client learns the output label and the server learns nothing.

The work of Bost et al. [BPTG15] presented a solution to the above problem using Paillier encryption and an argmax protocol based on additive homomorphic encryption. Our ROOM functionality provides a direct solution for this two-party problem, in which the server reveals an upper bound of its number of features. This solution works as follows: for each class $l \in C$, the server and the client invoke the ROOM functionality with input values $\log(P(t|l))$ for all keys $t$, as well as default values $1/(N_c + N)$ for the server, and query $(t)_{t \in d}$ for the client. This gives the parties additive shares of the vector $(\log(P(t|l))_{t \in d}$. Then, the parties can compute locally shares of the vector $(\text{score}(l))_{l \in C}$, which contains the scores of $d$ with respect to all classes. Finally, the class with highest score, which will be revealed only to the client, can be computed using any generic MPC protocol involving only $|C|$ comparisons.

### 5.5.3 Logistic Regression Training

A drawback of non-parametric approaches like $k$-NN is that each query depends on the entire training database. To circumvent this, parametric approaches such as logistic regression first train a smaller *model* $\theta$, which is then used to answer classification queries faster.

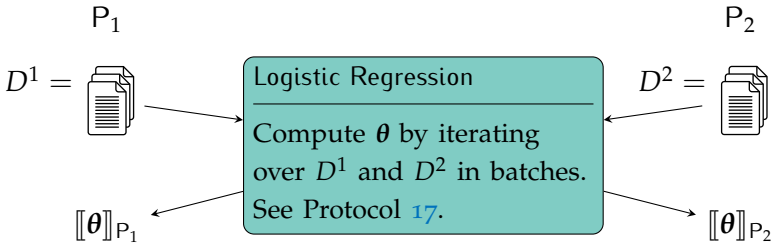



**Figure 5.4:** Setting for two-party logistic regression. Each party holds a set of labeled documents, and the output is a model that is secret-shared between the two parties.

Here, we assume a two-party setting where parties $P_1$ and $P_2$ hold a horizontally partitioned database, i.e., each party holds a *sparse* dataset, where $P_i$ holds $\boldsymbol{X^i} \in \mathbb{R}^{n_i \times d}, \boldsymbol{y^i} \in \mathbb{R}^{n_i}$, with $\boldsymbol{X^i}$ being the set of $n_i$ records with $d$ features and $\boldsymbol{y}^i$ being the vector of corresponding binary target labels. This corresponds to a training dataset of size $n \times d, n = n_1 + n_2$ distributed among $P_1$ and $P_2$, and the goal is to build a shared model $\boldsymbol{\theta}$ that is able to accurately predict a target value for an unlabeled record, while keeping the local training datasets private (cf. Figure 5.4).

A widely used algorithm for building this kind of model is *mini-batched stochastic gradient descent (SGD)*. Here, the empirical loss of the model $\boldsymbol{\theta}$ is minimized by iteratively refining it with an update rule of the form $\boldsymbol{\theta^{i+1}} = \boldsymbol{\theta^i} - \eta \boldsymbol{g}$, for a step size $\eta$ and a gradient quantity $\boldsymbol{g}$. The training dataset is partitioned in so-called *mini-batches*, each of which is used to compute a model update: In a forward pass, the prediction loss of the current batch is computed, and the gradient $\boldsymbol{g}$ of that loss is obtained as the result of a backward pass.

Protocol 17 shows a secure protocol for two-party SGD training. It relies on a secure matrix multiplication protocol MvMult, and an approximation of the logistic function $\mathrm{Sigmoid}(x) = 1/(1+e^x)$ introduced by Mohassel and Zhang [MZ17], implemented with a garbled circuit. The security of the protocol follows from the fact that $\boldsymbol{\theta}$ is always kept secret shared, and secure implementations of the two sub-protocols above.

We now discuss how the calls to MvMult in lines 6 and 9 of the protocol are instantiated with our protocols from Section 5.4.2. First note that, as the $\boldsymbol{X}^j$'s are sparse, so will be their mini-batches $\boldsymbol{B}^j$ contributed by $P_1$ or $P_2$ in line 5. In fact, as common mini-batch sizes are as small as 64 or at most 128, the mini-batches will be sparse in their columns. We show that this is the case in the context of concrete real-world datasets in Section 5.7. Hence, the call to MvMult in line 6 involves a column-sparse matrix and a dense vector, and thus we choose Protocol 15 instantiated with Basic-ROOM. The choice of Basic-ROOM is justified by the fact that the keys of $[\![\boldsymbol{\theta}]\!]_{P_2}$ span the whole key domain $\mathcal{K} = [d]$, as it is a secret share, and hence Basic-ROOM

**Protocol 17:** Secure two-party gradient descent on sparse distributed training data.

**Inputs:** $\mathsf{P}_1$: $D^1 = (\boldsymbol{X^1}, \boldsymbol{y^1})$, $\boldsymbol{X^1} \in \mathbb{R}^{n \times d}$, $\boldsymbol{y^1} \in \{0,1\}^n$,
$\mathsf{P}_2$: $D^2 = (\boldsymbol{X^2}, \boldsymbol{y^2})$, $\boldsymbol{X^2} \in \mathbb{R}^{n \times d}$, $\boldsymbol{y^2} \in \{0,1\}^n$.
**Output:** Shared model $[\![\boldsymbol{\theta}]\!]$

$[\![\boldsymbol{\theta}]\!] \leftarrow (0)_{i \in [d]}$
**for** $T$ *epochs* **do**
  **for** $i \in \lfloor \frac{n}{b} \rfloor$ **do**
    **for** $j \in [2]$ **do**
      $\boldsymbol{B^j} \leftarrow \boldsymbol{X^j}_{[i..i+b]}$
      $[\![\boldsymbol{u^j}]\!] \leftarrow \mathsf{MvMult}(\boldsymbol{B^j}, [\![\boldsymbol{\theta}]\!])$, $[\![\boldsymbol{v^j}]\!] \leftarrow \mathsf{Sigmoid}([\![\boldsymbol{u^j}]\!])$
      $[\![\boldsymbol{w^j}]\!] \leftarrow [\![\boldsymbol{v^j}]\!] - \boldsymbol{y^j}_{[i..i+b]}$
      $[\![\boldsymbol{g^j}]\!] \leftarrow \mathsf{MvMult}(\boldsymbol{B^j}^\mathsf{T}, [\![\boldsymbol{w^j}]\!])$
    $[\![\boldsymbol{g}]\!] \leftarrow \frac{1}{2b}([\![\boldsymbol{g^1}]\!] + [\![\boldsymbol{g^2}]\!])$
    $[\![\boldsymbol{\theta}]\!] \leftarrow [\![\boldsymbol{\theta}]\!] - \eta [\![\boldsymbol{g}]\!]$

does not incur unneccesary overhead in this case. On the other hand, the computation of $\boldsymbol{g^j}$ in line 9 is a multiplication between a row sparse matrix and a dense vector, for which we use our protocol from Section 5.4.2.2.

In Section 5.7.4, we compare the runtimes of our sparse implementation to those reported in [MZ17]. With the exception of the smallest dataset, we improve computation time by a factor of 2×–11× (LAN) and 12×–94× (WAN), and communication by 26×–215× (LAN) and 4×–10× (WAN).

## 5.6 IMPLEMENTATION OF OUR FRAMEWORK

*Our source code is available at* `https://github.com/schoppmp/room-framework`.

For our implementation, we follow the general architecture presented in Figure 5.1. For each layer of abstraction, we define generic interfaces that are then matched by our concrete implementations. This allows, for example, to use the same matrix multiplication function for different ROOM instantiations, which in turn simplifies development and makes sure our framework can be extended seamlessly.

Most of our library is written as generic C++ templates that abstract away from concrete integer, vector and matrix types. This allows us to use Eigen's expression templates [GJ+10], and thus avoid unnecessary local matrix operations. For generic two-party computation based on garbled circuits, we use Obliv-C [ZE15]. As a PRF, we use the AES-128 implementation in Obliv-C by Doerner [Doe]. The fast polynomial interpolation and evaluation that we need for Poly-ROOM and ScatterInit is done using Yanai's FastPolynomial library [Yan].

| *Dataset* | *Documents* | *Classes* | *Nonzero Features* | | *Accuracy* | | |
|---|---|---|---|---|---|---|---|
| | | | *Single (avg.)* | *Total* | *Log. Regression* | *Naive Bayes* | *k-NN* |
| Movies (M) [Maa+11] | 34341 | 2 | 136 | 95626 | 0.88 | 0.85 | (*) 0.74 |
| Newsgroups (N) [RL08] | 9051 | 20 | 98 | 101631 | 0.73 | 0.76 | 0.57 |
| Languages (L1) [The], ngrams=1 | 783 | 11 | 43 | 1033 | 0.96 | 0.87 | 0.96 |
| Languages (L2) [The], ngrams=2 | 783 | 11 | 231 | 9915 | 0.99 | 0.99 | 0.99 |

**Table 5.5:** Datasets used in the experiments. These comprise a variety of classification tasks such as sentiment analysis of movie reviews (Movies), topic identification (Newsgroups), and language identification (Languages). We also report the accuracy achievable using out-of-the-box classification algorithms. For the the Languages dataset, we further investigate the effect of analyzing larger n-grams instead of single characters.
(*) $k$-NN was trained on a subsample of 10k examples due to memory limitations.

## 5.7 EXPERIMENTAL EVALUATION

Given the large number of parameters and tradeoffs that our framework exhibits, a complete layer-by-layer evaluation of all components from Figure 5.1 with all ranges of useful parameters is both infeasible and not very useful. Instead, we chose to run experiments on only two abstraction layers: ROOM micro-benchmarks, which allow to compare our constructions with each other and with future improvements, and entire applications, which allow us to compare against previous work on application-specific protocols.

All our experiments are performed on Azure DS14 v2 instances with 110 GB of memory each, using a single core. We note that the memory bottleneck in our experiments is the "dense" case that we use as a baseline, not our ROOM-based implementations. For LAN experiments, we use instances in the same region, while for WAN experiments, we place one in the US and one in Europe. The measured roundtrip time was 0.7ms in the LAN setting, and 85ms in the WAN setting. The average data transfer rates were 2.73 Gbit/s and 245 Mbit/s, respectively. We use 64-bit integers for $\mathcal{K}$ and $\mathcal{V}$. Garbled circuits are run with 80-bit security, due to the default settings in Obliv-C. For Poly-ROOM, we use $s = 40$ bits of statistical security.

### 5.7.1 ROOM Micro-Benchmarks

Table 5.2 presents the runtimes for Circuit-ROOM and Poly-ROOM and how they depend on the database size $n$ and the query size $m$. We first measure the runtimes of each algorithm for a range of parameters $n \in \{500, 5000, 50000\}$ and $m \in \{0.1n, 0.2n, \ldots, n\}$. The results can be seen in Figure 5.6. Each plot corresponds to one choice of $n$, while values of $m$ are given on the x-axes. The runtime of both ROOM variants increases as $m$ grows, but Circuit-ROOM is outperformed by Poly-ROOM as $n$ increases, as long as $m \ll n$. The reason is that the

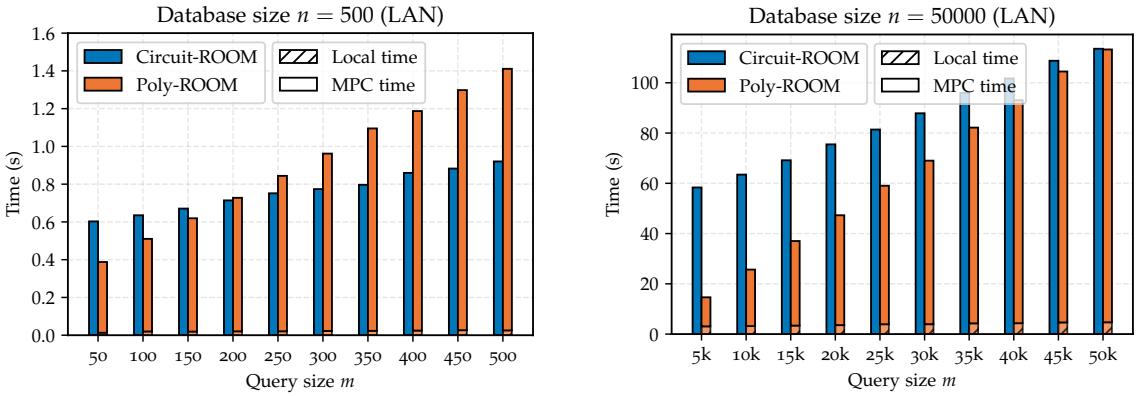



**Figure 5.6:** Running times of Circuit-ROOM and Poly-ROOM in the LAN setting, for several choices of query size and database size. We distinguish between local time (for time spent doing local computation) and MPC time, for running time of MPC sub-protocols.

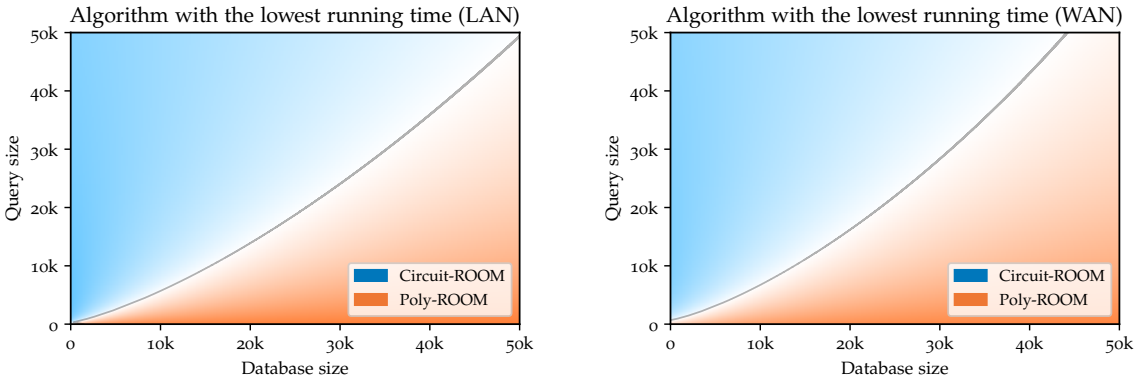



**Figure 5.7:** Estimated performance of our two instantiations of sparse ROOM in the LAN (left) and WAN (right) settings. Running times were measured for length $m$ queries to a ROOM of size $n$, with $n \in \{500, 5000, 50000\}$ and $m \in \{0.1n, 0.2n, \ldots, n\}$. Then, for each of our algorithms, a model of the running time was computed using nonlinear least-squares from `scipy.optimize.curve_fit`, where the function to be fitted was chosen according to the asymptotics in Table 5.2. Each pixel was computed by averaging over the colors corresponding to each algorithm, weighted by the inverse of their respective running times. Thus, the dominant color of a region corresponds to the algorithm that performs the best in that setting. The solid lines indicate the cutoffs in each setting.

complexity of Circuit-ROOM depends significantly on $n$, as its runtime is dominated by oblivious merging and shuffling, which scales with the sum $m + n$. The time of Poly-ROOM in mainly determined by $m$, while Circuit-ROOM remains more stable across the choices of $m$.

To investigate the cutoff point between the two instantiations, as well as their performance *relative to each other*, we then fit functions of the runtime to the collected data, which gives us a model to estimate the performance even for parameter choices not directly measured. Figure 5.7 shows the results in the LAN and WAN settings, respectively. For each set of choices for $m$ and $n$, the color in the plot indicates the relative performance of our two algorithms. Intuitively, in regions

where one of those two colors is prevalent, the corresponding algorithm is the optimal choice for that setting. Regions in between (turquoise) correspond to parameters where both of our algorithms perform equally well, with a solid line indicating the cutoff.

In the LAN setting, Poly-ROOM clearly wins in all cases where $m \ll n$, while Circuit-ROOM is only viable for large queries on small databases. For $m \approx n$, both approaches seem equally viable. Similar observations can be made in the WAN, where Poly-ROOM even outperforms Circuit-ROOM for $m > n$ when $n$ is large.

### 5.7.2 Datasets

We implement each of the applications presented in Section 5.5 in our framework. We analyze three real-world datasets that represent common classification tasks: sentiment analysis [Maa+11], topic identification [RL08], and language detection [The]. Table 5.5 summarizes the properties of each of the datasets, including the average number of features of single documents. We also report, for reference, the classification accuracies that can be achieved using the different methods outlined in Section 5.5: logistic regression, naive Bayes, and $k$-nearest neighbors. These were obtained in the clear using out-of-the-box Scikit-Learn [Ped+11] model fitting, without any sophisticated hyper-parameter tuning.

For the Movie Reviews and 20Newsgroups datasets, features correspond to words, using a TF-IDF representation. As in the previous chapter, we assume a public vocabulary of 150000 words for the first two datasets (Movie reviews and 20newsgroups). For the language classification task, $n$-grams of $n$ consecutive characters are used instead. We assume the set of characters is public.

### 5.7.3 $k$-Nearest Neighbors

For $k$-NN, the efficiency bottleneck is the computation of scores of the query document with respect to each training sample, which reduces to a secure matrix-vector multiplication where the matrix is sparse in its columns and the vector is sparse. Thus, we can implement this protocol using Protocol 16, instantiated with any of our ROOMs.

As observed in Section 5.5.1, the approach we used in Section 4.6 is equivalent to Circuit-ROOM, which is why we use it as the baseline here. In most of our experiments (Figure 5.8), this already turns out to be faster in than a simple dense multiplication.

Our new constructions using Basic-ROOM and Poly-ROOM achieve a similar improvement over the dense case (up to 82×) when it comes to total time, and at the same time a 2–5× improved online time compared Circuit-ROOM. Note that the online time includes top-$k$

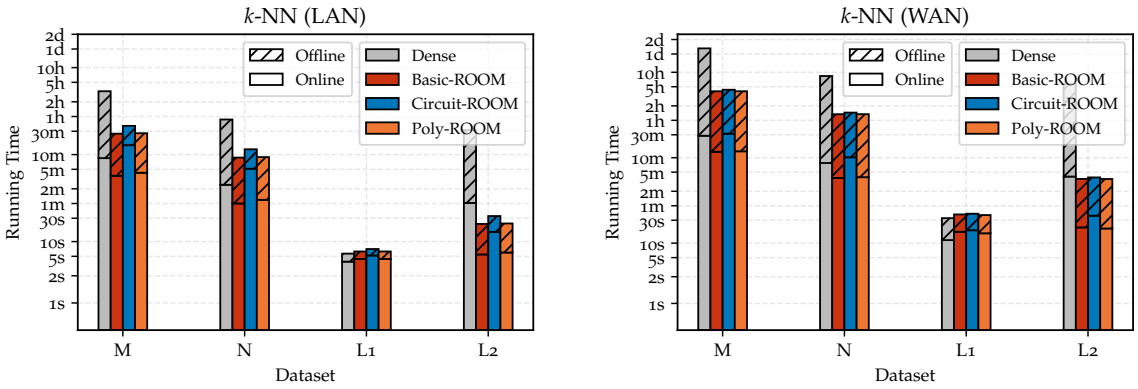



**Figure 5.8:** Running times of a single $k$-NN classification in LAN (left) and WAN (right) settings. Online time includes the time taken for ROOM queries, multiplication time with the reduced matrix, and time for top-$k$ selection. See Table 5.5 for a description of the datasets used.

selection (implemented using generic MPC) and multiplication of the reduced matrices.

#### 5.7.3.1 *Experiments on Naive Bayes*

We implemented our protocol described above, which consists of (i) a ROOM query for each of the $|C|$ potential classes composed with (ii) a protocol for securely computing the argmax of $|C|$ values. We do not include the latter here, since it only depends on the number of classes, which is usually significantly smaller than the dataset sizes (cf. Table 5.5). The running times, for each of the datasets, are shown in Figure 5.9. In both the LAN and WAN settings, Poly-ROOM generally outperforms Circuit-ROOM. This is consistent with the results from the previous section, since the queries are extremely sparse (i.e., $m$ is small). Note that neither Circuit-ROOM nor Poly-ROOM require a public vocabulary. If the vocabulary is public, then Basic-ROOM can be used as well. The plots in Figure 5.9 show the runtime for a public vocabulary of size 150000. In the LAN setting this gives a huge advantage: for the Movie Reviews dataset with >95k features, our protocol takes less than 2s. In contrast, the total classification time for a dataset with only 70 features took over 3 seconds in [BPTG15].

### 5.7.4 Logistic Regression Training

For each of our datasets, we also evaluate the time needed to build a logistic model using Protocol 17. We compare two approaches. One uses the state-of-the-art dense matrix multiplication protocol to instantiate MvMult (cf. lines 6 and 9 in Protocol 17), which is the extension of Beaver triples [Bea91] to matrices proposed in [MZ17]. The second approach uses Protocol 15 and our row-sparse protocol from Sec-

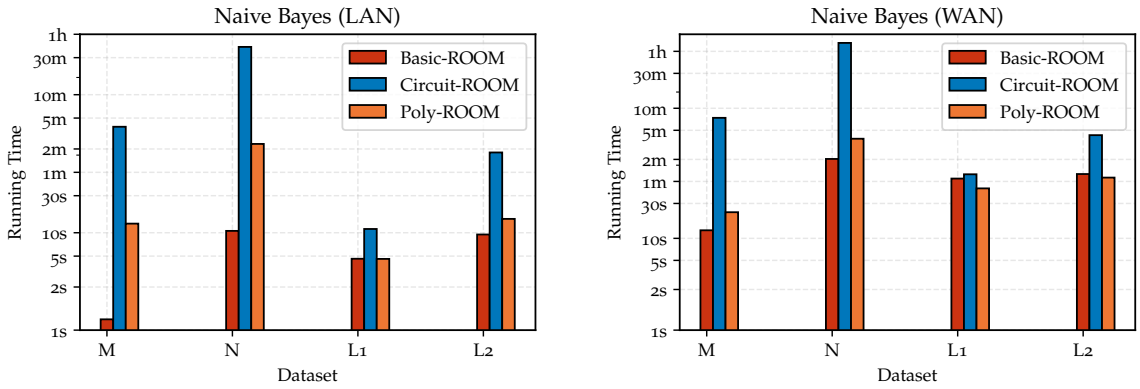



**Figure 5.9:** Running times for computing the conditional probabilities for a single Naive Bayes classification for each of the datasets in Table 5.5 and each of our ROOM constructions. Note that for Basic-ROOM, the vocabulary needs to be public. For private vocabularies, Poly-ROOM is the fastest in all cases.

tion 5.4.2.2 for forward and backward pass, respectively. We measure the online running time of a full run using both approaches, as well as the total amount of data transfered. We also estimate offline times using the measurements provided by Mohassel and Zhang [MZ17, Table II], and we present it together with the total time that includes both phases in Table 5.10.

While the dense solution [MZ17] achieves fast online computation, this comes at a significant offline computation and communication cost, requiring hours, even days in the WAN setting, and sometimes terabytes of communication. Our solution on the other hand saves a factor of 2×–11× in total runtime and a factor 26×–215× in communication in all reasonably large datasets (in "Languages, ngrams=1", SecureML is faster in total time, but both executions take just a few seconds).

Finally, we investigate how our solution scales with different dataset sparsities and the batch size used for training. For that, we run experiments on synthetic datasets. We use 1024 documents for each of the two servers, and vary the batch size between 128 and 1024. For each batch, we set the number of nonzero values between 1% and 10%. For comparison, the sparsity of a batch of 128 documents from the Movies or Newsgroups datasets is about 3%.

The results are shown in Figure 5.11. It can be seen that our sparse implementation benefits a lot from increasing the batch size. However, increasing the batch size will also increase the number of nonzeros per batch in real datasets, albeit sub-linearly. Thus, the batch size can be optimized to account for the sparsity of the dataset being used for training. Research on training ML models in the clear suggests larger batch sizes can be used without losing accuracy [HHS17], and we conjecture that this allows us to achieve even better speedups than those reported in Table 5.10, at the same level of accuracy. However,

| Dataset | Offline Time | | Total Time | | Offline Communication | | Total Communication | |
|---|---|---|---|---|---|---|---|---|
| | SecureML | Ours | SecureML | Ours | SecureML | Ours | SecureML | Ours |
| | | | | LAN | | | | |
| M | 6h 17m 45.06s | **14m 19.29s** | 6h 29m 28.37s | **2h 43m 46.09s** | 4.8 TiB | **186.25 GiB** | 4.8 TiB | **187.42 GiB** |
| N | 1h 39m 33.66s | **3m 34.55s** | 1h 42m 38.14s | **42m 37.68s** | 1.26 TiB | **46.5 GiB** | 1.26 TiB | **47.63 GiB** |
| L1 | 3.56s | **1.76s** | **5.9s** | 29.89s | 789.88 MiB | **390.75 MiB** | 790.9 MiB | **500.61 MiB** |
| L2 | 1h 01m 16.34s | **13.07s** | 1h 03m 07.12s | **6m 17.51s** | 796.82 GiB | **2.83 GiB** | 797.85 GiB | **3.69 GiB** |
| | | | | WAN | | | | |
| M | 4d 16h 34m 55.36s | **4h 18m 52.67s** | 4d 18h 35m 26.99s | **9h 29m 23.53s** | 19.19 GiB | **761.73 MiB** | 19.33 GiB | **1.92 GiB** |
| N | 1d 05h 40m 20.66s | **1h 05m 15.15s** | 1d 06h 09m 32.82s | **2h 26m 21.82s** | 5.01 GiB | **190.38 MiB** | 5.13 GiB | **1.31 GiB** |
| L1 | 1m 07.18s | **35.65s** | **1m 39.36s** | 3m 18.35s | 3.4 MiB | **1.9 MiB** | **4.42 MiB** | 111.76 MiB |
| L2 | 18h 15m 18.24s | **4m 1.77s** | 18h 34m 34.36s | **13m 36.84s** | 3.05 GiB | **11.71 MiB** | 4.08 GiB | **893.73 MiB** |

**Table 5.10:** Comparison of our protocols with SecureML [MZ17] for logistic regression training. Offline times are extrapolated from the results reported in [MZ17, Table II]. In all experiments, we use a batch size of 128. The total time represents a full training epoch, including forward pass, sigmoid activation function, and backward pass. Note that in the WAN setting, we use SecureML's homomorphic encryption-based offline phase that requires less communication. See Table 5.5 for a description of the datasets used.

in order to stay functionally equivalent to previous work [MZ17], we omit such optimizations at this point.

