Sharing without Showing: Building Secure Collaborative Systems

Wenting Zheng

August 13, 2020

Sharing without Showing: Building Secure Collaborative Systems

by

Wenting Zheng

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Raluca Ada Popa, Co-chair
Professor Ion Stoica, Co-chair
Professor Deirdre Mulligan

Summer 2020

Sharing without Showing: Building Secure Collaborative Systems

Abstract

Sharing without Showing: Building Secure Collaborative Systems

by

Wenting Zheng

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Raluca Ada Popa, Co-chair

Professor Ion Stoica, Co-chair

The recent revolution in advanced data analytics gave rise to a growing demand among organizations for high quality data. However, in many domains such as finance and medicine, organizations have encountered obstacles in data acquisition because their target applications need sensitive data that reside across multiple parties. One promising solution to this data scarcity problem is *collaborative computation*, where several organizations pool together their data and compute on the joint dataset. This type of computation enables parties to acquire a larger volume of data, as well as more diverse data. Unfortunately, organizations are often unwilling or unable to share their data in plaintext due to business competition or government regulation.

My dissertation focuses on solving this problem by enabling organizations to run complex computations on the joint dataset without revealing their sensitive input to the other parties. This dissertation presents four systems that utilize hardware enclaves as well as advanced cryptographic techniques for secure computation on workloads that range from SQL analytics to machine learning. By utilizing a wide range of tools from both systems and cryptography and also innovating on them, my systems provide strong and provable security guarantees and are often orders of magnitude faster compared to prior work or the more straightforward ways of integrating cryptography into systems.

To my parents, friends, mentors, and colleagues

Contents

List of Figures

List of Tables

Acknowledgments

This dissertation would not have been possible without the support of many people.

First, I want to thank my two advisors for their guidance and support during graduate school. I would not be here without them.

Raluca Ada Popa: I first saw Raluca when she was giving her job talk at MIT. I was taking my first security class that semester and was immediately fascinated by her work. I came to Berkeley in the fall and was happy to discover that Raluca had joined as well! I was very fortunate to be able to collaborate with her on my first research project here at Berkeley. Even though I came in with zero experience in cryptography, she was still willing to take me on as a student. She has been a tremendous help on every single paper that I have written with her, and guided me every step of the way from technical parts of research, to writing papers and giving talks.

Ion Stoica: Ion was the person I wanted to work with the most when I was applying to graduate school. I first met Ion during visit days and immediately knew that I wanted to come to Berkeley. Ion is full of wisdom on a surprisingly large number of both academic and personal topics. Over the years, he has taught me to become a better systems researcher, and to not only come up with good technical solution, but also *focus* on the bigger picture.

I also was very fortunate to have worked with other faculty at Berkeley and University of Washington, including Alessandro Chiesa, Joseph E. Gonzalez, Arvind Krishnamurthy and Scott Shenker. These faculty gave me valuable feedback on my work, and inspired my research to be more interdisciplinary.

Throughout my six years at Berkeley, I also met many people who became close collaborators and friends:

Ankur Dave: I first met Ankur at OSDI 2014, and we later became close collaborators on Opaque [221]. It was very fun working with him, and I enjoyed the many hours of pair programming that we did in the fishbowl room.

Grant Ho: I roomed with Grant for my first two years in Berkeley, though somehow we managed to become even closer later on in grad school. Grant is not only an excellent security researcher, but also a humorous and supportive friend. Grant also has always has the best comebacks!

Pratyush Mishra: I knew of Pratyush from the classes we took together, in which he was very active and had many interesting technical insights to share. We eventually became friends as well as collaborators on Delphi [161]. I especially enjoyed the intense and polarizing political discussions.

Aurojit Panda: I met Panda my first day in the NetSys lab, and we quickly became close friends. Panda is one of the smartest people I know, and also has interesting opinions and views on literally almost every topic that one can think of. I'm very happy to have collaborated with him on multiple projects ([175, 218]) and have learned so much from him. He's also a food connoisseur and I can always rely on him to give the best restaurant recommendations!

Rishabh Poddar: Rishabh and I did not meet each other until my third year, but we quickly became inseparable, mostly due to the amount of time we spend bantering with one another. We always grab lunch together (so much so that people on Euclid would recognize us!) and would also shame each other into going to the gym! One big regret: we never wrote a paper together despite having many interesting technical discussions.
Shivaram Venkataraman: I first met Shivaram during visit days, when he took the new admits on a quest to find a mysterious waterfall. Though the end goal was ultimately never achieved, what was important was the journey itself, during which we were all able to take in the beauty of Berkeley nature and have interesting discussions. Over the years, we became close friends – he was a tremendous source of support and also shared a common interest for classical music (with a soft spot for Shasti).
Yifan Wu: Yifan and I met in my second year and went on many enjoyable long walks near the lab. We ended up rooming together at the end of my second year and have been roommates ever since. I always enjoyed and will miss the satirical roommate banter.

I have also worked closely with a number of other collaborators, listed here in alphabetical order: Rachit Agarwal, Jethro Beekman, Weikeng Chen, Ryan Deng, Ryan Lemkuhl, Frank Li, Akshayaram Srinivasan, Xin Wang, and Howard Wu.

I also had the privilege of interacting with many cool people in NetSys and the RISE lab, including Moustafa Abdelbaky, Peter Bailis, Michael Chang, Emma Dauterman, Vivian Fang, Silvery Fu, Yuncong Hu, Sukrit Kalra, Anurag Khandelwal, Sam Kumar, Chester Leung, Lydia Liu, Horia Mania, Nathan Malkin, Murphy McCauley, Radhika Mittal, Robert Nishihara, Kay Ousterhout, Ashwinee Panda, Qifan Pu, Gur-Eyal Sela, Justine Sherry, Jeongseok Son, Vikram Sreekanti, Amin Tootoonchian, Chia-Che Tsai, Stephen Tu, Stephanie Wang, Jean-Luc Watson, Michael Whittaker, Chenggang Wu, Rashmi Vinayak, Neeraja Yadwadkar, Zongheng Yang, and Wen Zhang.

I also want to thank my best friend C.J. (whom I've known for more than 10 years!) for always being supportive and understanding. She is always willing to listen to any complaint I have!

Last but not least, I want to thank my parents for their continuous support and encouragement throughout grad school!

Chapter 1

Introduction

1.1 Motivation

The recent revolution in advanced data analytics gave rise to a growing demand among organizations for high quality data. A large amount of such data, when fed into increasingly advanced data analytics and machine learning techniques, can significantly enhance the capabilities of existing applications. Though there has been much progress on developing faster and more intelligent analytics systems, there is still a glaring problem at hand: how can organizations acquire even more high quality data than they currently have?

One promising solution to this data scarcity problem is *collaborative computation*, where several organizations pool their data together and compute on the joint dataset in order to mutually benefit from the results. This type of computation enables parties to acquire a larger volume of high quality data, as well as more diverse data. Unfortunately, many organizations are often unwilling or unable to share their data in plaintext due to business competition or government regulation. This obstacle not only hinders organizations from improving their existing applications, but also prevents any growth of new applications that can only be enabled by cross-institutional data sharing. Below, I analyze two use cases in this setting that were shared with us by our close industry collaborators.

The first use case arose out of two North American banks' need to effectively detect money laundering activities. Recent development in machine learning classification algorithms could identify such activities, and thus enhance the banks' yields by multiple orders of magnitude [190]. Since criminals often hide their traces by moving assets across different financial institutions, an accurate model would require training on multiple banks' customer transaction dataset. Unfortunately, even though such a model would benefit all participating banks, they are not willing to share their customers' data in plaintext because of privacy regulations and business competition.

The second use case was shared with us by a major healthcare provider who needs to distribute vaccines during the annual flu cycle. In order to launch an effective vaccination campaign (e.g., sending vans to vaccinate people in remote areas), this organization would like to identify areas that have high probabilities of flu outbreaks using machine learning. Once again, such training is impossible at this moment because the seven organizations cannot share their patient data with each other due to privacy regulations.

1.2 Secure collaborative computation

My dissertation work aims to solve the problem of collaborative computation among mutually distrusting parties by building systems that enable such computation on encrypted data. My systems allow multiple organizations to collaboratively run complex computations on their joint dataset without revealing their sensitive input data to each other. The security guarantees are formulated to emulate an *ideal trusted third party* (but without using such a physical trusted entity). At the end of the protocol, the parties should only additionally learn the final output of the secure computation. In fact, none of the intermediate results is revealed.

The past few decades have seen much progress towards making secure collaborative computation a reality. Currently, there are two main techniques for securing computation and emulating a trusted third party: *hardware enclaves* and *secure multi-party computation (MPC)*. Hardware enclaves [155, 129, 140] – or trusted execution environments (TEEs) – are special hardware mechanisms that can be used to create isolated environments on untrusted machines. By leveraging a trusted processor that is able to store encrypted data in memory (and decrypt only within the on-chip cache), this isolated execution environment protects against even a malicious privileged piece of software (such as the operating system). The second approach is via leveraging purely cryptographic techniques such as secure multi-party computation (MPC) [24, 103, 215]. Using MPC, parties can collaboratively compute with each other in a completely decentralized manner. The parties only exchanges encrypted messages and executes encrypted local computation, and no specialized hardware is needed at all. These two techniques are quite different, and also have different performance, security, and usability trade offs (see a detailed comparison in Chapter 2). Therefore, there is no single technique that works for all applications, and an application (and the application developer) must choose the right approach based on its needs.

1.3 Overview of thesis

My dissertation uses both of the above approaches to build secure and practical systems that enable parties to *share* their data for collaborative computation, but not *show* their data in the plaintext to the other participants. Although the field of secure collaborative computation has made amazing progress over the past decades, these advances are still lacking in achieving the goal of combining security, functionality, and performance. On the one hand, though the enclave approach is performant and functional, it is lacking in security guarantees because it is prone to an important type of side-channel attack called *access* *pattern leakage* [214, 171]. An adversary can infer the underlying encrypted data from only observing memory and network access patterns. On the other hand, while MPC is quite secure, it is still difficult to execute more complex functionalities efficiently. Moreover, most prior MPC systems only work within the honest-but-curious threat model, and does not defend against an adversary who can choose to deviate from the protocol. Finally, many of the advances in MPC over the past decades have been purely theoretical and have not been incorporated into practical systems that are programmable and easy to use.

My approach to solving this problem is via a *co-design of systems and cryptography*. By bridging the gap between theory and practice, as well as taking a systems-oriented approach that is backed by a deep understanding of cryptography, I build systems for secure collaborative computation that are secure, functional, and performant. Throughout my work, I not only utilize a wide range of tools from both systems and cryptography, but also innovate on them to make practical solutions possible. My systems provide strong and provable security guarantees and are often orders of magnitude faster compared to prior work or the more straightforward ways of integrating cryptography into systems.

For my thesis, I decided to secure two important workloads: SQL analytics and machine learning. The rest of this dissertation has a detailed presentation of my research work and will be arranged as follows. Chapter 2 first presents some background information on the two approaches for secure computation as well as their trade-offs. The next two chapters are focused on systems that secure the SQL analytics workload: Chapter 3 presents MiniCrypt, an encrypted and compressed cloud-based key-value store that can aid data analytics by enabling secure and efficient storage capabilities; Chapter 4 presents Opaque, an encrypted SQL analytics system built on top of Spark SQL and hardware enclaves, and also efficiently provides data access pattern protection by using oblivious computation. The following two chapters are focused on systems that secure machine learning: Chapter 5 presents Helen, an MPC-based system for training regularized linear models in the multi-party, dishonest majority and maliciously secure setting, where the system is able to defend against an adversary who can deviate from the protocol; Chapter 6 presents Cerebro, a programmable, end-to-end system for executing collaborative machine learning training and inference on MPC, and supports various levels of threat models. Finally, Chapter 7 concludes this thesis.

The technical material is adapted from existing published or co-authored works. Chapter 2 has some material adapted from [219, 222]. Chapter 3 is adapted from [220], chapter 4 is adapted from [222], chapter 5 is adapted from [219], and chapter 6 is adapted from [218].

1.4 Adoption

As a system security researcher, one of my goals is to design, build, and open source system artifacts that provide strong security guarantees. For example, Opaque (see Chapter 4) is open sourced, and IBM Research deployed the system for a volunteer matching application. Alibaba and Ericsson have used it in internal use cases. It was also used in a proof-of-concept by Scotiabank and Microsoft on an anti-money laundering use case.

Chapter 2

Background

This chapter presents some background on the two main approaches for secure collaborative computation. The first approach is via utilizing hardware enclaves such as Intel SGX [155], The second approach is by using a purely cryptographic approach like secure multi-party computation. This chapter will first introduce these two approaches in Section 2.1 and Section 2.2. Both of these approaches are valuable because they have different security and performance trade offs, and a comparison of the two techniques will be discussed in Section 2.3.

2.1 Hardware Enclaves

Secure enclaves are a recent advance in computer processor technology providing three main security properties: fully isolated execution, sealing, and remote attestation. The exact implementation details of these properties vary by platform (e.g. Intel SGX [154] or AMD Memory Encryption [129]), but the general concepts are the same.

The general notion of an enclave has several properties. First, *isolated execution* of an enclave process restricts access to a subset of memory such that only that particular enclave can access it. No other process on the same processor, not even the OS, hypervisor, or system management module, can access that memory. Additional security measures may be provided by the platform such as memory encryption or a separate memory bank used solely for secure enclaves. Generally, as part of the isolated execution, the enclaves have no or limited access to the I/O system. This isolation dramatically reduces the Trusted Computing Base (TCB) for the enclave, which comprises of only the enclave code itself and the secure enclave platform. This is a break from the traditional hierarchical kernel–userspace privilege model, in which the entire operating system kernel is generally considered to be part of the TCB of a user process. Second, *sealing* enables encrypting and authenticating the enclave's data such that no process other than the exact same enclave can decrypt or modify it (undetectably). This enables other parties, such as the operating system, to store information on behalf of the enclave. Third, *remote attestation* is the ability to prove that the desired code is indeed running securely and unmodified within the enclave of a particular device.

While hardware enclaves are very powerful, they are also prone to various *side-channel* attacks such as data access pattern leakage. This will be explained more in Section 4.1.

2.2 Cryptographic approach

The second approach for secure collaborative computation is a purely cryptographic approach. This section will focus on secure multi-party computation (MPC) instead of fully homomorphic encryption [94] because, while the latter has seen much improvement over the years, it is still mainly used for linear workloads and cannot efficiently handle non-linear workloads.

MPC enables multiple participants to *emulate* an ideal trusted third party who can help the participants compute a function f over P parties' private inputs $x_{i \in [1 \ldots P]}$. The private input x_i is only known to party i and is never directly revealed to any other party. The computation's final result is often released in plaintext to every participant.

There are two main MPC paradigms for generic computations: *arithmetic MPC* [24, 69] and *boolean MPC* (in particular, garbled circuits [215, 211]). In arithmetic MPC, data is represented as finite field elements, and the only supported operations are addition and multiplication (called "gates"). On the other hand, boolean MPC represents data as boolean values, and the only supported operations are XOR and AND.

One interesting commonality between these two frameworks is that they are both split into two phases: preprocessing and online execution. At a high level, both frameworks use preprocessing to improve the online execution time for certain gates. In arithmetic circuits, addition gates can be computed locally without communication, while multiplication gates are much more expensive to compute. Similarly, in boolean circuits, XOR is fast to compute while AND is much slower to compute. The preprocessing phase for these frameworks precomputes a majority of the cost of executing multiplication/AND gates. Furthermore, the preprocessing phase can execute without knowing the *input*; it only needs to know the *functionality*.

The online execution for both arithmetic MPC and boolean MPC requires the parties to input their private data. At the end of this phase, the secure protocol will generate an encrypted output. All parties jointly decrypt this encrypted output to get the plaintext result.

2.3 Comparison

In this section, I analyze these two approaches' trade offs in terms of setup, security guarantees, and performance.

Setup In secure collaborative computation, the application usually requires access to multiple parties' data. The parties have two different setups to choose from. The first is to utilize a cloud provider as an aid for secure computation. Utilizing the cloud can be very beneficial because it can provide elastic resources, and clients do not have to hire dedicated IT to manage their on-prem resources. However, we do not want the cloud provider to see the plaintext data, so the cloud needs to support some kind of platform for secure computation such that it only sees encrypted data. In this scenario, hardware enclaves are a perfect fit since the cloud can more easily obtain and upgrade its hardware, and a purely cryptographic solution is either too slow or does not fit the setup very well.

The second setup is a completely decentralized model where the parties collaboratively compute among themselves and without the aid of a cloud. If this is the setup, then MPC's execution model works very well. Enclaves can also be utilized if a subset of the parties can procure the necessary hardware, but this may not be easy to accomplish. Such hardware updates can usually be delegated to the cloud instead.

Security guarantees Hardware enclaves and a purely cryptographic approach have very different security guarantees. Enclave's threat model does need to rely on the hardware being trustworthy. For example, if Intel SGX is used for computation on encrypted data, then by default the threat model assumes that SGX's hardware cannot be compromised, the implementation is correct, and that Intel is trustworthy. Enclaves are additionally prone to *side-channel attacks* [214, 171], where the adversary can learn additional information about the encrypted data from observing information such as data access patterns or power usage of a computation. Utilizing cryptography does not have the extra hardware assumption, though the cryptographic implementation would also need to be trusted. However, many MPC systems also only defend against an honest-but-curious adversary (i.e., an eavesdropper) in order to improve performance, while hardware enclaves have integrity protection that is useful against a malicious attacker who has compromised the operating system.

Performance Performance varies greatly for these two approaches. Since trusted processors are able to decrypt encrypted data inside an isolated environment, it can run secure computation at near-processor speed. The performance will be slower if the enclave computation is augmented with protection against data access pattern leakage, but it will still be faster than utilizing MPC. MPC relies on complex mathematical operations on big integers for processing secure computation and are thus often orders of magnitude slower than their plaintext counterparts.

Chapter 3

MiniCrypt

This chapter is the first of two chapters that focus on systems that enable secure collaborative analytics. As mentioned in the background, many users who wish to utilize cloud resources for secure computation also store their sensitive data in cloud-hosted storage [156, 184, 72, 163, 47, 57, 138, 6, 133] for easy retrieval. However, the current set of solutions is problematic because the users store their sensitive data in plaintext and must trust the cloud providers. Therefore, applications need data storage systems that are able to preserve data confidentiality by only operating on encrypted data while also handling common workloads very efficiently.

In this chapter we present MiniCrypt, the first cloud-based key-value store that achieves the benefits of both compression and encryption. Many big data stores employ compression [135, 47] to significantly increase performance, sometimes by up to an order of magnitude [6, 133]. Compression is effective at providing performance gains because it enables servers to fit more data in main memory, thus decreasing the number of accesses to persistent storage. At the same time, a common technique for protecting data confidentiality is to encrypt the data stored on servers [181, 30, 99, 128] and keep the key at the client.

Unfortunately, existing systems choose either compression or encryption – but not both – because there is a *fundamental tension* between encryption and compression. Compressing encrypted data (which is randomized) is not viable because pseudorandom data is not compressible. Second, while encrypting compressed data works well in some systems, it is problematic in the database setting. Compressing a single row of data typically provides limited compression ratio, while compressing multiple rows together means the server cannot maintain fine-grained access to these rows/attributes and makes it more difficult to maintain correct semantics. There are a range of effective database compression techniques [3, 27, 114] that also permit querying data, but their layouts leak significant information about the data.

MiniCrypt is able to address this problem by first making an empirical observation that, while compressing a single row does not produce a good compression ratio, compressing together *only relatively few* key-value rows can actually achieve a compression ratio that is close to compressing the entire dataset. As we discuss in Section 3.2, we observed this behavior for a wide range of datasets that could be stored in key-value stores, such as data from Github, genomics, Twitter, gas sensors, Wikipedia, and anonymized user data from Conviva (internet-scale video access logs) [56]. Based on this observation, MiniCrypt's design works by packing few key-value pairs and compressing and encrypting these packs. Of course, this introduces some significant challenges because encrypting packs removes the server's ability to to manage the individual key-value pairs within the same pack since the server does not have the key to decrypt these packs. MiniCrypt solves these challenges by using an efficient client-centric design to maintain the same semantics.

3.1 Overview

3.1.1 Model and threat model

We adopt a cloud-hosting model, consisting of two roles: the hosting service (the server, which can be distributed), and the company/organization that uses the hosting service (the customer). The customer consists of one or more client machines within the same trust domain; these share a single encryption key, which is not available to the server. The server hosts the encrypted key-value store, and the clients issue queries to the server.

MiniCrypt protects against an attacker who manages to gain access to the server, can see all server side data (including messages from the clients), and attempts to exfiltrate the data. MiniCrypt considers a passive attacker, which follows the MiniCrypt protocol (e.g., a curious system administrator). In particular, it does not corrupt the data and does not provide incorrect query results. We also assume that the clients are trusted and allowed to see all server-side data (e.g., because the customer is the data owner). For MiniCrypt, we are concerned only with protecting the data from a server attacker. As a consequence, we assume that the attacker cannot control any client and hence cannot execute any queries through the client (e.g., insert, delete, get).

Many applications implement finer grained client access control to ensure that certain clients have access to only part of the data. Such access control can be implemented in various ways in a MiniCrypt setup and is complementary to MiniCrypt. For example, the customer can add a proxy between the clients and the server and the proxy acts as a MiniCrypt client: the proxy restricts access to queries and query results to the clients. Another way is to let the clients maintain different keys for each group of data with the same access permissions, essentially running a separate MiniCrypt instance for each group. Since pack compression is done on the client side, the client can pack data with different permissions in separate packs. Although both of these designs can be easily integrated into MiniCrypt, client access control is not the focus of MiniCrypt – MiniCrypt is concerned with protecting the data from the cloud provider.

Finally, access patterns (e.g., which keys are being retrieved) or timing attacks are out of scope for MiniCrypt.

Function	**Description**
`get` (`key`)	returns value associated with `key`
`put` (`key`, *val*)	sets the value for `key` to *val*
`delete` (`key`)	deletes the record with key `key`
`get` (`low`, `high`)	returns all the (`key`, `value`) pairs where `low` $\leq$ `key` $\leq$ `high`

3.1.2 Goals

MiniCrypt has the following design goals. First, MiniCrypt aims to provide end-to-end encryption for the values in the key-value store. Second, MiniCrypt aims to provide significant compression, which promises better performance in many situations. For example, MiniCrypt should achieve higher read throughput than a standard encrypted service without compression, because MiniCrypt can fit more data in memory. Third, MiniCrypt aims to work as a layer on top of *unmodified key-value stores*. This makes it easier to adopt MiniCrypt into different key-value stores and enables MiniCrypt to benefit from their performance and fault tolerance. Finally, MiniCrypt aims to provide eventual consistency guarantees, which is often utilized in big data key-value stores due to its performance.

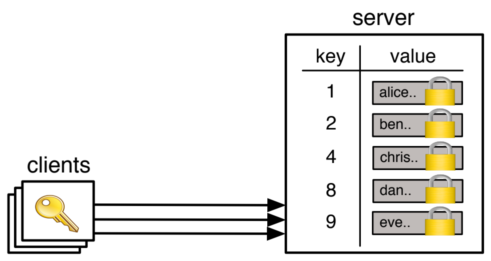
3.1.3 System API

MiniCrypt exposes the basic key-value store API, as well as support for range queries. Range queries (in the form of `get` (`low`, `high`)) are common for time-series big data [150, 45]. MiniCrypt supports the following API:

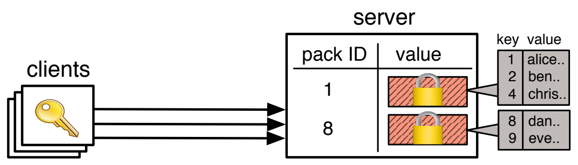
3.1.4 Strawman designs

Compression is a widely studied topic in databases [3, 27, 114, 6]. We briefly discuss two compression strawman designs and show their limitations.

The first approach is utilizing compression techniques that allow queries to be run on the compressed data [3, 114, 6]. Directly adding encryption will leak significant information about the data due to the data layout and the data access patterns. For example, run-length encoding (RLE) [3], commonly used in column-oriented databases, encodes runs of values (a contiguous sequence of the same values) together. Ten consecutive rows with the value “female" are encoded as (“female", 10). One possible way to integrate encryption is to encrypt the run value (e.g., “female”) separately, while leaving the run length (e.g., 10) in plaintext. This allows the server to answer a `get` (`key`) query, during which it can use the run length information to return the correct value. However, this arrangement easily leaks important information such as data frequency. For example, if the column is gender (F/M) or letter grades (A, B, C) and is sorted by this value, the server-side attacker can learn which row has what grade or gender by observing the run lengths that are stored in the plaintext.



(a) Typical setup with encryption.



(b) MiniCrypt's setup (encryption and compression).

Figure 3.1: System architecture for a typical encrypted key-value store and MiniCrypt. A lock indicates an encrypted item. Orange indicates a compressed item. MiniCrypt's values consist of compressed and encrypted packs.

Dictionary encoding [27] is another common compression technique. The clients share a compression table that maps uncompressed values to compressed codes. To compress, the clients look up specific values in this table and construct the correct compressed version. To decompress, the clients refer to the same shared table and translate the compressed values back into the correct uncompressed text. The advantage of this design is that it combines compression and encryption without introducing the complexity of packs. However, this approach has significant disadvantages. First, dictionary encoding works well for some columns (such as columns with few distinct values), but it does not work well in general. As an example, we ran this technique on Conviva data and found that, even though the compression rate was very high for some columns, the overall compression ratio was only 1.6. Second, each client needs to use the compression table for both reads and updates. If the table is stored on the server side, each client must do extra reads in order to decompress and compress, which will reduce the read throughput as well as leak significant information through access patterns on this shared table. Storing the table at the client imposes performance overhead. The compression table can be very big: for example, for Conviva, the table was 80% of the entire compressed data. Finally, as data gets modified, the contents of the table change over time. The database must provide a protocol for synchronizing the compression table stored on different clients, adding further complexity. Not updating the dictionary table might result in an out-of-date dictionary table that wastes space storing past encodings.