## 5.8 DISCUSSION

Privacy preserving machine learning algorithms often need to handle large inputs, and thus scalability is crucial in any solution of practical significance. Mirroring existing (centralized) computation frameworks, we leverage data sparsity to achieve this scalability, not only at the application level, but also in terms of lower-level operations.

A practical and principled approach to this problem calls for a modular design, where in analogy to the architecture of scientific computing frameworks, algorithms for linear algebra are built on top of a small set of low-level operations. With the ROOM framework presented in this chapter, we took a step in this direction, by defining sparse data structures with efficient access functionalities, which we then used to implement fast secure multiplication protocols for sparse matrices, a core building block in numerous ML applications. By implementing three different applications within our framework, we demonstrated the efficiency gain of exploiting sparsity in the context of secure computation for non-parametric ($k$-nearest neighbors and Naive-Bayes classification) and parametric (logistic regression) models, achieving manyfold improvements over the state of the art techniques. Beyond our three applications, the sparse linear algebra protocols

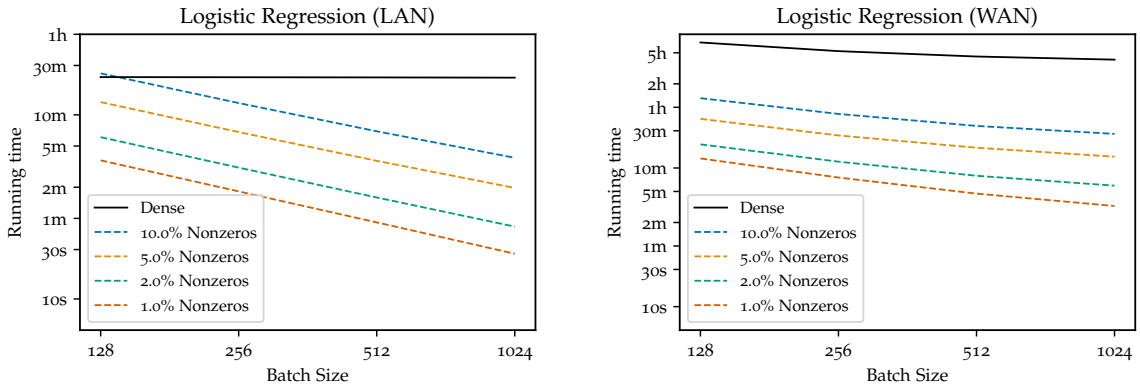



**Figure 5.11:** Total running time of a Stochastic Gradient Descent (SGD) training epoch for logistic regression. We use synthetic datasets with 1024 documents per server, a vocabulary size of 150k, and varying sparsity per batch. As in Table 5.10, offline times were extrapolated from SecureML [MZ17]. Note that the SecureML's HE-based offline phase used in the WAN setting also benefits from larger batch sizes, which is why the dense case is not constant.

implemented in our framework represent main building blocks for many machine learning algorithms.

At the same time, our modular design allows any of our protocols to be replaced by more efficient variants, thus making our framework somewhat future-proof. In the next chapter, we present two such improvements. The first, called *known-indices multi-point function secret sharing*, can serve as a more communication-efficient replacement of our ScatterInit protocol from Section 5.4.1. The second improvement, a pseudorandom *Vector-OLE* generator, can in some settings be used to speed up the offline phase of the dense matrix-vector multiplication protocol underlying our sparse constructions.

# 6 EFFICIENT DISTRIBUTED VECTOR OLE GENERATION

*The contents of this chapter have previously appeared in* Proceedings of the 2019 ACM Conference on Computer and Communications Security *[SGRR19].*

## 6.1 OVERVIEW

The ability to distribute correlated randomness between two parties in the absence of a trusted dealer is a central problem to cryptography. In the context of secure computation, this ability enables splitting the computation in an offline phase that is input independent and can be executed in advance, and an online phase that is very efficient.

Many previous works have focused on improving and optimizing methods for generation of correlated randomness in the context of oblivious transfer extension [Bea96; IKNP03; ALSZ17], which provides offline precomputation for two party computation based on garbled circuits [Yao86] or GMW [GMW87]; Beaver multiplication triples[1] [Bea91; DPSZ12; Dam+13; KOS16; KPR18], which are used in the offline phases of secure arithmetic computation protocols; and oblivious linear evaluation (OLE) [NP06; DKM12; Döt+17; BCGI18], which can be viewed as the equivalent of OT in arithmetic setting and which can be used for multiplicative triple generation.

[1] *The offline phases of the dense inner product and matrix multiplication protocols presented in previous chapters (Protocols 2 and 6) can be seen as generalizations of Beaver triples to vectors and matrices.*

In this chapter we focus on vector OLE (VOLE), which is the vectorized variant of an oblivious linear evaluation. More concretely, one party, the sender, holds vectors $\boldsymbol{u}, \boldsymbol{v}$ and a second party, the receiver has value $x$. The goal of the protocol is to enable the receiver to learn $\boldsymbol{w} = \boldsymbol{u}x + \boldsymbol{v}$ without revealing any further information to any of the parties. The concept of VOLE was introduced by Applebaum et al. [App+17]. In the recent work of Boyle et al. [BCGI18], the authors showed that VOLE is implied by a pseudorandom variant of the protocol where the vectors $\boldsymbol{u}, \boldsymbol{v}$ are pseudorandom and are generated during the execution of the protocol as outputs to the first party.

Succinctness is a crucial property for correlated randomness protocols, which aim to distribute long correlated outputs between the parties by communicating only short seeds. Boyle et al. [BCGI18] showed how to achieve succinctness in the setting of pseudorandom VOLE. The idea of their approach is to use a random linear code to extend short (sub-linear) seed vectors to long pseudorandom vectors. By masking the encoded vectors with a shared, sparse noise vector, they reduce security of their VOLE generator to the LPN assumption [BKW03]. The authors leverage functions secret sharing (FSS) for multi-point functions as a way to distribute the LPN noise vector to the inputs of the parties in an oblivious manner with succinct communication. This requires a two-party computation protocol for distributed generation of the FSS keys for the underlying multi-point function. The

proposed approach reduces the multi-point FSS to several executions of single point FSS by leveraging batching techniques. For distributed single-point FSS key generation, the authors suggest using the two party FSS key generation protocol of Doerner and Shelat [DS17].

In this work we address pseudorandom VOLE generation from a *practical* perspective. In particular, we focus on the *primal* variant of the protocol proposed in [BCGI18]. This is because to the best of our knowledge, there is no practical (i.e., implemented) construction of the "LPN-friendly" codes required for the *dual* variant. While the primal variant has a lower bound of $\tilde{O}(\sqrt{n})$ on its communication overhead, our implementation is still very efficient in practice. This is due to several improvements we make to the construction in [BCGI18]. Our main observation is that the non-zero *indices* of the shared sparse noise vector needed for LPN are part of the output to one of the parties. We use this observation in two ways. First, it allows us to use a more efficient batching scheme than what is proposed in [BCGI18]. Similar to previous work [ACLS18; DRRT18], we use cuckoo hashing to do *probabilistic batching*. This allows us to split up an instance of $t$-point FSS into $m = O(t)$ instances of single-point FSS, where $m$ is in practice only slightly larger than $t$. Second, we modify the FSS construction itself, which gives us a large constant-factor speedup in each FSS generation and evaluation. Our protocol is constant round, does not require secure PRG evaluations, has sub-linear communication, and like the distributed construction proposed by [BCGI18], provides security in the semi-honest model.

Our VOLE construction implies efficiency improvements in a wide range of applications such as secure linear algebra, sparse matrix multiplications and machine learning computations over sparse data, oblivious polynomial evaluation and private set intersection, and improved efficiency for semi-private data accessed in some ORAM constructions.

### 6.1.1 Chapter Contributions

As building blocks for our distributed VOLE protocol we develop constructions of several primitives of independent interest.

**A PROTOCOL FOR $(n-1)$-OUT-OF-$n$ RANDOM OT (SECTION 6.3)** An important component of our solution is a novel protocol for $(n-1)$-out-of-$n$ Random OT that requires one round and logarithmic communication. While any $m$-out-of-$n$ OT protocol requires communication $\Omega(m)$, we show how to leverage the fact that all messages are random to compress $n-1$ messages in a logarithmic number of seeds in manner oblivious to the sender. In terms of computation, an execution involves (i) $2n$ *local* PRG evaluations per party, and (ii) $\log_k(n)$ *parallel* executions of an $(k-1)$-out-of-$k$ OT protocol. In our implementation

we choose $k = 2$, and thus rely on 1-out-of-2 OT. Our $(n-1)$-out-of-$n$ Random OT implies a construction private puncturable PRF [BLW17], which enables a party to obtain a punctured PRF key at a location that remains secret to the full PRF key owner.

**KNOWN-INDEX SPFSS (SECTION 6.4).** The VOLE generation of Boyle et al. [BCGI18] uses SPFSS but assumes that one of the parties knows the input that evaluate to non-zero in the point function, while the function value us secret-shared. We propose a protocol for known-index SPFSS that outperforms the alternatives proposed in [BCGI18]. The protocol uses a reduction to $(n-1)$-out-of-$n$ Random OT, and thus leverages the protocol mentioned above. While known-index SPFSS implies distributed VOLE, its relevance is not limited to this application. It can be also viewed as a type of "scatter" vector operation, which is a core component in secure protocols for machine learning tasks (see Section 5.4.1 in Chapter 5).

**EFFICIENT KNOWN-INDICES MPFSS FROM SPFSS (SECTION 6.5).** To obtain a solution for distributed VOLE generation, we show an efficient reduction from known-indices MPFSS, where one party chooses the indices of the point function, to known-index SPFSS. Our reduction is based on Cuckoo hashing [PR04], and in practice it is very efficient, in particular when compared with the alternatives proposed by Boyle et al. [BCGI18].

**DISTRIBUTED VOLE (SECTION 6.6).** We combine the above protocol building blocks with some further optimizations, to obtain a full protocol for distributed VOLE generation.

**APPLICATIONS (SECTION 6.7).** We investigate several applications of our protocols, including linear algebra and matrix manipulation primitives commonly used in data analysis tasks. We show how our protocols yield concretely efficient secure two-party instantiations for Oblivious Polynomial Evaluation. We further show that our known-index SPFSS protocol can be used to improve the efficiency (both in asymptotic round complexity and concrete efficiency) of semi-private accesses in a recent FSS-based distributed ORAM construction [DS17].

**EXPERIMENTAL EVALUATION (SECTION 6.8).** Boyle et al. [BCGI18] provide estimates for runtime and communication, but they do not provide an implementation or an experimental evaluation. We implement all of our protocols, both over finite fields and integer rings, as well as the *primal* variant of the VOLE protocol proposed by Boyle et al. [BCGI18]. Instantiated over a finite field, we can generate a random VOLE of length $n = 2^{20}$ over a 60-bit prime field in about 1.2s. For

comparison, generating the same using standard Gilboa multiplication takes over 20s and has a 28× higher communication overhead.

**CONCURRENT AND SUBSEQUENT WORK** In recent concurrent work [Boy+19a], Boyle et al. present a two-round OT extension protocol based on Vector-OLE. As in our work, they observe that VOLE key generation can be performed in a constant number of rounds. Unlike us, they implement the more communication-efficient dual VOLE generator and provide malicious security. However, their implementation is limited to binary extension fields, as this is sufficient for the OT extension application they consider.

After publication ouf our work [SGRR19], Yang et al. [Yan+20] and Weng et al. [WYKW20] improved on it in multiple ways. First, Yang et al. [Yan+20] show that, because the primal variant of VOLE works by expanding a short (sublinear) VOLE correlation to a longer one, this process can be used iteratively to reduce the overall communication overhead. They also present an improved consistency check, obtaining malicious security at nearly no overhead compared to the semi-honest variant. However, as with Boyle et al. [Boy+19a], their work focuses on the application to OT extension and is thus limited to binary extension fields. Weng et al. [WYKW20] generalize Yang et al.'s protocols to arbitrary fields and apply them to efficient zero-knowledge proofs. We implement all of the improvements of [Yan+20; WYKW20] in our library. In Section 6.8.3.4 we show that the iterative bootstrapping approach of [Yan+20] reduces the communication overhead of our library by up to 96% in the semi-honest case.

## 6.2 PRELIMINARIES

### 6.2.1 $m$-out-of-$n$ Oblivious Transfer

As described in Section 2.4, oblivious transfer (OT) is a fundamental primitive in cryptography that allows a receiver to obliviously select one out of two messages held by a sender without revealing the selection bit to the sender, and without the receiver learning anything about the second message. In this chapter, we will generalize this functionality to the case where the sender has $n$ messages, and the receiver chooses $m < n$ of them. We formalize this in the following definition.

**Definition 6.1** ($m$-out-of-$n$ Oblivious Transfer (OT))**.** *An $m$-out-of-$n$ oblivious transfer is a protocol between two parties, sender and a receiver, where the sender has $n$ messages as input and the receiver has $m$ selection indices. The receiver obtains the messages corresponding to its indices while learning nothing about the remaining messages, and the sender learns nothing. If the $n$ messages are random and generated during the execution of the*

*protocol as output for the sender, the protocol is called random m-out-of-n OT, or m-out-of-n ROT.*

The communication complexity of 1-out-of-2 OT constructions is linear in the number of messages that the sender has. Naor and Pinkas [NP99] showed a reduction of 1-out-of-$n$ OT to $\log n$ instances of 1-out-of-2 OT, which yields logarithmic communication complexity. Observing that 1-out-of-$n$ OT is equivalent functionality to symmetric private information retrieval [CKGS98] is another approach to obtain an OT protocol with logarithmic complexity. Considering the general $m$-out-of-$n$ functionality the communication complexity naturally scales linearly in $m$ because we need to transfer at least that many messages. In Section 6.3 we show that this is no longer the case when we consider the $m$-out-of-$n$ ROT functionality and we present a protocol that requires only logarithmic communication. Just as random OT extension can be viewed as a PRF with single oblivious evaluation, we can view $(n-1)$-out-of-$n$ ROT as a privately punctured PRF [BLW17] where we generate a partial PRF key that enables evaluation of the PRF on all but one point, without revealing the punctured point to the PRF key holder.

#### 6.2.1.1 OT-based secure product a.k.a Gilboa multiplication

Gilboa [Gil99] proposed a two-party secure multiplication protocol of two $l$-bit numbers. The protocol outputs additive shares, and requires $l$ 1-out-of-2 OT that can be run in parallel (throughout this chapter we assume $l$ to be a constant, and set it to 64 in our experiments). Due to the practical efficiency of OT Extension protocols [IKNP03; ALSZ17], Gilboa multiplication is a common approach to secure cmultiplication. In particular, this approach has been used in several works [MZ17; DSZ15], as well as in Chapters 3–5, to compute Beaver triples for secure multiplication in the preprocessing model of MPC. In the context of this chapter, Gilboa multiplication is used for scalar vector multiplications. Considering practical implementations, one should note that this protocol can be implemented from correlated OT [ALSZ17], a more efficient variant of OT. Moreover, for the problem of scalar-vector multiplication, one can employ optimizations based on batching for concrete efficiency (see [MZ17] for details). We employ these optimizations in our implementation of secure scalar-vector multiplication based on Gilboa's protocol, which we use as a baseline.

### 6.2.2 Cuckoo Hashing

Cuckoo hashing [PR04] is an algorithm to build hash tables for (key, value) pairs with worst-case constant lookup. A cuckoo hash table is determined by $\kappa$ hash functions, where the value corresponding to a key is guaranteed to reside in one of the $\kappa$ locations determined by

the hash function evaluations on the key. Hash collisions are resolved using the cuckoo approach: if a collision occurs when placing an item in the hash table, the item residing in the location is evicted and then placed in the table using a different hash function, potentially evicting another item in the case of collision. This process continues until all evicted items are placed, if possible. Due to possible cycles in this graph of evictions, the insertion algorithm for cuckoo hashing has a chance to fail. For two hash functions, it is known that inserting $n$ items in a cuckoo tables of size $O(n)$ incurs more than $s$ insertion failures with probability bounded by $O(n^{-s})$ [KMW09]. The exact constants in this asymptotic bound are not known, but multiple papers have studied them empirically [DRRT18; PSZ18]. This is done by estimating, for any fixed statistical security parameter $\eta$, the number of hash functions and the cuckoo table size such that inserting $n$ items in the table fails with probability at most $2^{-\eta}$.

In cryptography, cuckoo hashing has been used as a probabilistic bath code to optimize Private Set Intersection (PSI) [CLR17; PSZ18; KKRT16] and Private Information Retrieval (PIR) [ACLS18] protocols. We introduce these ideas in Section 6.5, where we apply cuckoo hashing to obtain an optimized multi-point function secret sharing protocol.

### 6.2.3 Function Secret Sharing

Function secret sharing [BGI15; BGI16] is a primitive that allows a key generator to distribute the evaluation of a function between two parties in way that neither of the two parties learns anything about the evaluated function, but jointly the two parties can recover the evaluation at any point.

**Definition 6.2** (Function Secret Sharing). *Let $\mathcal{F} = \{f : I \to \mathbb{G}\}$ be a class of functions with input domain $I$ and output group $\mathbb{G}$, and let $\lambda \in \mathbb{N}$ denote a security parameter. A function secret sharing scheme consists of the following two algorithms:*

- $(K_1, K_2) \leftarrow \mathsf{FSS.Gen}(1^\lambda, f)$ – *given a description of $f : I \to \mathbb{G}$, output two keys $K_1, K_2$.*
- $f_b(x) \leftarrow \mathsf{FSS.Eval}(b, K_b, x)$ – *given an evaluation key $K_b$ for $b \in \{1, 2\}$ and an input $x$, output a share $f_b(x)$ of the value $f(x)$.*

*We require the following guarantees from the above algorithms:*

***CORRECTNESS.*** *For any $f \in \mathcal{F}$, and any $x \in I$, when $(K_1, K_2) \leftarrow \mathsf{FSS.Gen}(1^\lambda, f)$, we have*

$$\Pr\Big[\sum_{b\in\{1,2\}} \mathsf{FSS.Eval}(b, K_b, x) = f(x)\Big] = 1.$$

**SECURITY.** *For any $b \in \{1,2\}$, there exists a ppt simulator $\mathsf{Sim}_b$ such that for any polynomial-size function sequence $f_\lambda \in \mathcal{F}$,*

$$\left\{K_b \;\middle|\; (K_1, K_2) \leftarrow \mathsf{FSS.Gen}(1^\lambda, f_\lambda)\right\} \stackrel{c}{\equiv} \left\{K_b \leftarrow \mathsf{Sim}_b(1^\lambda, \mathsf{Leak}_b(f_\lambda))\right\}. \quad (6.1)$$

Note that the only difference between this definition and the one of Boyle, Gilboa, and Ishai [BGI16] is the leakage function is allowed to be different for each party. In the standard FSS construction, $\mathsf{Leak}_1(f_\lambda) = \mathsf{Leak}_2(f_\lambda) = (I, \mathbb{G})$, i.e., FSS keys must be simulated given only the input and output domains for $f$.

While FSS is defined for any function, an FSS instantiation is non-trivial if the length of the FSS keys is sub-linear in the size of the function domain. In this regime of operation we have single point FSS (SPFSS) constructions for point functions which evaluate to zero on all but one of their domain points. Boyle et al. [BGI15] introduced an FSS constructions for point functions where the keys are of length logarithmic in the function domain size.

Multi-point FSS (MPFSS) is a generalization of FSS where the shared functions has a larger number of non-zero evaluations. However, for the purposes of Vector-OLE (cf. Section 6.2.4), we observe that it is enough to consider a relaxed variant of MPFSS, where one party knows the where $f$ is nonzero in the clear. We call this variant *known-indices MPFSS*, and we provide a reduction to cuckoo hashing and known-index SPFSS in Section 6.5.

### 6.2.4 Vector OLE

Oblivious linear evaluation (OLE) is functionality that enables two parties to obtain correlated outputs. One party has input values $u, v$. The second party has input $x$ and obtains as output $w = ux + v$. Similarly to the use of OT for garbled circuits, OLE is a basic building block for secure arithmetic computation enabling the generation of multiplicative triples. Vector OLE (VOLE) [App+17; BCGI18] is a generalization of OLE to the setting of vector inputs, i.e., one party has input vectors $\boldsymbol{u}, \boldsymbol{v}$, the other party has input value $x$ and obtains a vector $\boldsymbol{w} = \boldsymbol{u}x + \boldsymbol{v}$. Boyle et al. [BCGI18] present application of VOLE to secure computation and zero-knowledge constructions.

Analogously to OT there is a variant of VOLE referred to as pseudorandom VOLE, where the vectors $\boldsymbol{u}, \boldsymbol{v}$ are generated randomly during the protocol execution. They are then provided as output to the first party. This primitive suffices for the construction of VOLE as well as its applications [BCGI18]. In Section 6.6 we present a new pseudorandom VOLE construction that requires a weaker version of the distributed MPFSS functionality compared to the approach of Boyle et

al. [BCGI18], which can be implemented efficiently as we demonstrate in Section 6.8.

**Definition 6.3** (Pseudorandom VOLE)**.** *A pseudorandom VOLE consists of the following algorithms:*

- $(\mathsf{seed}_1, \mathsf{seed}_2) \leftarrow \mathsf{VOLE.Setup}(1^\lambda, n, \mathbb{F}, x)$ – *this algorithms takes vector length $n$, field $\mathbb{F}$ and value $x$ and outputs two seeds.*
- $\mathsf{VOLE.Expand}(b, \mathsf{seed}_b)$ – *if $b = 1$, output $(\boldsymbol{u}, \boldsymbol{v}) \in \mathbb{F}^n \times \mathbb{F}^n$, else if $b = 2$, then output $\boldsymbol{w} \in \mathbb{F}^n$.*

*The correctness of the protocol guarantees that $\boldsymbol{w} = \boldsymbol{u}x + \boldsymbol{v}$. The security property requires that $\mathsf{seed}_1$ does not reveal any information about $x$ and that $\mathsf{seed}_2$ does not allow to distinguish $(\boldsymbol{u}, \boldsymbol{v})$ from random vectors subject to the correctness property, i.e., for any ppt algorithm $\mathcal{A}$ the following holds:*

$$
\begin{aligned}
&\Big| \Pr[b = b' \;\Big|\; b' \leftarrow \mathcal{A}(\mathsf{seed}_1), \\
&\qquad\qquad (\mathsf{seed}_1, \mathsf{seed}_2) \leftarrow \mathsf{VOLE.Setup}(1^\lambda, n, \mathbb{F}, x_b), \\
&\qquad\qquad (\mathbb{F}, n, x_1, x_2) \leftarrow \mathcal{A}(1^\lambda)] - 1/2 \Big| < \mathsf{negl}. \\
&\Big| \Pr[b = b' \;\Big|\; b' \leftarrow \mathcal{A}(\boldsymbol{u}_b, \boldsymbol{v}_b, \mathsf{seed}_2), \\
&\qquad\qquad (\mathsf{seed}_1, \mathsf{seed}_2) \leftarrow \mathsf{VOLE.Setup}(1^\lambda, n, \mathbb{F}, x), \\
&\qquad\qquad (\mathbb{F}, n, x) \leftarrow \mathcal{A}(1^\lambda), (\boldsymbol{u}_1, \boldsymbol{v}_1) \leftarrow \mathsf{VOLE.Expand}(1, \mathsf{seed}_1), \\
&\qquad\qquad \boldsymbol{w} \leftarrow \mathsf{VOLE.Expand}(2, \mathsf{seed}_2), \\
&\qquad\qquad \boldsymbol{u}_2 \leftarrow_R \mathbb{F}^n, \boldsymbol{v}_2 \leftarrow \boldsymbol{w} - \boldsymbol{u}_2 x] - 1/2 \Big| < \mathsf{negl}.
\end{aligned}
$$

### 6.2.5 LPN Assumption

The learning parity with noise (LPN) assumption [BKW03] states that given the noisy dot product of many public binary vectors $\boldsymbol{a}_i$ with a secret binary vector $\boldsymbol{s}$ is indistinguishable from a string of random bits. Adding noise to a bit is equivalent the flipping the bit with a fixed probability. We use the following generalization of the LPN assumption to larger fields.

**Definition 6.4** (LPN Assumption)**.** *Let $\mathbb{C}$ be a probabilistic code generation algorithm which given inputs values $k, q$ and a field $\mathbb{F}$, outputs a matrix $\mathsf{A} \in \mathbb{F}^{k \times q}$. The LPN assumption with respect to $\mathbb{C}$ with dimension $k = k(\lambda)$, $q = q(\lambda)$ queries and noise rate $r = r(\lambda)$ states that for any PPT algorithm $\mathcal{A}$ the following holds:*

$$
\begin{aligned}
&\Pr[1 \leftarrow \mathcal{A}(\mathsf{A}, \boldsymbol{b}) \;\big|\; \mathbb{F} \leftarrow \mathcal{A}(1^\lambda), \mathsf{A} \leftarrow \mathbb{C}(k, q, \mathbb{F}), \boldsymbol{e} \leftarrow \mathsf{Ber}_r(\mathbb{F})^q, \\
&\qquad\qquad \boldsymbol{s} \leftarrow \mathbb{F}^k, \boldsymbol{b} \leftarrow \boldsymbol{s} \cdot \mathsf{A} + \boldsymbol{e}] \\
&\approx \Pr[1 \leftarrow \mathcal{A}(\mathsf{A}, \boldsymbol{b}) \;\big|\; \mathbb{F} \leftarrow \mathcal{A}(1^\lambda), \mathsf{A} \leftarrow \mathbb{C}(k, q, \mathbb{F}), \boldsymbol{b} \leftarrow \mathbb{F}^q].
\end{aligned}
$$

In our construction (Section 6.6) we use LPN instantiations with parapets settings as those described by Boyle et al. [BCGI18], i.e., high

dimension $k$, low noise rate $1/k^{\epsilon}$ for a constant $\epsilon$ and a polynomial number of queries $q = k + o(k)$. Since we focus on the primal variant of VOLE, we can instantiate $\mathbb{C}$ using a *local linear code*, where LPN is assumed to hold [App+17; BCGI18].

Ring-LPN is a variant of the LPN assumption defined over rings rather than fields. The security of this assumption is less studied but there are works that explore its use in the context of protocols for the purposes of efficiency [Hey+12; DP12]. In our implementation we evaluate the performance of our protocol both in the setting of a field and a ring which rely on the two variants of the LPN assumption.

### 6.2.6 Definitions, Functionalities, and Secure Two-Party Protocols

All the constructions in this chapter describe communication efficient two-party protocols for computing correlated vectors, for different types of correlations. This notion has recently been formalized as a *Pseudorandom Correlation Generator (PCG)* [Boy+19b]. As observed there, communication-efficient PCGs don't lend themselves to direct simulation-based security proofs. Intuitively, this stems from the fact that any simulator that takes as input the ideal pseudorandom output and produces succinct messages for the protocol would be able to compress pseudorandom strings, which is impossible. However, it was shown that in many applications (including all the applications of vector OLE we consider), a weaker security definition is sufficient [BCGI18; Boy+19b]. We will therefore use the same approach as Boyle et al. and split up our protocols in two phases, namely *setup* (or *generation*), and *expansion* (or *evaluation*). This allows us to use the following structure in our security proofs: First, (i) we define correctness and security requirements of the generation and expansion algorithms. Then, (ii) we define ideal functionalities for the two phases and show that they satisfy our definition. And finally (iii), we show that our protocols securely (and efficiently) implement the key generation functionality. We focus on presenting our two-party protocols in this chapter, and giving the intuition behind for both security and efficiency. Nevertheless, detailed definitions, functionalities, and security proofs for all our novel constructions are given in the chapter appendix.

## 6.3 $(n-1)$-OUT-OF-$n$ RANDOM OT

In this section we consider the question of oblivious selection of $n-1$ items out of $n$ in the case when all items are pseudorandom. This corresponds to Functionality 18, namely $(n-1)$-out-of-$n$ random oblivious transfer. If we allow linear communication, a protocol for Functional-

| **Functionality 18:** $(n-1)$-out-of-$n$-ROT |
|---|
| **Parties:** $\mathsf{P}_1$, $\mathsf{P}_2$ |
| **Input:** $\mathsf{P}_2$: Index $i \in [n]$ |
| **Output (for $\mathsf{P}_1$):** Pseudorandom vector $\boldsymbol{u} \in \mathbb{F}^n$ |
| **Output (for $\mathsf{P}_2$):** vector $\boldsymbol{v} = (\boldsymbol{u}_j)_{j \neq i}$ |

ity 18 can be easily obtained using oblivious selection techniques. We instead propose a protocol with sub-linear communication and linear computation. Our protocol consists of a key generation phase where $\mathsf{P}_1$ learns a key $K_1$ consisting of a single PRG seed $s_0$, and $\mathsf{P}_2$ learns a key $K_2$ consisting of $\log_k(n)$ PRG seeds, via $\log_k(n)$ *parallel* executions of a $(k-1)$-out-of-$k$ OT protocol, for parameter $k > 1$. Expanding the respective seeds to obtain their length-$n$ outputs takes $O(n)$ PRG evaluations per party.

**KEY GENERATION VIA A GGM TREE** We crucially leverage the fact that values are generated pseudo-randomly in order to obtain a protocol with the above communication complexity. Let us assume, without loss of generality, that $\log_k(n)$ is an integer. The $n$ values of $\boldsymbol{u}$ are generated from a single random seed $s_0$ using a GGM tree $T$ [GGM86] constructed using a PRG $G$ of stretch $k$, i.e. $G : \{0,1\}^\lambda \mapsto \{0,1\}^{k\lambda}$, for security parameter $\lambda$. More concretely, $T$ is an ordered complete $k$-ary tree of depth $\log_k(n)$ and $n$ leaves, with its nodes labeled with seeds in $\{0,1\}^\lambda$ (we will refer to nodes and their seeds/labels indistinctly). The label of the root is $s_0$, and the label $s_j$ of the $j$th child of a node $v$ is obtained from the seed of $v$, by applying the PRG $G$ and parsing the output as $(s_1|\cdots|s_j|\cdots|s_k)$.

**THE 2-PARTY PROTOCOL** Our protocol is presented as Protocol 19. First, $\mathsf{P}_1$, the sender, computes the tree $T$ locally from a seed $s_0$ (note that this can be done with $2n-1$ calls to $G$) and sets $s_0$ to be its key $K_1$. The rest of the protocol allows $\mathsf{P}_2$, the receiver, to recover all the seeds of $T$, except for the ones in the path to the $i$th leaf. This is done in a way that does not leak $i$ to $\mathsf{P}_1$, and requires only $\log(n)$ seeds, which will constitute $\mathsf{P}_2$'s key $K_2$, to be expanded locally. We now informally discuss the correctness and security of our protocol, as well as associated communication and computation costs.

Let $(i_1, \ldots, i_{\log_k(n)})$ be the path to the $i$-th leaf (this is a sequence of values in $\{0, \ldots, k-1\}$, indicating which children to follow at each level to reach the $i$th leaf from the root, and in fact corresponds to the $k$-ary encoding of the integer $i-1$). For example, Figure 6.1 shows how for $n = 8$ and $i = 3$, the path the receiver should not learn is 010. As mentioned above, our goal is that the receiver can reconstruct all the tree except for the nodes on this path.

**Protocol 19:** $(n-1)$-out-of-$n$ Random OT

**Public Params:** PRG $\mathsf{G}$ of stretch $k > 1$ and security parameter $\lambda$, integer $n = k^c$ with $c > 0$
**Inputs:** $\mathsf{P}_1$: $\perp$; $\mathsf{P}_2$: index $i \in [n]$
**Output:** $\mathsf{P}_1$: $n$ random values $(r_j)_{j\in[n]}$
$\mathsf{P}_2$: $n-1$ random values $(r_j)_{j\in[n],j\neq i}$

**Key Generation (**$\mathsf{ROT.Gen}(1^\lambda, n, i)$**):**

(1) $\mathsf{P}_1$ generates a PRG seed $s_0 \xleftarrow{R} \{0,1\}^\lambda$.

(2) $\mathsf{P}_1$ computes a $k$-ary GGM tree of depth $\alpha = \log_k(n)$, denoted $T = T(s_0, \alpha)$, by associating $s_0$ to $T$ and, if $\alpha > 1$, constructing the $k$ children of $T$ recursively as $T(s_j, \alpha - 1)$, with $j \in [k]$ and seeds $s_1, \ldots, s_k$ computed as $(s_1 \mid s_2 \mid \ldots \mid s_k) := \mathsf{G}(s_0)$.

(3) $\mathsf{P}_2$ computes $(b_1, \ldots, b_\alpha)$, the $k$-ary encoding of $i-1$.

(4) The parties execute $\alpha$ instances of $(k-1)$-out-of-$k$ OTs:

- $\mathsf{P}_1$ acts as sender. For the $l$th OT, let $(p_1, \ldots, p_{k^l})$ be the seeds of the $l$th level of $T$. The $j$th message in the OT is set to be

$$m_j := \bigoplus_{s\in\{p_x \,:\, x \equiv j \bmod k\}} s$$

  (the $j$th message is the XOR of the seeds of the $j$th children of trees at level $l-1$).

- $\mathsf{P}_2$ acts as the chooser and inputs, in the $l$th OT, the set $\{0, \ldots, k-1\} \setminus \{b_l\}$, and obtains $k-1$ seeds $q_{l,j}$ with $j \in [k] \setminus \{b_l\}$.

(5) $\mathsf{P}_1$ outputs $K_1 \leftarrow s_0$, $\mathsf{P}_2$ outputs $K_2 \leftarrow (q_{l,j})_{l\in[\alpha],j\in[k]\setminus\{b_l\}}$.

**Expansion (**$\mathsf{ROT.Expand}(b, K_b)$**):**

(i) If $b = 1$: $\mathsf{P}_1$ returns the list of leaves of $T$.

(ii) If $b = 2$: $\mathsf{P}_2$ uses the seeds $q_{l,j}$ to reconstruct $T$, except for the path to the $i$th leaf (recall that $(b_1, \ldots, b_\alpha)$ is the $k$-ary encoding of $i-1$).

- For the first level, $\mathsf{P}_2$ constructs trees $T_j = T(q_{1,j}, \alpha - 1)$ with $j \in [k] \setminus \{b_1\}$.
- For each level $l \in [\alpha]$, let $T_1, \ldots, T_{k^l}$ be the sub-trees of $T$ at level $l$. In previous iterations $\mathsf{P}_2$ has computed all such sub-trees except for $T_{i_j}$, with $i_l = \sum_{x\in[l]} k^{x-1} \cdot b_x + 1$. $\mathsf{P}_2$ then collects the seeds of the direct children of each $T_j$ as $\{s_{j,1}, \ldots, s_{j,k}\}_{j\in[k]\setminus i_l}$. Then, additional seeds $\{s_{b_l,j}\}_{j\neq i_{l+1}}$ can be obtained from $(q'_{l,j})_{j\neq b_{l+1}}$ as

$$s_{b_l,j} := \bigoplus_{s\in\{s_{j,x} \,:\, x \equiv j \bmod k\}} s \oplus q_{l,j}$$

  By expanding those seeds using $G$, $\mathsf{P}_2$ computes all sub-trees of $T$ at level $l+1$, except for the one at position $i_{l+1} = \sum_{x\in[l+1]} k^{x-1} \cdot b_x + 1$.

$\mathsf{P}_2$ returns the list of seeds of leaves of $T$, except for the one at position $i = \sum_{x\in[\alpha]} k^{x-1} \cdot b_x + 1$.

Although it will become clear that the protocol can be parallelized across levels, for explanatory purposes it is useful to think of it as processing $T$ level by level from the root guaranteeing that, for each level $l \in [\log_k(n)]$, the receiver can reconstruct $T$ up to level $l$, except for the nodes in the path $(i_1, \ldots, i_l)$. This property obviously holds for $l = 0$ and, to argue the correctness of our protocol, we now argue inductively how to extend it from level $l$ to level $l + 1$.

By induction assume that the receiver can reconstruct all sub-trees $T_1, \ldots T_{k^l}$ of depth $\alpha - l$ rooted at the nodes of level $l$ except for exactly one: the one rooted at path $(i_1, \ldots, i_l)$. This is, precisely, $T_{(\sum_{x\in[l]} k^{x-1}\cdot i_x+1)}$, which we denote $T^*$ for simplicity. Now, let us show how a single execution of $(k-1)$-out-of-$k$ OT is enough to extend the above property to level $l + 1$. Intuitively, we want to ensure that the receiver learns all direct children of $T^*$, except for the $i_{l+1}$th one. As $T^*$ has $k$ direct children, this corresponds to a $(k-1)$-out-of-$k$-OT. However, for privacy, it is important that the sender never learns that $T^*$ is in fact the sub-tree that the receiver cannot reconstruct at level $l$, as this reveals too much about the index $i$. This difficulty can be overcome by constructing the messages in the $(k-1)$-out-of-$k$-OT as follows.

Let $s_{j,0}, \ldots, s_{j,k-1}$ be the seeds of the nodes that are direct children of each tree $T_j$. As the receiver knows all the $T_j$s except for $T^*$, she has all such seeds except for the ones with $j = (\sum_{x\in[l]} k^{x-1} \cdot i_x + 1)$, i.e., the children of $T^*$. The key idea to achieve the above goal is to have the sender compute $k$ values $m_0 = \left(\bigoplus_{j=1}^{k^l} s_{j,0}\right), \ldots, m_{k-1} = \left(\bigoplus_{j=1}^{k^l} s_{j,k-1}\right)$. Here, $m_0$ is the XOR of all direct first children of nodes at level $l$, $m_1$ is the XOR of all second children, and so on. Now observe that, given any value $m_y$ the receiver can compute the seed $s_{(\sum_{x\in[l]} k^{x-1}\cdot i_x+1),j}$ (the $y$th child of $T^*$) since she knows all the other values XOR-ed into the $m_j$ value. On the other hand, $m_j$ does not reveal anything about the seeds $s_{k^l,x}$ with $x \neq j$. Thus, the sender and the receiver run $(k-1)$-out-of-$k$ OT where the sender's inputs are $m_0, \ldots, m_{k-1}$ and the receiver's input is the set $\{0, \ldots, k-1\} \setminus \{i_l\}$. After running this sub-protocol the receiver can reconstruct $T$ up to level $l + 1$, except for the nodes in the path $(i_1, \ldots, i_{l+1})$. This shows how to extend the construction from level $l$ to $l + 1$, and the protocol finishes when $l = n$.

An important observation is that the instances of $(k-1)$-out-of-$k$ OT used in the above construction can all be run in parallel. The correctness of our construction follows from the above discussion, and its security, stated in the next lemma, follows directly from the security of $G$, and the underlying protocol for $(k-1)$-out-of-$k$ OT. A detailed proof can be found in Appendix 6.A.1. In Section 6.8 we describe how $G$ is instantiated in our implementation, as well as other practical considerations and optimizations.

**Lemma 6.5.** *For any constant $k > 1$, Protocol 19 is a secure two party computation protocol for the $(n-1)$-out-of-$n$ ROT functionality in the $(k-1)$-out-of-$k$ OT hybrid model assuming a secure PRG $G$. The protocol is one*

*round, and requires $O(\lambda \log(n))$ communication and $O(\lambda n)$ computation per party, including $2n$ PRG evaluations, where $\lambda$ is the length of the PRG seed.*

**PROOF SKETCH** Showing the security of the above protocol consists of two steps: first, showing that the keys that the parties receive have the desired pseudorandom properties (Definition 6.9), which follows from the pseudorandom properties of the GGM construction and which we formally prove in Theorem 6.10. And second, showing that the key generation protocol is a secure two party computation protocol for the generation of the keys, which follows from the OT security and which we prove formally in Theorem 6.11. The communication overhead follows from the fact that the parties execute $\log_k n$ OTs, which have linear communication in $\lambda$. The computation $O(\lambda n)$ for each comes from the execution of the $\log_k n$ OTs and the expansion of the keys which uses $2n$ PRG calls.

**HOW TO SET $k$, AND INSTANTIATIONS OF $(k-1)$-OUT-OF-$k$ OT** The construction of Protocol 19 works for any integer $k > 1$. Choosing $k$ constant results in logarithmic communication, and in fact in our implementation we use $k = 2$. In practice, this allows us to leverage very efficient implementations of 1-out-of-2 OT based on OT Extension. When instantiated with $k = 2$, our protocol resembles the Function Secret sharing construction by Boyle et al. [BGI15].

**PRIVATELY PUNCTURED PRF** Our $(n-1)$-out-$n$ random OT protocol also provides a construction for a privately punctured pseudorandom function, where one party has the PRF key and can evaluate the PRF on any input (in our case this is $P_1$ who has the GGM root) and the other party has a punctured key which allows it to evaluate the PRF on all but one inputs ($P_2$ in our case). The OT protocol enables $P_2$ to obtain its punctured PRF key without revealing the punctured point to $P_1$ (the punctured key is the output that $P_2$ has at the end of the KeyExchange phase of the OT protocol). We note the difference in the punctured key generation algorithm from the one defined in other contexts for privately puncturable PRFs [BLW17], where the party who has the full PRF key generates the punctured key and knows the point at which it is punctured.

## 6.4 KNOWN-INDEX SPFSS

In this section we use our $(n-1)$-out-of-$n$ random OT protocol to construct a 2-party computation protocol to jointly generate FSS keys for point functions. The setup for our distributed FSS protocol assumes that one party knows the non-zero evaluation point while the value at

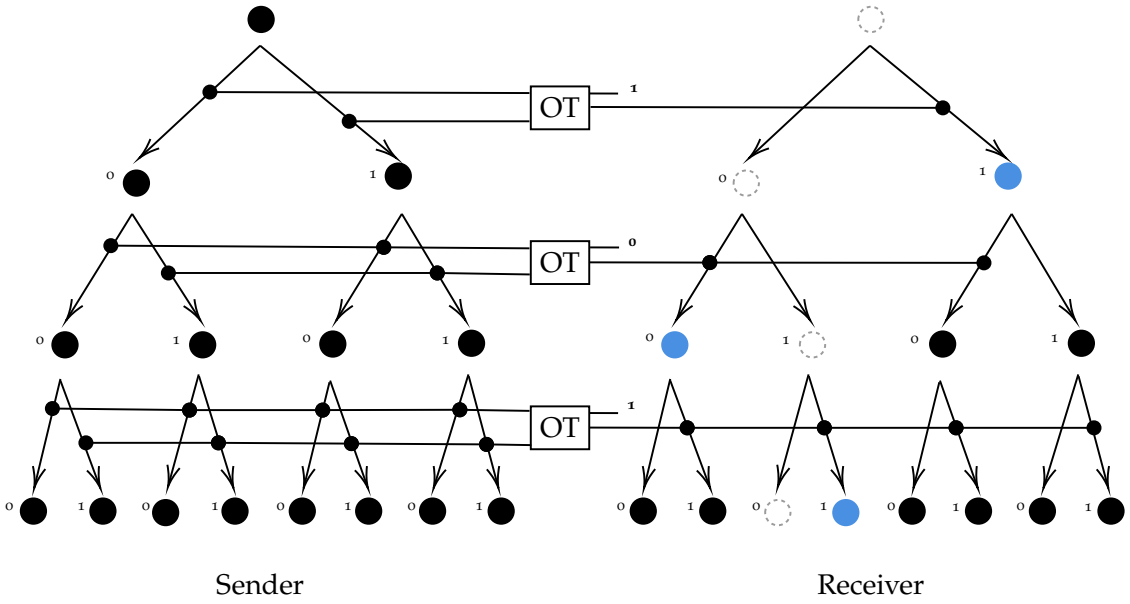



**Figure 6.1:** Example of the GGM tree generated by the sender and partially learnt by the receiver. Here, $k = 2$, $n = 8$, and $i = 3$. Thus, the path not learnt by the receiver is $(010)$. For each level, the parties run an OT where the receiver learns an XOR of either the left children or the right children of that level. Using previously expanded sub-trees, this information allows the receiver to learn a new seed (nodes filled in blue) which can be expanded by repeatedly calling $G$ (the nodes resulting from such expansions are filled in black). Figure contributed by co-author Leonie Reichert.

that point is shared between the two parties. Thus, it is not equivalent to a generic distributed FSS scheme for point functions, as for example described by Doerner and Shelat [DS17]. However, this relaxed version suffices for our VOLE construction described in Section 6.6. We call our FSS variant *Known-Index SPFSS* to emphasize that one party knows the non-zero index.

Conceptually, the existing construction of point function FSS [BGI15] generates two PRF keys $K_1$ and $K_2$ such that $\mathsf{PRF}_{K_1}(x) = -\mathsf{PRF}_{K_2}(x)$ for all values of $x$ except the input with non-zero evaluation $i$. The values $\mathsf{PRF}_{K_1}(i)$ and $\mathsf{PRF}_{K_2}(i)$ are random shares of the function evaluation $\beta$ at the input $i$. If two parties need to generate $K_1$ and $K_2$ in a distributed way, they can use secure general computation for this task, and Doerner and Shelat [DS17] show a more efficient way to construct such an MPC protocol in the semi-honest setting.

When one of the parties, $\mathsf{P}_2$, knows $i$, we can construct $K_1$ and $K_2$ in a distributed fashion as follows. First, $\mathsf{P}_1$ and $\mathsf{P}_2$ run a secure $(n-1)$-out-of-$n$-ROT key generation protocol (the construction from the previous section), for the parties to obtain keys $K_1^{\mathsf{ROT}}$ and $K_2^{\mathsf{ROT}}$. Note that, if the parties compute $\boldsymbol{r}^b = \mathsf{ROT.Expand}(b, K_b^{\mathsf{ROT}})$, the vectors $\boldsymbol{r^1}, \boldsymbol{r^2}$ coincide at every position except for a position $i$ known to $\mathsf{P}_2$. As $\mathsf{P}_2$ can negate its vector, we can think of $\boldsymbol{r^1}$ and $-\boldsymbol{r^2}$ as additive shares of a vector of all zeroes except for the $i$th position. Now all that

**Protocol 20:** Distributed Known-Index Single Point FSS

**Params and Building Blocks**: $(n-1)$-out-of-$n$-ROT;
Point function $f : [n] \to \mathbb{G}, f(i) = \beta, f(j) = 0 \; \forall \; j \neq i$
Random shares $\beta_1, \beta_2 : \beta_1 + \beta_2 = \beta; b \in \{0, 1\}$
**Parties:** $\mathsf{P}_1$, $\mathsf{P}_2$
**Inputs:** $\mathsf{P}_1 : \beta_1$, $\mathsf{P}_2 : \beta_2, i$

**Key Generation (**$\mathsf{SPFSS.Gen}(1^\lambda, f_{i,\beta})$**):**

(1) The parties run a secure $\mathsf{ROT.Gen}(1^\lambda, n, i)$ protocol to obtain keys $K_1^{\mathsf{ROT}}$ and $K_2^{\mathsf{ROT}}$.

(2) The parties execute locally ROT.Expand, from which $\mathsf{P}_1$ gets $n$ random values $\{r_i\}_{j\in[n]}$, and $\mathsf{P}_2$ obtains $\{r_i\}_{j\in[n],j\neq i}$.

(3) Let $R = \sum_{j\in[n]} r_i$. $\mathsf{P}_1$ sends to $\mathsf{P}_2$ the value $R_\beta = R - \beta_1$.

(4) $\mathsf{P}_2$ computes $\tilde{r} = \beta_2 - R_\beta + \sum_{j\in[n]\setminus\{i\}} r_i$.

(5) $\mathsf{P}_1$ outputs $K_1 \leftarrow K_1^{\mathsf{ROT}}$

(6) $\mathsf{P}_2$ outputs $K_2 \leftarrow (K_2^{\mathsf{ROT}}, \tilde{r})$.

**Expansion (**$\mathsf{SPFSS.Eval}(b, K_b, x)$**):**

- If $b = 1$, compute $\boldsymbol{v}^1 \leftarrow \mathsf{ROT.Expand}(1, K_1)$ and output $v_x^1$.
- If $b = 2$, parse $K_2$ as $\left(K_2^{\mathsf{ROT}}, \tilde{r}\right)$. If $x = i$, output $\tilde{r}$. Otherwise, compute $\boldsymbol{v}^2 \leftarrow \mathsf{ROT.Expand}\left(2, K_2^{\mathsf{ROT}}\right)$ and output $-v_x^2$.

remains is to modify $\boldsymbol{r^1}$ and $-\boldsymbol{r^2}$ to fix the $i$th position to be a share of a value $\beta$ shared among $\mathsf{P}_1$ and $\mathsf{P}_2$ (see Protocol 20). Crucially, this needs to be done in a way that does not leak $\beta$ to either party, and keeps $i$ private from $\mathsf{P}_1$. To do this we leverage the observation that $\mathsf{P}_1$ and $\mathsf{P}_2$ can compute sums $R = \sum_j r_j^1$ and $R' = \sum_{j\neq i} r_j^2$. The difference $R - R'$ will be the evaluation of $\mathsf{PRF}_{K_1}(j)$. Since the parties have shares $\beta_1$ and $\beta_2$ of the point function evaluation $\beta$ at $i$, we complete the protocol by $\mathsf{P}_1$ sending $R_\beta = R - \beta_1$ to $\mathsf{P}_2$ (note that this hides $\beta_1$ because $\mathsf{P}_2$ does not know $r_i^1$, which is a random mask) who can then appropriately set the $i$th entry of $\boldsymbol{r^2}$ so that $r_i^1 + r_i^2 = \beta$. Note that, as long as $\mathsf{P}_2$ obtains $R_\beta$ in the key generation phase, the corrections can be applied during expansion. Our construction is presented in Protocol 20 in terms of the key generation and expansion procedures for $(n-1)$-out-of-$n$-ROT from the previous section, which encompasses the steps from above.