MiniCrypt aims to be a generalized system that can handle both reads and updates on a variety of data types while providing strong confidentiality guarantees. Our packing technique is independent of workloads, data types, and compression algorithms.

3.1.5 MiniCrypt's design overview

Figure 3.1 summarizes MiniCrypt's architecture in comparison to a regular key-value store that provides encryption in a straightforward way. This is a logical baseline for MiniCrypt because it provides similar security. The data in MiniCrypt is stored in packs, where each pack is a group of key-value pairs, compressed together and encrypted. The keys in a pack represent a contiguous range in a sequence sorted on the keys.

Each pack has a packID. We choose the packID to be the smallest key in the pack, and assume there is a sorted index on the packID. This means that to find a given key, we simply need to retrieve the pack with the largest ID smaller than or equal to the key. Each pack also has some metadata information such as a hash of its value and status messages (to be introduced in later sections).

To read a key, a client fetches the corresponding pack, decrypts and decompresses it. To write a key, a client updates a pack. As keys get deleted or inserted, some packs become too small or too large; in this case, MiniCrypt merges or splits them to maintain performance.

Security guarantees. MiniCrypt protects each pack with a strong and standard encryption scheme (AES-256 in CBC mode), which provides semantic security. The clients never send the decryption key to the server. Thus, this encryption protects the values in the original key-value store. The encryption leaks nothing about the contents of the pack, except for the size of each compressed pack. While MiniCrypt reveals the size of a compressed pack, it does not reveal the sizes of the original rows within the pack, which are revealed by the vanilla system. MiniCrypt enables reducing the information leaked by the size of the pack by padding the encrypted packs to a tier of a few possible sizes. MiniCrypt allows the customer to specify the padding tiers, such as small-medium-large or exponential scale, and pads each pack to the smallest tier value that is at least the pack size. This strategy provides a tradeoff between compression and security. Note that, in our threat model, an attacker does not have the ability to issue write queries to the storage through the client, which prevents the attacker from doing injection attacks that exploit pack size.

The keys in a key-value store are often not sensitive (e.g., random identifiers, counters, timestamps). However, if the keys are deemed sensitive, then the packIDs should be encrypted since a packID reveals the lowest key in the pack. Note that the rest of the keys are automatically encrypted by MiniCrypt as part of a pack. MiniCrypt supports packID encryption only in `GENERIC` mode and for a system that does not perform range queries, as follows. MiniCrypt applies a pseudorandom function [100] to the packIDs, keyed using a different symmetric key for each table, and treats that as the new packID. While this is a deterministic encryption scheme, it is essentially as secure as randomized encryption because the keys in a key-value store are *unique*. This determinism allows MiniCrypt to proceed as normal by maintaining a sorted index on the *encrypted* keys and allows the clients to directly query on them. MiniCrypt does not support range queries or `APPEND` mode in this case, because there is currently no encryption scheme that enables ranges while providing semantic security and being efficient in our setting. A number of different schemes for range queries on encrypted data exist, trading off either security or efficiency. For example, order-preserving encryption (OPE) schemes [32, 180] enable efficient range queries on encrypted data in exchange for revealing the order of packIDs to the server. In the rest of the chapter, we treat packIDs as unencrypted for simplicity, but our evaluation uses encrypted packIDs. Finally, MiniCrypt does not hide the number of rows/packs in the database or the structure of the database (e.g., number of tables).

Mode	Type of write	Pack op.	Perfor. note
Generic	all types (`get`, `delete`)	split	`put` uses update-if
Append	append, `put`, no `delete`	merge	fast appends

Table 3.1: Comparison between MiniCrypt's modes, including the pack maintenance operation.

3.1.5.1 The underlying key-value store

To achieve the goal of working on top of unmodified key-value stores, MiniCrypt should not rely on specialized or expensive primitives that are not supported by most key-value stores. For example, most key-value stores do not support transactions covering multiple keys. The few that do (e.g. Redis and Cassandra) support them with limitations and significant performance overhead.

In fact, MiniCrypt can work on top of any key-value store that supports an ordered index on keys and provides a conditional atomic update (update-if) primitive for a single row. The first property enables MiniCrypt to support range queries and to efficiently locate the packID for a given *key*, as discussed in Section 3.3.1 and Section 3.3.2. Most key-value stores support an index on the key and many of these enable ordered clustering or range queries on this key.

The second property is a lightweight transactional primitive that executes an update on a single row at the server only if the condition is true. This primitive can be of the form "UPDATE ... IF condition" or "INSERT ... IF NOT EXISTS". This type of transaction is relatively lightweight because (1) it operates on a single row and (2) it contains only one statement as opposed to a multi-statement procedure. Most key-value stores support such transactions. For example, in Cassandra, these are called *lightweight transactions* [81].

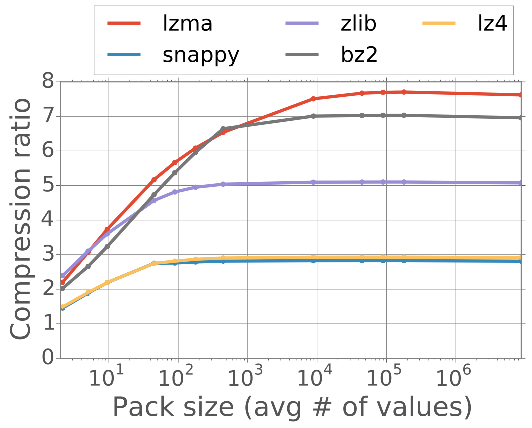
MiniCrypt maintains the semantics of the underlying store for eventual consistency, which is commonly used in the key-value store setting. MiniCrypt also benefits from the fault tolerance and replication mechanisms of the underlying store.

3.1.5.2 Modes of operation

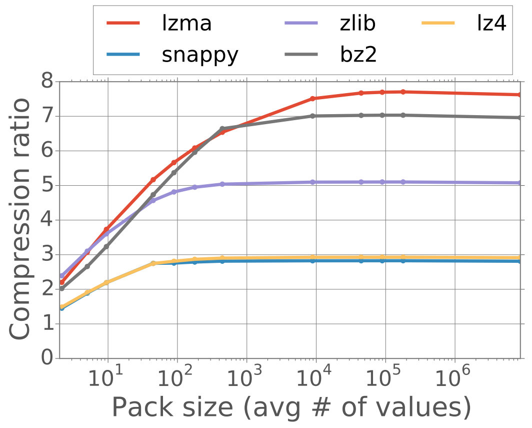
MiniCrypt provides two modes of operation: `GENERIC` and `APPEND`. Table 3.1 compares the two modes.

The `GENERIC` mode applies to any application: clients can update, insert, and delete key-value pairs. Writes use the update-if primitive to prevent clients from overwriting each other's updates. When a pack becomes too large, it is split. Since supporting both split and merge at the same time is overly complex, MiniCrypt does not support merging packs in this mode.

The `APPEND` mode supports applications in which writes are appends of new keys, and there are no deletes. Many big data applications are append-only. For example, a common



(a) Conviva



(b) Genomics

All datasets	Total number of rows	Average value size	Max compression ratio	Pack size (rows)
Conviva	8.7M	1.1KB	5.1	25
Genomics	1.6M	1.3KB	4.8	14.5
Twitter	2.5M	5.5KB	8.3	4.2
Gas Sensor	4.2M	135B	3.4	75
GitHub	50.8K	11.6KB	4.5	2.5
Wikipedia	202.5K	13.2KB	3.1	1.4

Figure 3.2: Compression ratios for different datasets. Note the x-axis is log-scale. The table summarizes the trend for each dataset using the zlib compression algorithm. For each dataset, it lists the total number of rows, the average size of the value in each row, the maximum compression ratio achieved (on the entire dataset), and the average number of rows that must be in a single pack to achieve a compression ratio that is $\geq 75\%$ of the maximum compression ratio.

pattern for big data is time-series data [150, 45], where the keys are timestamps, while values are typically actions or measurements. In this mode, MiniCrypt delivers high performance for appends, essentially as fast as the underlying system. Appends get inserted directly into the key-value store and not into packs. Then, background processes running on clients merge these keys in packs.

3.2 Key packing

In this section, we empirically justify the observation that compressing a *relatively small* number of key-value pairs together yields a compression ratio that is a significant fraction of the compression ratio obtained when compressing an entire dataset into one unfunctional blob.

The six datasets we examined are: anonymized Conviva time-series data on user behavior, genomics data where each entry consists of an identifier and a partial sequence of the genome, time-series Twitter data consisting of tweets and metadata, time-series data from gas sensors, Wikipedia files, and Github files from the Linux source code. We examined 5 different compression algorithms (bz2, zlib, lzma, lz4, snappy), which provide different tradeoffs in compression ratio and speed. For each pair of dataset and compression algorithm (30 pairs), we plotted the compression ratio against the number of rows in the pack. We first re-format each dataset into key-value pairs, then group the values into packs by adjusting a maximum threshold (in bytes) for each pack and calculate the average number of values present in each pack for a given pack size.

Figure 3.2 shows the results. We include here the graphs for Conviva and genomics with a table summary of the other datasets. The x-axis shows the average number of values, while the y-axis shows the compression ratio. We can see from Figure 3.2 that the compression ratio grows very fast as the number of rows increases, then quickly plateaus close to the maximum compression ratio achieved when compressing the entire dataset. For example, for Conviva, compressing 1 row yields a compression ratio of 1.6, compressing 50 rows yields a compression ratio of 4.6, and compressing the entire dataset of 1.5 million rows yields a compression ratio of 5.1. Hence, a relatively small number of rows per pack suffices for a significant compression ratio. We explain how MiniCrypt determines the pack size for a given dataset and system parameters in Section 3.7.3.

As surveyed in [71], there is a sharp tradeoff between compression ratio and the speed of compression/decompression. For example, bz2 and lzma have high compression ratios but poor compression/decompression speed, which affects client latency in MiniCrypt. Considering this tradeoff, we chose to use zlib in MiniCrypt as it achieves both a good compression ratio and good compression/decompression speeds.

3.3 Read operations

In this section, we describe read operations in MiniCrypt, which work the same in both the `GENERIC` and `APPEND` modes.

3.3.1 Get

Since key-value pairs are packed and encrypted in MiniCrypt, clients can only fetch at the granularity of a pack. A question is: how does a client know the packID given to the key of interest? One option is to maintain a table mapping keys to packIDs at the server. This strategy is undesirable because it increases server side space usage and query latency, and is difficult to keep consistent with the main data table during concurrent updates and client failures.

Instead, MiniCrypt enables clients to fetch the correct pack without knowing the packID. Since the packID is chosen to be smaller than or equal to the smallest key in the pack, the pack corresponding to a key k *is the pack with the highest ID from all the packIDs that are at most* k*.* This query can be run efficiently at the server because the underlying key-value store keeps an ordered index on packID. To find the right pack, the server simply locates k by retrieving the packID immediately preceding it. Once the client receives the result of this query, it decrypts and decompresses the pack. It then scans the content and retrieves the value for the key k. Figure 3.3 presents the overall procedure. The hash is a hash of the encrypted pack.

```
1: procedure GET(key)
2:     Fetch data from server:
       SELECT packID, value, hash FROM table
3:              WHERE packID ≤ key
                ORDER BY packID DESC LIMIT 1
4:     Decrypt and decompress value
5:     Return the entire row
```

Figure 3.3: `get` pseudocode.

3.3.2 Range queries

Range queries fit easily into MiniCrypt's design. In fact, for large range queries that touch many keys, MiniCrypt utilizes less network bandwidth than a regular key-value store because MiniCrypt compresses multiple keys together based on the query range. MiniCrypt performs a range query on packIDs using a key range [`low`, `high`]. Since a packID indicates the lowest key in the pack, a MiniCrypt client fetches the packIDs in [`low`, `high`]. If `low` is not equal to the smallest packID in the results, then MiniCrypt needs to fetch the pack that potentially contains keys from `low` to the smallest packID. The packs that contain key `low` and key `high` will need to be filtered by the MiniCrypt client as they can contain keys outside of the range [`low`, `high`]. The pseudocode for range queries is in Figure 3.4.

```
1: procedure GET(low, high)
2:     Fetch range from server:
       res ← SELECT packID, value, hash
3:                    FROM table
                      WHERE low ≤ packID ≤ high
4:     Decrypt and decompress each value in res
5:     if low < smallest packID in res then
6:         Run get(low) and add to res
7:     Filter out keys not in [low, high] from res
8:     return res
```

Figure 3.4: `get` by range pseudocode

3.4 Writes in the generic mode

In the `GENERIC` mode, clients can perform any type of write from the API in Section 3.1.3. MiniCrypt additionally supports pack splitting because a pack may grow too large if there are repeated inserts, and retrieving overly large packs will increase bandwidth usage and hurt performance.

3.4.1 Put

Writes are more challenging than reads in MiniCrypt. Since the server cannot update an individual key in a pack due to encryption, each client has to retrieve an entire pack to execute a write. The client updates the value of the specific key in the pack, compresses the pack, encrypts it and writes it back to the database. However, if designed naïvely, concurrent clients may overwrite each other's updates if they are all modifying the same pack.

To prevent such contention, we rely on the *update-if* primitive explained in Section 3.1.5.1. We use this primitive and hashes to ensure that clients do not overwrite changes from each other as follows. Consider a client C1 who wants to update a key in a pack. The client reads the pack and records its hash `h`. C1 then updates the contents of the pack and issues an *update-if* at the server, specifying that the value should be updated only if the hash of the pack is still `h`. If the hash is no longer `h`, it means that another client C2 has recently updated the pack. Thus, C1 should not perform the update because it can overwrite C2's update. Instead, C1 will retry the update by performing a read of the current pack value. Figure 3.5 presents the overall procedure.

Another possible design that preserves eventual consistency is to do a blind write of the pack without any update-if mechanism. However, a read of the pack is still necessary to decrypt and modify the pack. As we show in the evaluation section, the extra read incurs significantly more cost than the update-if mechanism. Hence, we chose to use update-if because its cost on top of the extra read is not significant and it preserves better the original data store's semantics.

3.4.2 Split

Pack splitting is useful when there are inserts in the system that cause packs to grow large.

When to split a pack Whenever a client runs `put` or `delete`, the client first checks the size of the retrieved pack after reading the pack. If the pack contains more than `max_keys`, the client proceeds to split the pack. The parameter `max_keys` is a system-wide constant, and can be set to $1.5 \cdot n$, where n is the desired number of keys in a pack. The client can proceed with the original operation once a split successfully completes.

```
 1: procedure PUT(key, value)
 2:     repeat
 3:         (packID, pvalue, h) ← get (key)
 4:         if # keys in pvalue > max_keys then
 5:             Do split as in Section 3.4.2. continue
 6:         Inside pvalue, set key's value to value
 7:         Compress and encrypt pvalue
 8:         Compute its hash phash
            ok ← UPDATE table SET value = pvalue,
 9:                                   hash = phash
                  WHERE packID = packID IF hash = h
10:     until ok
```

Figure 3.5: `put` pseudocode

How to split a pack Figure 3.6 shows the pseudocode for `split` (`packID`, `pack`, `h`), where `pack` and the hash `h` are the retrieved values from reading `packID`. During `split`, the client divides the pack by creating a left pack from the first half of the keys (rounded up) and a right pack from the rest of the keys. Note that we require this operation to be deterministic so that every client that reads the same pack will divide the pack in exactly the same way. The client then compresses each pack and encrypts it as usual. It inserts the right pack and then the left pack, both using the *update-if* primitive.

The `split` procedure is safe in the presence of multiple clients as well as client failures. For example, two clients, A and B, may attempt to split the same pack. These clients will read the same pack, and then split in a deterministic way, resulting in the same left and right packs. If client A inserts the new right pack into the database first, client B will attempt to insert the same right pack. The second insert operation will fail because client A's insert has succeeded.

What if a client fails in the middle of a `split` operation? As an example, let's assume that a client fails its `split` right before step 5 of Figure 3.6. This means there are two copies of the new right pack's rows in the database: one copy in the newly inserted right pack, and one (stale) copy in the original pack. This is still safe because a new client that attempts to read/modify those keys will retrieve the newly inserted right pack instead of the stale copies. Any client that attempts to modify the original pack will execute the `split` procedure. Note that the new right pack will not be overwritten since we do not delete packs.

3.4.3 Delete operations

`delete` is similar to `put` except that the key is removed from the pack. The ID of the pack does not change even when the lowest key in the pack gets removed. We do not remove packs when they become empty. The protocol for removing an pack is very complex because a split might reinsert a right half of the pack that was deleted.

```
1: procedure SPLIT(packID, pack, $h$)
2:     Assemble the right half of the pack:
       $right\_pack$ with $rightID$ and hash $rh$
3:     INSERT INTO table VALUES ($rightID$,
                 $right\_pack$, $rh$) IF NOT EXISTS
4:     Assemble $left\_pack$ with hash $lh$
5:     UPDATE table SET value = $left\_pack$, hash = $lh$
                        WHERE packID = packID
                        IF hash = $h$
```

Figure 3.6: `split` pseudocode

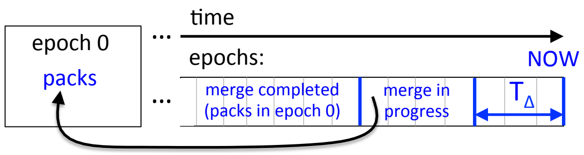


Figure 3.7: `APPEND` mode timeline.

3.4.4 Correctness

Recall that our notion of correctness was that MiniCrypt maintains the eventual consistency and liveness of the underlying key-value store. We will provide an intuition here.

The algorithm is safe when the operations are read-only. Updates that do not split the packs are also safe because of the hash check. The situation is trickier when there are concurrent updates with `split` operations. However, any client that issues a `put` or `delete` to a pack that has a number of keys greater than `max_keys` will block, since that client will choose to run `split` first. This allows concurrent modifications to safely co-exist with `split`. Furthermore, synchronization mechanisms such as "insert if not exists" and "update-if" ensure that concurrent splits on the *same* pack are safe. A client that is delayed during its split will not overwrite changes made after the split finished (due to other clients) to either of the two resulting packs. Finally, the liveness property is maintained because a client will not be stuck in an infinite loop — at least one live client will succeed to do a `put` or complete a `split`.

3.5 The `APPEND` mode

In `APPEND` mode, data is inserted into the system in order of roughly increasing keys. Clients can read keys, but no keys are updated or deleted once a certain time has elapsed since a key's first insertion. This mode fits applications whose writes are appends and enables these writes to be very fast. There are many `APPEND` mode use cases. For example, a common pattern for big data is time-series data [150, 45], where the keys are timestamps while the values are actions or events.

Let us define concretely MiniCrypt's assumption in this setting. First, MiniCrypt assumes that the keys are roughly inserted in order, but not in a perfectly increasing manner. MiniCrypt

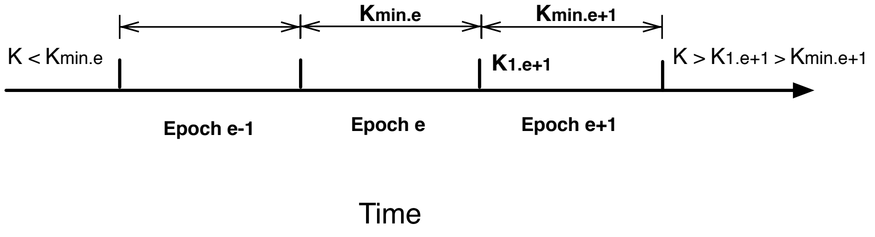


Figure 3.8: `APPEND` mode: an illustrated timeline showing the key constraints in each epoch. Let $k_{1.x}$ be the first key in epoch x, and $k_{min.x}$ be the minimum key in epoch x. The epoch time is designed to be large enough to provide guarantees that $k_{1.e+1}$ is less than all keys in epochs $e+2$ and beyond. This implies that $k_{min.e+1}$ is also less than all keys in epochs $e+2$ and beyond. We can also guarantee that $k_{min.e}$ is greater than all keys in epochs less than $e-1$, since $k_{1.e}$ is greater than all keys in those epochs. From this, we can guarantee that all keys between $k_{min.e}$ and $k_{min.e+1}$ occur in $e-1, e$ and $e+1$.

allows for a time lag in which keys do not appear (to `get` operations) in increasing order and requires that there is an upper bound on this lag denoted as T_Δ. Let key k be a key inserted at time t_k. The assumption is, for every key k, that no key less than k is inserted beyond $t_k + T_\Delta$; similarly, no key greater than k is inserted before $t_k - T_\Delta$.

What is T_Δ? T_Δ should be a conservative upper bound on the sum of the relevant time bounds, which include a bound on the time lag in which keys can arrive out of order (which could be due to the client), a bound on the network transfer time (the time it takes for a client request to reach the servers), and a bound on the server side time delay for new updates to be propagated to all available servers.

In the following section, we first present a basic design for the `APPEND` mode. We follow up with various improvements that can be made on the base design.

3.5.1 Design

The `put` operation is slow in the generic mode because MiniCrypt clients perform an extra read per `put` and employ synchronization (using *update-if*) to avoid overwrites due to concurrent `put` operations. To enable `put` to be fast in `APPEND` mode, MiniCrypt executes a `put` directly into the database (by compressing and encrypting a single row), and arranges for the clients to merge these inserts into packs in the background. In principle, this is possible because no key is updated or deleted once a certain amount of time has elapsed since it was first inserted, and a new key is not inserted between two such keys. In append mode, `put` should be very fast compared to regular put operations that do not operate on packs.

The main challenge that we face in `APPEND` mode is that the merge process can be expensive, but must keep the same rate as `put`s. Clients must be in charge of merging packs since the server cannot decrypt data. This means that the clients must read and delete a lot of keys, and that multiple clients might attempt to merge the same keys (potentially causing pack overwrites) while leaving other keys unmerged.

MiniCrypt addresses this challenge through a careful design of the system in `APPEND` mode. First, MiniCrypt groups keys into monotonically increasing *epochs*, which are useful in enabling batch processing. Each insert or update belongs to a single epoch. In many key-value stores, retrieving an entire epoch (e.g., if it is a partition as in Cassandra) and deleting the entire epoch is much faster than performing many server-side operations for reading and deleting each key.

We create a service called the *epoch management* (EM) service to manage epochs and the associated metadata. The EM service maintains a global epoch that is periodically increased. Each epoch is `EPOCH` seconds long. Clients periodically synchronize their local epochs with the global epoch. We define T_{drift} to be the maximum amount of time any client can be out of sync with the current global epoch. To maintain correctness, we need to make sure that `EPOCH` $> T_{\Delta} + T_{drift}$. In practice, T_{drift} is very small compared to `EPOCH` (we set it to be 10 seconds in our experiments), and we require any new/recovered client to synchronize its local epoch before doing an insertion.

Clients merge at the granularity of epochs in a deterministic way: sort the keys in an epoch and group them in packs of a given size starting with the smallest key. Thus, even if two clients concurrently merge the same epoch, the results are the same. The merged packs are placed in a special epoch 0, and the keys in the merged epochs are eventually deleted. Figure 3.7 shows the timeline of merge operations.

One way for the clients to merge epochs is to randomly pick an epoch to merge. Even though determinism ensures correctness, this method is inefficient because clients might do wasted work by merging the same epochs. MiniCrypt attempts to avoid having many multiple clients merge the same epoch by allowing the EM service to assign epochs to clients so that only one client is merging an epoch. If a client fails, the EM service will assign new clients to those potentially unmerged epochs.

3.5.1.1 The EM service

For availability, the EM service runs on the server. The EM does not see the contents of the packs, but only manages information about what client should merge what packs; hence, hosting the EM on the server does not pose a security issue within our threat model.

The EM maintains three pieces of information: the `stats` table, the `clients` table, and `g_epoch`, which we explain below. All this information is stored in the server database, so the EM is essentially a client of the underlying key-value store, and requires no changes to this store.

The `stats` table contains an entry per epoch of the form: epoch ID, client ID (the client that is in charge of merging the epoch), and status (current status of epoch, could be `NOT_MERGED`, `MERGED`, or `DELETED`). When a new client comes online, the client updates the `clients` table on the server by inserting its client ID and its local timestamp. Each client periodically refreshes that timestamp to indicate that it is still alive. The EM service periodically reads the `clients` table to assign clients to epochs, and to make sure that currently active clients are still alive. If a client times out, the EM service will scan through the `stats` table and assign all unmerged epochs with the failed client's ID to new clients.

The EM also maintains `g_epoch`, the current global epoch number, and updates it once every every `EPOCH` seconds.

3.5.1.2 Put

A `put` operation in `APPEND` mode is simply a single-row insertion of the key-value pair. When a client wishes to execute a `put`, it looks at its locally stored variable `c_epoch` for the current global epoch. The client loosely synchronizes the `c_epoch` variable with the server-side `g_epoch` by periodically querying `g_epoch`. In MiniCrypt, we adjust the period to be short enough so that `c_epoch` is at most one epoch behind of `g_epoch`. The client uses `(c_epoch, key)` as the new key and inserts `((c_epoch, key), value)` into the table, where `value` is compressed and encrypted.