**Lemma 6.6.** *Protocol 20 securely implements Known-Index SPFSS over a domain of size $n$ in the $(n-1)$-out-of-$n$-ROT hybrid model. With $(n-1)$-out-of-$n$-ROT instantiated by the construction of Protocol 19, Protocol 20 requires $O(\lambda \log n)$ communication and $O(\lambda n)$ computation per party where $\lambda$ is the security parameter of the ROT.*

**PROOF SKETCH.** The main argument in the security proof is that $R_\beta$ is a one-time pad that masks $\beta_1$, given the property of $(n-1)$-out-of-$n$-ROT that the output of $\mathsf{P}_1$ is a random vector. A detailed proof is given in Appendix 6.A.2.

## 6.5 KNOWN-INDICES MPFSS VIA CUCKOO HASHING

In this section we present a reduction from known-index multi-point FSS to known index single point FSS. The multi-point setting is analogous to the SPFSS functionality of Protocol 20, but extended to functions that fix the value of $t \geq 1$ points. We formalize our *Known-Indices MPFSS* variant in Definition 6.15 in the appendix. A naive reduction executes $t$ independent instances of known-index SPFSS on the original database. However, as observed by Boyle et al. [BCGI18], this requires evaluating all $t$ SPFSS instances on the whole domain, which results in an $\Omega(tn)$ computational overhead.

A general idea to improve on this baseline is to rely on batching schemes that split the domain of size $n$ into $m$ small parts in a way that allows to distribute the $t$ SPFSS instances across the $m$ smaller parts. One can instantiate this general idea using a combinatorial object called *batch codes* (see Ishai et al. [IKOS04] for an introduction). A batch code with parameters $n, t, k, m$ gives a partition of a database of size $n$ into $m$ parts such that any $t$ indices from $[n]$ can be recovered by reading at most $k$ entries in each of the $m$ parts. Although batch codes are attractive in that they offer very strong provable guarantees, they can be hard to instantiate in practice. This issue arises in the construction proposed by Boyle et al. [BCGI18], who explore Combinatorial Batch Codes (CBCs) for batching multiple FSS instances to obtain MPFSS. Since explicit constructions of the expander graphs required for instantiating a CBC do not satisfy their efficiency requirements, Boyle et al. propose a heuristic construction of a CBC. This leads to a small failure probability, which asymptotically depends on $t$ and the expansion factor of the batch code. However, concrete parameters for the heuristic CBC construction are not given by Boyle et al., and in their running time estimates, the authors assume $t$ SPFSS instances on disjoint subsets of $[n]$ instead of full MPFSS.

A second approach to baching is given by Angel et al. [ACLS18], who introduce a relaxed notion of Probabilistic Batch Codes (PBCs). Unlike the heuristic CBC construction of Boyle et al. [BCGI18], batching here may fail *on each insertion* of $t$ indices with a certain probability (which can be made arbitrarily small). The PBC construction of Angel et al. [ACLS18] is inspired by many works in the PSI literature [FHNP16; PSZ18; CLR17; DRRT18], where cuckoo hashing [PR04] is commonly used to reduce PSI to private set membership queries.

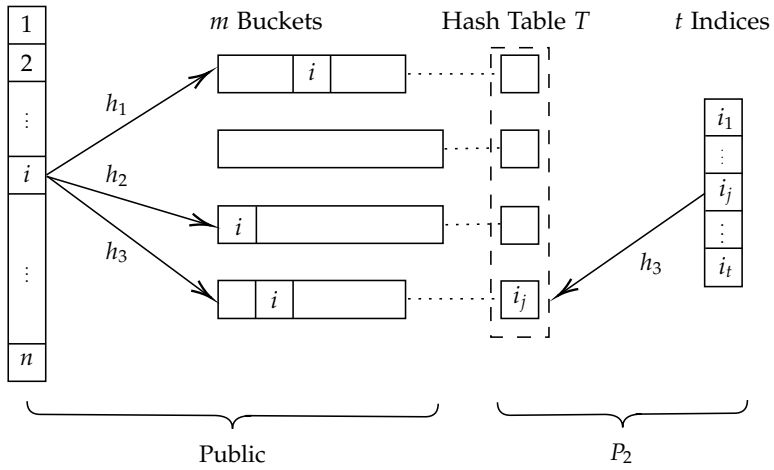



**Figure 6.2:** Simple hashing and cuckoo hashing for mapping indices to buckets in MPFSS. The domain of the multi-point function is hashed $\kappa$ times into $m$ Buckets using $\kappa$ different hash functions (this arrangement is public). $P_2$ privately builds a cuckoo hash table of the indices of the MP function. Then, an instance of known-index SPFSS is executed for each bucket. Figure contributed by co-author Leonie Reichert.

We follow this line of work, and base our MPFSS construction on probabilistic batching.

### 6.5.1 Batching Known-Index SPFSS

#### *6.5.1.1 Cuckoo hashing as a PBC*

Our approach to build MPFSS from single point FSS is to use Cuckoo hashing [PR04] and simple hashing in a similar manner as in PSI and PIR protocols [FHNP16; PSZ18; CLR17; DRRT18; ACLS18]. Cuckoo hashing [PR04] is a multi-choice hashing scheme with eviction parameterized by $\kappa$ universal hash functions $h_1, \ldots, h_\kappa$. Cuckoo hashing achieves the goal of distributing $t$ items in a table $T$ of size $m$ in a manner that guarantees that each location in $T$ is occupied by at most one item. The insertion algorithm puts the item to be inserted $x$ at $T[h_1(x)]$ and, if this position is occupied, evicts the item in that position and relocates it using $h_2$, which may cause yet another eviction resolved using $h_3$, and so on. This insertion algorithm may fail when a cycle of evictions is found, and thus cuckoo hashing has a failure probability that depends on parameters $\kappa, n, t$ and $m$. Multiple works [PSZ18; DRRT18] that use cuckoo hashing in secure computation protocols have empirically studied such parameters and how they relate to the failure probability. In our work we use the estimates of Demmler et al. [DRRT18], which leads to the same parameter choices used by Angel et al. [ACLS18] and Chen, Laine, and Rindal [CLR17].

While we present our concrete parameter choices in Section 6.8.2, we keep these symbolic in the protocol description for presentation purposes. We therefore introduce a statistical security parameter $\eta$, meaning that the probability of failing at hashing $t$ items is bounded by $2^{-\eta}$. More specifically, we denote by $\mathsf{ParamGen}(n, t, \eta)$ the function that generates cuckoo hashing parameters, i.e., number of hash functions $\kappa$ and cuckoo table size $m$, that guarantee this statistical bound on the insertion failure probability. Note that in the case of such an insertion failure, $\mathsf{P}_1$ learns of it in Step (1) of the protocol and thus can handle this case in several ways in practice. For example, it could simply abort the protocol, or it could sample new hash functions until the hashing step succeeds or a maximum number of trials is reached. In these cases, hashing failures result in leakage, as the adversary can infer information about the indices from the fact that they failed (or did not fail) to hash. A second option is to sacrifice correctness instead, and simply ignore indices that failed to hash. This way, no information is leaked from the generated MPFSS keys, but the multi-point function changes with a small probability. As discussed in [ACLS18; CLR17], the strategy for handling hashing failures depends a lot on the exact use case. In the case of vector OLE, we choose to drop indices that fail to hash (cf. Section 6.6). This is also the approach suggested by Boyle et al. [BCGI18] for their heuristic batch code construction. Our protocol therefore achieves the same type of security guarantee, while at the same time being concretely efficient.

#### 6.5.1.2 *Our protocol.*

Our construction is shown in Protocol 21. We use a cuckoo hashing scheme with capacity $t$ instantiated with $\kappa$ hash functions mapping $[n]$ to $[m]$, where $(m, \kappa) = \mathsf{ParamGen}(n, t, \eta)$ as described above. In step (1), the party holding the $t$ non-zero evaluation points of the multi-point function computes a cuckoo hash table $T$ of size $m$ that contains them. In step (2), the two parties use all of the $\kappa$ hash functions to simple-hash the whole domain $[n]$. This results in $m$ buckets $I_1, \ldots, I_m$, with $\kappa$ copies of each integer in $[n]$ distributed across them uniformly at random. An important point is that this arrangement is public (see left side of Figure 6.2). The parties also fix an order within each bucket $I_l$, and compute the reverse mapping $\mathsf{pos}_l$ from items to positions.

After having assigned indices to buckets, our protocol securely runs an SPFSS key generation for each bucket $I_l$. First, in step (3), the parties obtain shares of the vector $v$ of values to be fixed in each of the SPFSS instances (the value $\beta$ in Protocol 20). This needs to be done in a secure computation because both parties share all $\beta_i$'s, while only $\mathsf{P}_2$ knows which $\beta_i$ maps to which bucket. The secure computation can be implemented using permutation networks [Wak68] in a garbled circuit, or using additive homomorphic encryption. However, as we

**Protocol 21:** Distributed Known-Indices MPFSS

**Public Params**: Input domain $[n]$, number of points $t$, statistical security parameter $\eta$,
Cuckoo hash parameters: table size $m$, and number of hash functions $\kappa$, $(m, \kappa) = \mathsf{ParamGen}(n, t, \eta)$
Point function $f_{i,\beta} : [n] \to \mathbb{F}$, $f_{i,\beta}(i_j) = \beta_j^1 + \beta_j^2$ for all $j \in [t]$, $f_{i,\beta}(j') = 0$ for all other inputs.
**Parties:** $\mathsf{P}_1$, $\mathsf{P}_2$
**Inputs:** $\mathsf{P}_1$: $x$, $\beta_1^1, \ldots, \beta_t^1$; $\mathsf{P}_2$: $i_1, \ldots i_t$, $\beta_1^2, \ldots, \beta_t^2$

**Key Generation (**$\mathsf{MPFSS.Gen}(1^\lambda, f_{i,\beta})$**):**

(1) $\mathsf{P}_2$ randomly chooses $\kappa$ hash functions $(h_j)_{j\in[\kappa]}$, with $h_j : [n] \to [m]$. $\mathsf{P}_2$ inserts $i_1, \ldots i_t$ into a Cuckoo hash table $T$ of size $m$ using $h_1, \ldots, h_\kappa$, and it sends the $\kappa$ hash functions to $\mathsf{P}_1$. Let empty bins in $T$ be denoted by $\perp$.

(2) $\mathsf{P}_1$ and $\mathsf{P}_2$ do simple hashing with all $k_1, \ldots, h_\kappa$ on the domain $[n]$, to independently build $m$ buckets $I_1, \ldots, I_m$, i.e. $I_l = (x \in [n] \mid \exists p \in [\kappa] : h_p(x) = l)$, for $l \in [m]$, each sorted in some canonical order. The parties compute functions $\mathsf{pos}_l : I_l \to [|I_l|]$ that map values to their position in the $l$-th bucket.

(3) Let $\boldsymbol{u} = \left((\beta_j^1 + \beta_j^2, l_j)\right)_{j\in[t]}$, where $l_j$ is the location of $i_j$ in $T$. The parties run a secure 2PC protocol to obtain random shares $\boldsymbol{v^1}, \boldsymbol{v^2}$ of the vector $\boldsymbol{v} \in \mathbb{F}^m$ defined as

$$\boldsymbol{v}_j = \begin{cases} a & \text{if } (a, j) \in \boldsymbol{u}, \\ 0 & \text{otherwise.} \end{cases}$$

(4) For all $l \in [m]$, $\mathsf{P}_1$ and $\mathsf{P}_2$ run $SPFSS.Gen(1^\lambda, g_l)$ (Protocol 20) to obtain seeds $(K_1^l, K_2^l)$, with $g_l : [|I_l|] \to \mathbb{F}$ defined as

$$g_l(x) = \begin{cases} \boldsymbol{v_l^1} + \boldsymbol{v_l^2} & \text{if } T[l] \neq \perp \text{ and } x = \mathsf{pos}_l(T[l]), \\ 0 & \text{otherwise.} \end{cases}$$

(5) $\mathsf{P}_1$ outputs $K_1 = \left(K_1^l\right)_{l\in[m]}$ and $\mathsf{P}_2$ outputs $K_2 = \left(K_2^l\right)_{l\in[m]}$.

**Expansion (**$\mathsf{MPFSS.Eval}(b, K_b, x)$**):**
Output $\sum_{p=1}^{\kappa} \mathsf{SPFSS.Eval}\left(b, K_b^{h_p(x)}, \mathsf{pos}_{h_p(x)}(x)\right)$.

will see at the end of this section, this step can be omitted in the special case of Vector OLE.

In Steps (4) and (5), we generate and return SPFSS keys for each of the buckets, where the values are the shares obtained in the previous step, and the indexes are known to $\mathsf{P}_2$. Note that, since $m \geq t$,

some positions in $T$ might be empty, so those instances have the zero function associated to them (which is known only to $P_2$).

Finally, the evaluation of an MPFSS key on an input $x$ is the sum the evaluations of the SPFSS keys corresponding to the buckets into which $x$ is mapped by the cuckoo hash functions.

**Lemma 6.7.** *Assume a secure Known-Index SPFSS scheme with a secure two-party key generation protocol, both with security parameter $\lambda$. Then Protocol 21 implements a secure two-party protocol for generating Known-Indices MPFSS keys in the semi-honest model with statistical security $\eta$. Using Yao garbled circuits to instantiate step (2), the MPFSS.Gen protocol is constant round, and requires $O(m\lambda \log n)$ communication and $O(\lambda\kappa n + \lambda m \log n)$ local computation per party, where $(m, \kappa) = \mathsf{ParamGen}(n, t, \eta)$ are cuckoo hashing parameters.*

**PROOF SKETCH.** We outline the intuition for the proof of Lemma 6.7 here and provide the full proof in Appendix 6.A.3. Proving the security of the MPFSS protocol involves two steps: first, proving that the keys generated from the generation algorithm satisfy the FSS security requirements, and second, proving the generation protocol is a secure two party computation protocol that reveals to each party only its corresponding key. The first claim follows directly from the security guarantee of the SPFSS construction used to generate a key for each bucket. We prove this formally in Theorem 6.16. The second claim follows from the security of the two party protocol used for the SPFSS key generation, which we prove formally in Theorem 6.17.

The communication and computation for the garbled circuit used for Step (2) is $O(\lambda m \log m)$ since it needs to implement an oblivious permutation protocol over $m$ items. For each SPFSS instance in Step (3), we need $O(\lambda \log n)$ communication, since in the worst case each bucket has size $O(n)$. The computation that each party does includes simple hashing of all elements in $O(\lambda\kappa n)$, SPFSS distributed key generation for each bucket in $O(\lambda m \log n)$ and the MPFSS evaluation in $O(\lambda\kappa n)$.

**AN OPTIMIZATION FOR VECTOR-OLE** We leverage another observation related to the use of MPFSS in the context of vector OLE, which allows us to construct a more efficient solution. In the VOLE generator of Boyle et al. [BCGI18], the non-zero values for $t$-point MPFSS are of the form $xy_1, \ldots, xy_t$, where one party knows the indices of the non-zero function values and $y_1, \ldots, y_t$, and the other party knows $x$. Thus, we can have a secure two party computation protocol where one party inputs $y_1, \ldots, y_t$ padded with zero up to the size of the cuckoo table, in the order in which they are mapped to the cuckoo bins, and the other party inputs $x$. The protocol multiplies $x$ with the permuted vector and outputs shares of the result to the two parties. That way, we can generate the MPFSS keys needed for VOLE generation without

**Protocol 22:** MPFSS Optimization for VOLE

**Public Params**: Input domain $[n]$, number of points $t$, hash table size $m = \tilde{O}(t)$, and number of hash functions $\kappa$.
Point function $f_{i,xy} : [n] \to \mathbb{F}$, $f_{i,xy}(i_j) = (xy)_j$ for all $j \in [t]$, $f_{i,xy}(j') = 0$ for all other inputs.
**Parties:** $\mathsf{P}_1$, $\mathsf{P}_2$
**Inputs:** $\mathsf{P}_1$: $x$; $\mathsf{P}_2$: $i_1, \ldots i_t, y_1, \ldots, y_t$

**Key Generation (**$\mathsf{MPFSS.Gen}(1^\lambda, f_{i,xy})$**):**

(1, 2) *These are the same as in Protocol 21.*

(3a) Let $\boldsymbol{u} = ((y_j, l_j))_{j\in[t]}$, where $l_j$ is the location of $i_j$ in $T$. $\mathsf{P}_2$ locally computes the vector $\boldsymbol{w} \in \mathbb{F}^m$ defined as:

$$\boldsymbol{w}_j = \begin{cases} a & \text{if } (a, j) \in \boldsymbol{u}, \\ 0 & \text{otherwise.} \end{cases}$$

(3b) The parties run an MPC to compute shares of $\boldsymbol{v} = x\boldsymbol{w}$.

(4, 5) *The rest of the protocol is as Protocol 21.*

the expensive secure permutation in Step (3), and instead use a cheap multiplication protocol such as Gilboa multiplication [Gil99].

## 6.6 DISTRIBUTED VECTOR-OLE FROM MPFSS

In this section we present a new construction for two party computation of pseudorandom vector OLE that relies on multi-point function secret sharing. The main difference between our construction and the reduction described in the work of Boyle et al. [BCGI18] is the observation that the multi-point function that the two parties evaluate does not need to be completely hidden from both of them, since one of the keys contains the non-zero points in the clear. Thus, it suffices to use our distributed *Known-Indices MPFSS* from the previous section. We present our construction in Protocol 23.

The goal of a pseudorandom VOLE is to enable two parties $\mathsf{P}_1$ and $\mathsf{P}_2$ to obtain the following correlated outputs: $\mathsf{P}_1$ obtains vectors $\boldsymbol{u}$ and $\boldsymbol{v}$, and $\mathsf{P}_2$ obtains integer value $x$ and a vector $\boldsymbol{w}$ such that $\boldsymbol{u}x + \boldsymbol{v} = \boldsymbol{w}$. The requirements for these correlated outputs are that 1) $\boldsymbol{u}$ and $\boldsymbol{v}$ do not reveal information about $x$ and 2) given $\boldsymbol{w}$, $\boldsymbol{u}$ and $\boldsymbol{v}$ are indistinguishable from random vectors generated subject to the above relation. Without any further efficiency constraints the above functionality can be realized using standard MPC techniques. However, the goal here is to generate a VOLE correlation with much less communication than the length of the vectors. In this case, distributed VOLE faces the same

problems as other correlation generators (cf. Section 6.2.6 and Boyle et al. [Boy+19b]), i.e., that protocol messages of sublinear size can't be simulated from an ideal uniform output. Hence, the VOLE functionality is divided into two parts: an interactive setup protocol VOLE.Setup that produces short seeds for each party, and an expansion protocol VOLE.Expand that involves only local computation in which each party expands the short seed it has obtained from the setup to generate its long output vectors. It was shown that if these two phases satisfy Definition 6.3, the resulting pseudorandom correlation can securely be used for various applications of VOLE, such as secure arithmetic computation [BCGI18; Boy+19b].

The idea of the construction of Boyle et al. [BCGI18] is to start from short vector $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$ of length $k < n$ that have the required correlation, i.e., $\boldsymbol{c} = \boldsymbol{a}x + \boldsymbol{b}$, which the two parties can generate efficiently using MPC, and to expand them to long pseudorandom vectors using the LPN assumption. This assumption states that for appropriate code generating matrix $\boldsymbol{C} \in \mathbb{F}^{k \times n}$, the vector $\boldsymbol{u} = \boldsymbol{a} \cdot \boldsymbol{C} + \boldsymbol{\mu}$ is pseudorandom, where $\boldsymbol{\mu}$ is a sparse random vector. Now if we compute $\boldsymbol{v} = \boldsymbol{b} \cdot \boldsymbol{C} - \boldsymbol{v}_1$ and $\boldsymbol{w} = \boldsymbol{c} \cdot \boldsymbol{C} + \boldsymbol{v}_2$, where $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are shares of $\boldsymbol{\mu}x$, we will achieve the correctness property that $\boldsymbol{u}x + \boldsymbol{v} = \boldsymbol{w}$. Additionally, in order to get security, we need that $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are pseudorandom and do not reveal any information about $\boldsymbol{\mu}x$. This guarantees the pseudorandom properties of $\boldsymbol{u}$ and $\boldsymbol{v}$ under the correlation and that $\boldsymbol{v}_1$ (and hence $\boldsymbol{u}$ and $\boldsymbol{v}$) does not reveal any information about $x$.

Given the above idea, the heart of the VOLE generation is obtaining the shares $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ in a communication efficient manner. Boyle et al. [BCGI18] propose using a distributed multi-point FSS protocol. Our observation is that this functionality is more than what is needed for the pseudorandom VOLE construction. More specifically, an FSS protocol will guarantee that both shares $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ do not reveal any information about the multi-point function defined by $\boldsymbol{\mu}x$. However, while $x$ needs to remain hidden, $\boldsymbol{\mu}$ is revealed to $\mathsf{P}_1$, which in turn reveals the non-zero indices of $\boldsymbol{\mu}x = \boldsymbol{v}_1 + \boldsymbol{v}_2$. This observation is what allows us to use our known index MPFSS from Section 6.5 to generate the shares $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ more efficiently.

We note that as discussed in Section 6.5, our batching scheme introduces a small probability $2^{-\eta}$ of failing to batch all $t$ non-zero indices. This is also the case for the heuristic batch code construction of Boyle et al. [BCGI18]. However, as also pointed out there, this only strengthens the required LPN assumption a little: If batching fails (which results in some elements of $\boldsymbol{\mu}x$ becoming zero instead of nonzero), the distribution of noise values will only slightly deviate from uniform, but LPN for such a distribution remains a very conservative assumption.

**Theorem 6.8.** *Protocol 23 implements a secure distributed vector OLE generator in the semi-honest model. With step (3) instantiated with OT-*

**Protocol 23:** Distributed Vector OLE

**Public Params**: Vector length $n$, LPN parameters $t$, $k$, code generating matrix $\boldsymbol{C} \in \mathbb{F}^{k \times n}$.
**Parties:** $\mathsf{P}_1$, $\mathsf{P}_2$.
**Inputs:** None.
**Outputs:** $\mathsf{P}_1 : \boldsymbol{u}, \boldsymbol{v} \in \mathbb{F}^n$; $\mathsf{P}_2 : \boldsymbol{w} \in \mathbb{F}^n$, $x \in \mathbb{F}$, such that $\boldsymbol{u}x + \boldsymbol{v} = \boldsymbol{w}$.

**Share Generation (**$\mathsf{VOLE.Setup}(1^\lambda, \mathbb{F}, n)$**)**

(1) $\mathsf{P}_1$ chooses a set of $S$ random positions $S = \{s_1, \ldots, s_t\}$, with $s_i \in [n]$, $t$ random values $\boldsymbol{y} = (y_1, \ldots, y_t) \in \mathbb{F}^t$, and a pair of random vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{F}^k$. $\mathsf{P}_2$ chooses random $x \in \mathbb{F}$.

(2) $\mathsf{P}_1$ and $\mathsf{P}_2$ run $\mathsf{MPFSS.Gen}$ to obtain keys $K_1, K_2$ of the multi-point function $f_{S,x\boldsymbol{y}}$.

(3) $\mathsf{P}_1$ and $\mathsf{P}_2$ run an MPC with inputs $\boldsymbol{a}, \boldsymbol{b}$ and $x$ respectively, from which $\mathsf{P}_2$ obtains a vector $\boldsymbol{c} = \boldsymbol{a}x + \boldsymbol{b}$.

(4) $\mathsf{P}_1$ outputs $\mathsf{seed}_1 \leftarrow (K_1, S, \boldsymbol{y}, \boldsymbol{a}, \boldsymbol{b})$ and $\mathsf{P}_2$ outputs $\mathsf{seed}_2 \leftarrow (K_2, x, \boldsymbol{c})$.

**Expansion (**$\mathsf{VOLE.Expand}(b, \mathsf{seed}_b)$**)**

(i) If $b = 1$, $\mathsf{P}_1$ runs $\boldsymbol{v}_1[i] \leftarrow \mathsf{MPFSS.Eval}(1, K_1, i)$ for $i \in [n]$ and defines a vector $\boldsymbol{\mu} \in \mathbb{F}^n$ such that $\boldsymbol{\mu}[s_i] = y_i$ for all $i \in [t]$ and $\boldsymbol{\mu}[s] = 0$ for all $s \notin S$. $\mathsf{P}_1$ outputs $\boldsymbol{u} = \boldsymbol{a} \cdot \boldsymbol{C} + \boldsymbol{\mu}, \boldsymbol{v} = \boldsymbol{b} \cdot \boldsymbol{C} - \boldsymbol{v}_1$.

(ii) If $b = 2$, $\mathsf{P}_2$ runs $\boldsymbol{v}_2[i] \leftarrow \mathsf{MPFSS.Eval}(2, K_2, i)$ for $i \in [n]$ and outputs $\boldsymbol{w} = \boldsymbol{c} \cdot \boldsymbol{C} + \boldsymbol{v}_2$.