3.5.1.3 Get

A `get` operation in `APPEND` mode is similar to the `GENERIC` mode `get`. A client first queries epoch 0 using the `GENERIC` mode query method. If the key is not found in the retrieved pack, the client retrieves the `stats` table, which additionally contains the minimum key for each epoch. The client finds the epoch with the largest "minimum key" that is smaller than the queried key. Let this epoch be e. Since the keys can be inserted roughly out of order, the actual record could be in either epoch e or $e-1$ (but not beyond $e-1$ since each epoch is long enough to cover $T_{\Delta} + T_{drift}$). The client will execute `get` for at most two epochs. Note that due to concurrent merges, one could miss the key if that key is merged right before either of the queries. Therefore, if both reads miss, the client performs an extra read of epoch 0 to attempt to find the key. Note that there are still cases where the key exists but is not found partly due to the fact that the underlying key-value store is itself eventually consistent, so the client must retry after a delay.

3.5.1.4 Merge

Each client's merge process periodically reads the `stats` table to find epochs that the client is responsible for. Consider a client that is responsible for merging epoch e. Because of the loose epoch synchronization on the client side and the fact that key inserts are also only roughly increasing in order, we cannot take keys from epoch e, order by key, and directly merge them into packs. We still wish to maintain the pack semantics – that each pack uses the minimum key as its pack id, and that all key-value pairs reside in the correct pack. An `APPEND` mode `get` should be able to use a range query to retrieve the correct pack for a query key once that key-value pair has been inserted. Therefore, the merge process begins by reading back all key-value pairs from $e-1$, e, and $e+1$. We use the *minimum key* from epochs $e, e+1$ (denoted $k_{min.e}, k_{min.e+1}$), as markers for deciding which keys to merge: all key-value pairs from $e-1, e, e+1$ with keys that are greater than or equal to $k_{min.e}$ but less than $k_{min.e+1}$ are grouped together and merged. Once the correct key-value pairs are retrieved, the merging process is easy: we simply order every key-value pair by key, then split them into packs based on a pack threshold. These packs are then inserted into epoch 0. After the packs have been inserted, the client updates the `stats` table with a status that the epoch has been merged by setting status to `MERGED`.

Each client also periodically deletes epochs. An epoch e can be safely deleted if its status is `MERGED` and epochs $e-1$ and $e+1$ are either `DELETED` or `MERGED`. After deletion, e's status is set to `DELETED`.

3.5.2 Fault tolerance for the EM

There need only be one EM machine running at a time because the job of the EM is light. For reliability, we distribute the EM service into multiple replicas. In our design, we assign each server replica an *EM instance*. One of these instances is the master EM, which is in charge of modifying the `stats` table and updating the global epoch. To survive EM master failures, the server contains an entry `EMreplica` identifying which replica is the master EM. The only task of the other EM replicas is to ping the master replica periodically to check if the master is alive.

If a replica believes the master is down, it updates `EMreplica` to designate itself as the master; this update is performed with an update-if, and thus relies on the underlying store's lightweight transactional mechanism to agree on the next replica to run the EM service. Hence, every update to `g_epoch` and the other EM data structures is performed using an update-if. This makes our design safe even if multiple EM replicas believe they are masters. We assume that the EM replicas' clocks are roughly synchronized (using a protocol such as NTP [160]), which means that multiple masters can safely update the global epoch by checking its local timestamp with the timestamp of the `g_epoch`. Since there is a further timing overhead from synchronization, the epoch time needs to be adjusted appropriately. Second, our merge protocol is deterministic, which means that it is safe even if multiple clients attempt to merge the same epoch.

3.5.3 Correctness

We provide an intuition for the correctness of `APPEND` mode. MiniCrypt provides eventual consistency guarantees because all replicas will eventually reach the same state after updates stop. MiniCrypt's protocols are also correct. The writes are regular single-key inserts, thus inheriting the correctness of the underlying system. The `merge` process preserves correctness for three reasons. First, there are no concurrent `put` and `delete` operations because clients only merge epochs that are two epochs older than `g_epoch`. We carefully choose the epoch time to ensure that the inserts/updates have settled and that the values will not change further (essentially becoming immutable). Second, `merge` operations by two clients on the same epoch result in the same outcome because the merge operation is deterministic. Since the client always reads the `stats` table (executing a read that reads back the *latest* status) before attempting to merge epochs, it will never attempt to merge a partially deleted epoch because the epoch's status is always set to `DELETED` before the actual deletion begins. Finally, the `merge` protocol first inserts keys in epoch 0 before deleting them; `get`-s are not affected because a `get` queries epoch 0 as well as unmerged epochs.

Figure 3.8 also shows an explanation of our append mode's correctness through an epoch timeline analysis.

3.6 Implementation

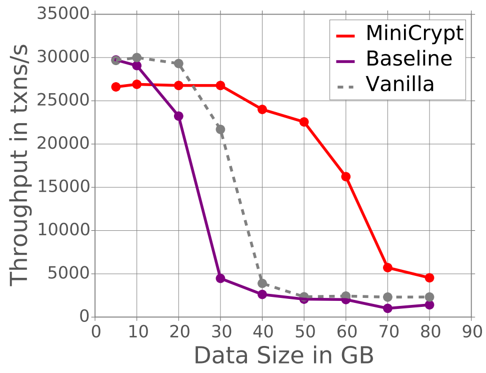
We implemented MiniCrypt in approximately 5000 lines of C++ code on top of an existing key-value store, Cassandra [138]. Cassandra is a widely used open-source key-value storage system that is both scalable and highly available. Our implemented interface does not make *any* internal modifications to Cassandra, and simply calls Cassandra's C++ driver. We used zlib to compress packs and OpenSSL AES-256-CBC to encrypt them.

Cassandra uses consistent hashing to distribute its data. Primary keys in Cassandra consist of a partition key and optional clustering keys. To find a particular piece of data, Cassandra simply hashes the partition key. Cassandra does not support range queries directly on the partition key, but clients can order rows within a partition based on the clustering keys. This allows for range queries within a Cassandra partition. MiniCrypt sorts data in descending order to optimize for get queries.

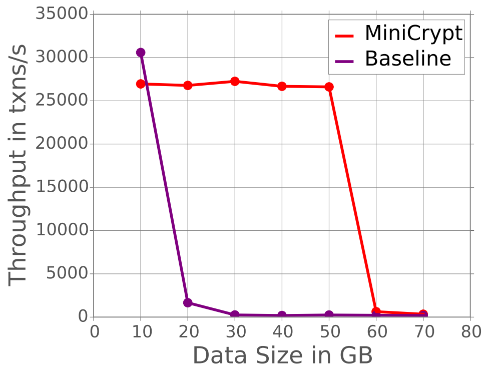
To support a generic key-value store interface, MiniCrypt makes some small adjustments to fit Cassandra's design. For a key-value pair `(key, value)`, MiniCrypt takes `key` and hashes it to a hash value. Using this hash value, MiniCrypt is able to assign the keys to `N` partitions. The default number of hash partitions is 8, though the user may adjust this parameter. Therefore, the primary key used in Cassandra is a key pair `(part_key, key)` where `part_key = SHA256(key) mod N`. Within each hash partition, the data is ordered by `key`. If a user wishes to perform `get(k)`, MiniCrypt will hash `k` to get a partition number, then use the get operation described in Section 3.3.1. For range queries, MiniCrypt has to make a range query request to each partition. In addition to a generic interface, MiniCrypt also has the ability to support a compound primary key `(partition key, clustering key)` (similar to what Cassandra supports). The primary key used in this situation is key pair `(part_key, key)` where `part_key = SHA56(partition key) mod N`.

3.7 Evaluation

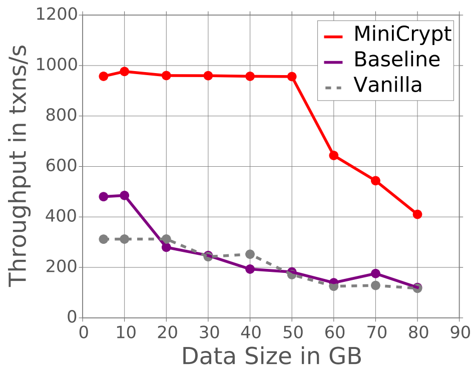
Our experiments were conducted on Amazon EC2. All benchmarks were run on a small cluster of 3 c4.2xlarge instances, and each instance has 15 GB of memory and 8 cores. The SSD experiments were run with 2 provisioned SSD drives per instance. The disk experiments were run with 2 magnetic drives per instance. The Cassandra replication factor was set to 3. All benchmarks use the Conviva dataset row values [56], consisting of approximately 1100



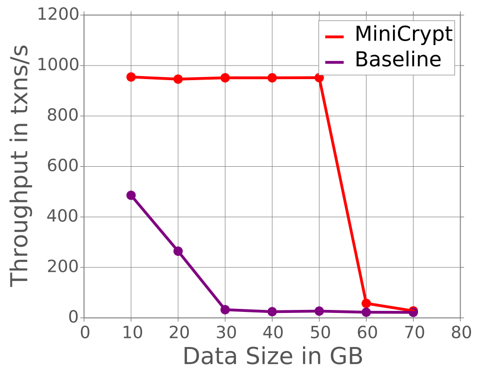
(a)



(b)



(c)



(d)

Figure 3.9: Point queries on (a) SSD (b) disk; range queries on (c) SSD (d) disk

bytes of anonymized customer data. All experiments use one 8-byte integer as the key, and one Conviva row as the value. Unless indicated otherwise, all MiniCrypt experiments (in both `APPEND` and `GENERIC` modes) set pack size to 50 rows.

We compare the performance of MiniCrypt with a baseline encrypted client. The baseline embodies a typical encrypted system that gives confidentiality guarantees by encrypting each row individually. This client has similar security to MiniCrypt, but it does not have the compression benefits of MiniCrypt. Nevertheless, we also use the same compression algorithm on each row before encrypting it to give this client an advantage. The compression ratio for single rows is roughly 1.6. Additionally, we compare MiniCrypt with a vanilla Cassandra client that uses no encryption and no compression for the 100% read and range query benchmarks.

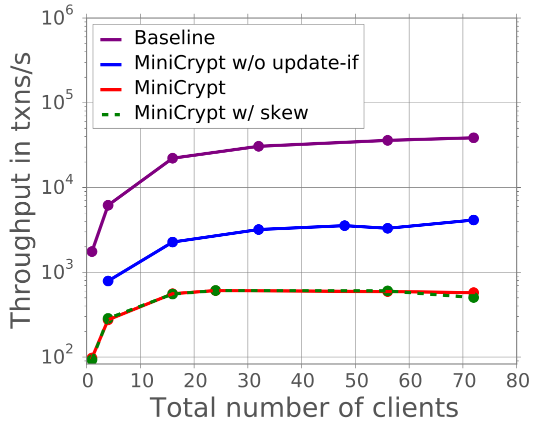


Figure 3.10: `GENERIC` mode 100% write

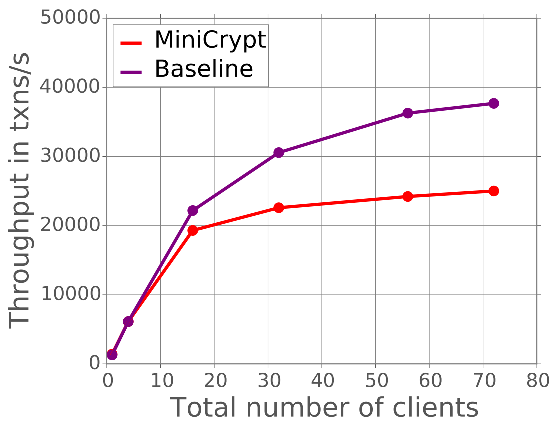


Figure 3.11: `APPEND` mode 100% write

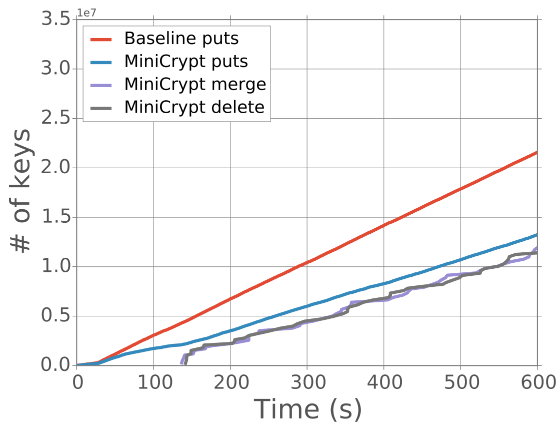


Figure 3.12: `APPEND` mode 100% write, long run

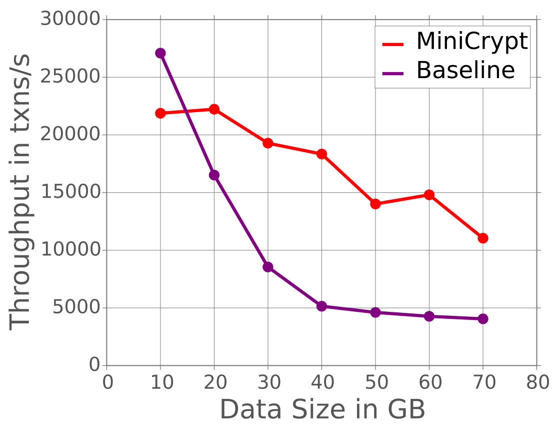


Figure 3.13: `APPEND` mode 50% read, 50% write

3.7.1 Read performance

We ran a modified YCSB read workload on a small three-instance cluster. We pre-loaded data into Cassandra and ran a 100% uniform read/range query workload on tables of different sizes. We compare MiniCrypt with a baseline encrypted client that compresses and encrypts each row separately, as well as a vanilla Cassandra client. The system was warmed up for 5 minutes for SSDs, and 10 minutes for disks. After warmup, each experiment was run for 60 seconds.

3.7.1.1 Point queries

Figure 3.9 plots the maximum server throughput (achieved by saturating the server with as many clients as possible) against varying overall dataset sizes. The same experiment was run with both SSD and magnetic disks.

We first compare MiniCrypt with the encrypted baseline client. When the dataset's size is small, both MiniCrypt and the baseline client fit the data in memory. The baseline client has a higher maximum throughput than MiniCrypt because the latter is retrieving more data and does extra processing (decompression, decryption) on the client. As the dataset size is slowly increased, the baseline client cannot fit in memory anymore. Once it starts accessing persistent storage for reads, the throughput drops significantly. Because MiniCrypt achieves higher data compression ratio, the same dataset that does not fit in memory for the baseline client still fits in memory for a MiniCrypt client. Thus, MiniCrypt continues to maintain high throughput until the compressed data can no longer fit in memory. In this situation, MiniCrypt is able to achieve roughly *100x* performance gain over the baseline client when the server is backed by disk, and *9.2x* performance gain when the server is backed by SSD. If both clients cannot fit data in memory anymore, MiniCrypt still manages to maintain good performance for a while because a large majority of the data it accesses is still in memory. For larger data sizes, we expect a crossover point where the baseline client becomes better than MiniCrypt because the query overhead starts to be dominated by accesses to persistent storage. MiniCrypt is weak in this scenario because it accesses an entire pack for a single point query. Compared to the SSD graph, the disk graph has much sharper drops. This behavior is expected because disks have a significantly lower read throughput for uniform access than SSDs.

MiniCrypt is also able to achieve roughly *6.2x* performance gain over the vanilla Cassandra client (SSD). The vanilla client's graph is similar to the encrypted baseline's graph, except shifted to the right. Since Cassandra utilizes compression on the server side, it is able to compress plaintext value to a certain extent. However, the compression ratio Cassandra achieves is not as good as that of MiniCrypt. These graphs show that MiniCrypt provides a significant throughput increase for a significant range of data sizes.

To compare latency numbers for both MiniCrypt and the baseline, we ran the same 100% read benchmark on SSD for a MiniCrypt client and a baseline client (both of which are single threaded) on a database size of 5 GB. This latency measurement is advantageous for the baseline client because its data fits in memory. The baseline achieves 1.019 ms per query, and MiniCrypt achieves 1.199 ms per query. MiniCrypt's extra latency gains come from the extra processing on the client.

3.7.1.2 Range queries

Range queries are very common in many workloads. For example, time series data (such as session logs) are frequently append-only and immutable when inserted. The logs are later retrieved by time range. The Conviva analytical query workload also retrieves customer data within a time range, where the range can be as short as one hour and as long as one week.

MiniCrypt's design makes range queries very efficient because MiniCrypt orders all key-value pairs and groups them into packs. For a point query, the space overhead is (pack size / compression ratio). For range queries (especially large scans that touch many records), the bandwidth overhead is reduced. For example, if the number of records queried is significantly greater than the pack size, the baseline client will have more bandwidth overhead than MiniCrypt (by a factor of C, where C is the ratio of MiniCrypt's pack compression to a single row's compression).

Our range query experiments are based on YCSB's short ranges workload. Each query selects a key k uniformly from the keyspace, and attempts to query all items between $(k - 1000, k]$. Figure 3.9 shows that MiniCrypt *consistently* experiences a significantly higher maximum range query throughput compared to both the encrypted baseline client and the vanilla client, both when the data fits in memory and when it does not. MiniCrypt is able to achieve up to 5x performance gain over the encrypted baseline client. Note that the vanilla client is slightly slower than the encrypted baseline client in the 100% range query experiment for small database sizes. This is due to the vanilla client being bottlenecked by the network. The vanilla client may achieve compression in memory, but still has to return the result in uncompressed format. The baseline encrypted client is able to achieve some compression for a single row, and can therefore return the result in the compressed format as well. As the size of the database increases, disk becomes more of a bottleneck, and the vanilla client and the encrypted baseline client converge to same throughput.

These experimental observations align with our analysis. Since our range is 1000 records, MiniCrypt is able to achieve much better performance because the compressed data has a higher compression ratio. When data can no longer fit in memory, the performance drops because of disk accesses. The drop is more significant for disk than for SSD.

3.7.2 Write performance

Generic mode In `GENERIC` mode, each write has two overheads: an extra read and a lightweight transaction (*update-if*). Figure 3.10 shows the result of running 100% uniform random writes on a pre-loaded 10 GB database, with the y-axis on a log scale. Each experiment ran for 120 seconds. The baseline client is fast because it is able to execute blind writes. MiniCrypt `GENERIC` mode with update-if is slow, but is mainly dominated by the extra read while the usage of the lightweight transaction introduces further stress to the system. This experiment justifies our decision to use the update-if mechanism because it gives much better guarantees of the system's semantics. Even if we revert the system to do blind writes, each write still requires an extra read, which is where the performance loss is coming from. Figure 3.10 also shows a skewed workload that is generated using a Zipfian distribution. The Zipfian parameter is set to 0.2 (with 0 being pure Zipfian, and 1 being uniformly random). The skew has almost no effect on the write performance.

Append mode Our append mode writes increase the performance of `put` by several orders of magnitude. We ran two sets of experiments in MiniCrypt `APPEND` mode: modified YCSB 100% write and 50% read/50% write. Under `APPEND` mode assumptions, all writes are actually inserts where inserted keys are roughly increasing. Each experiment is run for 120 seconds (except for the long 100% write).

Write-only. We start with an empty database for both the encrypted baseline and MiniCrypt. Figure 3.11 compares the baseline client with MiniCrypt in `APPEND` mode. Compared to Figure 3.10, MiniCrypt is able to keep up with the baseline client's put speed much better because `put` in `APPEND` mode does not have an extra read and does not use *update-if*. The difference between MiniCrypt's and baseline's throughput is due to the overhead of the merge process. This overhead is not visible when the number of clients is small, but it appears when the number of clients increases. The merge process has to read back inserted keys, as well as re-insert them (though in a compressed format). This interferes with regular inserts because of both disk reads and extra insertion costs. MiniCrypt settles to about 40% of maximum write throughput achieved by baseline client.

Figure 3.12 shows the performance of MiniCrypt for a long run (approximately 10 minutes). We scale up the number of clients to 72, which corresponds to the right most data point in Figure 3.11. This graph plots cumulative number of keys against time. The baseline client line shows cumulative number of keys inserted during the 10 minute run. MiniCrypt has three different lines: "insert", "merge", and "delete". Insert indicates the cumulative number of keys inserted during the run; "merge" is the total number of keys merged from the inserted keys; "delete" is the total number of keys deleted. This graph shows that the merge process is able to keep up with key insertions, albeit at a lower insertion rate than that of the baseline client.

Read/write mix. The read/write mix workload is aimed to emulate one of YCSB's "read-most-recent" workloads, which is a common case when a workload inserts new data. All of the runs were executed on a pre-loaded 70 GB database. We adjust an "interval" parameter that indicates the range of the keys read. For example, an interval of 5 GB will allow the clients to read a uniformly random key from the most recently inserted 5 GB worth of data. Both the baseline client and MiniCrypt were warmed up for 5 minutes before each run. Each experiment runs for 120 seconds.

Figure 3.13 shows MiniCrypt's performance for a 50% reads and 50% writes workload. Since writes are faster than reads, the baseline's performance settles to a point that is higher than the baseline's performance in Figure 3.9 (a). The performance of MiniCrypt `APPEND` falls off as the size of the query interval increases because the merge process also needs to read recently inserted values. If more values are read into memory in the normal benchmark, the two processes will interfere with each other.

We also benchmarked the latency of the append MiniCrypt client as compared to the baseline client (again using single-threaded clients) on 5 GB of data. The baseline achieves a latency of 1.103 ms per read query, and 0.718 ms per write query. MiniCrypt achieves a latency of 1.743 ms per read query, and 0.781 ms per write query. The writes are very fast for both clients because both execute appends, while MiniCrypt's read is slower because it may have to do more work if an initial attempt to read misses.

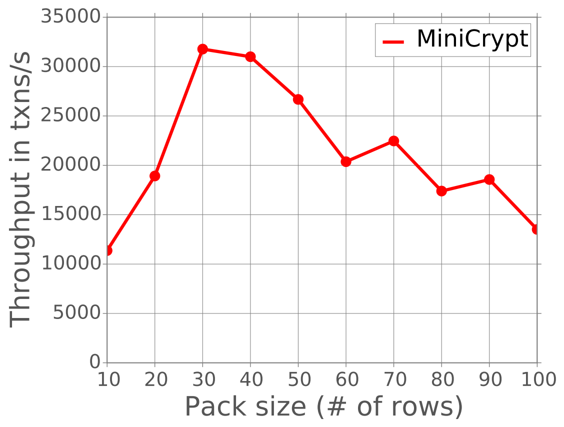


Figure 3.14: Pack size versus maximum throughput.

3.7.3 Determining the pack size

Writing an equation for the optimal pack size is not feasible because there are too many factors that affect this choice. Instead, MiniCrypt provides a tool to empirically determine a good pack size. This tool takes in a representative dataset and workload and can generate a graph of throughput plotted against various pack sizes. MiniCrypt then chooses the pack size that provides the highest throughput. Figure 3.14 shows running YCSB 100% uniform read workload for 50 GB of Conviva data. In our experiments, we noticed consistently that the optimal pack size was the following: the smallest pack size for which the data fits in memory, namely, $\mathsf{argmin}_n\{\mathsf{compratio}(n) \cdot \text{data size} < \text{memory size}\}$, where $\mathsf{compratio}(n)$ is the compression ratio obtained when compressing a pack of size n.

We recommend using MiniCrypt when all or most of the data fits in memory when compressed by MiniCrypt (i.e., fits in memory on each machine), but would not fit in memory without MiniCrypt. We show in the previous sections that there is a significant data size range when this is the case. If a significant fraction of the data does not fit in memory, we do not recommend using MiniCrypt.

3.7.4 Network bandwidth

MiniCrypt's network bandwidth overhead can be determined by (# of rows in each pack / pack compression ratio). In our experiments, network bandwidth did not become a bottleneck. We expect MiniCrypt to be used in a setting where the network is not the bottleneck.

3.8 Related work

We now discuss other related work in addition to the strawman designs described in Section 3.1.4.

Key-Value Stores. Some key-value stores (*e.g.*, Cassandra [138] and MongoDB [163]) compress and then encrypt the data at rest (in permanent storage). However, the decryption key is available to the server so that the server can decrypt and decompress the data when a client requests a key. This strategy does not protect against a server compromise (e.g., hacker, administrator of the server) because the attacker can get access to the decrypted data by compromising the server-side key. On the other hand, if a client inserts data encrypted with a key unavailable to the server, the compression mechanisms in these systems become ineffective due to the pseudorandom properties of the encryption. In comparison, MiniCrypt provides a significant compression ratio even in this case.

A recent system, Succinct [6] supports compression for a key-value store, while enabling rich search capabilities. However, Succinct does not support encryption. Adding encryption directly on top of Succinct would cause significant data access pattern leakage.

Encrypted databases. A number of recent proposals in databases support queries on encrypted data [181, 15, 222]. However, encryption introduces a significant storage overhead compared to the unencrypted data (e.g., 5 times larger for [181]). These systems do not support compression while executing queries on encrypted data. As presented in the MiniCrypt evaluation, querying on data that does not fit in memory will cause a significant performance hit.

Compressed databases. Compression is a common technique explore in databases [3, 70, 139, 6, 27, 114]. We discuss some simple strawman designs that utilize these techniques in Section 3.1.4. MiniCrypt is a generalized system that does not rely on a particular compression algorithm – users can choose any algorithm that fits their application requirements.

MiniCrypt differs from these two types of databases in two main ways. First, it focuses on NoSQL stores and does not support the more generic SQL operations. Second, MiniCrypt achieves both data confidentiality through encryption and a significant compression ratio.

File systems. There is previous work on designing encrypted file systems [30, 99, 128, 153] to protect data confidentiality from an untrusted server. One can compress a file before encrypting it. However, as discussed, compressing a single key-value pair alone does not provide good performance.