*based Gilboa multiplication,* MPFSS *instantiated using Protocol 22, and* $\boldsymbol{C}$ *instantiated by a local linear code, the protocol is constant round, and requires* $O(\lambda m \log n + \lambda k)$ *communication and* $O(\lambda \kappa n + \lambda m \log n)$ *computation per party, where* $(m, \kappa) \leftarrow \mathsf{ParamGen}(n, t, \eta)$*,* $\lambda$ *is a computational security parameter, and* $\eta$ *is the statistical security parameter of the MPFSS scheme.*

**PROOF SKETCH** As mentioned above, our protocol is obtained using a simple modification of the scheme of Boyle et al. [BCGI18], i.e., using known-index MPFSS instead of full MPFSS. Since the only additional information our variant reveals is already included in the VOLE keys, their proof [BCGI18, Section 3.2.2] can be trivially adapted to our protocol. We will give an overview here, but refer the reader to [BCGI18] for the full details.

Correctness follows from the observation that $\boldsymbol{\mu}x = \boldsymbol{v}_1 + \boldsymbol{v}_2$. It follows that

$$\begin{aligned}\boldsymbol{u}x + \boldsymbol{v} &= (\boldsymbol{a} \cdot \boldsymbol{C} + \boldsymbol{\mu})x + \boldsymbol{b} \cdot \boldsymbol{C} - \boldsymbol{v}_1 \\ &= (\boldsymbol{a}x + \boldsymbol{b})\boldsymbol{C} + \boldsymbol{\mu}x - \boldsymbol{v}_1 = \boldsymbol{c} \cdot \boldsymbol{C} + \boldsymbol{v}_2 = \boldsymbol{w}.\end{aligned}$$

To prove security we need to show that the two security properties from Definition 6.3 hold. To show the first property we observe that the only part of $\text{seed}_1$ that depends on $x$ is $K_1$. However, since it is generated using the distributed MPFSS construction, it follows by the security of known-index MPFSS (see Appendix 6.A.3) that there is a simulator that can simulate $K_1$ without knowledge of $x$. Note that the non-zero indices needed to simulate $K_1$ are also included in $\text{seed}_1$.

To prove the second property we show a transition between the distributions $(\boldsymbol{u}_1, \boldsymbol{v}_1, \text{seed}_2)$ and $(\boldsymbol{u}_2, \boldsymbol{v}_2, \text{seed}_2)$ in two steps and argue that an adversary cannot distinguish the changes applied in each of them. In the first step the input to the adversary is the same but we replace the $K_2$ with the simulated MPFSS key, which is generated from $\mathbb{F}$ and $n$ alone. Security of the MPFSS scheme guarantees that this simulated key is indistinguishable from the real one. In this distribution $\boldsymbol{u}_1 = \boldsymbol{a} \cdot \boldsymbol{C} + \boldsymbol{\mu}$ and $\boldsymbol{v}_1 = \boldsymbol{b} \cdot \boldsymbol{C} - \boldsymbol{v}_1 = \boldsymbol{b} \cdot \boldsymbol{C} + \boldsymbol{v}_2 - \boldsymbol{\mu}x = \boldsymbol{c} \cdot \boldsymbol{C} + \boldsymbol{v}_2 - (\boldsymbol{a} \cdot \boldsymbol{C} + \boldsymbol{\mu})x = \boldsymbol{c} \cdot \boldsymbol{C} + \boldsymbol{v}_2 - \boldsymbol{u}_1 x$. In the next step we replace $\boldsymbol{u}_1$ and $\boldsymbol{v}_1$ with $\boldsymbol{u}_2 \xleftarrow{R} \mathbb{F}^n$ and $\boldsymbol{v}_2 \leftarrow \boldsymbol{w} - \boldsymbol{u}_2 x = \boldsymbol{c} \cdot \boldsymbol{C} + \boldsymbol{v}_2 - \boldsymbol{u}_2 x$. By the LPN assumption, $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ are indistinguishable and since $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are computed in the same way, the change in the second step in indistinguishable for the adversary.

The communication in the protocol consists of the execution of the distributed MPFSS key generation and the secure computation for $\boldsymbol{c}$, which have cost $O(\lambda m \log n)$ and $O(\lambda k)$, respectively. The computation overhead additionally consists of the expansion of the MPFSS, which is $O(\lambda \kappa n)$ and the vector matrix multiplications with the matrix $C$, which using a local linear code is in $O(n)$.

In our evaluation (Section 6.8), rely on previous work [DRRT18] to choose a constant $\kappa$ and $m = O(t)$ such that $\eta \geq 40$ for the parameter ranges we're interested in. Together with the observation that $t$ and $k$ are both in $O(\sqrt{n})$ [BCGI18], this simplifies the communication overhead of our protocol to $O(\lambda \sqrt{n} \log n)$ and the computation to $O(\lambda n)$.

## 6.7 APPLICATIONS

Our distributed pseudorandom vector OLE protocol can be seen as a communication efficient precomputation that enables arbitrary secure two-party scalar-vector multiplications. This is thanks to a simple reduction from VOLE to pseudorandom vector OLE. The reduction is analogous to how a random multiplication triple can be exploited to compute extremely efficiently a secure multiplication in just a round of communication. The reduction from VOLE to pseudorandom VOLE is given in [BCGI18] (Proposition 10). The overhead of the reduction with respect to running pseudorandom VOLE generation and expanding the resulting seeds is just the cost of performing the scalar vector

multiplication in the clear, and transmitting a vector of the same length as the input vector. For that reason, in the context of multi-party computation, distributed pseudorandom vector OLE should be considered as a data independent preprocessing step that enables fast secure distributed scalar-vector multiplication, aka vector OLE. In this section we overview some applications that fit in this paradigm and thus can benefit from our protocol for distributed vector-OLE, as well as applications of our sub-protocol for known-Indices MPFSS.

Generally speaking, vector OLE can be used to batch one-against-many OLE computations, and thus directly provides a way to batch applications that rely on OLE computations. Such applications include, for example, PSI [GN19], and keyword search [FIPR05]. The latter relies on Oblivious Polynomial Evaluation (OPE) for which, as we will discuss later in this section, an efficient reduction to vector OLE exists. Our VOLE generator can also be used to implement PSI and variants thereof directly [RS21]. From a general MPC perspective, vector OLE enables communication efficient evaluation of arithmetic circuits with multiplication gates with large fan-out. This includes several important settings, including protocols for secure distributed data analysis.

### 6.7.1 Secure Linear algebra

As mentioned above, vector OLE is directly applicable in settings were OLE computations, i.e., secure multiplications, can be *vectorized* and thus computed by invoking several instances of vector OLE. This is the case, for example, in matrix-vector multiplication, as this operation can be computed, for a matrix of dimensions $n \times m$, by $m$ invocations of length $n$ vector OLE. Hence interesting settings for our protocols are the ones where $n$ is a lot larger than $m$. This corresponds to datasets with many records, and a limited number of features per record, which are natural in the context of training and evaluation of machine learning models, such as logistic regression. Similarly, matrix convolutions operations, the main ingredient of convolutional neural networks, rely on multiplying a small matrix called kernel (common kernel sizes are $3 \times 3$, $5 \times 5$, and $9 \times 9$) in a sliding fashion at each position of a input image (or layer input for intermediate layers). This corresponds to a small number of vector OLE computations of length the size of the image (which is commonly $255 \times 255$).

A natural approach to distributed vector OLE is (vectorized) Gilboa multiplication, as discussed in Section 5.2, and thus it has been used as a way to precompute multiplication triples for MPC in several works [MZ17; DSZ15]. This approach requires linear communication and computation in the size of the matrix. In contrast our Protocol 23 has sub-linear communication. In Section 6.8 we compare these two

approaches empirically, both in terms of communication and computation.

#### 6.7.1.1 Sparse matrix manipulations

As mentioned in the Section 6.1, known-indices MPFSS can also be seen as a type of "scatter" vector operation. This functionality was presented in Section 5.4.1 under the name ScatterInit. In that setting, two parties hold a share of a sparse vector, represented as a list of index-value pairs for which one party knows the indices and the values are additively shared. The goal is to securely convert the vector into a dense representation, where it is represented as an array of shared values of length equal to the size of the domain of indices. The protocol presented in Section 5.4.1 has computation and communication linear in the size of the dense vector. In contrast, known-indices MPFSS only needs linear computation, but has sub-linear communication overhead. Thus, it can be used directly as a replacement for ScatterInit in the row-sparse matrix-vector multiplication protocol of Section 5.4.2.2.

### 6.7.2 Oblivious Polynomial Evaluation

The problem of oblivious polynomial evaluation (OPE) considers the setting where one party, the server, has the coefficients of a polynomial $P(x)$ and a second party, the client, has an input $z$ and the goal of the protocol is to enable the client to learn $P(z)$ without learning anything more about the polynomial and without the server learning anything about the input. OPE has applications to privacy preserving set operations and data comparison, anonymous initialization for metering and anonymous coupons [NP06]. The OPE setting can be viewed as a generalization of the OLE problem to a higher degree polynomial.

We show that we can implement the OPE protocol leveraging the VOLE functionality. We use the OPE construction introduced in the work of Naor and Pinkas [NP06]. The idea is to reduce the evaluation of a degree $n$ polynomial to $n$ evaluations of linear polynomials, which can be executed in parallel. Next we overview the main idea of the reduction. Let $P(x) = a_n x^n + \cdots + a_1 x + a_0$ be a degree $n$ polynomial. It can be expressed as $P(x) = xQ(x) + b_0$ where $Q(x)$ is a degree $n-1$ polynomial. If the client and the server have obtained respectively additive shares $q_C, q_S$ of the evaluation of $Q(x) = q_C + q_S$, then $P(x) = q_C x + q_S x + b_0$. If the server fixes its share $q_S$ in advance, then the client's share $q_C = Q(x) - q_S = Q'(x)$ can be computed using oblivious polynomial evaluation of $Q'(x)$, which is of degree $n-1$ and its coefficients are known to the server. Now $P(x) = xQ'(x) + P'(x)$ where $P'(x) = q_S x + b_0$ is a linear polynomial. Therefore the OPE of $P(x)$ reduces to the oblivious evaluation of $Q'(x)$ and $P'(x)$, which

can be done in parallel. By induction we obtain that the evaluation of $P(x)$ can be reduced to the parallel evaluation of $n$ linear polynomials of the forms $w_i = P_i(x) = u_i x + v_i$ for $i \in [n]$ where the server knows the values $(u_i, v_i)_{i\in[n]}$ and the client knows $x$ and obtains $\{w_i\}_{i\in[n]}$. These corresponds to $n$ OLE evaluations, with the crucial aspect that one of the inputs is common to all of them. Hence an OPE of degree $n$ can be implemented with a single vector OLE computation where the server has two vectors of length $n$: $\boldsymbol{u}$ and $\boldsymbol{v}$, which consist of the values $\{u_i\}_{i\in[n]}$ and $\{v_i\}_{i\in[n]}$ respectively, and the client obtains $\boldsymbol{w} = \boldsymbol{u}x + \boldsymbol{v}$, which contains the values $\{w_i\}_{i\in[n]}$.

### 6.7.3 Partially Private Distributed ORAM

Doerner and Shelat [DS17] presented a distributed ORAM construction (FLORAM) that has asymptotically linear access time but achieves practically very competitive concrete efficiency. This advantage is even more pronounced in the RAM secure computation setting where this ORAM construction is used for memory access and the access queries are executed jointly by the two parties. The authors also consider semi-private queries which consist of both data dependent and data independent queries. In the latter type the parties know the accessed index. For these types of queries the FLORAM construction enables access in constant time.

We consider semi-private queries where the query index is known only to one of the parties. This corresponds to situations when data held by one party is indexed at private locations by the other party. We show that in this setting we can use our SPFSS construction and avoid having a Write-Only ORAM structure in the overall construction.

First, we briefly overview the FLORAM construction [DS17]. The ORAM in this construction consists of a Read-only ORAM, a Write-Only ORAM and a stash. The Read-Only ORAM consists of encryptions of the data under a key shared among the two parties. Each party has a copy of the Read-Only ORAM. The two parties execute an access query using a two server PIR construction based on SPFSS to retrieve the corresponding data item. They generate the distributed query running the distributed FSS key generation. The Write-Only ORAM consists of two XOR shares of the database, where each party holds one of the shares. It is updated with a write for a new item again using an SPFSS which evaluates to a non-zero value at the location of the write and this evaluation there is the XOR of the old value and the new value. The stash contains all the items that are currently in Write-Only ORAM. An ORAM access that hides read and writes consists of one Read-Only ORAM access, and one addition to the stash of the item that is written. Periodically all the content of the Write-Only ORAM is moved to the Read-Only ORAM using a special protocol with linear communication.

We observe that in setting of partially private queries where one of the parties knows the access index we can use our distributed only shared value FSS key generation presented in Section 6.4. This results in an improvement in terms of round communication, as the general SPFSS construction by Doerner and Shelat requires a logarithmic number of rounds. In Section 6.8 we show empirically the benefits of using our variant in the specific setting of semi-private queries by comparing two implementations of these protocols. Our results show improvements of up to an order of magnitude.

## 6.8 EXPERIMENTAL EVALUATION

### 6.8.1 Implementation and Setup

*Our source code is available at* `https://github.com/schoppmp/distributed-vector-ole`.

We implement all the protocols needed for Vector-OLE (Protocols 19, 20, 22, 23). Our implementation is written in C++. For OT extension we use EMP [WMK16], for finite field computations we use NTL [Sho+01], and for matrix multiplications needed in Protocol 23 we rely on Eigen [GJ+10]. We use AES to implement the PRG needed for Protocol 19. Just as the FLORAM implementation of Doerner and Shelat [DS17], we rely on the Davies-Meyer construction [Win84] to avoid repeated expansions of AES keys. We further interleave the setup and expansion phases in our implementation, and therefore only report the total time in each of our experiments.

All our experiments are done on Azure Dsv3 machines in the same region, using 2.4 GHz Intel Xeon E5-2673 v3 CPUs. For our comparisons against other protocols, we used a single thread. Note that this does not penalize any protocol in particular, since their local computations all parallelize well. To show the scalability of our protocol, we also implement a parallel version of it using OpenMP [DM98].

### 6.8.2 Parameter Selection

In our experiments, we use $\lambda = 128$ as the computational security parameter. Following the analysis in [BCGI18, Section 5.1], we choose the parameters for Vector-OLE (i.e., number of noise indices $t$ and number of rows in the code matrix $k$) such that known attacks on LPN require at least $2^{80}$ arithmetic operations. The concrete parameters depending on the vector size $n$ are given in Table 6.3. To instantiate the code generator $\boldsymbol{C} \in \mathbb{F}^{k \times n}$, we choose a local linear code with $d = 10$ non-zeros per column, which is also suggested by previous work on Vector-OLE from LPN [BCGI18; App+17]. Finally, we rely on the estimates in [DRRT18, Appendix B] to choose cuckoo hashing parameters such that hashing of the $t$ random indices fails with probability at most $2^{-40}$, i.e., $\eta = 40$. For the values of $t$ in Table 6.3 and $\kappa = 3$,

| $n$ | $2^{10}$ | $2^{12}$ | $2^{14}$ | $2^{16}$ | $2^{18}$ | $2^{20}$ | $2^{22}$ | $2^{24}$ |
|---|---|---|---|---|---|---|---|---|
| $t$ | 57 | 98 | 192 | 382 | 741 | 1422 | 2735 | 5205 |
| $k$ | 652 | 1589 | 3482 | 7391 | 15336 | 32771 | 67440 | 139959 |

**Table 6.3:** Vector-OLE parameters used in our evaluation. These were computed by Boyle et al. such that solving the corresponding LPN instance requires at least $2^{80}$ operations using either low-weight parity check, Gaussian elimination, or Information Set Decoding [BCGI18].

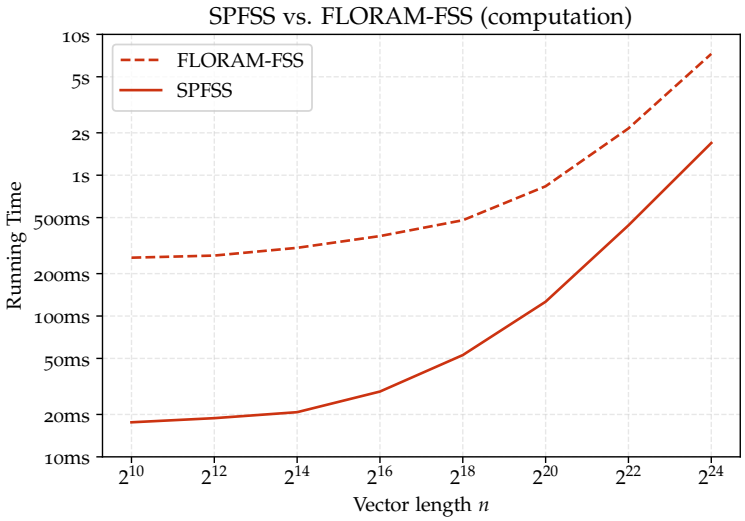



**Figure 6.4:** Comparison of our single-point FSS variant (Section 6.4) with the implementation of Doerner and Shelat [DS17].

this yields $m = 1.5t$. Those exact parameters have also been used in a previous work that uses cuckoo hashing for batching [ACLS18].

### 6.8.3 Results

#### *6.8.3.1 Comparison of Known-Index SPFSS with FLORAM*

First, we compare our distributed Known-Index SPFSS variant (Protocol 20) with the SPFSS implementation of Doerner and Shelat [DS17] in order to demonstrate the efficiency gain we can obtain in settings where the index might be known to one of the parties, e.g., semi-private accesses. The results are shown in Figure 6.4. Our implementation performs better for all vector lengths we tested. For short vectors, this is not surprising given that our protocol does not require expensive garbled circuits, but only $\log(n)$ oblivious transfers. Even for large vectors, where both protocols take time approximately linear in $n$, our implementation remains very efficient, which is made possible by the simplicity of our construction.

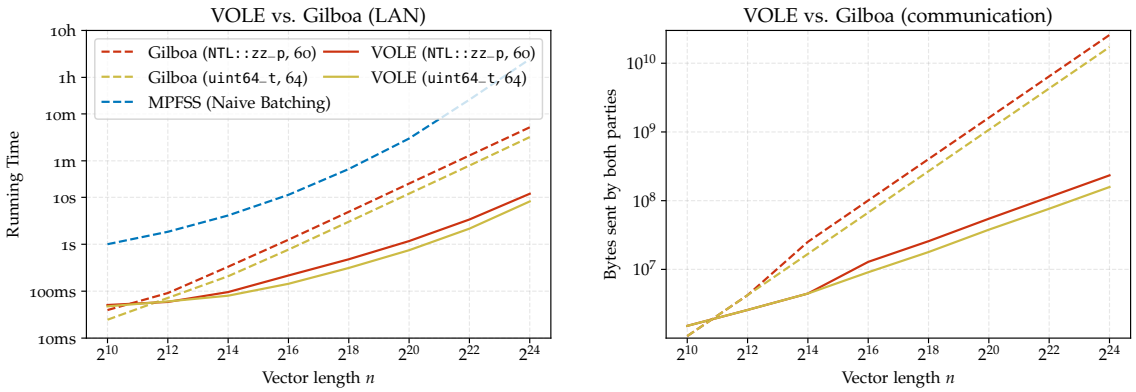



**Figure 6.5:** (Left) running time of our Vector-OLE (*VOLE*) implementation for generating a single random Vector-OLE. We compare against two baselines: Our known-indices MPFSS, but using naive batching instead of cuckoo hashing (cf. Section 6.5); and Gilboa multiplication [Gil99], which is also commonly used in the literature to implement two-party multiplications [MZ17; DSZ15]. We also compare two multiplication types: a 60-bit finite field (`NTL::zz_p`), and a 64-bit integer ring (`uint64_t`). It can be seen that VOLE outperforms both baselines as soon as $n > 2^{12}$. (Right) communication overhead of our Vector-OLE and Gilboa multiplication. Here, our VOLE implementation outperforms Gilboa multiplication for $n > 2^{11}$.

#### 6.8.3.2 Vector-OLE computation

We also measure the time it takes to generate a full Vector-OLE. Here, we compare our implementation of Protocol 23 against two baselines. First, our variant of MPFSS, but using naive batching by repeatedly evaluating over the whole domain. And second, our own implementation of Gilboa's multiplication protocol [Gil99]. We already heavily optimized this second baseline. In particular, we employ all of the optimizations from [MZ17], and our time per single-element multiplication is lower than the one reported in [MZ17].

Figure 6.5 (left) shows a comparison of wall-clock running times of all three approaches. Our first baseline, known-indices MPFSS with naive batching, is worse than both Gilboa and our VOLE in terms of asymptotics as well as concrete efficiency. As for the second baseline, Gilboa's multiplication is only slightly faster than our protocol for vector lengths below $2^{12}$, and slower for larger values of $n$. For both Gilboa and our VOLE, the finite field variant is slightly slower than the integer variant. This is due to the fact that in addition to the reduced computational overhead from the lack of modular reductions, using 64-bit integers directly allows us to use correlated OT [ALSZ17].

#### 6.8.3.3 Communication Experiments

We also investigate the communication overhead of both our VOLE implementation and Gilboa multiplication. To that end, we measure the number of bytes sent by both parties during the protocol ex-

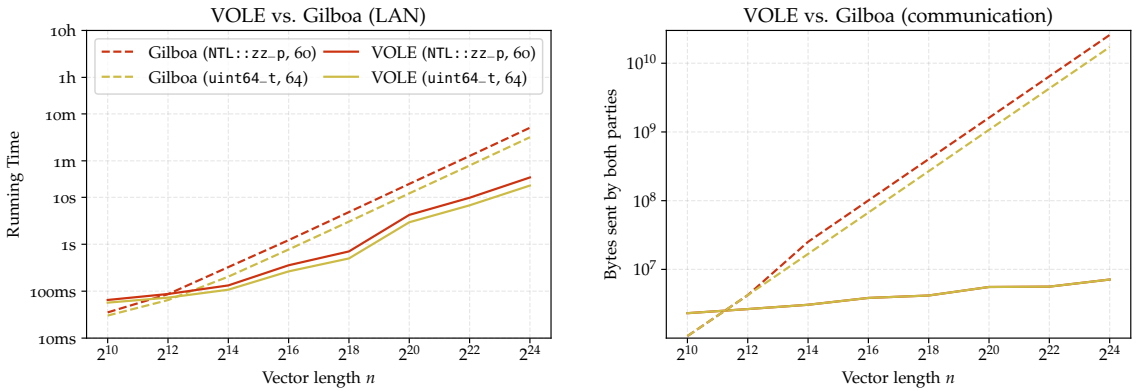



**Figure 6.6:** Communication and computation cost of our VOLE generator with the iterative bootstrapping approach of Yang et al. [Yan+20].

ecution. The results are shown in Figure 6.5 (right). Compared to Figure 6.5 (left), the cutoff point where our protocol outperforms Gilboa is slightly lower. We can also clearly observe the difference in asymptotic communication complexity given the different slopes in the log-log plot.

#### 6.8.3.4 *Incorporating Improvements of Yang et al. [Yan+20]*

Finally, we investigate how the iterative bootstrapping approach of Yang et al. [Yan+20] affects communication and computation when applied to our vector OLE generator. To that end, we re-run the experiments from Figure 6.5 with this variant. The results are shown in Figure 6.6. It can be seen that communication is drastically reduced by up to 96% when using bootstrapping, at the cost of a slightly increase running time.

## 6.9 DISCUSSION

In this chapter we presented a new protocol for shared randomness generation in the form of a random vector oblivious linear evaluation, which generates vectors with linear correlations. On the way to our final construction we also developed several new protocols, which are of independent interest, in the areas of random OT, private puncturable PRFs, and function secret sharing for single and multi-point functions with known indices. We showed how our VOLE construction can be leveraged in the context of several secure computation constructions, and compared them experimentally with two alternatives. We also implemented the bootstrapping approach of Yang et al. [Yan+20] and showed how it improves the communication complexity of our library.

Further possible improvements can be achieved in our lowest-level primitive, $(n-1)$-out-of-$n$-ROT. While our construction is based on GGM trees with arbitrary arity, our implementation is limited to binary trees and 1-out-of-2-OT. We believe that by using efficient $(k-1)$-out-of-$k$-OT sub-protocols for larger $k$, e. g.from homomorphic encryption, we can gain additional concrete efficiency. In terms of asymptotics, to our knowledge, ours is the first implementation of general Vector OLE with sub-linear communication. However, it does not reach the asymptotical guarantees that alternative constructions (in particular the dual version by Boyle et al.) provide, namely poly-logarithmic communication. This is due to the lack of concretely efficient, LPN-friendly encoding schemes, and we believe that if such encoding schemes become available, our implementation can yield poly-logarithmic communication complexity while staying concretely efficient.

# CHAPTER APPENDIX

## 6.A SECURITY PROOFS

In this section, we will prove security of all our main constructions, that is, Protocols 19, 20, and 21. We do not provide a full proof for Protocol 23, but as we discuss in Section 6.6, this proof can be obtained directly by taking the one given by Boyle et al. [BCGI18] and replacing their MPFSS construction by ours. As described in Section 6.2.6, we split our proofs in three phases, i.e., we (i) define correctness and security requirements, (ii) define ideal functionalities that satisfy these requirements, and (iii) prove our key generation protocols securely implement the ideal functionalities.