3.9 Conclusion

This chapter presented MiniCrypt, the first big data key-value store that reconciles encryption and compression. MiniCrypt makes an empirical observation about data compression trends and provides a set of distributed systems techniques for retrieving, updating, merging and splitting encrypted packs while preserving consistency and performance. Our evaluation shows that MiniCrypt can increase the server's throughput by up to two orders of magnitude. Such an encrypted key-value store is an important first step in providing secure computation in the cloud, as the users can use it as a storage service for their sensitive data.

Chapter 4

Opaque

SQL is widely used to analyze vast amounts of sensitive data such as user information (emails, social interactions, shopping history), medical data, and financial data because it not only offers database analytics but also supports machine learning [157, 108] and graph processing [107]. It is also useful for multi-party use cases such as money laundering and fraud detection across multiple financial institutions where customer transaction patterns must be traced. However, this cloud-based data analysis is currently done on plaintext data today. Even if the data storage service can be secured at rest using encryption, this data must be decrypted in memory for processing, which means that the cloud again sees the sensitive plaintext data.

Recent innovation in trusted hardware enclaves (such as Intel SGX [154] and AMD Memory Encryption [129]) promise support for arbitrary computation [22, 189] at processor speeds while protecting the data. Unfortunately, enclaves still suffer from an important attack vector: *access pattern leakage* [214, 171]. Such leakage occurs at the *memory level* and the *network level*. Memory-level access pattern leakage happens when a compromised OS is able to infer information about the encrypted data by monitoring an application's page accesses. Previous work [214] has shown that an attacker can extract hundreds of kilobytes of data from confidential documents in a spellcheck application, as well as discernible outlines of JPEG images from an image processing application running inside the enclave. Network-level access pattern leakage occurs in the distributed setting because tasks (e.g., sorting or hash-partitioning) can produce network traffic that reveals information about the encrypted data (e.g., key skew), even if the messages sent over the network are encrypted. For example, Ohrimenko et al [171] showed that an attacker who observes the metadata of network messages, such as source and destination (but not their content), in a MapReduce computation can identify the age group, marital status, and place of birth for some rows in a census database. Therefore, to truly secure the data, the computation should be *oblivious*: i.e., it should not leak any access patterns.

This chapter presents Opaque[1], an oblivious and distributed data analytics platform. Utilizing Intel SGX hardware enclaves, Opaque is able to provide strong security guarantees

[1]The name "Opaque" stands for Oblivious Platform for Analytic QUEries, as well as opacity, because

including computation integrity and computation obliviousness. The rest of this chapter focuses on a system design where there is a single client. Nevertheless, such design also works for the multi-party scenario because the parties can pool their data together on the cloud, and Opaque can execute SQL queries over this combined data.

The main challenge that Opaque faced is the question of how to *efficiently* provide protection against data access pattern leakage. To address this challenge, we propose a two-part solution. First, we introduce a set of new distributed relational operators that protect against both memory and network access pattern leakage at the same time. These include operators for joins and group-by aggregates. The contribution of these relational operators is to achieve obliviousness in a *distributed and parallel* setting. These operators also come with computation integrity guarantees, called *self-verifying computation*, preventing an attacker from affecting the computation result.

Second, we provide novel query planning techniques, both rule-based and cost-based, to further improve the performance of oblivious computation. Oblivious SQL operators in Opaque consist of fine-grained oblivious computation blocks called Opaque operators. We observe that by taking a global view across these Opaque operators and applying Opaque-specific rules, some operators can be combined or removed while preserving security. We also develop a cost model for oblivious operators that lets us evaluate the cost of a physical plan. This model introduces security as a new dimension to query optimization. We show that it is possible to achieve significant performance gains by using join reordering to minimize the number of oblivious operators. One key aspect used by our cost model is that *not all* tables in a database are sensitive: some contain public information. Hence, we can query such tables using non-oblivious operators to improve performance. Opaque allows database administrators to specify which tables are sensitive. However, sensitive tables can be related with seemingly insensitive tables. To protect the sensitive tables in this case, Opaque leverages a technique in the database literature called inference detection [113, 73] to propagate sensitivity through tables based on their schema information. Additionally, Opaque propagates operator sensitivity as well for all operators that touch sensitive tables.

4.1 Background

4.1.1 Access pattern leakage attacks

To understand access pattern leakage concretely, consider an example query in the medical setting:

```
SELECT COUNT(*) FROM patient WHERE age > 30 GROUP BY disease
```

The "group by" operation commonly uses hash bucketing: each machine iterates through its records and assigns each record to a bucket. The records are then shuffled over the network

they system is able to hide sensitive information.

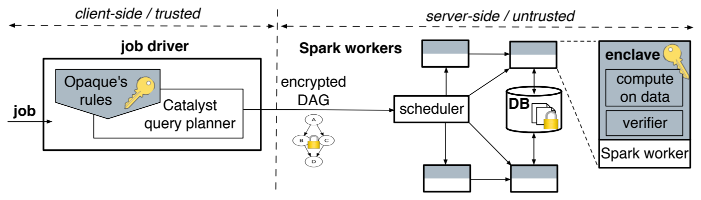


Figure 4.1: Opaque's architecture overview.

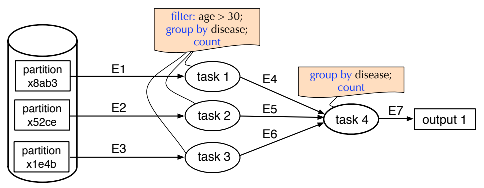


Figure 4.2: Example task DAG.

so that records within the same bucket are sent to the same machine. For simplicity, assume each bucket is assigned to a separate machine. By watching network packets, the attacker sees the number of items sent to each machine. Combined with public knowledge about disease likelihood, the attacker infers each bucket's disease type.

Moreover, the attacker can learn the disease type for a *specific database record*, as follows. By observing page access patterns, the attacker can track a specific record's bucket assignment. If the bucket's disease type is known, then the record's disease type is also known. A combination of page-based access patterns and network-level access patterns thus gives attackers a powerful tool to gain information about encrypted data.

4.1.2 Spark background

We implemented Opaque on top of Spark SQL [216, 16], a popular cluster computing framework, and we use Spark terminology in our design for concreteness. We emphasize that the design of Opaque is not tied to Spark or Spark SQL: the oblivious operators and query planning techniques are applicable to other relational frameworks.

The design of Spark SQL [216, 16] is built around two components: master and workers. The user interacts with the master which is often running with the workers in the cloud. When a user issues a query to Spark SQL, the command is sent to the master which constructs and optimizes a physical query plan in the form of a DAG (directed acyclic graph) whose nodes are tasks and whose edges indicate data flow. The conversion of the SQL query into a physical query plan is mediated by the Catalyst query optimizer.

4.2 Overview

4.2.1 Threat model and assumptions

In this section, we first describe the definitions and requirements of an *ideal enclave* and a real world *enclave implementation*. We then discuss Opaque's threat model. Opaque's design assumes and relies on the existence of an ideal enclave to fully provide obliviousness, but we also explain the security implications of using a real world enclave implementation.

4.2.1.1 Ideal enclave vs. enclave implementation

Ideal enclave In an ideal enclave environment, the processor is trusted and what happens inside the processor should be isolated from an attacker. This means that the attacker should not be able to observe control flow information from uncleared registers, etc.

We define two regions of memory: *internal* and *external*. The properties of these memory regions are:

1. Internal memory: this memory region should be completely isolated from the attacker and the attacker cannot observe anything that happens in this region. This includes which memory locations are accessed, how many memory accesses are executed, the order of the accesses, the timing of each memory access, etc. An attacker should not be able to use root-level software attacks to learn information about these statistics.

2. Secure external memory: all data in this region should be encrypted and authenticated so that the attacker does not see the data content. However, the attacker could still observe the memory access patterns.

3. Insecure external memory: the attacker can see all accessed data in plaintext.

This internal memory region should be large enough to contain all of the enclave code binary plus the minimum amount of working space needed by the code. Once the code and data pages are loaded in the internal memory region, the attacker cannot observe anything that happens in this region.

Enclave implementation Real world enclave implementations do not have an internal memory region. Using Intel SGX as an example, previous research [214] has shown that a malicious operating system could utilize page faults to observe page-level memory accesses. The cache is located within the package, but it does not provide any isolation. The enclave memory, or the enclave page cache (EPC), can be considered as a secure external memory because it is encrypted but still allows an attacker to observe memory accesses.

Nevertheless, recent work, such as Sanctum [61], GhostRider [143], and T-SGX [191], propose enclave designs that protect against access patterns to the EPC. For example, T-SGX uses Intel TSX to intercept page faults and protect against the controlled leakage attack [214]. Therefore, Opaque can utilize T-SGX to create a small pool of oblivious internal memory that is safe from the controlled leakage attack. The rest of Opaque can protect access to the untrusted memory and network using our own oblivious algorithms.

Note that Opaque inherits the prior work's security properties. T-SGX also leaks up to 10 page faults per transaction area because TSX cannot distinguish between page faults from benign OS interrupts. T-SGX can be adjusted to leak fewer than 10 page faults per transaction area, but there is a trade-off between security guarantees and functionality since a lower threshold means more opportunities for aborting from false positives. If Opaque utilizes T-SGX, Opaque will also inherit its leakage based on how T-SGX is configured.

Hardware mechanisms can also be used to create internal memory. For example, hardware-enforced page pinning (a malicious OS cannot swap out pages) or superpages can be used to provide isolation for page fault attacks.

For the rest of the chapter (other than the implementation and evaluation sections), we assume that Opaque has access to ideal enclaves.

4.2.1.2 Threat model

We assume a powerful adversary who controls the cloud provider's software stack. The attacker may gain root access to the operating system, modify data or communications that are not inside a secure enclave, and observe the content and order of memory accesses by an enclave to the external memory. The adversary can observe and modify the network traffic between different nodes in the cloud as well as between the cloud and the client. The adversary may also perform a *rollback attack* that restores sealed data to a previous state.

We assume that the adversary cannot compromise the trusted hardware, relevant enclave keys, or client software. In particular, the attacker cannot issue queries or change server-side data through the client. We also assume that the adversary does not launch hardware based attacks such as memory bus snooping.

4.2.1.3 Out of scope attacks

We assume that the attacker's power is limited to what is stated in the previous section. Therefore, the attacker does not use hardware attacks, software timing attacks, power analysis, or any other side channel attack not included in the threat model. Opaque also does not prevent denial-of-service attacks. A cloud provider may destroy all customer data or deny or delay access to the service, but this is not in the provider's interest because customers have the option to choose a different provider. Side-channel attacks based on power analysis or timing attacks (including those that measure the time spent in the enclave or the time when queries arrive) are also out of scope for us.

4.2.2 Opaque's architecture

Figure 4.1 shows Opaque's architecture. Opaque does not change the layout of Spark and Spark SQL, except for one aspect. Opaque moves the query planner to the client side because a malicious cloud controlling the query planner can result in incorrect job execution. However, we keep the scheduler on the server side, where it runs in the untrusted domain. We augment Opaque with a computation verification mechanism (Section 4.3.2) to prevent an attacker from corrupting the computation results.

The Catalyst planner resides in the job driver and is extended with Opaque optimization rules. Given a job, the job driver outputs a task DAG and a unique job identifier JID for this job. For example, the query from Section 4.1.1 translates to the DAG shown in Figure 4.2.

The job driver annotates each edge with an ID, e.g., E1, and each node with a task ID, e.g., task 4. The input data is split in partitions, each having its own identifier.

Oblivious memory parameter. Opaque can use prior work to establish an oblivious pool of memory under certain threat model assumptions. Since the size of the oblivious pool depends on the architecture used, we parameterize Opaque with a variable specifying the size of the oblivious memory. This parameter can range from as small as the registers (plus Opaque's enclave code size) to as large as the entire EPC [61, 191] or main memory. Bigger oblivious memory allows faster oblivious execution in Opaque.

4.2.3 Security guarantees

4.2.3.1 Encryption mode

In encryption mode, Opaque provides data encryption and authentication guarantees. Opaque's self-verifying integrity protocol (Section 4.3.2) guarantees that, if the client verifies the received result of the computation successfully, then the result is correct, i.e., not affected by a malicious attacker. The proof of security for the self-verifying integrity protocol is rather straightforward, and similar to the proof for VC3 [189].

4.2.3.2 Oblivious modes

Opaque has two oblivious modes: *oblivious mode* and *oblivious pad mode*. Oblivious mode hides the data access patterns of each SQL operator, but reveals the input/output sizes of each operator. Oblivious pad mode, explained in Section 4.4.3, hides even this information by pushing up all filters and padding the final output to a public upper bound, in exchange for more performance overhead.

In this section, we will focus oblivious mode's security properties; we expand on the two modes. Opaque does not hide the computation/queries run at the server, the schema, or other public information like the number of rows in each table. As stated before, we will assume that Opaque has access to *ideal enclaves*.

In oblivious mode, Opaque provides oblivious SQL operator execution with respect to memory and network accesses under the ideal enclave assumption. We can load both the code and data pages into internal memory (defined in Section 4.2.1.1). Anything located in the internal memory is isolated from the attacker (i.e., the attacker cannot passively or actively observe data and data accesses).

The standard way to formalize that a system hides access patterns is to construct a simulator that takes as input a query plan and data sizes but not the data content, yet is able to produce the *same* trace of memory and network accesses as the system itself. Intuitively, since the simulator's input does not include the data content, the data accesses reveal nothing about the content. The simulator's input provides an upper bound on the system's leakage.

To specify Opaque's leakage, we first define the following (informal) notation. Let $\mathcal{D}$ be a dataset and Q a query. Let $\mathsf{Size}(\mathcal{D})$ be the sizing information of $\mathcal{D}$, which includes the size of each table, row, column, attribute, the number of rows, the number of columns, but does not include the value of each attribute. Let S be the schema information, which includes table and column names in $\mathcal{D}$, as well as the sensitive tables' names (Opaque can additionally hide table and column names via encryption). The sensitive tables include those marked by the administrator, as well as those marked by Opaque after sensitivity propagation (Section 4.5.3). Let $\mathsf{IOSize}(\mathcal{D}, Q)$ be the input/output size of each SQL operator in Q when run on $\mathcal{D}$. We define $P = \mathsf{OpaquePlanner}(\mathcal{D}, Q)$ as the physical plan generated by Opaque. We define Trace as the trace of memory accesses and network traffic patterns (the source, destination, execution stage, and size of each message) for sensitive operators.

Theorem 1. *For all $\mathcal{D}$, S, where $\mathcal{D}$ is a dataset and S is its schema, and for any query Q, there exists a probabilistic polynomial-time simulator* Sim *such that, for $P =$* $\mathsf{OpaquePlanner}(\mathcal{D}, Q)$*,*

$$\mathsf{Sim}(\mathsf{Size}(\mathcal{D}), S, \mathsf{IOSize}(\mathcal{D}, Q), P) = \mathsf{Trace}(\mathcal{D}, P).$$

The existence of Sim demonstrates that access patterns of the execution are oblivious, and that the attacker does not learn the data content $\mathcal{D}$ beyond sizing information and the query plan. The fact that the planner chose a certain query plan over other possible plans for the same query might leak some information about the statistics on the data maintained by the planner. Nevertheless, the planner maintains only a small amount of such statistics that contain much less information than the actual data content. Further, the attacker does not see these statistics directly and does not have the power to change data or queries and observe changes to the query plan.

Oblivious pad mode's security guarantees are similar to the above, except that the simulator no longer takes as input $\mathsf{IOSize}(\mathcal{D}, Q)$, but instead only a public upper bound on the size of a query's final output.

Note that Opaque protects most constants in a query using semantic security: for example it hides the constant in "age $\geq$ 30", but not in "LIMIT 30".

Coupling oblivious accesses with the fact that the content of every write to memory and every network message is freshly encrypted with semantic security enables Opaque to provide a strong degree of data confidentiality. In particular, Opaque protects against the memory and network access patterns attacks presented in [214] and [171].

4.3 Opaque's encryption mode

In this section, we describe Opaque's encryption mode, which provides data encryption, authentication and computation integrity.

4.3.1 Data encryption and authentication

Similar to previous designs [22, 189], Opaque uses remote attestation to ensure that the correct code has been loaded into enclaves. A secure communication channel is then established and used to agree upon a shared secret key k between the client and the enclaves.

All data in an enclave is automatically encrypted by the enclave hardware using the processor key of that enclave. Before communicating with another enclave, an enclave always encrypts its data with AuthEnc using the shared secret key k. AuthEnc encrypts data with AES in GCM mode, a high-speed mode that provides authenticated encryption. In addition to encryption, this mode also produces a 128-bit MAC to be used for checking integrity.

4.3.2 Self-verifying computation

Ensuring computation integrity is necessary because a malicious OS could drop messages, alter data or computation. We call our integrity checking strategy *self-verifying computation* because the computation verifies itself as it proceeds. The mere fact that the computation finished without aborting means that it was not tampered with.

Let us first discuss how to check that the input data was not corrupted. As in VC3 [189], the identifier of a partition of input data is its MAC. The MAC acts as a *self-certifying* identifier because an attacker cannot produce a different partition content for a given ID. Finally, the job driver computes $C \leftarrow \textsc{AuthEnc}_k(\text{JID}, \text{DAG}, P_1, \dots, P_p)$, where $P_1, \dots, P_p$ indicates the identifiers of the partitions to be taken as input. Every worker node receives C. Opaque's verifier running in the enclave decrypts and checks the authenticity of the DAG in C.

Then, to verify the integrity of the computation, each task needs to check that the computation up to it has proceeded correctly. First, if $E_1, \dots, E_t$ are edges incoming into task T in the DAG, the verifier checks that it has received authentic input on each edge from the correct previous task and that it has received input for *all* edges. To ensure this invariant, each node producing an output o for an edge E encrypts this output using $\textsc{AuthEnc}_k(\text{JID}, E, o)$. The receiving node can check the authenticity of this data and that it has received data for *every* edge in the DAG. Second, the node will run the correct task T because the enclave code was set up using remote attestation and task T is integrity-verified in the DAG. Finally, each job ends with the job driver receiving the final result and checking its MAC. The last MAC serves as a proof of correct completion of this task.

This protocol improves over VC3 [189], which requires an extra stage where all workers send their inputs and outputs to a master which checks that they all received complete and correct inputs. Opaque avoids the cost of this extra stage and performs the verification during the computation, resulting in negligible cost.

Rollback attacks. Spark's RDDs combined with our verification method implicitly defend against rollback attacks, because the input to the workers is matched against the expected

MACs from the client and afterwards, the computation proceeds deterministically. The computation result is the same even with rollbacks.

4.3.3 Fault tolerance

In Spark, if the scheduler notices that some machine is slow or unresponsive, it reassigns that task to another machine. Opaque's architecture facilitates this process because the encrypted DAG is *independent* from the workers' physical machines. As a result, the scheduler can live entirely in the untrusted domain, and does not affect Opaque's security if compromised.

4.4 Oblivious execution

In this section, we describe Opaque's oblivious execution design. We first present two oblivious building blocks, followed by Opaque's oblivious SQL operator designs.

4.4.1 Oblivious building blocks

Oblivious sorting is central to the design of oblivious SQL operators. Opaque adapts existing oblivious sorting algorithms for both local and distributed sorting, which we now explain.

4.4.1.1 Intra-machine oblivious sorting

Sorting networks [58] are abstract networks that consist of a set of *comparators* that compare and swap two elements. Elements travel over wires from the input to comparators, where they are sorted and output again over wires. Sorting networks are able to sort any sequence of elements using a fixed set of comparisons.

Denote by OM, the oblivious memory available for query processing, as discussed in §4.2.2. In the worst case, this is only a part of the registers. If the total size of the data to be sorted on a single machine fits inside the OM, then it is possible to load everything into the OM, sort using quicksort, then re-encrypt and write out the result. If the data cannot fit inside the OM, Opaque will first partition the data into blocks. Each block is moved into the OM and sorted using quicksort. We then run a sorting network called *bitonic sort* over the blocks, treating each one as an abstract element in the network. Each comparator operation loads two blocks into the enclave, decrypts, merges, and re-encrypts the blocks. The merge operation only requires a single scan over the blocks.

4.4.1.2 Inter-machine oblivious sorting

A natural way to adapt the bitonic sorting network in the distributed setting is to treat each machine as an abstract element in the sorting network. We can sort within each machine

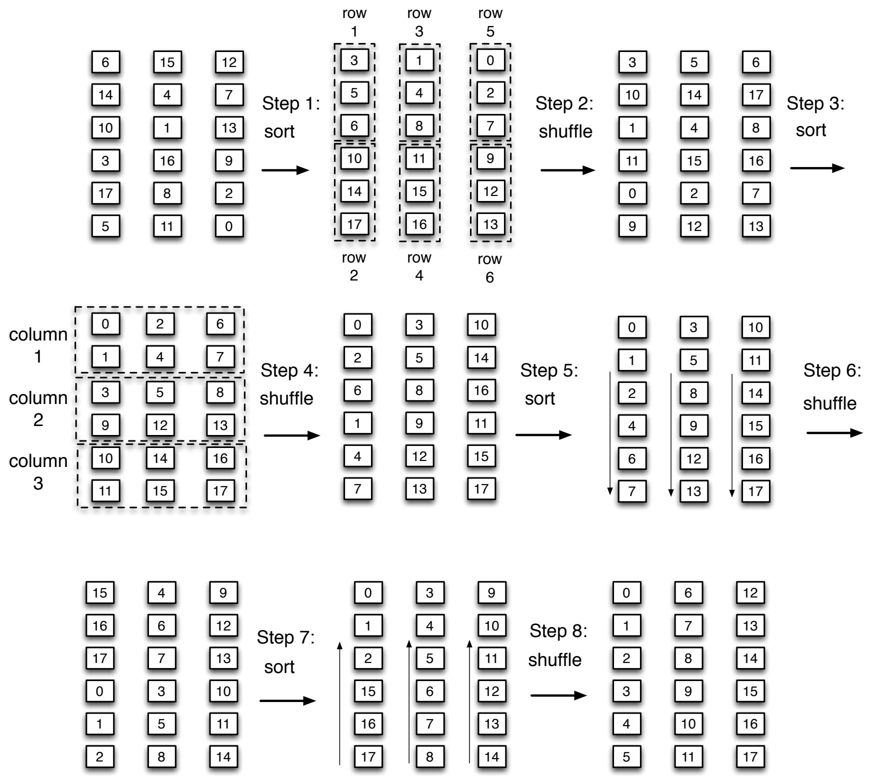


Figure 4.3: Column sort, used in the distributed setting. Each column represents a single partition, and we assume that each machine only has one partition. The algorithm has eight steps. Steps 1, 3, 5, 7 are sorts, and the rest are shuffle operations.

separately, then run the bitonic sorting network over the machines. However, each level of comparators now corresponds to a network shuffling of data. Given n machines, the sorting network will incur $O(\log^2 n)$ number of shuffles, which is high.

Instead, Opaque uses column sort [142], which sorts the data using a *fixed* number of shuffles (5 in our experiments) by exploiting the fact that a single machine can hold many items. Column sort works as follows: given a sequence of B input items, we split these items into s partitions, where each partition has exactly r items (with padding if necessary). Without loss of generality, we assume that each machine handles one partition. We treat each partition as a column in column sort. The sorting algorithm has 8 steps: the odd-numbered steps are per-column sorts (implemented as intra-machine oblivious sorting), and the even-numbered steps shuffle the data deterministically. Figure 4.3 gives a visual example of how column sort works. The sorting algorithm has the restriction that $r \geq 2(s-1)^2$, which applies well to our setting because there are many records in a single partition/column.

An important property of column sort is that, as an oblivious operator, it preserves the *balance* of the partitions. This means that after a sort, a partition will have exactly the same number of items as before. Partition balance is required to avoid leaking any information regarding the underlying data's distribution. However, balanced partitioning is incompatible with co-locating all records of a given group. Instead, records with identical grouping attributes may be split across partitions. Operators that consume the output of column sort must therefore be able to transfer information between adjacent partitions *obliviously* and *efficiently*. We address this challenge in our descriptions of the oblivious operators.

4.4.2 Oblivious operators

In this section, we show how to use the oblivious building blocks to construct oblivious relational algebra operators. The three operators we present are filter, group-by, and join. Opaque uses an existing oblivious filter operator [13], but provides new algorithms for the join and group-by operators, required by the distributed and parallel setting.

In what follows, we focus only on the salient parts of these algorithms. We also assume that during oblivious computation, every row is padded to the same size so that the sizing information will not leak anything about the computation.

4.4.2.1 Oblivious filter

An oblivious filter ensures that the attacker cannot track which encrypted input rows pass the filter. A naïve filter that streams data through the enclave to get rid of unwanted rows will leak which rows have been filtered out because the attacker can keep track of which input resulted in an output. Instead, the filter operator [13] first executes a projection by scanning and marking each row with a "0" (record should be kept) or a "1" (record should be filtered). It then obliviously sorts all rows with "0" before "1" and removes the "1" rows.

4.4.2.2 Oblivious Aggregate

Aggregation queries group items with equal *grouping attributes* and then aggregate them using an aggregation function. For example, for the query in Section 4.1.1, the grouping attribute is `disease` and the aggregation function is `count`.

A naïve aggregation implementation leaks information about group sizes (some groups may contain more records than others), as well as the actual mapping from a record to a group. For example, a reduce operation that sends all rows in the same group to a single machine reveals which and how many rows are in the group. Prior work [171] showed that an attacker can identify age group or place of birth from such protocols.

Opaque's oblivious aggregation starts with an oblivious sort on the grouping attributes. Once the sort is complete, all records that have the same grouping attributes are located next to each other. A single scan might seem sufficient to aggregate and output a value for each group, but this is incorrect. First, the number of groups per machine can leak the number of values in each group. A further challenge (mentioned in Section 4.4.1.2) is that a set of rows with the same grouping attributes might span multiple machines, leaking such information. We need to devise a parallel solution because a sequential scan is too slow.

We solve the above problems by designing a distributed group-by operator that reveals neither row-to-group mapping nor the size of each group. The logical unit for this algorithm is a partition, which is assumed to fit on one machine. The intuition for this algorithm is that we want to *simulate* a global sequential scan using per-partition parallel scans. If all records in a group are in one partition, the group will be aggregated immediately. Once the last record in that group has been consumed in the scan, the aggregation result is complete. If records in a group are split across partitions, we want to pass information across partitions efficiently and obliviously so that later partitions have the information they need to finish the aggregation.

High-cardinality aggregation. This aggregation algorithm should be used when the number of groups is large.

Stage 1 [sort]: Obliviously sort all records based on the grouping attributes.

Stages 2–4 are the *boundary processing* stages. These stages solve the problem of a single group being split across multiple machines after column sort. Figure 4.4 illustrates an example.

Stage 2 [per-partition scan 1]: Each worker scans its partition once to gather some statistics, which include the partition's first and last rows, as well as *partial aggregates* of the last group in this partition. The partial aggregate consists of "grouping attributes" and "aggregate values". We define "grouping attributes" to be the grouping columns' values, and "aggregate values" to be the aggregated values.

In Figure 4.4, each column represents one partition. Each worker calculates statistics including R_i, the partial aggregate. In partition 0, $R_0 = (C, 2)$ is the partial aggregate that corresponds to the last row in that partition, C.

Stage 3 [boundary processing]: All of the statistics from stage 2 are collected into a single partition. The worker assigned to this partition will scan all of the statistics and compute one global partial aggregate (GPA) per partition. If the collected partial aggregates are all unique, then each partition's GPA should be given to the next partition. If the partial aggregates from multiple partitions are the same, then they need to be aggregated together before being passed onto the next partition. The middle partitions will receive dummy elements instead. Finally, each partition passes its first row to the previous partition because this will identify whether a particular group extends into the next partition.

Figure 4.4's stage 3 shows an example of how the GPA is computed. The first partition always receives a dummy GPA since it is not preceded by any other partition. Partition $P1$ receives $(C, 2)$ from $P0$. With this information, $P1$ can correctly compute the aggregation result for group C, even though the records are split across $P0$ and $P1$.

Stage 4 [per-partition scan 2]: Each partition receives a GPA. This GPA is used to do aggregation. One record is output for every input record. The last record of a group outputs the group's aggregation results, while the other records output dummies.

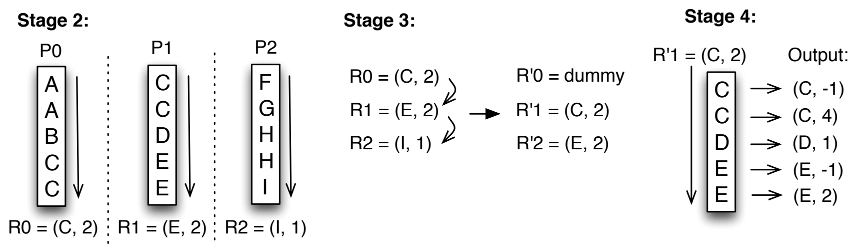


Figure 4.4: Stages 2 - 4 of oblivious aggregation

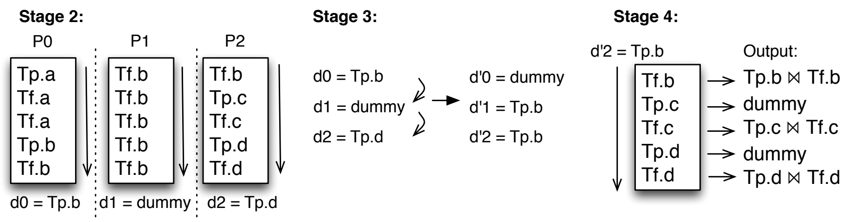


Figure 4.5: Stages 2–4 of oblivious join.

Figure 4.4's stage 4 shows how $P1$ can aggregate groups C, D and E using R_1'.
Stage 5 [sort and filter]: Execute a projection that maps dummy records to "1", and other records to "0" Obliviously sort on the projected column and filter out the dummies.

Low-cardinality aggregation. If the number of groups is small (e.g., age groups, states), Opaque provides an alternative algorithm that avoids the second oblivious sort.

4.4.2.3 Oblivious sort-merge join

Regular joins leak information about how many and which records are joined together on the same join attributes. For example, a regular primary-foreign key join may sort the two tables separately, maintain a pointer to each table, and merge the two tables together while advancing the pointers. The pointer locations reveal information about how many rows have the same join attributes and which rows are joined together.

We developed an oblivious equi-join algorithm based on the sort-merge join algorithm. Let T_p be the primary key table, and T_f be the foreign key table.
Stage 1 [union and sort]: We union T_p with T_f, then obliviously sort them together based on the join attributes. We break ties by ordering T_p records before T_f records.

As with oblivious aggregation, stages 2–4 are used to handle the case of a join group (e.g., a set of rows from T_p and T_f that are joined together) that is split across multiple machines. We use Figure 4.5 to illustrate these three stages.

Stage 2 [per-partition scan 1]: Each partition is scanned once and the last row from T_p in that partition, or a dummy (if there is *no* record from T_p on that machine) is returned. We call this the boundary record.

Figure 4.5 explains stage 2 with an example, where Tp.x indicates a record from the primary key table with join attribute x, and Tf.x indicates a record from the foreign key table with join attribute x. In partition $P0$, Tp.b is the last record of T_p in that partition, so the boundary record is set to Tp.b. $P1$ does not contain any row from T_p, so its boundary record is set to a dummy value.

Stage 3 [boundary processing]: In stage 3, we want to generate primary key table records to give back to each data partition so that all of the foreign key table records in each partition (even if the information spans across multiple machines) can be joined with the corresponding primary key record. We do so by first collecting all of the boundary records to one partition. This list is scanned once, and we output a new boundary record for every partition. Each output is set to the value of the most recently encountered *non-dummy* boundary.

For example, Figure 4.5's stage 3 shows that three boundary records are collected. Partition 0 will always get a dummy record. Record Tp.b is passed from partition 0 to partitions 1 and 2 because d_1 is a dummy. This ensures that any record from T_f with join attribute b (e.g., the first record of partition 2) will be joined correctly.

Stage 4 [per-partition scan 2]: Stage 4 is similar to a normal sort-merge join, where the worker linearly scans the tables and joins primary key records with the corresponding foreign key records. There are some variations to preserve obliviousness. First, the initial record in the primary key table should come from the boundary record received in stage 3 (except for the first partition). Second, during the single scan, we need to make sure that one record is output for every input record, outputting dummy records as necessary.

Figure 4.5's stage 4 shows how the algorithm works on partition 2. The boundary record's value is Tp.b, which is successfully joined with the first row of partition 2. Since P_2's second row is a new record from T_p, we change the boundary record to Tp.c, and a dummy is output.

Stage 5 [sort and filter]: Execute a projection that maps dummy records to "1", and other records to "0" Obliviously sort on the projected column and filter out the dummies.

4.4.3 Oblivious pad mode

Oblivious execution provides strong security guarantees and prevents access pattern leakage. However, it does not hide the output *size* of each relational operator (i.e., how many rows are in the output). This means that in a query with multiple relational operators, the size of each intermediate result is leaked. To solve this problem, Opaque provides a stronger variant of oblivious execution: oblivious with padding.

The idea is to never reduce the output size of a relational operator until the end of the query. This can be easily achieved by using "filter push up." For example, a query that has a join followed by an aggregation will skip stage 5 of the join. After the aggregation, all dummies will be filtered out in a single sort with filter. We also require the user to provide an upper bound on the final result size, and Opaque will pad the final result to this size. In this case, the query plan also no longer depends on data statistics, as we discuss in §Section 4.5.4.

Note that this mode is more inefficient because Opaque cannot take advantage of selectivity (e.g., of filters). Therefore, we recommend using padding on extremely sensitive datasets.

4.5 Query planning

Even with parallelizable oblivious algorithms, obliviousness is still expensive. We now describe Opaque's query planner, which reduces obliviousness overheads by introducing novel techniques that build on rule-based and cost-based optimization, as well as entity-relational modeling. We first formalize a cost model for our oblivious operators to allow a standard query planner to perform basic optimizations on oblivious plans. We then describe several new optimizations specific to Opaque, enabled by a decomposition of oblivious relational operators into lower-level Opaque operators. Finally, we describe a mixed sensitivity setting where a database administrator can designate tables as sensitive. Opaque applies a technique in databases known as second path analysis that uses foreign-key relationships in a data model to identify tables that are *not* sensitive, accounting for inference attacks. We also demonstrate that such sensitivity propagation occurs within a single query plan, allowing us to substantially speed up certain queries using join reordering.

4.5.1 Cost model

Cost estimation in Opaque differs from that of a traditional SQL database because sorting, the core database operation, is more costly in the oblivious setting than otherwise. Oblivious sorting has very different algorithmic behavior from conventional sorting algorithms because the sequence of comparisons can be constructed based only on the input *size* and not the input data. Therefore, our cost model must accurately model oblivious sorting, which is the dominant cost in our oblivious operators.

Similarly to a conventional sort, the cost of an oblivious sort depends on two factors: the number of input items and the padded record size. Even for datasets that fit in memory, cost modeling for an oblivious sort is similar to that of a traditional external sort because the latency penalty incurred by the enclave for accessing pages outside of the oblivious memory or EPC effectively adds a layer to the memory hierarchy. We therefore use a two-level sorting scheme for oblivious sort, described in Section 4.4.1.1, having a runtime complexity of $O(n \log^2 n)$.

We now formalize the cost of oblivious sort and use this to model oblivious join. The costs of other oblivious operators can be similarly modeled.

Let T be a relation, and r be a padded record. We denote $|T|$ to be the size of the relation T, and $|r|$ to be the size of a padded record. Let $|\text{OMem}|$ be the size of the oblivious memory, and K a constant scale factor representing the cost of executing a compare-and-swap on two records. We denote n to be the number of records per block, and B to be the required

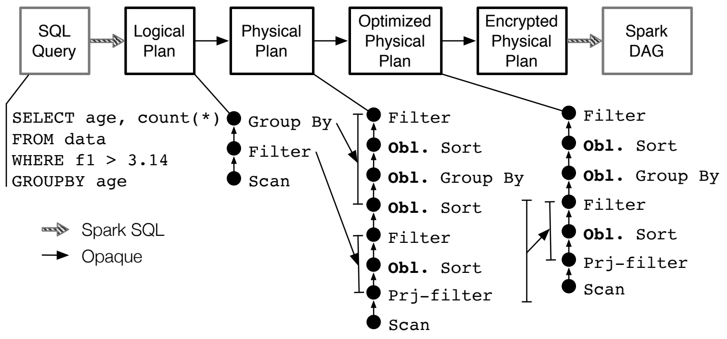


Figure 4.6: Catalyst oblivious query planning.

number of blocks. We can estimate n, B, and the resulting sort cost $C_{\text{o-sort}}$ and join cost $C_{\text{o-join}}$ as follows:

$$n = \frac{|\text{OMem}|}{2\,|r|}, \quad B = |T|\,/n, \quad C_{\text{o-join}} \approx 2 \cdot C_{\text{o-sort}}$$

$$C_{\text{o-sort}}(|T|\,, |r|) = \begin{cases} K\,|T| \log |T| & \text{if } |T| \cdot |r| \leq |\text{OMem}| \\ K\,[Bn \log n + nB \log B(1 + \log B)/2] \\ \qquad\qquad\qquad \text{otherwise} \end{cases}$$

The number of records n per block follows from the fact that two blocks must fit in oblivious memory at a time for the merge step. The expression for the sort cost follows from the two-level sorting scheme. If the input fits inside the oblivious memory, we bypass the sorting network and instead use quicksort within this memory, so the estimated cost is simply the cost of quicksort. Otherwise, we sort each block individually using quicksort, run a sorting network on the set of blocks and merge blocks pairwise. The sorting network performs $B \log B(1 + \log B)/4$ merges, each incurring a cost of $2n$ to merge two blocks. We experimentally verify this cost model in Section 4.7.4.

4.5.2 Oblivious query optimization

We now describe new optimization rules for a sequence of oblivious operators. Our rules operate on the lower-level operations within each oblivious operator, which we call Opaque operators.

4.5.2.1 Overview of the query planner

Before describing the Opaque operators, we provide an overview of the planning process, illustrated in Figure 4.6. Opaque leverages the Catalyst query planner to transform a *SQL query* into an operator graph encoding the *logical plan*. Opaque interposes in the planning process to mark all logical operators that process sensitive data as oblivious. Catalyst can apply standard relational optimizations to the logical plan such as filter pushdown and join reordering.

Catalyst then generates a *physical plan* where each logical operator is mapped to one or more physical operators representing the choice of execution strategy. For example, a logical non-oblivious join operator could be converted to a physical hash join or a broadcast join based on the input cardinalities. Oblivious operators are transformed into physical Opaque operators at this stage, allowing us to express rules specific to combinations of oblivious operators. Similar to Catalyst, generating these physical operators allows Opaque to select from multiple implementations of the same logical operator based on table statistics. For example, if column cardinality is available, Opaque may use it to decide which oblivious aggregation algorithm to use. Catalyst then applies our Opaque rules to the physical plan.

The physical plan is then converted into an *encrypted representation* to hide information such as column names, constants, etc. Finally, Catalyst transforms the encrypted physical plan into a *Spark DAG* containing a graph of RDDs and executes it on the cluster.

4.5.2.2 Opaque operators

The following is a sampling of the physical Opaque operators generated during planning:

- `SORT`(C): obliviously sort on columns C
- `FILTER`: drop rows if predicate not satisfied
- `PROJECT-f`: similar to `FILTER`, but projects filtered out rows to 1, the rest to 0; preserves input size
- `HC-AGG`: stages 2–4 of the aggregation algorithm
- `SORT-MERGE-JOIN`: steps 2–4 of the sort-merge join algorithm

4.5.2.3 Query optimization

In this section, we give an example of an Opaque-specific rule:

$$\texttt{SORT}(C_2, \texttt{FILTER}(\texttt{SORT}(C_1, \texttt{PROJECT-f}(C_1)))) = \texttt{FILTER}(\texttt{SORT}(C_1, C_2, \texttt{PROJECT-f}(C_1)))$$

Let us take a look at how this rule would work with a specific query. We use the example query from Section 4.1.1, which translates to the following physical plan:

```
LC-AGG(disease,
  SORT(disease, FILTER(dummy,
    SORT(dummy_col, PROJECT-f(age, patient)))))
```

The filter will first do a projection based on the column `age`. To preserve obliviousness, the projected column is sorted and a real filter is applied. Since a sort-based aggregation comes after the filter, we need to do another sort on `disease`.

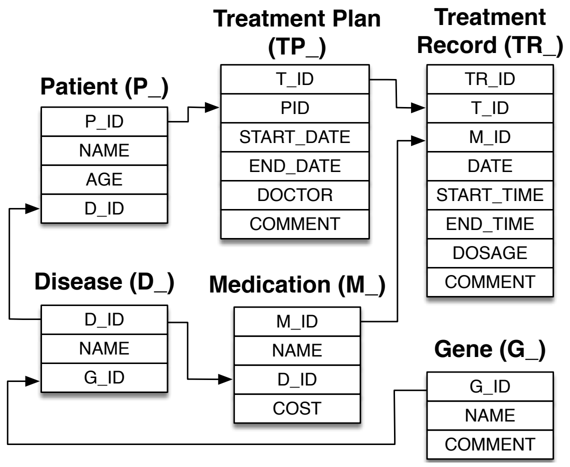


Figure 4.7: Example medical schema.

We make the observation that the second sort can be combined with the first sort into *one* oblivious sort on multiple columns. Since `PROJECT-f` always projects a column that is binary (i.e., the column contains only "0"s and "1"s), we can first sort on the binary column, then on the second sort's columns (in this example, the disease column). Therefore, the previous plan becomes:

$$\begin{aligned}&\texttt{LC-AGG}(\mathsf{disease}, \texttt{FILTER}(\mathsf{dummy\_col},\\ &\quad \texttt{SORT}(\{\mathsf{dummy\_col}, \mathsf{disease}\},\\ &\qquad \texttt{PROJECT-f}(\mathsf{age}, \texttt{patient}))))\end{aligned}$$

This optimization is rule-based instead of cost-based. Furthermore, our rule is different from what a regular SQL optimizer applies because it pushes *up* the filter, while a SQL optimizer pushes *down* the filter. Filter push-down is unsafe because it does not provide obliviousness guarantees. Applying the filter before sorting will leak which records are filtered out.

4.5.3 Mixed sensitivity

Many applications operate on a database where not all of the tables are sensitive. For example, a hospital may treat patient information as sensitive while information about drugs, diseases, and various hospital services may be public knowledge (see Figure 4.7).

4.5.3.1 Sensitivity propagation

Propagation on tables. In a mixed sensitivity environment, tables that are not marked as sensitive could still be sensitive if they reveal information about other sensitive tables. Consider the example schema in Figure 4.7. The Disease, Medication, and Gene tables are public datasets or have publicly known distributions in this example and therefore are not sensitive. Meanwhile the Patient table would likely be marked as sensitive. But what about Treatment Plan and Treatment Record? It turns out these tables are also sensitive because they *implicitly* embed patient information. Each treatment record belongs to a single patient, and each patient's plan may contain multiple treatment records. If an attacker has some prior knowledge, for example regarding what type of medication a patient uses, then observing only the Treatment Record table may allow the attacker to use an *inference attack* to gain further information about that patient such as their treatment frequency and other medication they may be taking.

To prevent such attacks, we use a technique from database literature called second path analysis [113]. The intuition for the inference attack is that information propagates along primary-foreign key relations: since each treatment record belongs to one treatment plan and one patient, the treatment record contains implicit information about patients. The disease table is connected to the patient table as well, except it has a primary key pointing *into* patient. This means that the disease table does not implicitly embed patient information.

Second path analysis accomplishes table sensitivity propagation by first directly marking user-specified tables as sensitive. After this is done, it recursively marks all tables that are reachable from every sensitive table via primary-foreign key relationships as sensitive as well. As in Figure 4.7, such relationships are marked in an entity-relationship diagram using an arrow from the primary key table to the foreign key table.

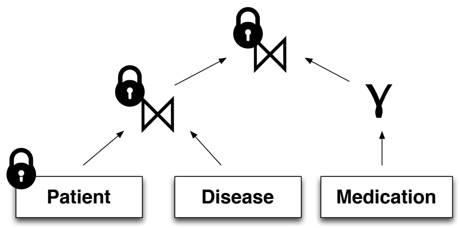
This approach has been generalized to associations other than explicit foreign keys and implemented in automated tools [73]. We do not reimplement such analysis in Opaque, instead referring to the existing work.

Propagation on operators. Another form of sensitivity propagation occurs when an operator (e.g., join) involves a sensitive and a non-sensitive table. In this case, we must run the entire operator obliviously. Additionally, for every leaf table that is marked sensitive in a query plan, sensitivity propagates on the path from the leaf to the root, and Opaque runs all the operators on this path obliviously.

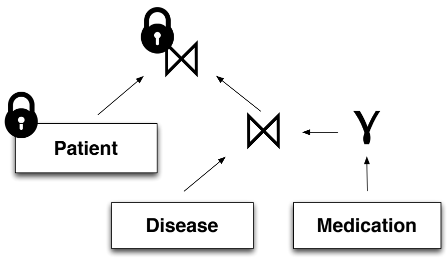
4.5.3.2 Join reordering

Queries involving both sensitive and non-sensitive tables may contain a mix of oblivious and non-oblivious operators. Due to sensitivity propagation on operators, some logical plans may involve more oblivious operators than others. For example, a three-way join query where one table is sensitive may involve two oblivious joins if the sensitive table is joined first, or only one oblivious join if it is joined last (e.g., the non-sensitive tables are pushed down in the join order).

Join reordering in a traditional SQL optimizer centers on performing the most selective joins first, reducing the number of tuples that need to be processed. The statistics regarding selectivity can be collected by running oblivious Opaque queries. In Opaque, mixed sensitivity introduces another dimension to query optimization because of operator-level sensitivity



(a) SQL join order



(b) Opaque join order

Figure 4.8: Join reordering in mixed sensitivity mode.

propagation and the fact that oblivious operators are much more costly than their non-oblivious counterparts. Therefore, a join ordering that minimizes the number of oblivious operators may in some cases be more efficient than one that only optimizes based on selectivity.

Consider the following query to find the least costly medication for each patient, using the schema in Figure 4.7:

```
SELECT    p_name, d_name, med_cost
FROM      patient, disease,
          (SELECT d_id, min(cost) AS med_cost
            FROM medication
            GROUP BY d_id) AS med
WHERE     disease.d_id = patient.d_id
          AND disease.d_id = med.d_id
```

We assume that the Patient table is the smallest, followed by Disease, then Medication ($|P| < |D| < |M|$), as might occur when considering only currently hospitalized patients and assuming there are multiple medications for each disease. The aggregation query reduces the cardinality of Medication to that of Disease and ensures a one-to-one relationship between the two tables.

Figure 4.8 shows two join orders for this query. A traditional SQL optimizer will execute the most selective join first, joining Patient with Disease, then with Medication. The optimal ordering for Opaque will instead delay joining Patient to reduce the number of oblivious joins. To see this, we now analyze the costs for both join orders.

Let C_{SQL} be the cost of this query using the SQL join order, C_{Opaque} the cost using the Opaque join order, and R the padded row size for all input tables. Note that the size of the Medication aggregate table is $|D|$.

$$\begin{aligned} C_{\text{SQL}} &= 2C_{\text{o-join}}(|P| + |D|, R) \\ C_{\text{Opaque}} &= C_{\text{join}}(2|D|, R) + C_{\text{o-join}}(|P| + |D|, R) \end{aligned}$$

Assuming $C_{\text{join}} \ll C_{\text{o-join}}$,

$$\frac{C_{\text{SQL}}}{C_{\text{Opaque}}} \leq \frac{2C_{\text{o-join}}(|P| + |D|, R)}{C_{\text{o-join}}(|P| + |D|, R)} = 2$$

Thus, this query will see at most 2x speedup from join reordering. However, other queries can benefit still further from this optimization. Consider a three-way join of Patient, Disease, and Gene to extract the gene mutation affecting each patient. We assume Gene is a very large public dataset, so that $|P| < |D| < |G|$. Because Disease contains a foreign key into Gene, the three-way join occurs only on primary-foreign key constraints with no need for aggregation. As before, a traditional SQL optimizer would execute $(P \bowtie D) \bowtie G$ while Opaque will run $(G \bowtie D) \bowtie P$. The costs are as follows:

$$\begin{aligned} C_{\text{SQL}} &= C_{\text{o-join}}(|P| + |D|, R) + C_{\text{o-join}}(|P| + |G|, R) \\ C_{\text{Opaque}} &= C_{\text{join}}(|G| + |D|, R) + C_{\text{o-join}}(|D| + |P|, R) \end{aligned}$$

Assuming $C_{\text{join}} \ll C_{\text{o-join}}$ and $|P| < |D| \ll |G|$,

$$\frac{C_{\text{SQL}}}{C_{\text{Opaque}}} = \frac{C_{\text{o-join}}(|P| + |G|, R)}{C_{\text{join}}(|G| + |D|, R)} \approx \frac{C_{\text{o-join}}(|G|, R)}{C_{\text{join}}(|G|, R)}$$

The maximum theoretical performance gain for this query therefore approaches the performance difference between the Opaque and non-oblivious join operators. We demonstrate this empirically in Figure 4.11b.

Limitations. Note that sensitivity propagation optimizes efficiently when the large tables in a database are not sensitive. This makes intuitive sense because computation on larger tables contributes more to the query runtime. If the larger tables are sensitive, then join reordering cannot help because any join with these tables must always be made oblivious. Therefore, the underlying schema will have a large impact on the effectiveness of our cost-based query optimizations.

4.5.4 Query planning for oblivious pad mode

As discussed in Section 4.2.3, the fact that the planner chose a query plan over another plan leaks some information about the selectivity of some operators. For example, generalized inner joins' costs depend on join selectivity information. This is not a problem for primary-foreign key joins because these costs can be estimated using only the size of each table: the output size of such a join is always the size of the foreign key table.

Oblivious pad mode does not leak such statistics information. All filters are pushed up and combined together at the end of the query. The optimizer does not need to use selectivity information because the overall size will not be reduced until the very end. Thus, our query planning stage only needs to use publicly-known information such as the size of each table.

4.6 Implementation

Opaque is implemented on top of Spark SQL, a big data analytics framework. Our implementation consists of 7000 lines of C++ enclave code and 3600 lines of Scala.

We implemented the Opaque operators and query optimization rules from Section 4.5 by extending Catalyst using its developer APIs with minimal modifications to Spark. Our operators are written in Scala and execute in the untrusted domain, making trusted calls to the enclave when necessary through JNI. For example, the `SORT` operator performs inter-machine sorting using an RDD-based implementation of distributed column sort in the untrusted domain (Section 4.4.1.2). Within each partition, the `SORT` operator serializes the encrypted rows and passes them using JNI to the worker node's enclave, which then performs the local sort in the trusted domain (Section 4.4.1.1). Our implementation currently does not support arbitrary user-defined functions (UDFs) due to the difficulty in making them oblivious.

Opaque encrypts and integrity-protects data on a block-level basis using AES in GCM mode, which provides data confidentiality as well as integrity. We pad all rows within a table to the same upper bound before encrypting. This is essential for tables with variable-length attributes as it prevents an attacker from distinguishing between different rows as they move through the system.