We note that our definitions are also closely related to the Pseudorandom Correlation Generators (PCGs) of Boyle et al. [Boy+19b]. However, as our key generators take additional arguments beyond the security parameter, we cannot use their definition out-of-the-box. Still, our $(n-1)$-out-of-$n$-ROT is defined in a similar way as PCGs. For our FSS variants, we stick to pure simulation-based proofs using Definition 6.2, which ensures they cen be used as a drop-in replacement for the constructions of Boyle et al. [BCGI18].

### 6.A.1 $(n-1)$-out-of-$n$-ROT

**Definition 6.9** (Pseudorandom $(n-1)$-out-of-$n$-OT Generator)**.** *A pseudorandom $(n-1)$-out-of-n-OT generator for a group $\mathbb{G}$ consists of the following two algorithms:*

- $(K_1, K_2) \leftarrow \mathsf{ROT.Gen}(1^\lambda, n, i)$ - *Outputs two keys when given an output size $n$ and a single index $i \in [n]$.*
- $\boldsymbol{v}^b \leftarrow \mathsf{ROT.Expand}(b, K_b)$ - *Given an evaluation key $K_b$ for $b \in \{1,2\}$, outputs a vector of length $n$.*

*Here, $\lambda \in \mathbb{N}$ denotes a security parameter. Additionally, the following properties must hold:*

***CORRECTNESS.*** *For any $n \in \mathbb{N}$ and $i \in [n]$, any pair $(K_1, K_2)$ in the image of $\mathsf{ROT.Gen}(1^\lambda, n, i)$, and $\boldsymbol{v}^b \leftarrow \mathsf{ROT.Expand}(b, K_b)$ for $b \in \{1,2\}$, we have that $\boldsymbol{v}^1$ is computationally indistinguishable from a uniformly random vector from $\mathbb{G}^n$, and $v^1_j = v^2_j$ for all $j \in [n] \setminus \{i\}$.*

**Functionality 24:** $(n-1)$-out-of-$n$-ROT

**Public Parameter**: $k$

**Key Generation (**$\mathsf{ROT.Gen}(1^\lambda, n, i)$**):**

(1) Run steps (1) and (2) from Protocol 19 as $\mathsf{P}_1$ to obtain a $k$-ary GGM tree $T$ with root $s_0$ and depth $\alpha = \log_k(n)$, using seeds of size $\lambda$.

(2) For each level $l \in [\alpha]$, let $(p_1, \ldots, p_{k^l})$ be the seeds of the $l$th level of $T$, and for each $j \in [k] \setminus \{b_l\}$, compute

$$q_{l,j} \leftarrow \bigoplus_{s \in \{p_x \,:\, x \equiv j \bmod k\}} s.$$

(3) Let $(b_1, \ldots, b_\alpha)$ be a $k$-ary encoding of $i-1$. Return $K_1 \leftarrow s_0$ and $K_2 \leftarrow \left(i, (q_{l,j})_{l \in [\alpha], j \in [k] \setminus \{b_l\}}\right)$.

**Expansion (**$\mathsf{ROT.Expand}(b, K_b)$**):**
Let $\alpha = \log_k(n)$.

- If $b = 1$, compute the GGM tree $T = T(K_b, \alpha)$ and output the $n$ leaves of $T$.
- If $b = 2$, parse $K_b$ as $\left(i, (q_{l,j})_{l \in [\alpha], j \in [k] \setminus \{b_l\}}\right)$, where $(b_1, \ldots, b_\alpha)$ is a $k$-ary encoding of $i-1$. Then run steps (5) and (7) of Protocol 19 as $\mathsf{P}_2$.

***SECURITY.*** *There are ppt simulators* $\mathsf{Sim}_b$ *for* $b \in \{1,2\}$ *such that for any* $n \in \mathbb{N}$ *and* $i \in [n]$,

$$\left\{K_1 \,\middle|\, (K_1, K_2) \xleftarrow{R} \mathsf{ROT.Gen}(1^\lambda, n, i),\right\} \stackrel{c}{\equiv} \left\{K_1 \,\middle|\, K_1 \xleftarrow{R} \mathsf{Sim}_1(1^\lambda, n)\right\}, \quad (6.2)$$

*and*

$$\left\{K_2, v_i^1 \,\middle|\, \begin{array}{l} (K_1, K_2) \xleftarrow{R} \mathsf{ROT.Gen}(1^\lambda, n, i), \\ \boldsymbol{v}^1 \leftarrow \mathsf{ROT.Expand}(1, K_1) \end{array}\right\} \stackrel{c}{\equiv} \left\{K_2, v_i^1 \,\middle|\, K_2 \xleftarrow{R} \mathsf{Sim}_2(1^\lambda, n, i), v_i^1 \xleftarrow{R} \mathbb{G}\right\}. \quad (6.3)$$

*Informally, the above security definition ensures that* $\mathsf{P}_1$ *does not learn anything about* $i$*, while* $\mathsf{P}_2$ *does not learn anything about* $v_i^1$*, i.e., the random value it chooses not to receive, beyond the fact that it is random.*

**Theorem 6.10.** *Functionality 24 is a pseudorandom generator for* $(n-1)$*-out-of-n-OT.*

*Proof. Correctness.* First, observe that a GGM tree $T$ with $n$ leaves and initial seed $s_0$ implements a PRF $F_{s_0} : [n] \to \{0,1\}^\lambda$ with key $s_0$, where $F_{s_0}(j)$ is the $j$-th leaf of $T$ [GGM86; KL14]. Since $s_0$ is chosen uniformly at random, $v^1 \leftarrow \mathsf{ROT.Expand}(1, K_1) = \left(F_{s_0}(j)\right)_{j\in[n]}$ is indistinguishable from a vector drawn uniformly at random from $\mathbb{G}^n$. Second, observe that in $\mathsf{ROT.Expand}(2, K_2)$ in Functionality 24, all seeds of sub-trees of $T$ that do not lie on the path to the $i$-th leaf are recovered. Since the expansion of $G$ is deterministic, all leaves of these sub-trees are equal to the corresponding leaves in $T$, and therefore $v^1_j = v^2_j$ for all $j \in [n] \setminus \{i\}$.

*Security.* We construct simulators $\mathsf{Sim}_b$ for $b \in \{1,2\}$ as follows.

$b = 1$. Sample a random seed $s'_0 \in \{0,1\}^\lambda$ and output $s'_0$. Indistinguishability of the two sides in Eq. (6.2) follows immediately as $K_1$ on the left hand side is also sampled uniformly from $\{0,1\}^\lambda$.

$b = 2$. Let $\alpha = \log_k(n)$, and let $(b_1, \ldots, b_\alpha)$ be a $k$-ary encoding of $i-1$. Construct a partial GGM tree by following the path from the root to the $i$-th leaf, sampling uniformly random seeds for all siblings of nodes on that path, and expanding them using the GGM construction. Now, for each level $l \in [\alpha]$ and each $j \in [k] \setminus \{b_l\}$, compute $q'_{l,j}$ as in Step (2) of Functionality 24, and output $K'_2 \leftarrow \left(i, (q'_{l,j})_{l\in[\alpha], j\in[k]\setminus\{b_l\}}\right)$.

We will now show the indistinguishability of the two sides of Eq. (6.3) using a hybrid argument. We construct $\alpha + 1$ hybrid distributions by successively modifying ROT.Gen as follows. Let $\mathcal{H}^0$ be the left-hand side of Eq. (6.3), and let $(p_1, \ldots, p_\alpha)$ denote the nodes on the path from the root to the $i$-th leaf of the GGM tree generated in ROT.Gen. Now, for each $l \in [\alpha]$, construct $\mathcal{H}^l$ from $\mathcal{H}^{l-1}$ by replacing the result of the PRG expansion of $p_l$ by $k$ uniformly random strings from $\{0,1\}^\lambda$, and proceeding normally from then on to compute $K'_2$. Note that neither $H^{l-1}$ nor $H^l$ contain $p_l$, but both contain at least one of the children. Thus, any distinguisher between $\mathcal{H}^{l-1}$ and $\mathcal{H}^l$ could be used to distinguish the output of a PRG from random. Now, by construction of $\mathsf{Sim}_2$ above, $\mathcal{H}^\alpha$ is precisely the right-hand side of Eq. 6.3 which concludes the security proof.

□

What remains to be shown is that the key generation of Protocol 19 securely implements Functionality 24. We reduce this to the security of the $(k-1)$-out-of-$k$-OT sub-protocol used in Protocol 19.

**Theorem 6.11.** *Steps (1) – (4) of Protocol 19 implement Functionality 24 in the $(k-1)$-out-of-k-OT-hybrid model with security against semi-honest adversaries.*

*Proof.* For each $b \in \{1,2\}$, we construct a simulator $\mathsf{Sim}_b$ for the view of $\mathsf{P}_b$ in the $(k-1)$-out-of-$k$-OT-hybrid model.

$b = 1$. Since $\mathsf{P}_1$ does not receive any messages in Protocol 19, $\mathsf{Sim}_1$ is the identity function. Since the computation performed is the same in Protocol 19 and Functionality 24, the simulated and real views are identically distributed.

$b = 2$. Here, in addition to the outputs of the ideal functionality, $\mathsf{P}_2$ receives the outputs of the OTs in Step (4). However, note that these are directly passed through to $\mathsf{P}_2$'s output and are therefore trivially simulatable. Since the values computed in Step (2) of Functionality 24 are precisely the ones selected by the OT functionality, the two views are again identically distributed.

□

We can now compose Protocol 19 with any $(k-1)$-out-of-$k$-OT protocol using a standard modular composition theorem, as for example given by Canetti [Canoo], thus obtaining a secure protocol in the plain model.

### 6.A.2 Known-Index SPFSS

Here, we define out Known-Index SPFSS as an instance of Definition 6.2 from the preliminaries section.

**Definition 6.12** (Known-Index SPFSS)**.** *Let $\mathcal{F} = \{f_{i,\beta} : [n] \to \mathbb{G}\}$ denote a class of point functions, where for all $x \in [n]$,*

$$f_{i,\beta} = \begin{cases} \beta & \text{if } x = i, \\ 0 & \text{otherwise.} \end{cases}$$

*A* Known-Index Single-Point Function Secret Sharing (Known-Index SPFSS) *scheme is a FSS scheme for $\mathcal{F}$, where $\mathsf{Leak}_1(f_{i,\beta}) = (I, \mathbb{G})$ and $\mathsf{Leak}_2(f_{i,\beta}) = (I, \mathbb{G}, i)$, i.e., we allow the recipient of $K_2$ to additionally learn the non-zero index $i$ (but not the value $\beta$).*

In Functionality 25, we define key generation and evaluation procedures for our known-index FSS scheme. We will now prove that this functionality indeed satisfies Definition 6.12, and that Protocol 20 implements the key generation phase securely.

**Theorem 6.13.** *Functionality 25 is a Known-Index Single-Point Function Secret Sharing scheme.*

*Proof. Correctness.* For any $j \in [n] \setminus \{i\}$, the correctness of the ROT scheme guarantees that $v_j^1 = v_j^2$, and hence $\mathsf{SPFSS.Eval}(1, K_1, j) + \mathsf{SPFSS.Eval}(2, K_2, j) = v_j^1 - v_j^2 = 0$. On the other hand, for $j = i$, we

**Functionality 25:** Known-Index SPFSS

**Key Generation** ($\mathsf{SPFSS.Gen}(1^\lambda, f_{i,\beta})$)**:**
Let $[n]$ denote the domain of $f_{i,\beta}$.

(1) Generate keys for a $(n-1)$-out-of-$n$-ROT scheme $\left(K_1^{\mathsf{ROT}}, K_2^{\mathsf{ROT}}\right) \leftarrow \mathsf{ROT.Gen}(1^\lambda, n, i)$.

(2) Compute $v^1 = \mathsf{ROT.Expand}(1, K_1)$ and $\tilde{r} = \beta - v_i^1$.

(3) Output $K_1 = K_1^{\mathsf{ROT}}$ and $K_2 = \left(K_2^{\mathsf{ROT}}, \tilde{r}\right)$.

**Expansion** ($\mathsf{SPFSS.Eval}(b, K_b, x)$)**:**
Let $\mathbb{G}$ denote the image of $f_{i,\beta}$.

- If $b = 1$, compute $v^1 \leftarrow \mathsf{ROT.Expand}(1, K_1)$ and output $v_x^1$.
- If $b = 2$, parse $K_2$ as $\left(K_2^{\mathsf{ROT}}, \tilde{r}\right)$. Note that $K_2^{\mathsf{ROT}}$ contains the non-zero index $i$. If $x = i$, output $\tilde{r}$. Otherwise, compute $v^2 \leftarrow \mathsf{ROT.Expand}\left(2, K_2^{\mathsf{ROT}}\right)$ and output $-v_x^2$.

have $\mathsf{SPFSS.Eval}(1, K_1, j) + \mathsf{SPFSS.Eval}(2, K_2, j) = v_i^1 + \tilde{r} = v_i^1 + \beta - v_i^1 = \beta$.

*Security.* We construct the following simulators $\mathsf{Sim}_b$ for $b \in \{1, 2\}$, assuming simulators $\mathsf{Sim}_b^{\mathsf{ROT}}$ for the random OT scheme used.

$b = 1$. Output $\mathsf{Sim}_1^{\mathsf{ROT}}(1^\lambda, n)$. Indistinguishability follows from Eq. (6.2) in Definition 6.9.

$b = 2$. Sample $r \xleftarrow{R} \mathbb{G}$ and output $\left(\mathsf{Sim}_2^{\mathsf{ROT}}(1^\lambda, n, i), r\right)$. Note that this distribution is the same as the right side of Eq. (6.3). Therefore, any distinguisher of the two sides of Eq. (6.1) could be used to distinguish the distributions in Eq. (6.3) by choosing a $\beta \leftarrow \mathbb{G}$ and replacing $v_i^1$ in Eq. (6.3) by $\beta - v_i^1$.

□

**Theorem 6.14.** *Steps (1)–(6) in Protocol 20 implement* $\mathsf{SPFSS.Gen}(1^\lambda, f_{i,\beta})$ *from Functionality 25 with security against semi-honest adversaries, where $i$ is input by $\mathsf{P}_2$ and $\beta$ is secret-shared between the two parties.*

*Proof.* We first prove that Protocol 20 is secure in the $(n-1)$-out-of-$n$-ROT-hybrid model when all calls to ROT.Gen are performed by the ideal Functionality 24. We construct simulators $\mathsf{Sim}_b$ for $b \in \{1, 2\}$ for the views of both parties in the ideal model.

$b = 1$. The only messages received by $\mathsf{P}_1$ come from the execution of ROT.Gen, and thus $\mathsf{Sim}_1$ is the identity function.

$b = 2$. Here, in addition to the output of ROT.Gen, $\mathsf{P}_2$ receives $R_\beta$. Simulate this with $\beta_2 - \tilde{r} + \sum_{j \in [n] \setminus \{i\}} \mathsf{SPFSS.Eval}(2, K_2, j)$. In the

$(n-1)$-out-of-$n$-ROT-hybrid model, this simulated view is distributed identically to the real view.

To prove security in the plain model, we again use the modular composition theorem for semi-honest security together with a secure protocol for ROT.Gen, as proven in Theorem 6.11 □

### 6.A.3 Known-Indices MPFSS

We will now prove security of our batched FSS implementation. However, as discussed in Section 6.5, there is a small probability that the batching fails (note that this is also the case for the heuristic batch code construction suggested by Boyle et al. [BCGI18]). Here we have two options if batching fails: We could abort the key generation, sacrificing security as this leaks some information about the non-zero indices that failed to be batched; or we could sacrifice correctness by returning keys that will result in shares of zeros for some indices that should be non-zero. Both are valid approaches depending on the concrete application, as also discussed in [ACLS18; CLR17]. For our VOLE construction, we will opt for the second choice, since this will allow us to achieve the same security guarantee as Boyle et al. [BCGI18], i. e., our scheme is either secure under standard LPN (if batching succeeds), or under a slightly stronger variant of LPN (if batching fails). See also the discussion in Section 6.6. We will not mention this explicitly in the following definitions and proofs, but whenever cuckoo hashing is performed, we assume that failures are handled by dropping indices that would result in a hashing failure.

**Definition 6.15** (Known-Indices MPFSS). *For any $t, n \in \mathbb{N}$, let $\mathcal{F} = \{f_{i,\beta} : [n] \to \mathbb{G}\}$ be a class of multi-point functions, where $\boldsymbol{i} \in [n]^t$, $\boldsymbol{\beta} \in \mathbb{G}^t$, and*

$$f_{i,\beta}(x) = \begin{cases} \beta_j & \text{if } x = i_j \text{ for some } j \in [t], \\ 0 & \text{otherwise.} \end{cases}$$

*Let further $\eta, \lambda \in \mathbb{N}$ denote statistical and computational security parameters, respectively. A* Known-Indices Multi-Point Function Secret Sharing (Known-Indices MPFSS) *scheme consists of the following two algorithms:*

- $(K_1, K_2) \leftarrow \mathsf{MPFSS.Gen}(1^\lambda, \eta, f)$ – *given a description of $f \in \mathcal{F}$, outputs two keys.*
- $f_b(x) \leftarrow \mathsf{MPFSS.Eval}(b, K_b, x)$ – *given a key for party $b \in \{1,2\}$ and an input $x \in [n]$, return a share of $f(x)$.*

*Where the following properties have to be satisfied:*

***CORRECTNESS.*** *For any $f \in \mathcal{F}$, and any $x \in I$, when $(K_1, K_2) \leftarrow \mathsf{MPFSS.Gen}(1^\lambda, \eta, f)$, we have*

$$\Pr\left[\sum_{b\in\{1,2\}} \mathsf{MPFSS.Eval}(b, K_b, x) = f(x)\right] \geq 1 - 2^{-\eta}.$$

**Functionality 26:** Known-Indices MPFSS

**Key Generation (**$\mathsf{MPFSS.Gen}(1^\lambda, \eta, f_{i,\beta})$**):**
Let $[n]$ denote the domain of $f_{i,\beta}$, and $t$ the number of non-zero points.

(1) Choose parameters $(\kappa, m) \leftarrow \mathsf{ParamGen}(n, t, \eta)$ for a cuckoo hashing scheme such that hashing any $t$ indices from $[n]$ fails with probability at most $2^{-\eta}$.

(2) Perform Steps (1) and (2) from Protocol 21, i. e., choose $\kappa$ random hash functions and use them to insert $(i_1, \ldots, i_t)$ into a cuckoo hash table $T$, and simple-hash the domain $[n]$. Let $\mathsf{pos}_l$ be defined as in Protocol 21.

(3) Let $\boldsymbol{u} = \left((\beta_j, l_j)\right)_{j\in[t]}$, where $l_j$ is the location of $i_j$ in $T$. Compute $\boldsymbol{v} \in \mathbb{G}^m$, where

$$v_j = \begin{cases} a & \text{if } (a, j) \in \boldsymbol{u}, \\ 0 & \text{otherwise.} \end{cases}$$

(4) Call $\mathsf{SPFSS.Gen}$ $m$ times as in Step (4) from Protocol 21 to obtain $m$ sets of keys $\left((K_1^l, K_2^l)\right)_{l\in[m]}$

(5) Output $K_b = \left((h_p)_{p\in[\kappa]}, (K_b^l)_{l\in[m]}\right)$ for $b \in \{1, 2\}$.

**Expansion (**$\mathsf{MPFSS.Eval}(b, K_b, x)$**):**
Parse $K_b$ as $\left((h_p)_{p\in[\kappa]}, (K_b^l)_{l\in[m]}\right)$ and output $\sum_{p=1}^{\kappa} \mathsf{SPFSS.Eval}\left(b, K_b^{h_p(x)}, \mathsf{pos}_{h_p(x)}(x)\right)$.

***SECURITY.*** *For any $b \in \{1, 2\}$, there exists a ppt simulator $\mathsf{Sim}_b$ such that for any polynomial-size function sequence $f_\lambda \in \mathcal{F}$,*

$$\left\{K_b \,\middle|\, (K_1, K_2) \leftarrow \mathsf{MPFSS.Gen}(1^\lambda, \eta, f_\lambda)\right\} \stackrel{c}{\equiv} \left\{K_b \leftarrow \mathsf{Sim}_b(1^\lambda, \eta, \mathsf{Leak}_b(f_\lambda))\right\}, \quad (6.4)$$

*where $\mathsf{Leak}_1(f_{i,\beta}) = ([n], \mathbb{G})$ and $\mathsf{Leak}_2(f_{i,\beta}) = ([n, \mathbb{G}], \boldsymbol{i})$.*

Note that the security guarantee of Definition 6.15 is the same as in Definition 6.2. The main difference is in the correctness guarantee, where we allow the output to be incorrect with a small probability depending on the statistical security parameter $\eta$.

Functionality 26 describes our MPFSS procedure. We will now prove its correctness and security guarantees according to Definition 6.15.

**Theorem 6.16.** *Functionality 26 is a Known-Index MPFSS scheme.*

*Proof. Correctness.* First, observe that the parameters for cuckoo hashing are chosen in Step (1) such that insertion fails with probability of

at most $2^{-\eta}$. Thus, it remains to show in the case that cuckoo hashing succeeds,

$$\begin{aligned} f_{i,\beta}(x) &= \sum_{b\in\{1,2\}} \mathsf{MPFSS.Eval}(b, K_b, x) \\ &= \sum_{b\in\{1,2\}} \sum_{p=1}^{\kappa} \mathsf{SPFSS.Eval}\big(b, K_b^{h_p(x)}, \mathsf{pos}_{h_p(x)}(x)\big) \\ &= \sum_{p=1}^{\kappa} g_{h_p(x)}(\mathsf{pos}_{h_p(x)}(x)) \end{aligned}$$

where $g_l(x)$ is defined as in Step (4) of Protocol 21. There are two cases.

1. $x = i_j$ for some $j \in [t]$. Then, since cuckoo hashing was successful, for exactly one $p^* \in [\kappa]$, $T\big[h_{p^*}(x)\big] = x$. Let $l^* = h_{p^*}$ be the location of $x$ in $T$. Then $g_{l^*}(\mathsf{pos}_{l^*}(x)) = \beta_j$, while $g_l(\mathsf{pos}_l(x)) = 0$ for all $l \in \big\{h_p(x) \,\big|\, p \in [\kappa] \setminus \{p^*\}\big\}$.

2. $x \notin \boldsymbol{i}$. Then for all possible locations $l \in \big\{h_p(x) \,\big|\, p \in [\kappa]\big\}$, $T[l] \neq x$ and thus $g_l(\mathsf{pos}_l(x)) = 0$.

*Security.* We construct simulators $\mathsf{Sim}_b$ for $b \in \{1,2\}$ by calling simulators $\mathsf{Sim}_b^{\mathsf{SPFSS.Gen}}$ for the SPFSS key generation algorithm used in Step (4) of Functionality 26.

Both simulators start by computing $(\kappa, m) \leftarrow \mathsf{ParamGen}(n, t, \eta)$ and sampling $\kappa$ random hash functions $(h_p)_{p\in[\kappa]}$. They then simple-hash the domain $[n]$, resulting in $m$ buckets of sizes $(I_l)_{l\in[m]}$.

$b = 1$. For each bucket $l \in [m]$, call $\mathsf{Sim}_1^{\mathsf{SPFSS.Gen}}(1^\lambda, I_l, \mathbb{G})$ to obtain keys $(K_1^l)$. Output $\Big((h_p)_{p\in[\kappa]}, (K_1^l)_{l\in[m]}\Big)$. Indistinguishability of the distributions in Eq. (6.4) follows from the fact that the $h_p$ (and therefore the bucket sizes $I_l$) are identically distributed, and for each bucket the simulated keys are indistinguishable from the real ones due to the security of the SPFSS.Gen procedure (Eq. (6.1)).

$b = 2$. Construct a cuckoo hash table $T$ of size $m$ using the hash functions $(h_p)_{p\in[\kappa]}$ and $i_1, \ldots, i_t$ as in Step (2) of Functionality 26. Now for each bucket $l \in [m]$, compute

$$K_2^l \leftarrow \mathsf{Sim}_1^{\mathsf{SPFSS.Gen}}\big(1^\lambda, I_l, \mathbb{G}, \mathsf{pos}_l(T[l])\big)$$

and output $\Big((h_p)_{p\in[\kappa]}, (K_2^l)_{l\in[m]}\Big)$. Again, indistinguishability follows from the fact that both views are identically distributed up to and including the creation of $T$, and then from the fact that the simulated and real keys for each bucket are indistinguishable by Eq. (6.1).

□

**Theorem 6.17.** *Protocol 21 implements* $\mathsf{MPFSS.Gen}(1^\lambda, \eta, f_{i,\beta})$ *from Functionality 26 with security against semi-honest adversaries, where* $\boldsymbol{i}$ *is input by* $\mathsf{P}_2$ *and* $\boldsymbol{\beta}$ *is secret-shared element-wise between the parties.*

*Proof.* We will first prove security assuming an ideal functionalities SPFSS.Gen (Functionality 25) for SPFSS key generation, and $\mathcal{F}^{\mathsf{2PC}}$ for generic two-party computation. Then we again rely on modular composition to obtain a protocol in the plain model. We construct simulators $\mathsf{Sim}_b$ for the the views of both parties $b \in \{1, 2\}$. Both simulators perform simple hashing to obtain bucket sizes consistent with the keys from the ideal output. Then, the simulation depends on $b$:

$b = 1$. The only messages $\mathsf{Sim}_1$ needs to simulate are the outputs of $\mathcal{F}^{\mathsf{2PC}}$, which by construction are equal to the inputs to the calls to SPFSS.Gen, since all other messages received by $\mathsf{P}_1$ are part of the output. Since by definition, $\boldsymbol{v^1}$ in Step (3) of Protocol 21 is a random share, this can be simulated by sampling $\boldsymbol{v^1} \xleftarrow{R} \mathbb{G}$. The resulting view is identical to the one in the (SPFSS.Gen, $\mathcal{F}^{\mathsf{2PC}}$)-hybrid model.