4.7 Evaluation

In this section, we demonstrate that Opaque represents a significant performance improvement over the state of the art in oblivious computation, quantify its overhead compared to an insecure baseline, and measure the gains from our query planning techniques. We evaluate Opaque's overhead without taking into account the overhead of T-SGX or other mechanisms that are needed for obtaining an oblivious SIM.

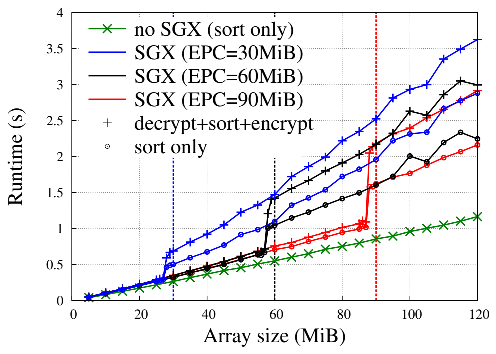
4.7.1 Experimental setup

Single-machine experiments were run using SGX hardware on a machine with Intel Xeon E3-1280 v5 (4 cores @ 3.70GHz, 8MiB cache) with 64GiB of RAM. This is the maximum number of cores available on processors supporting SGX at the time of writing.

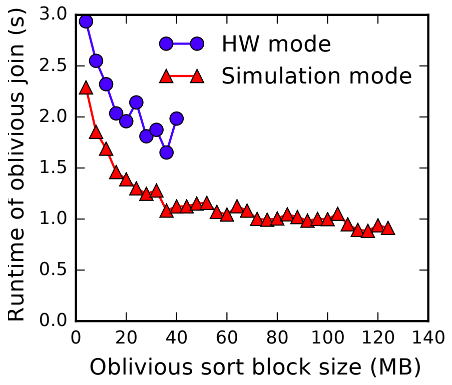
Distributed experiments were run on a cluster of 5 SGX machines with Intel Xeon E3-1230 v5 (4 cores @ 3.40GHz, 8MiB cache) with 64GiB of RAM.

4.7.2 Impact of oblivious memory size

We begin by studying the impact of the secure enclave memory size and show that Opaque will benefit significantly from future enclave implementations with more memory. SGX maintains an encrypted cache of memory pages called the Enclave Page Cache, which is small compared to the size of main memory. Once a page is evicted from the EPC, it



(a) Varying EPC size



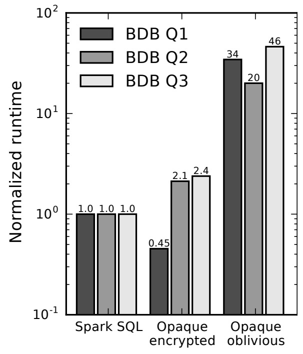
(b) Oblivious sort block size

Figure 4.9: Sort microbenchmarks. Figure 4.9a Non-oblivious sort in SGX. Exceeding EPC size causes a dramatic slowdown. Figure 4.9b Oblivious sort in SGX. Larger blocks improve performance until the EPC limit in HW mode, or indefinitely in simulation mode.

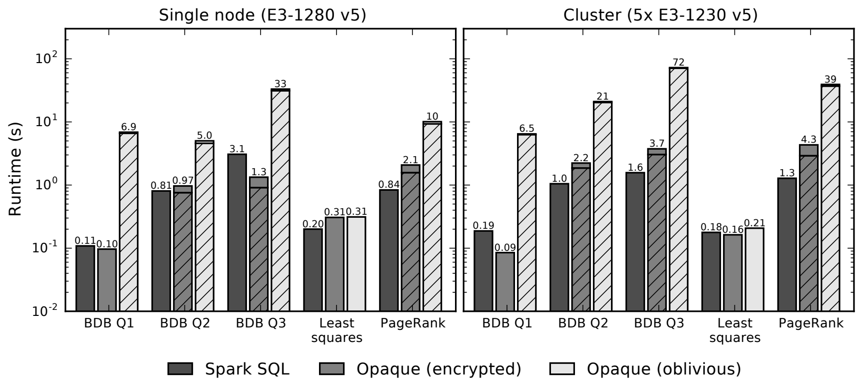
is decrypted if it was not entirely in CPU cache, re-encrypted under a different key, and stored in main memory. When an encrypted page in main memory is accessed, it needs to be decrypted again. This paging in and out of the EPC introduces a large overhead. Current implementations of SGX have a maximum effective EPC size of 93.5MiB, but this will be significantly increased in upcoming versions of SGX.

Sorting is the core operation in Opaque, so we studied how SGX affected its performance. In Figure 4.9a, we benchmark non-oblivious sorting (introsort) in SGX by sorting arrays of 64-bit integers of various sizes using EPCs of various sizes. We also measure the overhead incurred by decrypting input data and encrypting output data before and after sorting using AES-GCM-128. We see that exceeding the EPC size even by just a little incurs a 50 $\sim$ 60% overhead. When below the EPC limit, the overhead of encryption for I/O is just 7.46% on average. The overhead of the entire operation versus the insecure baseline is 31.7% on average.

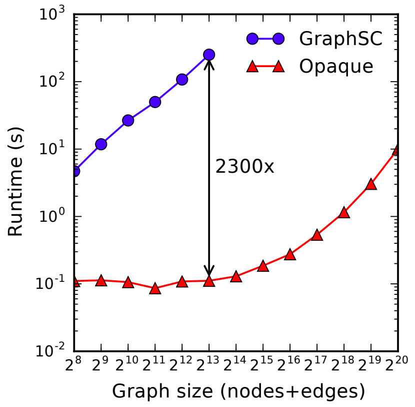
A larger internal memory improves performance radically. In Figure 4.9b, we call this an oblivious block size, and benchmarked the performance of oblivious sort with varying block sizes (Section 4.4.1.1). Within a block, regular quicksort can happen which speeds up performance. The case when only the registers are oblivious (namely an oblivious block of the same size as the available registers) did not fit in the graph: the overhead was 30x versus when the L3 cache (8MB) is oblivious. We see that in hardware mode, more oblivious memory improves performance until a sort block size of 40 MB, when the working set (two blocks for merging) exceeds the hardware EPC size, causing thrashing, as occurred in Figure 4.9a near EPC limits. In simulation mode, no thrashing occurs. In sum, Opaque's performance will improve significantly when run with more oblivious memory as a cache.



(a) Security mode comparison



(b) Comparison to Spark SQL



(c) Comparison to GraphSC

Figure 4.10: Figure 4.10a: Encryption mode is competitive with Spark SQL. Obliviousness (including network and memory obliviousness) adds up to 46x overhead. Figure 4.10b: Comparison across a wide range of queries. Hatched areas represent time spent sorting. Figure 4.10c: Single iteration of PageRank for various graph sizes.

4.7.3 System comparisons

4.7.3.1 Comparison with Spark SQL

We evaluated Opaque against vanilla Spark SQL, which provides no security guarantees, on three different workloads: SQL, machine learning, and graph analytics.

For the SQL workload, we benchmarked both systems on three out of four queries of Big Data Benchmark [12], a popular benchmark for big data SQL engines. The fourth query is an external script query and is not supported by our system. The three queries cover filter, aggregation (high cardinality), and join. For the machine learning workload, we chose least squares regression on 2D data; this query uses projection and global aggregation. Finally, we chose to benchmark PageRank for the graph analytics workload; this query uses projection and aggregation.

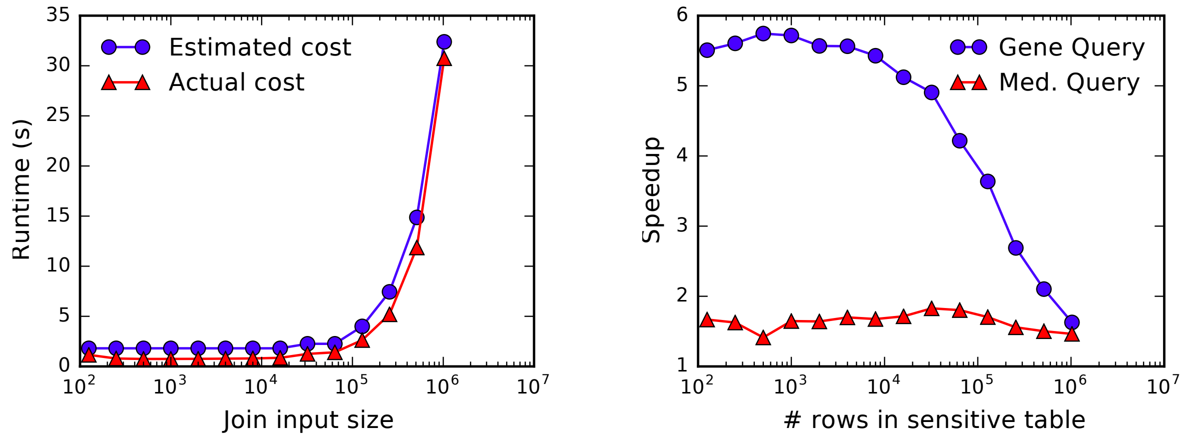
We show our results in two graphs, Figure 4.10a and Figure 4.10b. Figure 4.10a shows the performance of each of Opaque's security modes on the Big Data Benchmark in the distributed setting. Higher security naturally adds more overhead. Encryption mode is competitive with Spark SQL (between 52% improvement and 2.4x slowdown). The performance gain comes from the fact that Opaque runs C++ in the enclave, while Spark SQL incurs overhead from the JVM. Opaque's oblivious mode adds 20–46x overhead.

Figure 4.10b shows Opaque's performance on five queries. Hatched areas show the time spent in oblivious sort, the dominant cost. The left side of Figure 4.10b shows Opaque running on a single machine using SGX hardware compared to Spark SQL, while the right side shows the distributed setting. In the single-machine setting, Opaque's encryption mode performance varies from 58% performance gain to 2.5x performance loss when compared with the Spark SQL baseline. The oblivious mode (both network and memory oblivious) slows down the baseline by 1.6–62x. The right side shows Opaque's performance on a distributed SGX cluster. Encryption mode's performance ranges from a 52% performance improvement to a 3.3x slowdown, while oblivious mode adds 1.2–46x overhead. In these experiments, Opaque was configured with 80 MB of oblivious memory. As discussed in Section 4.7.2, more oblivious memory would give better performance, and such hardware proposals already exist (see Section 4.2.2).

4.7.3.2 Comparison with GraphSC

We use the same PageRank benchmark to compare with the existing state-of-the-art graph computation platform, GraphSC [166]. While Opaque is more general than graph computation, we compared Opaque with GraphSC instead of its more generic counterpart ObliVM [144], because ObliVM is about ten times slower than GraphSC.

We used data from GraphSC and ran the same experiment on both systems on our single node machine, with Opaque running in hardware mode with obliviousness. Figure 4.10c shows that Opaque is faster than GraphSC for all data sizes. For 8K graph size, Opaque is 2300x faster than GraphSC. This is consistent with the ObliVM and GraphSC papers: ObliVM reports a 9.3×10^6x slowdown, and GraphSC [166] a slowdown of 2×10^5x to 5×10^5x.



(a) Accuracy of obl. join cost model (b) Speedup from join reordering

Figure 4.11: Query planning benchmarks. Figure 4.11a: Our cost model closely approximates the empirical results for oblivious joins across a range of input sizes. Figure 4.11b: Join reordering provides up to 5x speedup for some queries.

Though GraphSC and Opaque share the high-level threat model of an untrusted service provider, they relax the threat model in different ways, explaining the performance gap. Opaque relies on trusted hardware, while GraphSC relies on two servers that must not collude and are semi-honest (do not cheat in the protocol) and so must use garbled circuits and secure two-party computation, which are much slower for generic computation than trusted hardware.

4.7.4 Query planning

We next evaluate the query planning techniques proposed in Section 4.5. First, we evaluate the cost model presented in Section 4.5.1 using a single-machine microbenchmark. We run an oblivious join and vary the input cardinality. We then fit the equation from Section 4.5.1 to the empirical results. Figure 4.11a shows that our theoretical cost model closely approximates the actual join costs.

Second, to evaluate the performance gain from join reordering, we run the two queries from Section 4.5.3.2. Figure 4.11b shows the speedup from reordering each query with varying sizes of the sensitive patient table. The medication query sees just under 2x performance gain because two equal-sized oblivious joins are replaced by one oblivious and one non-oblivious join. The gene query sees a 5x performance gain when the sensitive table is small because the larger oblivious join is replaced with a non-oblivious join. As the sensitive table increases in size, the benefit of join reordering approaches the same level as for the medication query.

4.8 Related work

4.8.1 Relevant cryptographic protocols

ORAM. Oblivious RAM [105, 198, 199, 197] is a cryptographic construct for protecting against access pattern leakage. However, ORAM does not fit in Opaque's setting because it has an intrinsically different computation model: serving key-value pairs. We show this problem by devising a simple strawman design using ORAM: put all data items in an in-memory ORAM in Spark.

How can ORAM be utilized if we attempt to sort data, which is an essential operation in SQL? One way to implement sorting on top of ORAM is to simply treat a sorting algorithm's compare-and-swap operation as two ORAM reads and two ORAM writes. This is not viable for three reasons. First, making an ORAM access for each data item is very slow. Second, current ORAM designs are not parallel and distributed, which means that the ORAM accesses will be serialized. Third, we cannot use a regular sorting algorithm because the number of comparisons may be different when run on different underlying data values. This could leak something about the encrypted data and would not provide obliviousness. Therefore, we must use a sorting network anyway, which means that adding ORAM will add an extra polylog(n) factor of accesses.

Other protocols. Fully homomorphic encryption [94, 95] permits computing any function on encrypted data, but is prohibitively slow. Oblivious protocols such as sorting and routing networks [58] are more relevant to Opaque, and Opaque builds on these as discussed in Section 4.4.1.

4.8.2 Non-oblivious systems

A set of database systems encrypt the data so that the service provider cannot see it. These databases can be classified into two types. The first type are encrypted databases, such as CryptDB [181], BlindSeer [178], Monomi [201], AlwaysEncrypted [159], and Seabed [177], that rely on cryptographic techniques for computation. The second type are databases, such as Haven [22], VC3 [189], TrustedDB [19], TDB [152] and GnatDb [206], that require trusted hardware to execute computation.

The main drawback of these systems is that they do not hide access patterns (both in memory and over the network) and hence leak data [214, 171]. Additionally, most of these systems do not fit the distributed analytics setting.

4.8.3 Oblivious systems

Non-distributed systems. Cipherbase [14] uses trusted hardware to achieve generic functionality for encrypted databases. The base Cipherbase design is not oblivious, but Arasu and Kaushik [13] have proposed oblivious protocols for SQL queries. However, unlike Opaque, their work does not consider the distributed setting. In particular, the proposed oblivious operators are not designed for a parallel setting resulting in sequential execution in Opaque, and do not consider boundary conditions. In addition, Cipherbase's contribution is a design proposal, while Opaque also provides a system and an evaluation.

Ohrimenko et al. [169] provide oblivious algorithms for common ML protocols such as matrix factorization or neural networks, but do not support oblivious relational operators or query optimization. Their focus is not on the distributed setting, and parts of the design (e.g., the choice of a sorting network) and the evaluation focus on single machine performance.

Distributed systems. ObliVM [144] is a platform for generic oblivious computation, and GraphSC [166] is a platform specialized to distributed graph computations built on ObliVM. As we show in Section 4.7.3, these systems are three orders of magnitude slower than Opaque. As explained there, they have a different threat model and use different techniques resulting in this higher overhead.

Ohrimenko et al. [171] and M2R [77] provide mechanisms for reducing network traffic analysis leakage for MapReduce jobs. Their solutions do not suffice for Opaque's setting because they do not protect in-memory access patterns. Moreover, they are designed for the simpler setting of a MapReduce job and do not suffice for Opaque's relational operators; further, they do not provide global query optimization of oblivious operators.

4.9 Conclusion

This chapter proposed Opaque, a distributed data analytics platform providing encryption, oblivious computation, and integrity. Opaque contributes a set of distributed oblivious relational operators as well as an oblivious query optimizer. Finally, we show that Opaque is three orders of magnitude faster than state-of-the-art specialized oblivious protocols. While Opaque is originally designed for the single party, it can be easily adapted to the collaborative setting if each party encrypts its data and uploads the encrypted data to the cloud. Every party can attest to all of the enclaves and give the enclaves its secret key during this process. With the secret keys. Opaque can then run SQL queries on the joint dataset.

Chapter 5

Helen

The two previous chapters presented systems that enable secure collaborative SQL analytics. In the next two chapters, we will explore systems that enable secure collaborative machine learning among mutually untrusting parties in a decentralized setup (i.e., without leveraging the aid of a third party like the cloud).

The motivation for this type of system comes from the fact that many organizations are increasingly interested in collaborative training or inference over their combined sensitive data. In the training scenario, the parties train on the joint dataset and also agree to release the final model to every participant so that everyone can benefit from the training process. In many existing applications, collaborative training is advantageous because training on more data tends to yield higher quality models [111]. Even more exciting is the potential of enabling new applications that are not possible to compute using a single party's data because they require training on complementary data from multiple parties (e.g., geographically diverse data).

While the general setup of collaborative learning in a decentralized setup fits within the cryptographic framework of secure multi-party computation (MPC) [24, 103, 215], implementing training using generic MPC frameworks is extremely inefficient, so recent training systems [167, 112, 162, 93, 55, 97, 10] opt for tailored protocols instead. However, many of these systems rely on outsourcing to non-colluding servers, and all assume a passive attacker who never deviates from the protocol. In practice, these assumptions are often not realistic because they essentially require an organization to base the confidentiality of its data on the correct behavior of other organizations. In fact, many parties such as financial institutions informed us that they are not comfortable with trusting the behavior of their competitors when it comes to sensitive business data.

Hence, we need a much stronger security guarantee: each organization should *only trust itself.* This goal calls for maliciously secure MPC in the setting where $m-1$ out of m parties can fully misbehave.

This chapter presents Helen, the first specialized platform for maliciously secure collaborative training. Helen supports regularized linear models, a significant slice of machine learning and statistics problems. This family of models includes ordinary least squares regression, ridge regression, and LASSO. Because these models are statistically robust and easily interpretable, they are widely used in cancer research [134], genomics [65, 176], financial risk analysis [188, 49], and are the foundation of basis pursuit techniques in signals processing. By combining insights from systems, cryptography, and machine learning, we show that it is possible to reformulate the training process so that the expensive cryptographic computation scales *independently* of the number of training samples. With our insights, Helen is able to achieve up to four orders of magnitude of performance improvement when compared to a common training protocol implemented using an existing state-of-the-art generic maliciously secure MPC framework.

Our first insight is to leverage a classic but under-utilized technique in distributed convex optimization called Alternating Direction Method of Multipliers (ADMM) [37]. The standard algorithm for training models today is SGD, which optimizes an objective function by iterating over the input dataset. Even though ADMM is less popular for training on plaintext data, we show that it is much more efficient for cryptographic training than SGD.

However, merely expressing ADMM in MPC does not solve an inherent scalability problem. As mentioned before, the malicious setting requires the protocol to ensure that the users' behavior is correct. A naïve way of solving this problem is to have each party commit to the entire input dataset and calculate the summaries using MPC. This is problematic because 1) the cryptographic computation will scale linearly in the number of samples, and 2) calculating the summaries would also require Helen to calculate complex matrix inversions within MPC (similar to [168]). Instead, we make a second observation that each party can use singular value decomposition (SVD) [106] to decompose its input summaries into small matrices that scale only in the number of features.

Finally, one important aspect of ADMM is that it enables decentralized computation. Expressing this computation directly in MPC would encode local optimization into a computation that is done by every party, thus losing the decentralization aspect of the original protocol. Instead, Helen uses partially homomorphic encryption to encrypt the global weights so that each party can solve the local problems in a decentralized manner, and enables each party to efficiently prove in zero-knowledge that it computed the local optimization problem correctly.

5.1 Background

5.1.1 Preliminaries

In this section, we describe the notation we use for the rest of the chapter. Let $P_1, ..., P_m$ denote the m parties. Let $\mathbb{Z}_N$ denote the set of integers modulo N, and $\mathbb{Z}_p$ denote the set of integers modulo a prime p. Similarly, we use $\mathbb{Z}_N^*$ to denote the multiplicative group modulo N.

We use z to denote a scalar, $\mathbf{z}$ to denote a vector, and $\mathbf{Z}$ to denote a matrix. We use $\texttt{Enc}_{\text{PK}}(x)$ to denote an encryption of x under a public key PK. Similarly, $\texttt{Dec}_{\text{SK}}(y)$ denotes a decryption of y under the secret key SK.

Each party P_i has a feature matrix $\mathbf{X}_i \in \mathbb{R}^{n\times d}$, where n is the number of samples per party and d is the feature dimension. $\mathbf{y}_i \in \mathbb{R}^{n\times 1}$ is the labels vector. The machine learning datasets use floating point representation, while our cryptographic primitives use groups and fields. Therefore, we represent the dataset using fixed point integer representation.

5.1.2 Cryptographic building blocks

5.1.2.1 Threshold partially homomorphic encryption

A partially homomorphic encryption scheme is a public key encryption scheme that allows limited computation over the ciphertexts. For example, Paillier [173] is an additive homomorphic encryption scheme: multiplying two ciphertexts together (in a certain group) generates a new ciphertext such that its decryption yields the sum of the two original plaintexts. Anyone with the public key can encrypt and manipulate the ciphertexts based on their homomorphic property. This encryption scheme also acts as a perfectly binding and computationally hiding homomorphic commitment scheme [110], another property we use in Helen.

A *threshold* variant of such a scheme has some additional properties. While the public key is known to everyone, the secret key is split across a set of parties such that a subset of them must participate together to decrypt a ciphertext. If not enough members participate, the ciphertext cannot be decrypted. The threshold structure can be altered based on the adversarial assumption. In Helen, we use a threshold structure where *all* parties must participate in order to decrypt a ciphertext.

5.1.2.2 Zero knowledge proofs

Informally, zero knowledge proofs are proofs that prove that a certain statement is true without revealing the prover's secret for this statement. For example, a prover can prove that there is a solution to a Sudoku puzzle without revealing the actual solution. Zero knowledge *proofs of knowledge* additionally prove that the prover indeed knows the secret. Helen uses modified Σ-protocols [67] to prove properties of a party's local computation. The main building blocks we use are ciphertext proof of plaintext knowledge, plaintext-ciphertext multiplication, and ciphertext interval proof of plaintext knowledge [62, 36], as we further explain in Section 5.3. Note that Σ-protocols are honest verifier zero knowledge, but can be transformed into full zero-knowledge using existing techniques [66, 82, 91]. We present our protocol using the Σ-protocol notation.

5.1.2.3 Malicious MPC

We utilize SPDZ [69], a state-of-the-art malicious MPC protocol, for both Helen and the secure baseline we evaluate against. Another recent malicious MPC protocol is authenticated garbled circuits [211], which supports boolean circuits. We decided to use SPDZ for our baseline because the majority of the computation in SGD is spent doing matrix operations, which is not efficiently represented in boolean circuits. For the rest of this section we give an overview of the properties of SPDZ.

An input $a \in \mathbb{F}_{p^k}$ to SPDZ is represented as $\langle a \rangle = (\delta, (a_1, \ldots, a_n), (\gamma(a)_1, \ldots, \gamma(a)_n))$, where a_i is a share of a and $\gamma(a)_i$ is the MAC share authenticating a under a SPDZ global key α. Player i holds $a_i, \gamma(a)_i$, and δ is public. During a correct SPDZ execution, the following property must hold: $a = \sum_i a_i$ and $\alpha(a + \delta) = \sum_i \gamma(a)_i$. The global key α is not revealed until the end of the protocol; otherwise the malicious parties can use α to construct new MACs.

SPDZ has two phases: an offline phase and an online phase. The offline phase is independent of the function and generates precomputed values that can be used during the online phase, while the online phase executes the designated function.

5.1.3 Learning and Convex Optimization

Much of contemporary machine learning can be framed in the context of minimizing the *cumulative error* (or loss) of a model over the training data. While there is considerable excitement around deep neural networks, the vast majority of real-world machine learning applications still rely on robust linear models because they are well understood and can be efficiently and reliably learned using established convex optimization procedures.

In this work, we focus on linear models with squared error and various forms of regularization resulting in the following set of multi-party optimization problems:

$$\hat{\mathbf{w}} = \arg\min_{w} \frac{1}{2} \sum_{i=1}^{m} \|\mathbf{X}_i \mathbf{w} - \mathbf{y}_i\|_2^2 + \lambda \mathbf{R}(\mathbf{w}), \tag{5.1}$$

where $\mathbf{X}_i \in \mathbb{R}^{n \times d}$ and $\mathbf{y}_i \in \mathbb{R}^n$ are the training data (features and labels) from party i. The regularization function $\mathbf{R}$ and regularization tuning parameter λ are used to improve prediction accuracy on high-dimensional data. Typically, the regularization function takes one of the following forms:

$$\mathbf{R}_{L^1}(\mathbf{w}) = \sum_{j=1}^{d} |\mathbf{w}_j|, \qquad \mathbf{R}_{L^2}(\mathbf{w}) = \frac{1}{2} \sum_{j=1}^{d} \mathbf{w}_j^2$$

corresponding to Lasso (L^1) and Ridge (L^2) regression respectively. The estimated model $\hat{\mathbf{w}} \in \mathbb{R}^d$ can then be used to render a new prediction $\hat{y}_* = \hat{\mathbf{w}}^T \mathbf{x}_*$ at a query point $\mathbf{x}_*$. It is worth noting that in some applications of LASSO (e.g., genomics [65]) the dimension d can be larger than n. However, in this work we focus on settings where d is smaller than n, and the real datasets and scenarios we use in our evaluation satisfy this property.

ADMM Alternating Direction Method of Multipliers (ADMM) [37] is an established technique for distributed convex optimization. To use ADMM, we first reformulate Eq. 5.1

by introducing *additional* variables and constraints:

$$\begin{aligned} \underset{\{\mathbf{w}_i\}_{i=1}^m, \mathbf{z}}{\textbf{minimize:}} \quad & \frac{1}{2}\sum_{i=1}^{m} \|\mathbf{X}_i\mathbf{w}_i - \mathbf{y}_i\|_2^2 + \lambda\mathbf{R}(\mathbf{z}), \\ \textbf{such that:} \quad & \mathbf{w}_i = \mathbf{z} \text{ for all } i \in \{1, \dots, p\} \end{aligned} \tag{5.2}$$

This equivalent formulation splits $\mathbf{w}$ into $\mathbf{w}_i$ for each party i, but still requires that $\mathbf{w}_i$ be equal to a global model $\mathbf{z}$. To solve this constrained formulation, we construct an *augmented Lagrangian*:

$$\begin{aligned} L\left(\{\mathbf{w}_i\}_{i=1}^m, \mathbf{z}, \mathbf{u}\right) = \frac{1}{2}\sum_{i=1}^{m} \|\mathbf{X}_i\mathbf{w}_i - \mathbf{y}_i\|_2^2 + \lambda\mathbf{R}(\mathbf{z}) + \\ \rho\sum_{i=1}^{m} \mathbf{u}_i^T\left(\mathbf{w}_i - \mathbf{z}\right) + \frac{\rho}{2}\sum_{i=1}^{m} ||\mathbf{w}_i - \mathbf{z}||_2^2, \end{aligned} \tag{5.3}$$

where the dual variables $\mathbf{u}_i \in \mathbb{R}^d$ capture the mismatch between the model estimated by party i and the global model $\mathbf{z}$ and the augmenting term $\frac{\rho}{2}\sum_{i=1}^{m} ||\mathbf{w}_i - \mathbf{z}||_2^2$ adds an additional penalty (scaled by the constant ρ) for deviating from $\mathbf{z}$.

The ADMM algorithm is a simple iterative dual ascent on the augmented Lagrangian of eq. (5.2). On the k^{th} iteration, each party locally solves this closed-form expression:

$$\mathbf{w}_i^{k+1} \leftarrow \left(\mathbf{X}_i^T\mathbf{X}_i + \rho\mathbf{I}\right)^{-1}\left(\mathbf{X}_i^T\mathbf{y}_i + \rho\left(\mathbf{z}^k - \mathbf{u}_i^k\right)\right) \tag{5.4}$$

and then shares its local model $\mathbf{w}_i^{k+1}$ and Lagrange multipliers $\mathbf{u}_i^k$ to solve for the new global weights:

$$\mathbf{z}^{k+1} \leftarrow \arg\min_{\mathbf{z}} \lambda\mathbf{R}(\mathbf{z}) + \frac{\rho}{2}\sum_{i=1}^{m} ||\mathbf{w}_i^{k+1} - \mathbf{z} + \mathbf{u}_i^k||_2^2. \tag{5.5}$$

Finally, each party uses the new global weights $\mathbf{z}^{k+1}$ to update its local Lagrange multipliers

$$\mathbf{u}_i^{k+1} \leftarrow \mathbf{u}_i^k + \mathbf{w}_i^{k+1} - \mathbf{z}^{k+1}. \tag{5.6}$$

The update equations (5.4), (5.5), and (5.6) are executed iteratively until all updates reach a fixed point. In practice, a fixed number of iterations may be used as a stopping condition, and that is what we do in Helen.

LASSO We use LASSO as a running example for the rest of the chapter in order to illustrate how our secure training protocol works. LASSO is a popular regularized linear regression model that uses the L^1 norm as the regularization function. The LASSO formulation is given by the optimization objective $\arg\min_{\mathbf{w}} \|\mathbf{Xw} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|$. The boxed section below shows the ADMM training procedure for LASSO. Here, the quantities in color are quantities that are intermediate values in the computation and need to be protected from every party, whereas the quantities in black are private values known to one party.

```
The coopetitive learning task for LASSO
Input of party $P_i$: $\mathbf{X}_i, \mathbf{y}_i$
    1. $\mathbf{A}_i \leftarrow \left(\mathbf{X}_i^T\mathbf{X}_i + \rho\mathbf{I}\right)^{-1}$
    2. $\mathbf{b}_i \leftarrow \mathbf{X}_i^T\mathbf{y}_i$
    3. $\mathbf{u^0}, \mathbf{z^0}, \mathbf{w^0} \leftarrow \mathbf{0}$
    4. For $k = 0$, ADMMIterations-1:
        a) $\mathbf{w}_i^{k+1} \leftarrow \mathbf{A}_i(\mathbf{b}_i + \rho\left(\mathbf{z}^k - \mathbf{u}_i^k\right))$
        b) $\mathbf{z}^{k+1} \leftarrow S_{\lambda/m\rho}\left(\frac{1}{m}\sum_{i=1}^{m}\left(\mathbf{w}_i^{k+1} + \mathbf{u}_i^k\right)\right)$
        c) $\mathbf{u}_i^{k+1} \leftarrow \mathbf{u}_i^k + \mathbf{w}_i^{k+1} - \mathbf{z}^{k+1}$
```

$S_{\lambda/m\rho}$ is the soft the soft thresholding operator, where

$$S_\kappa(a) = \begin{cases} a - \kappa & a > \kappa \\ 0 & |a| \leq \kappa \\ a + \kappa & a < -\kappa \end{cases} \tag{5.7}$$

The parameters λ and ρ are public and fixed.

5.2 System overview

Figure 5.1 shows the system setup in Helen. A group of m participants (also called parties) wants to jointly train a model on their data without sharing the plaintext data. As mentioned in the introduction, the use cases we envision for our system consist of a few large organizations (around 10 organizations), where each organization has a lot of data (n is on the order of hundreds of thousands or millions). The number of features/columns in the dataset d is on the order of tens or hundreds. Hence $d \ll n$.

We assume that the parties have agreed to publicly release the final model. As part of Helen, they will engage in an interactive protocol during which they share encrypted data, and only at the end will they obtain the model in decrypted form. Helen supports regularized linear models including least squares linear regression, ridge regression, LASSO, and elastic net. In the rest of the chapter, we focus on explaining Helen via LASSO, but we also provide update equations for ridge regression in Section 5.6.

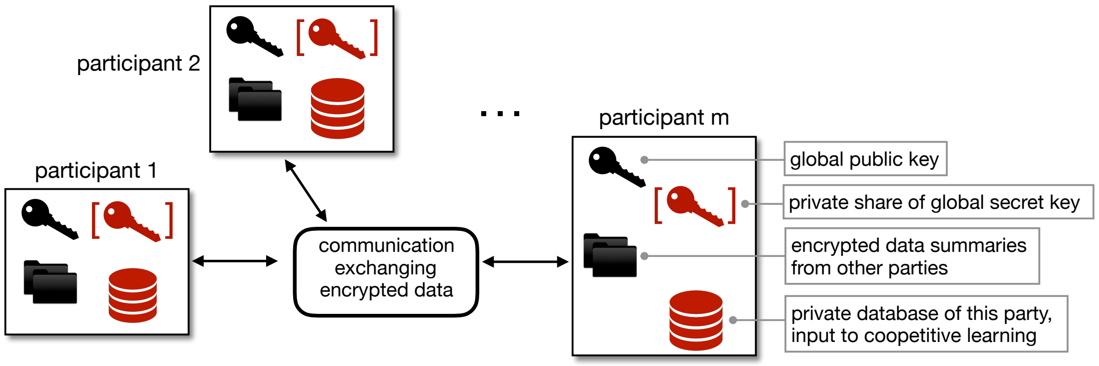

participant 2
participant 1
participant m
communication exchanging encrypted data
. . .
global public key
private share of global secret key
encrypted data summaries from other parties
private database of this party, input to coopetitive learning

Figure 5.1: Architecture overview of Helen. Every red shape indicates secret information known only to the indicated party, and black indicates public information visible to everyone (which could be private information in encrypted form). For participant m, we annotate the meaning of each quantity.

5.2.1 Threat model

We assume that all parties have agreed upon a single functionality to compute and have also consented to releasing the final result of the function to every party.

We consider a strong threat model in which all but one party can be compromised by a malicious attacker. This means that the compromised parties can deviate arbitrarily from the protocol, such as supplying inconsistent inputs, substituting their input with another party's input, or executing different computation than expected. In the flu prediction example, six divisions could collude together to learn information about one of the medical divisions. However, as long as the victim medical division follows our protocol correctly, the other divisions will not be able to learn anything about the victim division other than the final result of the function. We now state the security theorem.

Theorem 6. *Helen securely evaluates an ideal functionality f_{ADMM} in the (f_{crs}, f_{SPDZ})-hybrid model under standard cryptographic assumptions, against a malicious adversary who can statically corrupt up to $m - 1$ out of m parties.*

We formalize the security of Helen in the standalone MPC model. f_{crs} and f_{SPDZ} are ideal functionalities that we use in our proofs, where f_{crs} is the ideal functionality representing the creation of a common reference string, and f_{SPDZ} is the ideal functionality that makes a call to SPDZ. We present the formal definitions as well as proofs in Section 5.10.2.

Out of scope attacks/complementary directions. Helen does not prevent a malicious party from choosing a bad dataset for the coopetitive computation (e.g., in an attempt to alter the computation result). In particular, Helen does not prevent poisoning attacks [124, 53]. MPC protocols generally do not protect against bad inputs because there is no way to ensure that a party provides true data. Nevertheless, Helen will ensure that once a party supplies its input into the computation, the party is bound to using the same input consistently throughout the entire computation; in particular, this prevents a party from providing different inputs at different stages of the computation, or mix-and-matching inputs from other parties. Further, some additional constraints can also be placed in pre-processing, training, and post-processing to mitigate such attacks, as we elaborate in Section 5.8.2.

Helen also does not protect against attacks launched on the public model, for example, attacks that attempt to recover the training data from the model itself [192, 44]. The parties are responsible for deciding if they are willing to share with each other the model. Our goal is only to conduct this computation securely: to ensure that the parties do not share their raw plaintext datasets with each other, that they do not learn more information than the resulting model, and that only the specified computation is executed. Investigating techniques for ensuring that the model does not leak too much about the data is a complementary direction to Helen, and we expect that many of these techniques could be plugged into a system like Helen. For example, Helen can be easily combined with some differential privacy tools that add noise before model release to ensure that the model does not leak too much about an individual record in the training data. We further discuss possible approaches in Section 5.8.3.

Finally, Helen does not protect against denial of service – all parties must participate in order to produce a model.

5.2.2 Protocol phases

We now explain the protocol phases at a high level. The first phase requires all parties to agree to perform the coopetitive computation, which happens before initializing Helen. The other phases are run using Helen.

Agreement phase. In this phase, the m parties come together and agree that they are willing to run a certain learning algorithm (in Helen's case, ADMM for linear models) over their joint data. The parties should also agree to release the computed model among themselves.

The following discrete phases are run by Helen. We summarize their purposes here and provide the technical design for each in the following sections.

Initialization phase. During initialization, the m parties compute the threshold encryption parameters [83] using a generic maliciously secure MPC protocol like SPDZ [69]. The public output of this protocol is a public key PK that is known to everyone. Each party also receives a piece (called a *share*) of the corresponding secret key SK: party P_i receives the i-th share of the key denoted as $[\text{SK}]_i$. A value encrypted under PK can only be decrypted via all shares of the SK, so every party needs to agree to decrypt this value. Fig. 5.1 shows these keys. This phase only needs to run once for the entire training process, and does not need to be re-run as long as the parties' configuration does not change.

Input preparation phase. In this phase, each party prepares its data for the coopetitive computation. Each party P_i precomputes summaries of its data and commits to them by broadcasting encrypted summaries to all other parties. The parties also need to prove that they know the values inside these encryptions using zero-knowledge proofs of knowledge. From this moment on, party P_i will not be able to use different inputs for the rest of the computation.

By default, each party stores the encrypted summaries from other parties. This is a viable solution since these summaries are much smaller than the data itself. It is possible to also store all m summaries in a public cloud by having each party produce an integrity MAC of the summary from each other party and checking the MAC upon retrieval which protects against a compromised cloud.

Model compute phase. This phase follows the iterative ADMM algorithm, in which parties successively compute locally on encrypted data, followed by a coordination step with other parties using a generic MPC protocol.

Throughout this protocol, each party receives only encrypted intermediate data. No party learns the intermediate data because, by definition, an MPC protocol should not reveal any data beyond the final result. Moreover, each party proves in zero knowledge to the other parties that it performed the local computation correctly using data that is consistent with the private data that was committed in the input preparation phase. If any one party misbehaves, the other parties will be able to detect the cheating with overwhelming probability.

Model release phase. At the end of the model compute phase, all parties obtain an encrypted model. All parties jointly decrypt the weights and release the final model. However, it is possible for a set of parties to not receive the final model at the end of training if other parties misbehave (it has been proven that it is impossible to achieve fairness for generic MPC in the malicious majority setting [54]). Nevertheless, this kind of malicious behavior is easily detectable in Helen and can be enforced using legal methods.

5.3 Cryptographic Gadgets

Helen's design combines several different cryptographic primitives. In order to explain the design clearly, we split Helen into modular gadgets. In this section and the following sections, we discuss (1) how Helen implements these gadgets, and (2) how Helen composes them in the overall protocol.

For simplicity, we present our zero knowledge proofs as Σ-protocols, which require the verifier to generate random challenges. These protocols can be transformed into full zero knowledge with non-malleable guarantees with existing techniques [91, 82]. We explain one such transformation in Section 5.10.2.

5.3.1 Plaintext-ciphertext matrix multiplication proof

Gadget 1. *A zero-knowledge proof for the statement: "Given public parameters: public key PK, encryptions $E_{\mathbf{X}}$, $E_{\mathbf{Y}}$ and $E_{\mathbf{Z}}$; private parameters: $\mathbf{X}$,*

- $\mathtt{Dec}_{SK}(E_{\mathbf{Z}}) = \mathtt{Dec}_{SK}(E_{\mathbf{X}}) \cdot \mathtt{Dec}_{SK}(E_{\mathbf{Y}})$*, and*
- *I know* $\mathbf{X}$ *such that* $\mathtt{Dec}_{SK}(E_{\mathbf{X}}) = \mathbf{X}$*."*

Gadget usage We first explain how Gadget 1 is used in Helen. A party P_i in Helen knows a plaintext $\mathbf{X}$ and commits to $\mathbf{X}$ by publishing its encryption, denoted by $\mathtt{Enc}_{\mathrm{PK}}(\mathbf{X})$. P_i also receives an encrypted matrix $\mathtt{Enc}_{\mathrm{PK}}(\mathbf{Y})$ and needs to compute $\mathtt{Enc}_{\mathrm{PK}}(\mathbf{Z}) = \mathtt{Enc}_{\mathrm{PK}}(\mathbf{XY})$ by leveraging the homomorphic properties of the encryption scheme. Since parties in Helen may be malicious, other parties cannot trust P_i to compute and output $\mathtt{Enc}_{\mathrm{PK}}(\mathbf{Z})$ correctly. Gadget 1 will help P_i prove in zero-knowledge that it executed the computation correctly. The proof needs to be zero-knowledge so that nothing is leaked about the value of $\mathbf{X}$. It also needs to be a proof of knowledge so that P_i proves that it knows the plaintext matrix $\mathbf{X}$.

Protocol Using the Paillier ciphertext multiplication proofs [62], we can construct a naïve algorithm for proving matrix multiplication. For input matrices that are $\mathbb{R}^{l \times l}$, the naïve algorithm will incur a cost of l^3 since one has to prove each individual product. One way to reduce this cost is to have the prover prove that $\mathbf{tZ} = (\mathbf{tX})\mathbf{Y}$ for a randomly chosen $\mathbf{t}$ such that $\mathbf{t}_i = t^i \mathsf{mod} q$ (where $\mathbf{t}$ is a challenge from the verifier). For such a randomly chosen t, the chance that the prover can construct a $\mathbf{tZ}' = \mathbf{tXY}$ is exponentially small (see Theorem 4 for an analysis).

As the first step, both the prover and the verifier apply the reduction to get the new statement $\mathtt{Enc}_{\mathrm{PK}}(\mathbf{tZ}) = \mathtt{Enc}_{\mathrm{PK}}(\mathbf{tX})\mathtt{Enc}_{\mathrm{PK}}(\mathbf{Y})$. To prove this reduced form, we apply the Paillier ciphertext multiplication proof in a straightforward way. This proof takes as input three ciphertexts: E_a, E_b, E_c. The prover proves that it knows the plaintext a^* such that $a^* = \mathtt{Dec}_{\mathrm{SK}}(E_a)$, and that $\mathtt{Dec}_{\mathrm{SK}}(E_c) = \mathtt{Dec}_{\mathrm{SK}}(E_a) \cdot \mathtt{Dec}_{\mathrm{SK}}(E_b)$. We apply this proof to every multiplication for each dot product in $(\mathbf{tX}) \cdot \mathbf{Y}$. The prover then releases the individual encrypted products along with the corresponding ciphertext multiplication proofs. The verifier needs to verify that $\mathtt{Enc}_{\mathrm{PK}}(\mathbf{tZ}) = \mathtt{Enc}_{\mathrm{PK}}(\mathbf{tXY})$. Since the encrypted ciphers from the previous step are encrypted using Paillier, the verifier can homomorphically add them appropriately to get the encrypted vector $\mathtt{Enc}_{\mathrm{PK}}(\mathbf{tXY})$. From a dot product perspective, this step will sum up the individual products computed in the previous step. Finally, the prover needs to prove that each element of $\mathbf{tZ}$ is equal to each element of $\mathbf{tXY}$. We can prove this using the same ciphertext multiplication proof by setting $a^* = 1$.

5.3.2 Plaintext-plaintext matrix multiplication proof

Gadget 2. *A zero-knowledge proof for the statement: "Given public parameters: public key PK, encryptions $E_{\mathbf{X}}$, $E_{\mathbf{Y}}$, $E_{\mathbf{Z}}$; private parameters: $\mathbf{X}$ and $\mathbf{Y}$,*

- $\mathtt{Dec}_{SK}(E_{\mathbf{Z}}) = \mathtt{Dec}_{SK}(E_{\mathbf{X}}) \cdot \mathtt{Dec}_{SK}(E_{\mathbf{Y}})$*, and*
- *I know* $\mathbf{X}$, $\mathbf{Y}$, *and* $\mathbf{Z}$ *such that* $\mathtt{Dec}_{SK}(E_{\mathbf{X}}) = \mathbf{X}$, $\mathtt{Dec}_{SK}(E_{\mathbf{Y}}) = \mathbf{Y}$, *and* $\mathtt{Dec}_{SK}(E_{\mathbf{Z}}) = \mathbf{Z}$*."*

Gadget usage This proof is used to prove matrix multiplication when the prover knows *both* input matrices (and thus the output matrix as well). The protocol is similar to the plaintext-ciphertext proofs, except that we have to do an additional proof of knowledge of $\mathbf{Y}$.

Protocol The prover wishes to prove to a verifier that $\mathbf{Z} = \mathbf{XY}$ without revealing $\mathbf{X}, \mathbf{Y}$, or $\mathbf{Z}$. We follow the same protocol as Gadget 1. Additionally, we utilize a variant of the ciphertext multiplication proof that only contains the proof of knowledge component to show that the prover also knows $\mathbf{Y}$. The proof of knowledge for the matrix is simply a list of element-wise proofs for $\mathbf{Y}$. We do not explicitly prove the knowledge of $\mathbf{Z}$ because the matrix multiplication proof and the proof of knowledge for $\mathbf{Y}$ imply that the prover knows $\mathbf{Z}$ as well.

5.4 Input preparation phase

5.4.1 Overview

In this phase, each party prepares data for coopetitive training. In the beginning of the ADMM procedure, every party precomputes some summaries of its data and commits to them by broadcasting encrypted summaries to all the other parties. These summaries are then reused throughout the model compute phase. Some form of commitment is necessary in the malicious setting because an adversary can deviate from the protocol by altering its inputs. Therefore, we need a new gadget that allows us to efficiently commit to these summaries.

More specifically, the ADMM computation reuses two matrices during training: $\mathbf{A}_i = (\mathbf{X}_i^T\mathbf{X}_i + \rho\mathbf{I})^{-1}$ and $\mathbf{b}_i = \mathbf{X}_i^T\mathbf{y}_i$ from party i (see Section 5.1.3 for more details). These two matrices are of sizes $d \times d$ and $d \times 1$, respectively. In a semihonest setting, we would trust parties to compute $\mathbf{A}_i$ and $\mathbf{b}_i$ correctly. In a malicious setting, however, the parties can deviate from the protocol and choose $\mathbf{A}_i$ and $\mathbf{b}_i$ that are inconsistent with each other (e.g., they do not conform to the above formulations).

Helen does not have any control over what data each party contributes because the parties must be free to choose their own $\mathbf{X}_i$ and $\mathbf{y}_i$. However, Helen ensures that each party consistently uses the same $\mathbf{X}_i$ and $\mathbf{y}_i$ during the entire protocol. Otherwise, malicious parties could try to use different/inconsistent $\mathbf{X}_i$ and $\mathbf{y}_i$ at different stages of the protocol, and thus manipulate the final outcome of the computation to contain the data of another party.

One possibility to address this problem is for each party i to commit to its $\mathbf{X}_i$ in $\texttt{Enc}_{\text{PK}}(\mathbf{X}_i)$ and $\mathbf{y}_i$ in $\texttt{Enc}_{\text{PK}}(\mathbf{y}_i)$. To calculate $\mathbf{A}_i$, the party can calculate and prove $\mathbf{X}_i^T\mathbf{X}$ using Section 5.3.2, followed by computing a matrix inversion computation within SPDZ. The result $\mathbf{A}_i$ can be repeatedly used in the iterations. This is clearly inefficient because (1) the protocol scales linearly in n, which could be very large, and (2) the matrix inversion computation requires heavy compute.

Our idea is to prove using an alternate formulation via *singular value decomposition (SVD)* [106], which can be much more succinct: $\mathbf{A}_i$ and $\mathbf{b}_i$ can be decomposed using SVD to matrices that scale linearly in d. Proving the properties of $\mathbf{A}_i$ and $\mathbf{b}_i$ using the decomposed matrices is equivalent to proving using $\mathbf{X}_i$ and $\mathbf{y}_i$.

5.4.2 Protocol

5.4.2.1 Decomposition of reused matrices

We first derive an alternate formulation for $\mathbf{X}_i$ (denoted as $\mathbf{X}$ for the rest of this section). From fundamental linear algebra concepts we know that every matrix has a corresponding singular value decomposition [106]. More specifically, there exists unitary matrices $\mathbf{U}$ and $\mathbf{V}$, and a diagonal matrix $\mathbf{\Gamma}$ such that $\mathbf{X} = \mathbf{U}\mathbf{\Gamma}\mathbf{V}^T$, where $\mathbf{U} \in \mathbb{R}^{n\times n}$, $\mathbf{\Gamma} \in \mathbb{R}^{n\times d}$, and $\mathbf{V} \in \mathbb{R}^{d\times d}$. Since $\mathbf{X}$ and thus $\mathbf{U}$ are real matrices, the decomposition also guarantees that $\mathbf{U}$ and $\mathbf{V}$ are orthogonal, meaning that $\mathbf{U}^T\mathbf{U} = \mathbf{I}$ and $\mathbf{V}^T\mathbf{V} = \mathbf{I}$. If $\mathbf{X}$ is not a square matrix, then the top part of $\mathbf{\Gamma}$ is a diagonal matrix, which we will call $\mathbf{\Sigma} \in \mathbb{R}^{d\times d}$. $\mathbf{\Sigma}$'s diagonal is a list of singular values σ_i. The rest of the $\mathbf{\Gamma}$ matrix are 0's. If $\mathbf{X}$ is a square matrix, then $\mathbf{\Gamma}$ is simply $\mathbf{\Sigma}$. Finally, the matrices $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices. Given an orthogonal matrix $\mathbf{Q}$, we have that $\mathbf{Q}\mathbf{Q}^T = \mathbf{Q}^T\mathbf{Q} = \mathbf{I}$.

It turns out that $\mathbf{X}^T\mathbf{X}$ has some interesting properties:

$$\begin{aligned} \mathbf{X}^T\mathbf{X} &= (\mathbf{U}\mathbf{\Gamma}\mathbf{V}^T)^T\mathbf{U}\mathbf{\Gamma}\mathbf{V}^T \\ &= \mathbf{V}\mathbf{\Gamma}^T\mathbf{U}^T\mathbf{U}\mathbf{\Gamma}\mathbf{V}^T \\ &= \mathbf{V}\mathbf{\Gamma}^T\mathbf{\Gamma}\mathbf{V}^T \\ &= \mathbf{V}\mathbf{\Sigma}^2\mathbf{V}^T. \end{aligned}$$

We now show that $(\mathbf{X}^T\mathbf{X} + \rho\mathbf{I})^{-1} = \mathbf{V}\mathbf{\Theta}\mathbf{V}^T$, where $\mathbf{\Theta}$ is the diagonal matrix with diagonal values $\dfrac{1}{\sigma_i^2 + \rho}$.

$$\begin{aligned} (\mathbf{X}^T\mathbf{X} + \rho\mathbf{I})\mathbf{V}\mathbf{\Theta}\mathbf{V}^T &= \mathbf{V}(\mathbf{\Sigma}^2 + \rho\mathbf{I})\mathbf{V}^T\mathbf{V}\mathbf{\Theta}\mathbf{V}^T \\ &= \mathbf{V}(\mathbf{\Sigma}^2 + \rho\mathbf{I})\mathbf{\Theta}\mathbf{V}^T \\ &= \mathbf{V}\mathbf{V}^T = \mathbf{I}. \end{aligned}$$

Using a similar reasoning, we can also derive that

$$\mathbf{X}^T\mathbf{y} = \mathbf{V}\mathbf{\Gamma}^T\mathbf{U}^T\mathbf{y}.$$

5.4.2.2 Properties after decomposition

The SVD decomposition formulation sets up an alternative way to commit to matrices $(\mathbf{X}_i^T\mathbf{X}_i + \rho\mathbf{I})^{-1}$ and $\mathbf{X}_i\mathbf{y}_i$. For the rest of this section, we describe the zero knowledge proofs that every party has to execute. For simplicity, we focus on one party and use $\mathbf{X}$ and $\mathbf{y}$ to represent its data, and $\mathbf{A}$ and $\mathbf{b}$ to represent its summaries.