$b = 2$. $\mathsf{Sim}_2$ needs to first perform cuckoo hashing to generate a hash table $T$ consistent with the input indices $\boldsymbol{i}$ and hash functions from the ideal output. It can then call $\mathcal{F}^{\mathsf{2PC}}$ with a uniform vector $\boldsymbol{v^1} \xleftarrow{R} \mathbb{G}$ as above. The inputs to each SPFSS.Gen call are computed from $T$ as in Step (4) of Protocol 21. The resulting view is again identical to the one in the (SPFSS.Gen, $\mathcal{F}^{\mathsf{2PC}}$)-hybrid model.

□

# 7 CONCLUSION

As machine learning becomes an increasingly important tool in many disciplines, so do privacy-enhancing technologies that enable data holders to collaborate without revealing their private data. In this thesis, we have shown that secure multi-party computation (MPC) is well-suited for machine learning tasks on distributed data. By implementing and evaluating MPC versions of various machine learning algorithms, including linear and logistic regression, and $k$-nearest neighbors and naive Bayes classification, we have shown that MPC can scale to real-world dataset sizes, while providing provable security in well-defined threat models.

The scalability of our protocols was achieved by developing custom sub-protocols for the most expensive parts of each task, while using generic MPC for the rest of the computation. This modular approach also allows sub-protocols to be reused as building blocks across different ML tasks. In some settings, we have shown that further speedups can be obtained by relaxing the privacy guarantees and revealing additional information, such as the sparsity or other high-level statistics about the inputs. In these cases, differential privacy can be used to provably quantify and limit the amount of information revealed.

While the protocols in this thesis scale well with the sizes of the inputs, they do less so with regard to the number of computing parties. In fact, most of the protocols presented in this thesis are two-party protocols. In some cases, such as our linear regression protocol from Chapter 3, there can be many input parties, but the security in this case still relies on the fact that two of the computing parties are not allowed to collude. Extending the ideas from this thesis to the multi-party case without non-collusion assumptions is an important, albeit non-trivial next step. In terms of practical deployment, data harmonization and record linkage are two obstacles that have remained outside the scope of this thesis. Combining recent progress in secure record linkage based on MPC with the protocols in this thesis is another promising avenue towards real-world deployment of privacy-preserving machine learning.

Looking outside of the secure machine learning space, our work features several connections to other parts of cryptography. For example, as mentioned in Chapter 5, our ROOM functionality is close (but not equivalent) to private set intersection with secret-shared outputs (a.k.a. circuit-PSI), as well as the *private matching*-type protocols that are recently gaining traction in both academia and industry. We believe that progress in these areas will enable more efficient versions

of our protocols. In the other direction, our work in this thesis can be—and has been—used to improve cryptographic protocols for OT extension, zero-knowledge proofs, and (circuit-)PSI. We hope that our open-source implementations continue to help others advance research in these and other fields.

# BIBLIOGRAPHY

## REFERENCES


[Aba+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems." In: *CoRR* abs/1603.04467 (2016). URL: http://arxiv.org/abs/1603.04467 (visited on 10/20/2020).

[ASKG19] Nitin Agrawal, Ali Shahin Shamsabadi, Matt J. Kusner, and Adrià Gascón. "QUOTIENT: Two-Party Secure Neural Network Training and Prediction." In: *CCS*. ACM, 2019, pp. 1231–1247. DOI: 10.1145/3319535.3339819.

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. "PIR with Compressed Queries and Amortized Query Processing." In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2018, pp. 962–979. DOI: 10.1109/SP.2018.00062.

[App+17] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. "Secure Arithmetic Computation with Constant Computational Overhead." In: *CRYPTO (1)*. Springer, 2017, pp. 223–254. DOI: 10.1007/978-3-319-63688-7_8.

[Ash+20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. "OptORAMa: Optimal Oblivious RAM." In: *EUROCRYPT (2)*. Springer, 2020, pp. 403–432. DOI: 10.1007/978-3-030-45724-2_14.

[ALSZ17] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. "More Efficient Oblivious Transfer Ex-

tensions." In: *J. Cryptology* 30.3 (2017), pp. 805–858. DOI: 10.1007/s00145-016-9236-6.

[Atk] Kevin Atkinson. *Aspell dictionary creation*. URL: http://app.aspell.net/create (visited on 10/20/2020).

[BBGN19] Borja Balle, James Bell, Adrià Gascón, and Kobbi Nissim. "The Privacy Blanket of the Shuffle Model." In: *CRYPTO (2)*. Springer, 2019, pp. 638–667. DOI: 10.1007/978-3-030-26951-7_22.

[Bat68] Kenneth E. Batcher. "Sorting Networks and Their Applications." In: *AFIPS Spring Joint Computing Conference*. Vol. 32. AFIPS Conference Proceedings. Thomson Book Company, Washington D.C., 1968, pp. 307–314. DOI: 10.1145/1468075.1468121.

[Bea91] Donald Beaver. "Efficient Multiparty Protocols Using Circuit Randomization." In: *CRYPTO*. Springer, 1991, pp. 420–432. DOI: 10.1007/3-540-46766-1_34.

[Bea96] Donald Beaver. "Correlated Pseudorandomness and the Complexity of Private Computations." In: *STOC*. ACM, 1996, pp. 479–488. DOI: 10.1145/237814.237996.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. "The Round Complexity of Secure Protocols (Extended Abstract)." In: *STOC*. ACM, 1990, pp. 503–513. DOI: 10.1145/100216.100287.

[BGLB16] Jöran Beel, Bela Gipp, Stefan Langer, and Corinna Breitinger. "Research-paper recommender systems: a literature survey." In: *Int. J. Digit. Libr.* 17.4 (2016), pp. 305–338. DOI: 10.1007/s00799-015-0156-0.

[BNO08] Amos Beimel, Kobbi Nissim, and Eran Omri. "Distributed Private Data Analysis: Simultaneously Solving How and What." In: *CRYPTO*. Springer, 2008, pp. 451–468. DOI: 10.1007/978-3-540-85174-5_25.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. "Efficient Garbling from a Fixed-Key Blockcipher." In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013, pp. 478–492. DOI: 10.1109/SP.2013.39.

[BLO16] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. "Optimizing Semi-Honest Secure Multiparty Computation for the Internet." In: *CCS*. ACM, 2016, pp. 578–590. DOI: 10.1145/2976749.2978347.

[BL07] James Bennett and Stan Lanning. "The Netflix Prize." In: *KDDCup*. 2007, pp. 3–6. URL: https://www.cs.uic.edu/~liub/KDD-cup-2007/proceedings/The-Netflix-Prize-Bennett.pdf (visited on 10/20/2020).

[Ber11] Thierry Bertin-Mahieux. *YearPredictionMSD Data Set*. 2011. URL: https://archive.ics.uci.edu/ml/datasets/YearPredictionMSD (visited on 10/20/2020).

[BEWL11] Thierry Bertin-Mahieux, Daniel P. W. Ellis, Brian Whitman, and Paul Lamere. "The Million Song Dataset." In: *ISMIR*. University of Miami, 2011, pp. 591–596. URL: http://ismir2011.ismir.net/papers/OS6-1.pdf (visited on 10/20/2020).

[BKW03] Avrim Blum, Adam Kalai, and Hal Wasserman. "Noise-tolerant learning, the parity problem, and the statistical query model." In: *J. ACM* 50.4 (2003), pp. 506–519. DOI: 10.1145/792538.792543.

[BCG14] Carlo Blundo, Emiliano De Cristofaro, and Paolo Gasti. "EsPRESSO: Efficient privacy-preserving evaluation of sample set similarity." In: *J. Comput. Secur.* 22.3 (2014), pp. 355–381. DOI: 10.3233/JCS-130482.

[BKLS18] Dan Bogdanov, Liina Kamm, Sven Laur, and Ville Sokk. "Rmind: A Tool for Cryptographically Secure Statistical Analysis." In: *IEEE Trans. Dependable Secur. Comput.* 15.3 (2018), pp. 481–495. DOI: 10.1109/TDSC.2016.2587623.

[BLW17] Dan Boneh, Kevin Lewi, and David J. Wu. "Constraining Pseudorandom Functions Privately." In: *Public Key Cryptography (2)*. Springer, 2017, pp. 494–524. DOI: 10.1007/978-3-662-54388-7_17.

[BPTG15] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. "Machine Learning Classification over Encrypted Data." In: *NDSS*. The Internet Society, 2015. URL: https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/machine-learning-classification-over-encrypted-data (visited on 10/20/2020).

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. "Compressing Vector OLE." In: *CCS*. ACM, 2018, pp. 896–912. DOI: 10.1145/3243734.3243868.

[Boy+19a] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. "Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation." In: *CCS*. ACM, 2019, pp. 291–308. DOI: 10.1145/3319535.3354255.

[Boy+19b] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. "Efficient Pseudorandom Correlation Generators: Silent OT Extension and More." In: *CRYPTO (3)*. Springer, 2019, pp. 489–518. DOI: 10.1007/978-3-030-26954-8_16.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. "Function Secret Sharing." In: *EUROCRYPT (2)*. Springer, 2015, pp. 337–367. DOI: 10.1007/978-3-662-46803-6_12.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. "Function Secret Sharing: Improvements and Extensions." In: *CCS*. ACM, 2016, pp. 1292–1303. DOI: 10.1145/2976749.2978429.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping." In: *ACM Trans. Comput. Theory* 6.3 (2014), 13:1–13:36. DOI: 10.1145/2633600.

[Bud+20] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. "Private Matching for Compute." In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 599. URL: https://eprint.iacr.org/2020/599.

[Büs+18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. "HyCC: Compilation of Hybrid Protocols for Practical Secure Computation." In: *CCS*. ACM, 2018, pp. 847–861. DOI: 10.1145/3243734.3243786.

[BB13] Sahin Buyrukbilen and Spiridon Bakiras. "Secure Similar Document Detection with Simhash." In: *Secure Data Management*. Springer, 2013, pp. 61–75. DOI: 10.1007/978-3-319-06811-4_12.

[Buz12] Krisztian Buza. "Feedback Prediction for Blogs." In: *GfKl*. Studies in Classification, Data Analysis, and Knowledge Organization. Springer, 2012, pp. 145–152. DOI: 10.1007/978-3-319-01595-8_16.

[Buz14] Krisztian Buza. *BlogFeedback Data Set*. 2014. URL: https://archive.ics.uci.edu/ml/datasets/BlogFeedback (visited on 10/20/2020).

[Can00] Ran Canetti. "Security and Composition of Multiparty Cryptographic Protocols." In: *J. Cryptology* 13.1 (2000), pp. 143–202. DOI: 10.1007/s001459910006.

[CM20] Melissa Chase and Peihan Miao. "Private Set Intersection in the Internet Setting from Lightweight Oblivious PRF." In: *CRYPTO (3)*. Springer, 2020, pp. 34–63. DOI: 10.1007/978-3-030-56877-1_2.

[CD14] Kamalika Chaudhuri and Sanjoy Dasgupta. "Rates of Convergence for Nearest Neighbor Classification." In: *NIPS*. 2014, pp. 3437–3445. URL: http://papers.nips.cc/paper/5439-rates-of-convergence-for-nearest-neighbor-classification (visited on 10/20/2020).

[CV13] Kamalika Chaudhuri and Staal A. Vinterbo. "A Stability-based Validation Procedure for Differentially Private Machine Learning." In: *NIPS*. 2013, pp. 2652–2660. URL: http://papers.nips.cc/paper/5014-a-stability-based-validation-procedure-for-differentially-private-machine-learning (visited on 10/20/2020).

[Che+20] Hao Chen, Ilaria Chillotti, Yihe Dong, Oxana Poburinnaya, Ilya P. Razenshteyn, and M. Sadegh Riazi. "SANNS: Scaling Up Secure Approximate k-Nearest Neighbors Search." In: *USENIX Security Symposium*. USENIX Association, 2020, pp. 2111–2128. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hao (visited on 10/25/2020).

[CHLR18] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. "Labeled PSI from Fully Homomorphic Encryption with Malicious Security." In: *CCS*. ACM, 2018, pp. 1223–1237. DOI: 10.1145/3243734.3243836.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. "Fast Private Set Intersection from Homomorphic Encryption." In: *CCS*. ACM, 2017, pp. 1243–1255. DOI: 10.1145/3133956.3134061.

[Che+19] Albert Cheu, Adam D. Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. "Distributed Differential Privacy via Shuffling." In: *EUROCRYPT (1)*. Springer, 2019, pp. 375–403. DOI: 10.1007/978-3-030-17653-2_13.

[CGN98] Benny Chor, Niv Gilboa, and Moni Naor. "Private Information Retrieval by Keywords." In: *IACR Cryptol. ePrint Arch.* 1998 (1998), p. 3. URL: https://eprint.iacr.org/1998/003 (visited on 10/20/2020).

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. "Private Information Retrieval." In: *J. ACM* 45.6 (1998), pp. 965–981. DOI: 10.1145/293347.293350.

[CO15] Tung Chou and Claudio Orlandi. "The Simplest Protocol for Oblivious Transfer." In: *LATINCRYPT*. Springer, 2015, pp. 40–58. DOI: 10.1007/978-3-319-22174-8_3.

[CO18] Michele Ciampi and Claudio Orlandi. "Combining Private Set-Intersection with Secure Two-Party Computation." In: *SCN*. Springer, 2018, pp. 464–482. DOI: 10.1007/978-3-319-98113-0_25.

[CDNN15] Martine De Cock, Rafael Dowsley, Anderson C. A. Nascimento, and Stacey C. Newman. "Fast, Privacy Preserving Linear Regression over Distributed Datasets based on Pre-Distributed Data." In: *AISec@CCS*. ACM, 2015, pp. 3–14. DOI: 10.1145/2808769.2808774.

[Cor14] Paulo Cortez. *Student Performance Data Set*. 2014. URL: https://archive.ics.uci.edu/ml/datasets/Student+Performance (visited on 10/20/2020).

[Cor+09a] Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. "Modeling wine preferences by data mining from physicochemical properties." In: *Decis. Support Syst.* 47.4 (2009), pp. 547–553. DOI: 10.1016/j.dss.2009.05.016.

[Cor+09b] Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. *Wine Quality Data Set*. 2009. URL: https://archive.ics.uci.edu/ml/datasets/Wine+Quality (visited on 10/20/2020).

[CS08] Paulo Cortez and Alice Maria Gonçalves Silva. "Using data mining to predict secondary school student performance." In: *Future Business Technology Conference*. EUROSIS, 2008, pp. 5–12. URL: http://www3.dsi.uminho.pt/pcortez/student.pdf (visited on 10/20/2020).

[Cra+19] Mark Craddock, David W. Archer, Dan Bogdanov, Adrià Gascón, Borja Balle, Kim Laine, Andrew Trask, Mariana Raykova, Matjaz Jug, Robert McLellan, Ronald Jansen, Olga Ohrimenko, Simon Wardley, Kristin Lauter, Nigel Smart, Alekh Sharan, Ira Saxena, Rebecca N. Wright, Eddie Garcia, and Andy Wall. *UN Handbook on Privacy-Preserving Computation Techniques*. BigData UN Global Working Group, 2019. URL: https://publications.officialstatistics.org/handbooks/privacy-preserving-techniques-handbook/UN%20Handbook%20for%20Privacy-Preserving%20Techniques.pdf (visited on 10/26/2020).

[DM98] Leonardo Dagum and Ramesh Menon. "OpenMP: An industry-standard API for shared-memory programming." In: *Computational Science and Engineering* 5.1 (1998), pp. 46–55. DOI: 10.1109/99.660313.

[Dam+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. "Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits." In: *ESORICS*. Springer, 2013, pp. 1–18. DOI: 10.1007/978-3-642-40203-6_1.

[DP12] Ivan Damgård and Sunoo Park. "Is Public-Key Encryption Based on LPN Practical?" In: *IACR Cryptol. ePrint Arch.* 2012 (2012), p. 699. URL: http://eprint.iacr.org/2012/699 (visited on 10/20/2020).

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. "Multiparty Computation from Somewhat Homomorphic Encryption." In: *CRYPTO*. Springer, 2012, pp. 643–662. DOI: 10.1007/978-3-642-32009-5_38.

[Dat+17] Kushal Datta, Karthik Gururaj, Mishali Naik, Paolo Narvaez, and Ming Rutar. *GenomicsDB: Storing Genome Data as Sparse Columnar Arrays*. Tech. rep. Intel Health and Life Sciences, 2017. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/genomics-storing-genome-data-paper.pdf (visited on 10/20/2020).

[Dem+15] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. "Automated Synthesis of Optimized Circuits for Secure Computation." In: *CCS*. ACM, 2015, pp. 1504–1517. DOI: 10.1145/2810103.2813678.

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. "PIR-PSI: Scaling Private Contact Discovery." In: *Proc. Priv. Enhancing Technol.* 2018.4 (2018), pp. 159–178. DOI: 10.1515/popets-2018-0037.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. "ABY – A Framework for Efficient Mixed-Protocol Secure Two-Party Computation." In: *NDSS*. The Internet Society, 2015. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/08_2_1.pdf (visited on 10/20/2020).

[Doe] Jack Doerner. *Absentminded Crypto Kit*. URL: https://bitbucket.org/jackdoerner/absentminded-crypto-kit (visited on 10/20/2020).

[DS17] Jack Doerner and Abhi Shelat. "Scaling ORAM for Secure Computation." In: *CCS*. ACM, 2017, pp. 523–535. DOI: 10.1145/3133956.3133967.

[Döt+17] Nico Döttling, Satrajit Ghosh, Jesper Buus Nielsen, Tobias Nilges, and Roberto Trifiletti. "TinyOLE: Efficient Actively Secure Two-Party Computation from Oblivious Linear Function Evaluation." In: *CCS*. ACM, 2017, pp. 2263–2276. DOI: 10.1145/3133956.3134024.

[DKM12] Nico Döttling, Daniel Kraschewski, and Jörn Müller-Quade. "David & Goliath Oblivious Affine Function Evaluation - Asymptotically Optimal Building Blocks for Universally Composable Two-Party Computation from a Single Untrusted Stateful Tamper-Proof Hardware Token." In: *IACR Cryptol. ePrint Arch.* 2012 (2012), p. 135. URL: http://eprint.iacr.org/2012/135 (visited on 10/20/2020).

[DA01a] Wenliang Du and Mikhail J. Atallah. "Privacy-Preserving Cooperative Scientific Computations." In: *CSFW*. IEEE Computer Society, 2001, pp. 273–294. DOI: 10.1109/CSFW.2001.930152.

[DA01b] Wenliang Du and Mikhail J. Atallah. "Protocols for Secure Remote Database Access with Approximate Matching." In: *E-Commerce Security and Privacy*. Vol. 2. Advances in Information Security. Springer, 2001, pp. 87–111. DOI: 10.1007/978-1-4615-1467-1_6.

[DHC04] Wenliang Du, Yunghsiang S. Han, and Shigang Chen. "Privacy-Preserving Multivariate Statistical Analysis: Linear Regression and Classification." In: *SDM*. SIAM, 2004, pp. 222–233. DOI: 10.1137/1.9781611972740.21.

[DG17] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml (visited on 10/20/2020).

[DHP02] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. "An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum." In: *ACM Trans. Math. Softw.* 28.2 (2002), pp. 239–267. DOI: 10.1145/567806.567810.

[Dwo+06] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. "Our Data, Ourselves: Privacy Via Distributed Noise Generation." In: *EUROCRYPT*. Springer, 2006, pp. 486–503. DOI: 10.1007/11761679_29.

[DMNS16] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. "Calibrating Noise to Sensitivity in Private Data Analysis." In: *J. Priv. Confidentiality* 7.3 (2016), pp. 17–51. DOI: 10.29012/jpc.v7i3.405.

[DN04] Cynthia Dwork and Kobbi Nissim. "Privacy-Preserving Datamining on Vertically Partitioned Databases." In: *CRYPTO*. Springer, 2004, pp. 528–544. DOI: 10.1007/978-3-540-28628-8_32.

[DR14] Cynthia Dwork and Aaron Roth. "The Algorithmic Foundations of Differential Privacy." In: *Found. Trends Theor. Comput. Sci.* 9.3-4 (2014), pp. 211–407. DOI: 10.1561/0400000042.

[Efr17] Alexei Efros. *How to stop worrying and learn to love Nearest Neighbors*. NIPS workshop on Nearest Neighbors for Modern Applications with Massive Data. 2017. URL: https://nn2017.mit.edu/wp-content/uploads/sites/5/2017/12/Efros-NIPS-NN-17.pdf (visited on 10/20/2020).

[EL04] Miloš D Ercegovac and Tomás Lang. *Digital arithmetic*. Elsevier, 2004. ISBN: 978-1558607989.

[FV12] Junfeng Fan and Frederik Vercauteren. "Somewhat Practical Fully Homomorphic Encryption." In: *IACR Cryptol. ePrint Arch.* 2012 (2012), p. 144. URL: http://eprint.iacr.org/2012/144 (visited on 10/20/2020).

[FG13] Hadi Fanaee-T and João Gama. *Bike Sharing Dataset Data Set*. 2013. URL: https://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset (visited on 10/20/2020).

[FG14] Hadi Fanaee-T and João Gama. "Event labeling combining ensemble detectors and background knowledge." In: *Prog. Artif. Intell.* 2.2-3 (2014), pp. 113–127. DOI: 10.1007/s13748-013-0040-3.

[FR18] Maryam Fanaeepour and Benjamin I. P. Rubinstein. "Histogramming Privately Ever After: Differentially-Private Data-Dependent Error Bound Optimisation." In: *ICDE*. IEEE Computer Society, 2018, pp. 1204–1207. DOI: 10.1109/ICDE.2018.00111.

[FH15] Jordi Fonollosa and Ramon Huerta. *Gas sensor array under dynamic gas mixtures Data Set*. 2015. URL: https://archive.ics.uci.edu/ml/datasets/Gas+sensor+array+under+dynamic+gas+mixtures (visited on 10/20/2020).

[FSHM15] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. "Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring." In: *Sensors and Actuators B: Chemical* 215 (2015), pp. 618–629. DOI: 10.1016/j.snb.2015.03.028.

[FHNP16] Michael J. Freedman, Carmit Hazay, Kobbi Nissim, and Benny Pinkas. "Efficient Set Intersection with Simulation-Based Security." In: *J. Cryptol.* 29.1 (2016), pp. 115–155. DOI: 10.1007/s00145-014-9190-0.

[FIPR05] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. "Keyword Search and Oblivious Pseudorandom Functions." In: *TCC*. Springer, 2005, pp. 303–324. DOI: 10.1007/978-3-540-30576-7_17.

[Gen09] Craig Gentry. "Fully homomorphic encryption using ideal lattices." In: *STOC*. ACM, 2009, pp. 169–178. DOI: 10.1145/1536414.1536440.

[GIKM00] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. "Protecting Data Privacy in Private Information Retrieval Schemes." In: *J. Comput. Syst. Sci.* 60.3 (2000), pp. 592–629. DOI: 10.1006/jcss.1999.1689.

[GN19] Satrajit Ghosh and Tobias Nilges. "An Algebraic Approach to Maliciously Secure Private Set Intersection." In: *EUROCRYPT (3)*. Springer, 2019, pp. 154–185. DOI: 10.1007/978-3-030-17659-4_6.

[Gil99] Niv Gilboa. "Two Party RSA Key Generation." In: *CRYPTO*. Springer, 1999, pp. 116–129. DOI: 10.1007/3-540-48405-1_8.

[GI14] Niv Gilboa and Yuval Ishai. "Distributed Point Functions and Their Applications." In: *EUROCRYPT*. Springer, 2014, pp. 640–658. DOI: 10.1007/978-3-642-55220-5_35.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004. DOI: 10.1017/CBO9780511721656.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. "How to construct random functions." In: *J. ACM* 33.4 (1986), pp. 792–807. DOI: 10.1145/6490.6503.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority." In: *STOC*. ACM, 1987, pp. 218–229. DOI: 10.1145/28395.28420.

[GO96] Oded Goldreich and Rafail Ostrovsky. "Software Protection and Simulation on Oblivious RAMs." In: *J. ACM* 43.3 (1996), pp. 431–473. DOI: 10.1145/233551.233553.

[GLN12] Thore Graepel, Kristin E. Lauter, and Michael Naehrig. "ML Confidential: Machine Learning on Encrypted Data." In: *ICISC*. Springer, 2012, pp. 1–21. DOI: 10.1007/978-3-642-37682-5_1.

[Gra+11a] Franz Graf, Hans-Peter Kriegel, Matthias Schubert, Sebastian Pölsterl, and Alexander Cavallaro. "2D Image Registration in CT Images Using Radial Image Descriptors." In: *MICCAI (2)*. Springer, 2011, pp. 607–614. DOI: 10.1007/978-3-642-23629-7_74.

[Gra+11b] Franz Graf, Hans-Peter Kriegel, Matthias Schubert, Sebastian Pölsterl, and Alexander Cavallaro. *Relative location of CT slices on axial axis Data Set*. 2011. URL: https://archive.ics.uci.edu/ml/datasets/Relative+location+of+CT+slices+on+axial+axis (visited on 10/20/2020).

[GRR19] Adam Groce, Peter Rindal, and Mike Rosulek. "Cheaper Private Set Intersection via Differentially Private Leakage." In: *Proc. Priv. Enhancing Technol.* 2019.3 (2019), pp. 6–25. DOI: 10.2478/popets-2019-0034.

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org (visited on 10/20/2020).

[GJ+] Gaël Guennebaud, Benoît Jacob, et al. *Eigen: Sparse matrix manipulations*. URL: https://eigen.tuxfamily.org/dox/group__TutorialSparse.html (visited on 10/20/2020).

[HFN11] Rob Hall, Stephen E Fienberg, and Yuval Nardi. "Secure multiple linear regression based on homomorphic encryption." In: *Journal of Official Statistics* 27.4 (2011), p. 669. URL: https://www.scb.se/contentassets/ff271eeeca694f47ae99b942de61df83/secure-multiple-linear-regression-based-on-homomorphic-encryption.pdf (visited on 10/20/2020).

[HM16] Ruining He and Julian J. McAuley. "Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering." In: *WWW*. ACM, 2016, pp. 507–517. DOI: 10.1145/2872427.2883037.

[HMFS17] Xi He, Ashwin Machanavajjhala, Cheryl J. Flynn, and Divesh Srivastava. "Composing Differential Privacy and Secure Computation: A Case Study on Scaling Private Record Linkage." In: *CCS*. ACM, 2017, pp. 1389–1406. DOI: 10.1145/3133956.3134030.

[HK20] David Heath and Vladimir Kolesnikov. "Stacked Garbling – Garbled Circuit Proportional to Longest Execution Path." In: *CRYPTO (2)*. Springer, 2020, pp. 763–792. DOI: 10.1007/978-3-030-56880-1_27.

[Hey+12] Stefan Heyse, Eike Kiltz, Vadim Lyubashevsky, Christof Paar, and Krzysztof Pietrzak. "Lapin: An Efficient Authentication Protocol Based on Ring-LPN." In: *FSE*. Springer, 2012, pp. 346–365. DOI: 10.1007/978-3-642-34047-5_20.

[HHS17] Elad Hoffer, Itay Hubara, and Daniel Soudry. "Train longer, generalize better: closing the generalization gap in large batch training of neural networks." In: *NIPS*. 2017, pp. 1731–1741. URL: http://papers.

nips.cc/paper/6770-train-longer-generalize-better-closing-the-generalization-gap-in-large-batch-training-of-neural-networks (visited on 10/20/2020).

[HSV16] Sebastiaan de Hoogh, Berry Schoenmakers, and Meilof Veeningen. "Certificate Validation in Secure Computation and Its Use in Verifiable Linear Programming." In: *AFRICACRYPT*. Springer, 2016, pp. 265–284. DOI: 10.1007/978-3-319-31517-1_14.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. "Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?" In: *NDSS*. The Internet Society, 2012. URL: https://www.ndss-symposium.org/ndss2012/ndss-2012-programme/private-set-intersection-are-garbled-circuits-better-custom-protocols/.

[Hua+11] Yan Huang, Chih-Hao Shen, David Evans, Jonathan Katz, and Abhi Shelat. "Efficient Secure Computation with Garbled Circuits." In: *ICISS*. Springer, 2011, pp. 28–48. DOI: 10.1007/978-3-642-25560-1_2.

[Ion+20] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Mariana Raykova, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. "On Deploying Secure Computing: Private Intersection-Sum-with-Cardinality." In: *EuroS&P*. IEEE, 2020.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. "Extending Oblivious Transfers Efficiently." In: *CRYPTO*. Springer, 2003, pp. 145–161. DOI: 10.1007/978-3-540-45146-4_9.

[IKOS04] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. "Batch codes and their applications." In: *STOC*. ACM, 2004, pp. 262–271. DOI: 10.1145/1007352.1007396.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. "Founding Cryptography on Oblivious Transfer - Efficiently." In: *CRYPTO*. Springer, 2008, pp. 572–591. DOI: 10.1007/978-3-540-85174-5_32.

[JMCS08] Wei Jiang, Mummoorthy Murugesan, Chris Clifton, and Luo Si. "Similar Document Detection with Limited Information Disclosure." In: *ICDE*. IEEE Computer Society, 2008, pp. 735–743. DOI: 10.1109/ICDE.2008.4497482.

[JOP+] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *Sparse matrices (scipy.sparse) – SciPy v1.1.0 Reference Guide*. URL: https://docs.scipy.org/doc/scipy/reference/sparse.html (visited on 10/20/2020).

[JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "GAZELLE: A Low Latency Framework for Secure Neural Network Inference." In: *USENIX Security Symposium*. USENIX Association, 2018, pp. 1651–1669. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar (visited on 10/20/2020).

[Kai+19] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Keith Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaïd Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konecný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Mariana Raykova, Hang Qi, Daniel Ramage, Ramesh Raskar, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. "Advances and Open Problems in Federated Learning." In: *CoRR* abs/1912.04977 (2019). URL: http://arxiv.org/abs/1912.04977 (visited on 10/26/2020).

[KOV17] Peter Kairouz, Sewoong Oh, and Pramod Viswanath. "The Composition Theorem for Differential Privacy." In: *IEEE Trans. Inf. Theory* 63.6 (2017), pp. 4037–4049. DOI: 10.1109/TIT.2017.2685505.

[KO62] Anatolii Alekseevich Karatsuba and Yu P Ofman. "Multiplication of many-digital numbers by automatic computers." In: *Doklady Akademii Nauk*. Vol. 145. 2. Russian Academy of Sciences. 1962, pp. 293–294.

[KLSR04] Alan F. Karr, Xiaodong Lin, Ashish P. Sanil, and Jerome P. Reiter. "Regression on Distributed Databases via Secure Multi-Party Computation." In: *DG.O*. ACM International Conference Proceeding Series. Digital Government Research Center, 2004. URL: http://dl.acm.org/citation.cfm?id=1124299 (visited on 10/20/2020).

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. CRC Press, 2014. ISBN: 9781466570269.

[Kel20] Marcel Keller. "MP-SPDZ: A Versatile Framework for Multi-Party Computation." In: *CCS*. ACM, 2020, pp. 1575–1590. DOI: 10.1145/3372297.3417872.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. "Actively Secure OT Extension with Optimal Overhead." In: *CRYPTO (1)*. Springer, 2015, pp. 724–741. DOI: 10.1007/978-3-662-47989-6_35.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. "MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer." In: *CCS*. ACM, 2016, pp. 830–842. DOI: 10.1145/2976749.2978357.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. "Overdrive: Making SPDZ Great Again." In: *EUROCRYPT (3)*. Springer, 2018, pp. 158–189. DOI: 10.1007/978-3-319-78372-7_6.

[KMW09] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. "More Robust Hashing: Cuckoo Hashing with a Stash." In: *SIAM J. Comput.* 39.4 (2009), pp. 1543–1561. DOI: 10.1137/080728743.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley, 1997. ISBN: 978-0201896848.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. "Efficient Batched Oblivious PRF with Applications to Private Set Intersection." In: *CCS*. ACM, 2016, pp. 818–829. DOI: 10.1145/2976749.2978381.

[KS08] Vladimir Kolesnikov and Thomas Schneider. "Improved Garbled Circuit: Free XOR Gates and Applications." In: *ICALP (2)*. Springer, 2008, pp. 486–498. DOI: 10.1007/978-3-540-70583-3_40.

[Lau15] Peeter Laud. "Parallel Oblivious Array Access for Secure Multiparty Computation and Privacy-Preserving Minimum Spanning Trees." In: *Proc. Priv. Enhancing Technol.* 2015.2 (2015), pp. 188–205. DOI: 10.1515/popets-2015-0011.

[LP16] Peeter Laud and Martin Pettai. "Secure Multiparty Sorting Protocols with Covert Privacy." In: *NordSec*. 2016, pp. 216–231. DOI: 10.1007/978-3-319-47560-8_14.

[Laz+18] Ibrahim Lazrig, Toan C. Ong, Indrajit Ray, Indrakshi Ray, Xiaoqian Jiang, and Jaideep Vaidya. "Privacy Preserving Probabilistic Record Linkage Without Trusted Third Party." In: *PST*. IEEE Computer Society, 2018, pp. 1–10. DOI: 10.1109/PST.2018.8514192.

[Lep+20] Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Karn Seth, and Ni Trieu. "Private Join and Compute from PIR with Default." In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 1011. URL: https://eprint.iacr.org/2020/1011.

[LYRL04] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. "RCV1: A New Benchmark Collection for Text Categorization Research." In: *J. Mach. Learn. Res.* 5 (2004), pp. 361–397. URL: http://jmlr.org/papers/volume5/lewis04a/lewis04a.pdf (visited on 10/20/2020).

[LSP15] Frank Li, Richard Shin, and Vern Paxson. "Exploring Privacy Preservation in Outsourced K-Nearest Neighbors with Multiple Data Owners." In: *CCSW*. ACM, 2015, pp. 53–64. DOI: 10.1145/2808425.2808430.

[Lin17] Yehuda Lindell. "How to Simulate It - A Tutorial on the Simulation Proof Technique." In: *Tutorials on the Foundations of Cryptography*. Springer International Publishing, 2017, pp. 277–346. DOI: 10.1007/978-3-319-57048-8_6.

[LOP11] Yehuda Lindell, Eli Oxman, and Benny Pinkas. "The IPS Compiler: Optimizations, Variants and Concrete Efficiency." In: *CRYPTO*. Springer, 2011, pp. 259–276. DOI: 10.1007/978-3-642-22792-9_15.

[LP02] Yehuda Lindell and Benny Pinkas. "Privacy Preserving Data Mining." In: *J. Cryptology* 15.3 (2002), pp. 177–206. DOI: 10.1007/s00145-001-0019-2.

[LP09] Yehuda Lindell and Benny Pinkas. "A Proof of Security of Yao's Protocol for Two-Party Computation." In: *J. Cryptology* 22.2 (2009), pp. 161–188. DOI: 10.1007/s00145-008-9036-8.

[LJLA17] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. "Oblivious Neural Network Predictions via MiniONN Transformations." In: *CCS*. ACM, 2017, pp. 619–631. DOI: 10.1145/3133956.3134056.

[Maa+11] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. "Learning Word Vectors for Sentiment Analysis." In: *ACL*. The Association for Computer Linguistics, 2011, pp. 142–150. URL: https://www.aclweb.org/anthology/P11-1015/ (visited on 10/20/2020).

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. "Fairplay – A Secure Two-Party Computation System." In: *USENIX Security Symposium*. USENIX, 2004, pp. 287–302. URL: https://www.usenix.org/legacy/events/sec04/tech/malkhi/malkhi.pdf (visited on 10/20/2020).

[MRS08] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. "Scoring, term weighting and the vector space model." In: *Introduction to information retrieval*. 2008, pp. 100–122. ISBN: 978-0521865715.

[MG18] Sahar Mazloom and S. Dov Gordon. "Secure Computation with Differentially Private Access Patterns." In: *CCS*. ACM, 2018, pp. 490–507. DOI: 10.1145/3243734.3243851.

[McA] Julian McAuley. *Amazon Product Data*. URL: http://jmcauley.ucsd.edu/data/amazon/ (visited on 10/20/2020).

[MT07] Frank McSherry and Kunal Talwar. "Mechanism Design via Differential Privacy." In: *FOCS*. IEEE Computer Society, 2007, pp. 94–103. DOI: 10.1109/FOCS.2007.41.

[Meu06] G. Meurant. *The Lanczos and Conjugate Gradient Algorithms: From Theory to Finite Precision Computations*. Vol. 19. Software, Environments and Tools. SIAM, 2006. ISBN: 978-0898716160.

[MPRV09] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil P. Vadhan. "Computational Differential Privacy." In: *CRYPTO*. Springer, 2009, pp. 126–142. DOI: 10.1007/978-3-642-03356-8_8.

[MB72] R. Moenck and Allan Borodin. "Fast Modular Transforms via Division." In: *SWAT (FOCS)*. IEEE Computer Society, 1972, pp. 90–96. DOI: 10.1109/SWAT.1972.5.

[MF06] Payman Mohassel and Matthew K. Franklin. "Efficiency Tradeoffs for Malicious Two-Party Computation." In: *Public Key Cryptography*. Springer, 2006, pp. 458–473. DOI: 10.1007/11745853_30.

[MZ17] Payman Mohassel and Yupeng Zhang. "SecureML: A System for Scalable Privacy-Preserving Machine Learning." In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 19–38. DOI: 10.1109/SP.2017.12.

[Mur12] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive computation and machine learning. MIT Press, 2012. ISBN: 978-0262018029.

[Mur+10] Mummoorthy Murugesan, Wei Jiang, Chris Clifton, Luo Si, and Jaideep Vaidya. "Efficient privacy-preserving similar document detection." In: *VLDB J.* 19.4 (2010), pp. 457–475. DOI: 10.1007/s00778-009-0175-9.

[NP99] Moni Naor and Benny Pinkas. "Oblivious Transfer and Polynomial Evaluation." In: *STOC*. ACM, 1999, pp. 245–254. DOI: 10.1145/301250.301312.

[NP01] Moni Naor and Benny Pinkas. "Efficient oblivious transfer protocols." In: *SODA*. ACM/SIAM, 2001, pp. 448–457. URL: http://dl.acm.org/citation.cfm?id=365411.365502 (visited on 10/20/2020).

[NP06] Moni Naor and Benny Pinkas. "Oblivious Polynomial Evaluation." In: *SIAM J. Comput.* 35.5 (2006), pp. 1254–1281. DOI: 10.1137/S0097539704383633.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. "Privacy preserving auctions and mechanism design." In: *EC*. ACM, 1999, pp. 129–139. DOI: 10.1145/336992.337028.

[Nay+15] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. "GraphSC: Parallel Secure Computation Made Easy." In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015, pp. 377–394. DOI: 10.1109/SP.2015.30.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. "A New Approach to Practical Active-Secure Two-Party Computation." In: *CRYPTO*. Springer, 2012, pp. 681–700. DOI: 10.1007/978-3-642-32009-5_40.

[Nik+13a] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. "Privacy-preserving matrix factorization." In: *CCS*. ACM, 2013, pp. 801–812. DOI: 10.1145/2508859.2516751.

[Nik+13b] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. "Privacy-Preserving Ridge Regression on Hundreds of Millions of Records." In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013, pp. 334–348. DOI: 10.1109/SP.2013.30.

[NW99] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 1999. ISBN: 978-0387987934.

[PR04] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo hashing." In: *J. Algorithms* 51.2 (2004), pp. 122–144. DOI: 10.1016/j.jalgor.2003.12.002.

[PSW09] Maura B. Paterson, Douglas R. Stinson, and Ruizhong Wei. "Combinatorial batch codes." In: *Adv. Math. Commun.* 3.1 (2009), pp. 13–27. DOI: 10.3934/amc.2009.3.13.

[Ped+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. "Scikit-learn: Machine Learning in

Python." In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830. URL: http://dl.acm.org/citation.cfm?id=2078195 (visited on 10/20/2020).

[PRTY20] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. "PSI from PaXoS: Fast, Malicious Private Set Intersection." In: *EUROCRYPT (2)*. Springer, 2020, pp. 739–767. DOI: 10.1007/978-3-030-45724-2_25.

[PSTY19] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. "Efficient Circuit-Based PSI with Linear Communication." In: *EUROCRYPT (3)*. Springer, 2019, pp. 122–153. DOI: 10.1007/978-3-030-17659-4_5.

[PSZ18] Benny Pinkas, Thomas Schneider, and Michael Zohner. "Scalable Private Set Intersection Based on OT Extension." In: *ACM Trans. Priv. Secur.* 21.2 (2018), 7:1–7:35. DOI: 10.1145/3154794.

[Pow98] David M. W. Powers. "Applications and Explanations of Zipf's Law." In: *CoNLL*. ACL, 1998, pp. 151–160. URL: https://www.aclweb.org/anthology/W98-1218/ (visited on 10/20/2020).

[PS15] Pille Pullonen and Sander Siim. "Combining Secret Sharing and Garbled Circuits for Efficient Private IEEE 754 Floating-Point Computations." In: *Financial Cryptography Workshops*. Springer, 2015, pp. 172–183. DOI: 10.1007/978-3-662-48051-9_13.

[Qui93a] J. Ross Quinlan. *Auto MPG Data Set*. 1993. URL: https://archive.ics.uci.edu/ml/datasets/Auto+MPG (visited on 10/20/2020).

[Qui93b] J. Ross Quinlan. "Combining Instance-Based and Model-Based Learning." In: *ICML*. Morgan Kaufmann, 1993, pp. 236–243. DOI: 10.1016/b978-1-55860-307-3.50037-x.

[Rab81] Michael O. Rabin. *How to exchange secrets with oblivious transfer*. Tech. rep. TR-81. Aiken Computation Lab, Harvard University, 1981. URL: https://eprint.iacr.org/2005/187 (visited on 10/20/2020).

[Red09] Michael Redmond. *Communities and Crime Data Set*. 2009. URL: https://archive.ics.uci.edu/ml/datasets/Communities+and+Crime (visited on 10/20/2020).

[RB02] Michael Redmond and Alok Baveja. "A data-driven software tool for enabling cooperative information sharing among police departments." In: *Eur. J. Oper. Res.* 141.3 (2002), pp. 660–678. DOI: 10.1016/S0377-2217(01)00264-8.

[RL08] Jason Rennie and Ken Lang. *The 20 Newsgroups data set*. 2008. URL: http://qwone.com/~jason/20Newsgroups/ (visited on 10/20/2020).

[Ria+16] M. Sadegh Riazi, Beidi Chen, Anshumali Shrivastava, Dan S. Wallach, and Farinaz Koushanfar. "Sub-linear Privacy-preserving Search with Untrusted Server and Semi-honest Parties." In: *CoRR* abs/1612.01835 (2016). URL: http://arxiv.org/abs/1612.01835 (visited on 10/20/2020).

[RWLX16] Hong Rong, Huimei Wang, Jian Liu, and Ming Xian. "Privacy-Preserving k-Nearest Neighbor Computation in Multiple Cloud Environments." In: *IEEE Access* 4 (2016), pp. 9589–9603. DOI: 10.1109/ACCESS.2016.2633544.

[AWCK17] Mohammad Al-Rubaie, Pei Yuan Wu, J. Morris Chang, and Sun-Yuan Kung. "Privacy-preserving PCA on horizontally-partitioned data." In: *DSC*. IEEE, 2017, pp. 280–287. DOI: 10.1109/DESEC.2017.8073817.

[SKLR04] Ashish P. Sanil, Alan F. Karr, Xiaodong Lin, and Jerome P. Reiter. "Privacy preserving regression modelling via distributed computation." In: *KDD*. ACM, 2004, pp. 677–682. DOI: 10.1145/1014052.1014139.

[SL03] George A. F. Seber and Alan J. Lee. *Linear regression analysis*. 2nd ed. John Wiley & Sons, 2003. ISBN: 978-0471415404.

[Sho+01] Victor Shoup et al. *NTL: A library for doing number theory*. 2001. URL: https://www.shoup.net/ntl (visited on 10/20/2020).

[The+15] The 1000 Genomes Project Consortium et al. "A global reference for human genetic variation." In: *Nature* 526 (2015), pp. 68–74. DOI: 10.1038/nature15393.

[The] The Scikit-learn authors. *Scikit-learn language identification dataset*. URL: https://github.com/scikit-learn/scikit-learn/tree/master/doc/tutorial/text_analytics/data/languages (visited on 10/20/2020).

[Wak68] Abraham Waksman. "A Permutation Network." In: *J. ACM* 15.1 (1968), pp. 159–163. DOI: 10.1145/321439.321449.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. *EMP-toolkit: Efficient MultiParty computation toolkit*. 2016. URL: https://github.com/emp-toolkit (visited on 10/20/2020).

[Wei61] Martin H Weik. *A Third Survey of Domestic Electronic Digital Computing Systems*. Tech. rep. DTIC Document, Ballistic Research Laboratories, Aberdeen Proving Ground, Maryland, 1961. URL: https://archive.org/details/DTIC_AD0253212 (visited on 10/20/2020).

[WYKW20] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. "Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits." In: *IACR Cryptol. ePrint Arch.* 2020 (2020), p. 925. URL: https://eprint.iacr.org/2020/925 (visited on 01/10/2021).

[Wil88] James Hardy Wilkinson. *The algebraic eigenvalue problem*. Clarendon Press Oxford, 1988. ISBN: 978-0198534181.

[Win84] Robert S. Winternitz. "A Secure One-Way Hash Function Built from DES." In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1984, pp. 88–90. DOI: 10.1109/SP.1984.10027.

[Yan] Avishay Yanai. *FastPolynomial*. URL: https://github.com/AvishayYanay/FastPolynomial (visited on 10/20/2020).

[Yan+20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. "Ferret: Fast Extension for Correlated OT with Small Communication." In: *CCS*. ACM, 2020, pp. 1607–1626. DOI: 10.1145/3372297.3417276.

[Yao86] Andrew Chi-Chih Yao. "How to Generate and Exchange Secrets (Extended Abstract)." In: *FOCS*. IEEE Computer Society, 1986, pp. 162–167. DOI: 10.1109/SFCS.1986.25.

[YVJ06] Hwanjo Yu, Jaideep Vaidya, and Xiaoqian Jiang. "Privacy-Preserving SVM Classification on Vertically Partitioned Data." In: *PAKDD*. Springer, 2006, pp. 647–656. DOI: 10.1007/11731139_74.

[ZE15] Samee Zahur and David Evans. "Obliv-C: A Language for Extensible Data-Oblivious Computation." In: *IACR Cryptol. ePrint Arch.* 2015 (2015), p. 1153. URL: https://eprint.iacr.org/2015/1153 (visited on 10/20/2020).

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. "Two Halves Make a Whole – Reducing Data Transfer in Garbled Circuits Using Half Gates." In: *EUROCRYPT (2)*. Springer, 2015, pp. 220–250. DOI: 10.1007/978-3-662-46803-6_8.

[Zah+16] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. "Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation." In: *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2016, pp. 218–234.

[Zho09] Jingren Zhou. "Sort-Merge Join." In: *Encyclopedia of Database Systems*. Springer US, 2009, pp. 2673–2674. DOI: 10.1007/978-0-387-39940-9_867.

## AUTHOR'S PUBLICATIONS

[Ali+21] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. "Communication–Computation Trade-offs in PIR." In: *USENIX Security Symposium*. USENIX Association, 2021, pp. 1811–1828. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/ali.

[Gas+17] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. "Privacy-Preserving Distributed Linear Regression on High-Dimensional Data." In: *Proc. Priv. Enhancing Technol.* 2017.4 (2017), pp. 345–364. DOI: 10.1515/popets-2017-0053.

[RS21] Peter Rindal and Phillipp Schoppmann. "VOLE-PSI: Fast OPRF and Circuit-PSI from Vector-OLE." In: *EUROCRYPT (2)*. Vol. 12697. Lecture Notes in Computer Science. Springer, 2021, pp. 901–930. DOI: 10.1007/978-3-030-77886-6_31.

[SGRP19] Phillipp Schoppmann, Adrià Gascón, Mariana Raykova, and Benny Pinkas. "Make Some ROOM for the Zeros: Data Sparsity in Secure Distributed Machine Learning." In: *CCS*. ACM, 2019, pp. 1335–1350. DOI: 10.1145/3319535.3339816.

[SGRR19] Phillipp Schoppmann, Adrià Gascón, Leonie Reichert, and Mariana Raykova. "Distributed Vector-OLE: Improved Constructions and Implementation." In: *CCS*. ACM, 2019, pp. 1055–1072. DOI: 10.1145/3319535.3363228.

[SVGB20] Phillipp Schoppmann, Lennart Vogelsang, Adrià Gascón, and Borja Balle. "Secure and Scalable Document Similarity on Distributed Databases: Differential Privacy to the Rescue." In: *Proc. Priv. Enhancing Technol.* 2020.2 (2020), pp. 209–229. DOI: 10.2478/popets-2020-0024.

[Sta+20] Sebastian Stammler, Tobias Kussel, Phillipp Schoppmann, Florian Stampe, Galina Tremper, Stefan Katzenbeisser, Kay Hamacher, and Martin Lablans. "Mainzelliste SecureEpiLinker (MainSEL): Privacy-Preserving Record Linkage using Secure Multi-Party Computation." In: *Bioinformatics* (Sept. 2020). btaa764. DOI: 10.1093/bioinformatics/btaa764.

[Vog+20] Lennart Vogelsang, Moritz Lehne, Phillipp Schoppmann, Fabian Prasser, Sylvia Thun, Björn Scheuermann, and Josef Schepers. "A Secure Multi-Party Computation Protocol for Time-To-Event Analyses." In: *MIE*. Vol. 270. Studies in Health Technology and Informatics. IOS Press, 2020, pp. 8–12. DOI: 10.3233/SHTI200112.

# DECLARATION

I declare that I have completed the thesis independently using only the aids and tools specified. I have not applied for a doctor's degree in the doctoral subject elsewhere and do not hold a corresponding doctor's degree. I have taken due note of the Faculty of Mathematics and Natural Sciences PhD Regulations, published in the Official Gazette of Humboldt-Universität zu Berlin no. 42 on July 11 2018.

Phillipp Schoppmann