During the ADMM computation, matrices $\mathbf{A} = (\mathbf{X}^T\mathbf{X}+\rho\mathbf{I})^{-1}$ and $\mathbf{b} = \mathbf{X}^T\mathbf{y}$ are repeatedly used to calculate the intermediate weights. Therefore, each party needs to commit to $\mathbf{A}$ and $\mathbf{b}$. With the alternative formulation, it is no longer necessary to commit to $\mathbf{X}$ and $\mathbf{y}$ individually. Instead, it suffices to prove that a party knows $\mathbf{V}$, $\mathbf{\Theta}$, $\mathbf{\Sigma}$ (all are in $\mathbb{R}^{d\times d}$) and a vector $\mathbf{y}^* = (\mathbf{U}^T\mathbf{y})_{[1:d]} \in \mathbb{R}^{d\times 1}$ such that:

1. $\mathbf{A} = \mathbf{V}\mathbf{\Theta}\mathbf{V}^T$,
2. $\mathbf{b} = \mathbf{V}\mathbf{\Sigma}^T\mathbf{y}^*$,
3. $\mathbf{V}$ is an orthogonal matrix, namely, $\mathbf{V}^T\mathbf{V} = \mathbf{I}$, and
4. $\mathbf{\Theta}$ is a diagonal matrix where the diagonal entries are $1/(\sigma_i^2 + \rho)$. σ_i are the values on the diagonal of $\mathbf{\Sigma}$ and ρ is a public value.

Note that $\mathbf{\Gamma}$ can be readily derived from $\mathbf{\Sigma}$ by adding rows of zeros. Moreover, both $\mathbf{\Theta}$ and $\mathbf{\Sigma}$ are diagonal matrices. Therefore, we only commit to the diagonal entries of $\mathbf{\Theta}$ and $\mathbf{\Sigma}$ since the rest of the entries are zeros.

The above four statements are sufficient to prove the properties of $\mathbf{A}$ and $\mathbf{b}$ in the new formulation. The first two statements simply prove that $\mathbf{A}$ and $\mathbf{b}$ are indeed decomposed into *some* matrices $\mathbf{V}$, $\mathbf{\Theta}$, $\mathbf{\Sigma}$, and $\mathbf{y}^*$. Statement 3) shows that $\mathbf{V}$ is an orthogonal matrix, since by definition an orthogonal matrix $\mathbf{Q}$ has to satisfy the equation $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$. However, we allow the prover to choose $\mathbf{V}$. As stated before, the prover would have been free to choose $\mathbf{X}$ and $\mathbf{y}$ anyway, so this freedom does not give more power to the prover.

Statement 4) proves that the matrix $\mathbf{\Theta}$ is a diagonal matrix such that the diagonal values satisfy the form above. This is sufficient to show that $\mathbf{\Theta}$ is correct according to *some* $\mathbf{\Sigma}$. Again, the prover is free to choose $\mathbf{\Sigma}$, which is the same as freely choosing its input $\mathbf{X}$.

Finally, we chose to commit to $\mathbf{y}^*$ instead of committing to $\mathbf{U}$ and $\mathbf{y}$ separately. Following our logic above, it seems that we also need to commit to $\mathbf{U}$ and prove that it is an orthogonal matrix, similar to what we did with $\mathbf{V}$. This is not necessary because of an important property of orthogonal matrices: $\mathbf{U}$'s columns span the vector space $\mathbb{R}^n$. Multiplying $\mathbf{U}\mathbf{y}$, the result is a linear combination of the columns of $\mathbf{U}$. Since we also allow the prover to pick its $\mathbf{y}$, $\mathbf{U}\mathbf{y}$ essentially can be any vector in $\mathbb{R}^n$. Thus, we only have to allow the prover to commit to the product of $\mathbf{U}$ and $\mathbf{y}$. As we can see from the derivation, $\mathbf{b} = \mathbf{V}\mathbf{\Gamma}^T\mathbf{U}\mathbf{y}$, but since $\mathbf{\Gamma}$ is simply $\mathbf{\Sigma}$ with rows of zeros, the actual decomposition only needs the first d elements of $\mathbf{U}\mathbf{y}$. Hence, this allows us to commit to $\mathbf{y}^*$, which is $d \times 1$.

Using our techniques, Helen commits only to matrices of sizes $d \times d$ or $d \times 1$, thus removing any scaling in n (the number of rows in the dataset) in the input preparation phase.

5.4.2.3 Proving the initial data summaries

First, each party broadcasts $\texttt{Enc}_{\text{PK}}(\mathbf{V})$, $\texttt{Enc}_{\text{PK}}(\mathbf{\Sigma})$, $\texttt{Enc}_{\text{PK}}(\mathbf{\Theta})$, $\texttt{Enc}_{\text{PK}}(\mathbf{y}^*)$, $\texttt{Enc}_{\text{PK}}(\mathbf{A})$, and $\texttt{Enc}_{\text{PK}}(\mathbf{b})$. To encrypt a matrix, the party simply individually encrypts each entry. The encryption scheme itself also acts as a commitment scheme [110], so we do not need an extra commitment scheme.

To prove these statements, we also need another primitive called an interval proof. Moreover, since these matrices act as inputs to the model compute phase, we also need to prove that $\mathbf{A}$ and $\mathbf{b}$ are within a certain range (this will be used by Gadget 4, described in Section 5.5.5). The interval proof we use is from [36], which is an efficient way of proving that a committed number lies within a certain interval. However, what we want to prove is that an encrypted number lies within a certain interval. This can be solved by using techniques from [68], which appends the range proof with a commitment-ciphertext equality proof. This extra proof proves that, given a commitment and a Paillier ciphertext, both hide the same plaintext value.

To prove the first two statements, we invoke Gadget 1 and Section 5.3.2. This allows us to prove that the party knows all of the matrices in question and that they satisfy the relations laid out in those statements.

There are two steps to proving statement 3. The prover will compute $\texttt{Enc}_{\text{PK}}(\mathbf{V}^T\mathbf{V})$ and prove it computed it correctly using Gadget 1 as above. The result should be equal to the encryption of the identity matrix. However, since we are using fixed point representation for our data, the resulting matrix could be off from the expected values by some small error. $\mathbf{V}^T\mathbf{V}$ will only be close to $\mathbf{I}$, but not equal to $\mathbf{I}$. Therefore, we also utilize interval proofs to make sure that $\mathbf{V}^T\mathbf{V}$ is close to $\mathbf{I}$, without explicitly revealing the value of $\mathbf{V}^T\mathbf{V}$.

Finally, to prove statement 4, the prover does the following:

1. The prover computes and releases $\texttt{Enc}_{\text{PK}}(\mathbf{\Sigma}^2)$ because the prover knows $\mathbf{\Sigma}$ and proves using Gadget 1 that this computation is done correctly.

2. The prover computes $\texttt{Enc}_{\text{PK}}(\mathbf{\Sigma}^2 + \rho\mathbf{I})$, which anyone can compute because ρ and $\mathbf{I}$ are public. $\texttt{Enc}_{\text{PK}}(\mathbf{\Sigma}^2)$ and $\texttt{Enc}_{\text{PK}}(\rho\mathbf{I})$ can be multiplied together to get the summation of the plaintext matrices.

3. The prover now computes $\texttt{Enc}_{\text{PK}}(\mathbf{\Sigma}^2 + \rho\mathbf{I}) \times \texttt{Enc}_{\text{PK}}(\mathbf{\Theta})$ and proves this encryption was computed correctly using Gadget 1.

4. Similar to step 3), the prover ends this step by using interval proofs to prove that this encryption is close to encryption of the identity matrix.

5.5 Model compute phase

5.5.1 Overview

In the model compute phase, all parties use the summaries computed in the input preparation phase and execute the iterative ADMM training protocol. An encrypted weight vector is generated at the end of this phase and distributed to all participants. The participants can jointly decrypt this weight vector to get the plaintext model parameters. This phase executes in three steps: initialization, training (local optimization and coordination), and model release.

5.5.2 Initialization

We initialize the weights $\mathbf{w}_i^0, \mathbf{z}^0,$ and $\mathbf{u}_i^0$. There are two popular ways of initializing the weights. The first way is to set every entry to a random number. The second way is to initialize every entry to zero. In Helen, we use the second method because it is easy and works well in practice.

5.5.3 Local optimization

During ADMM's local optimization phase, each party takes the current weight vector and iteratively optimizes the weights based on its own dataset. For LASSO, the update equation is simply $\mathbf{w}_i^{k+1} \leftarrow \mathbf{A}_i(\mathbf{b}_i + \rho\left(\mathbf{z}^k - \mathbf{u}_i^k\right))$, where $\mathbf{A}_i$ is the matrix $(\mathbf{X}_i^T\mathbf{X}_i + \rho\mathbf{I})^{-1}$ and $\mathbf{b}_i$ is $\mathbf{X}_i^T\mathbf{y}_i$. As we saw from the input preparation phase description, each party holds encryptions of $\mathbf{A}_i$ and $\mathbf{b}_i$. Furthermore, given $\mathbf{z}^k$ and $\mathbf{u}_i^k$ (either initialized or received as results calculated from the previous round), each party can independently calculate $\mathbf{w}_i^{k+1}$ by doing plaintext scaling and plaintext-ciphertext matrix multiplication. Since this is done locally, each party also needs to generate a proof proving that the party calculated $\mathbf{w}_i^{k+1}$ correctly. We compute the proof for this step by invoking Gadget 1.

5.5.4 Coordination using MPC

After the local optimization step, each party holds encrypted weights $\mathbf{w}_i^{k+1}$. The next step in the ADMM iterative optimization is the coordination phase. Since this step contains non-linear functions, we evaluate it using generic MPC.

5.5.4.1 Conversion to MPC

First, the encrypted weights need to be converted into an MPC-compatible input. To do so, we formulate a gadget that converts ciphertext to arithmetic shares. The general idea behind the protocol is inspired by arithmetic sharing protocols [62, 69].

Gadget 3. *For m parties, each party having the public key PK and a share of the secret key SK, given public ciphertext $\texttt{Enc}_{PK}(a)$, convert a into m shares $a_i \in \mathbb{Z}_p$ such that $a \equiv \sum a_i \mathsf{mod} p$. Each party P_i receives secret share a_i and does not learn the original secret value a.*

Gadget usage Each party uses this gadget to convert $\texttt{Enc}_{\text{PK}}(\mathbf{w}_i)$ and $\texttt{Enc}_{\text{PK}}(\mathbf{u}_i)$ into input shares and compute the soft threshold function using MPC (in our case, SPDZ). We denote p as the public modulus used by SPDZ. Note that all of the computation encrypted by ciphertexts are dong within modulo p.

Protocol The protocol proceeds as follows:

1. Each party P_i generates a random value $r_i \in [0, 2^{|p|+\kappa}]$ and encrypts it, where κ is a statistical security parameter. Each party should also generate an interval plaintext proof of knowledge of r_i, then publish $\texttt{Enc}_{\text{PK}}(r_i)$ along with the proofs.
2. Each party P_i takes as input the published $\{\texttt{Enc}_{\text{PK}}(r_j)\}_{j=1}^m$ and compute the product with $\texttt{Enc}_{\text{PK}}(a)$. The result is $c = \texttt{Enc}_{\text{PK}}(a + \sum_{j=1}^m r_j)$.
3. All parties jointly decrypt c to get plaintext b.
4. Party 0 sets $a_0 = b - r_0 \mathsf{mod} p$. Every other party sets $a_i \equiv -r_i \mathsf{mod} p$.
5. Each party publishes $\texttt{Enc}_{\text{PK}}(a_i)$ as well as an interval proof of plaintext knowledge.

5.5.4.2 Coordination

The ADMM coordination step takes in $\mathbf{w}_i^{k+1}$ and $\mathbf{u}_i^k$, and outputs $\mathbf{z}^{k+1}$. The $\mathbf{z}$ update requires computing the soft-threshold function (a non-linear function), so we express it in MPC. Additionally, since we are doing fixed point integer arithmetic as well as using a relatively small prime modulus for MPC (256 bits in our implementation), we need to reduce the scaling factors accumulated on $\mathbf{w}_i^{k+1}$ during plaintext-ciphertext matrix multiplication. We currently perform this operation inside MPC as well.

5.5.4.3 Conversion from MPC

After the MPC computation, each party receives shares of $\mathbf{z}$ and its MAC shares, as well as shares of $\mathbf{w}_i$ and its MAC shares. It is easy to convert these shares back into encrypted form simply by encrypting the shares, publishing them, and summing up the encrypted shares. We can also calculate $\mathbf{u}_i^{k+1}$ this way. Each party also publishes interval proofs of knowledge for each published encrypted cipher. Finally, in order to verify that they are indeed valid SPDZ shares (the specific protocol is explained in the next section), each party also publishes encryptions and interval proofs of all the MACs.

5.5.5 Model release

5.5.5.1 MPC conversion verification

Since we are combining two protocols (homomorphic encryption and MPC), an attacker can attempt to alter the inputs to either protocol by using different or inconsistent attacker-chosen inputs. Therefore, before releasing the model, the parties must prove that they correctly executed the ciphertext to MPC conversion (and vice versa). We use another gadget to achieve this.

Gadget 4. *Given public parameters: encrypted value $\texttt{Enc}_{PK}(a)$, encrypted SPDZ input shares $\texttt{Enc}_{PK}(b_i)$, encrypted SPDZ MACs $\texttt{Enc}_{PK}(c_i)$, and interval proofs of plaintext knowledge, verify that*

1. $a \equiv \sum_i b_i \mathsf{mod} p$, *and*

2. b_i *are valid SPDZ shares and* c_i*'s are valid MACs on* b_i*.*

Gadget usage We apply Gadget 4 to all data that needs to be converted from encrypted ciphers to SPDZ or vice versa. More specifically, we need to prove that (1) the SPDZ input shares are consistent with $\texttt{Enc}_{\text{PK}}(\mathbf{w}_i^{k+1})$ that is published from each party, and (2) the SPDZ shares for $\mathbf{w}_i^{k+1}$ and $\mathbf{z}^k$ are authenticated by the MACs.

Protocol The gadget construction proceeds as follows:

1. Each party verifies that $\texttt{Enc}_{\text{PK}}(a)$, $\texttt{Enc}_{\text{PK}}(b_i)$ and $\texttt{Enc}_{\text{PK}}(c_i)$ pass the interval proofs of knowledge. For example, b_i and c_i need to be within $[0, p]$.

2. Each party homomorphically computes $\texttt{Enc}_{\text{PK}}(\sum_i b_i)$, as well as $E_d = \texttt{Enc}_{\text{PK}}(a - \sum_i b_i)$.

3. Each party randomly chooses $r_i \in [0, 2^{|a|+|\kappa|}]$, where κ is again a statistical security parameter, and publishes $\texttt{Enc}_{\text{PK}}(r_i)$ as well as an interval proof of plaintext knowledge.

4. Each party calculates $E_f = E_d \prod_i \texttt{Enc}_{\text{PK}}(r_i)^p = \texttt{Enc}_{\text{PK}}((a - \sum_i b_i) + \sum_i (r_i \cdot p))$. Here we assume that $\log |m| + |p| + |a| + |\kappa| < |n|$.

5. All parties participate in a joint decryption protocol to decrypt E_f obtaining e_f.

6. Every party individually checks to see that e_f is a multiple of p. If this is not the case, abort the protocol.

7. The parties release the SPDZ global MAC key α.

8. Each party calculates $\texttt{Enc}_{\text{PK}}(\alpha(\sum b_i + \delta))$ and $\texttt{Enc}_{\text{PK}}(\sum c_i)$.

9. Use the same method in steps 2 – 6 to prove that $\alpha(\sum b_i + \delta) \equiv \sum c_i \mathsf{mod} p$.

The above protocol is a way for parties to verify two things. First, that the SPDZ shares indeed match with a previously published encrypted value (i.e., Gadget 3 was executed correctly). Second, that the shares are valid SPDZ shares. The second step is simply verifying the original SPDZ relation among value share, MAC shares, and the global key.

Note that we cannot verify these relations by simply releasing the plaintext data shares and their MACs since the data shares correspond to the intermediate weights. Furthermore, the shares need to be equivalent in modulo p, which is different from the Paillier parameter N. Therefore, we use an alternative protocol to test modulo equality between two ciphertexts, which is the procedure described above in steps 2 to 6.

Since the encrypted ciphers come with interval proofs of plaintext knowledge, we can assume that $a \in [0, l]$. If two ciphertexts encrypt plaintexts that are equivalent to each other, they must satisfy that $a \equiv b \mathsf{mod} p$ or $a = b + \eta p$. Thus, if we take the difference of the two ciphertexts, this difference must be ηp. We could then run the decryption protocol to test that the difference is indeed a multiple of p.

If $a \equiv \sum_i b_i \mathsf{mod} p$, simply releasing the difference could still reveal extra information about the value of a. Therefore, all parties must each add a random mask to a. In step 3, r_i's are generated independently by all parties, which means that there must be at least one honest party who is indeed generating a random number within the range. The resulting plaintext thus statistically hides the true value of $a - \sum_i b_i$ with the statistical parameter κ. If $a \not\equiv \sum_i b_i \mathsf{mod} p$, then the protocol reveals the difference between $a - \sum_i b_i \mathsf{mod} p$. This is safe because the only way to reveal $a - \sum_i b_i \mathsf{mod} p$ is when an adversary misbehaves and alters its inputs, and the result is independent from the honest party's behavior.

5.5.5.2 Weight vector decryption

Once all SPDZ values are verified, all parties jointly decrypt $\mathbf{z}$. This can be done by first aggregating the encrypted shares of $\mathbf{z}$ into a single ciphertext. After this is done, the parties run the joint decryption protocol from [83] (without releasing the private keys from every party). The decrypted final weights are released in plaintext to everyone.

5.6 Extensions to Other Models

Though we used LASSO as a running example, our techniques can be applied to other linear models like ordinary least-squares linear regression, ridge regression, and elastic net. Here we show the update rules for ridge regression, and leave its derivation to the readers.

Ridge regression solves a similar problem as LASSO, except with L^2 regularization. Given a dataset $(\mathbf{X}, \mathbf{y})$ where $\mathbf{X}$ is the feature matrix and $\mathbf{y}$ is the prediction vector, ridge regression optimizes $\arg\min_{\mathbf{w}} \frac{1}{2}\|\mathbf{X}\mathbf{w}-\mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_2$. The update equations for ridge regression are:

$$\begin{aligned}
\mathbf{w}_i^{k+1} &= (\mathbf{X}_i^T\mathbf{X}_i + \rho I)^{-1}(\mathbf{X}_i^T\mathbf{y}_i + \rho(\mathbf{z}^k - \mathbf{u}_i^k)) \\
&\quad + (\rho/2)\|\mathbf{w}_i - \mathbf{z}^k + \mathbf{u}_i^k\|_2^2 \\
\mathbf{z}^{k+1} &= \frac{\rho}{2\lambda/m + \rho}(\bar{\mathbf{w}}^{k+1} + \bar{\mathbf{u}}^k) \\
\mathbf{u}_i^{k+1} &= \mathbf{u}_i^k + \mathbf{x}_i^{k+1} - \mathbf{z}^{k+1}
\end{aligned}$$

The local update is similar to LASSO, while the coordination update is a linear operation instead of the soft threshold function. Elastic net, which combines L^1 and L^2 regularization, can therefore be implemented by combining the regularization terms from LASSO and ridge regression.

5.7 Evaluation

We implemented Helen in C++. We utilize the SPDZ library [40], a mature library for maliciously secure multi-party computation, for both the baseline and Helen. In our implementation, we apply the Fiat-Shamir heuristic to our zero-knowledge proofs [82]. This technique is commonly used in implementations because it makes the protocols non-interactive and thus more efficient, but assumes the random oracle model.

We compare Helen's performance to a maliciously secure baseline that trains using stochastic gradient descent, similar to SecureML [162]. Since SecureML only supports two parties in the semihonest setting, we implemented a similar baseline using SPDZ [69]. SecureML had a number of optimizations, but they were designed for the two-party setting. We did not extend those optimizations to the multi-party setting. We will refer to SGD implemented in SPDZ as the "secure baseline" (we explain more about the SGD training process in Section 5.7.1). Finally, we do not benchmark Helen's Paillier key setup phase. This can be computed using SPDZ itself, and it is ran only once (as long as the party configuration does not change).

5.7.1 Experiment setup

We ran our experiments on EC2 using r4.8xlarge instances. Each machine has 32 cores and 244 GiB of memory. In order to simulate a wide area network setting, we created EC2 instances in Oregon and Northern Virginia. The instances are equally split across these two regions. To evaluate Helen's scalability, we used synthetic datasets that are constructed by drawing samples from a noisy normal distribution. For these datasets, we varied both the dimension and the number of parties. To evaluate Helen's performance against the secure baseline, we benchmarked both systems on two real world datasets from UCI [75].

Training assumptions. We do not tackle hyperparameter tuning in our work, and also assume that the data has been normalized before training. We also use a fixed number of rounds (10) for ADMM training, which we found experimentally using the real world datasets. We found that 10 rounds is often enough for the training process to converge to a reasonable error rate. Recall that ADMM converges in a small number of rounds because it iterates on a summary of the *entire dataset*. In contrast, SGD iteratively scans data from all parties at least once in order to get an accurate representation of the underlying distributions. This is especially important when certain features occur rarely in a dataset. Since the dataset is very large, even one pass already results in many rounds.

MPC configuration. As mentioned earlier, SPDZ has two phases of computation: an offline phase and an online phase. The offline phase can run independently of the secure function, but the precomputed values cannot be reused across multiple online phases. The SPDZ library provides several ways of benchmarking different offline phases, including MASCOT [131] and Overdrive [132]. We tested both schemes and found Overdrive to perform better over the wide area network. Since these are for benchmarking purposes only, we decided to estimate the SPDZ offline phase by dividing the number of triplets needed for a circuit by the benchmarked throughput. The rest of the evaluation section will use the estimated numbers for all SPDZ offline computation. Since Helen uses parallelism, we also utilized parallelism in the SPDZ offline generation by matching the number of threads on each machine to the number of cores available.

On the other hand, the SPDZ online implementation is not parallelized because the API was insufficient to effectively express parallelism. We note two points. First, while parallelizing the SPDZ library will result in a faster baseline, Helen also utilizes SPDZ, so any improvement to SPDZ also carries over to Helen. Second, as shown below, our evaluation shows that Helen still achieves significant performance gains over the baseline even if the online phase in the secure baseline is infinitely fast.

Finally, the parameters we use for Helen are: 128 bits for the secure baseline's SPDZ configuration, 256 bits for the Helen SPDZ configuration, and 4096 bits for Helen's Paillier ciphertext.

5.7.2 Theoretic performance

table 5.1 shows the theoretic scaling behavior for SGD and Helen, where m is the number of parties, n is the number of samples per party, d is the dimension, and C and c_i are constants. Note that c_i's are not necessarily the same across the different rows in the table. We split Helen's input preparation phase into three sub-components: SVD (calculated in plaintext), SVD proofs, and MPC offline (since Helen uses SPDZ during the model compute phase, we also need to run the SPDZ offline phase).

SGD scales linearly in n and d. If the number of samples per party is doubled, the number of iterations is also doubled. A similar argument goes for d. SGD scales quadratic in m because it first scales linearly in m due to the behavior of the MPC protocol. If we add more

Baseline	Secure SGD	$C \cdot m^2 \cdot n \cdot d$
Helen	SVD decomposition	$c_1 \cdot n \cdot d^2$
	SVD proofs	$c_1 \cdot m \cdot d^2 + c_2 \cdot d^3$
	MPC offline	$c_1 \cdot m^2 \cdot d$
	Model compute	$c_1 \cdot m^2 \cdot d + c_2 \cdot d^2 + c_3 \cdot m \cdot d$

Table 5.1: Theoretical scaling (complexity analysis) for SGD baseline and Helen. m is the number of parties, n is the number of samples per party, d is the dimension.

parties to the computation, the number of samples will also increase, which in turn increases the number of iterations needed to scan the entire dataset.

Helen, on the other hand, scales linearly in n only for the SVD computation. We emphasize that SVD is very fast because it is *executed on plaintext data*. The c_1 part of the SVD proofs formula scales linearly in m because each party needs to verify from every other party. It also scales linearly in d^2 because each proof verification requires d^2 work. The c_2 part of the formula has d^3 scaling because our matrices are $d \times d$), and to calculate a resulting encrypted matrix requires matrix multiplication on two $d \times d$ matrices.

The coordination phase from Helen's model compute phase, as well as the corresponding MPC offline compute phase, scale quadratic in m because we need to use MPC to re-scale weight vectors from each party. This cost corresponds to the c_1 part of the formula. The model compute phase's d^2 cost (c_2 part of the formula) reflects the matrix multiplication and the proofs. The rest of the MPC conversion proofs scale linearly in m and d (c_3 part of the formula).

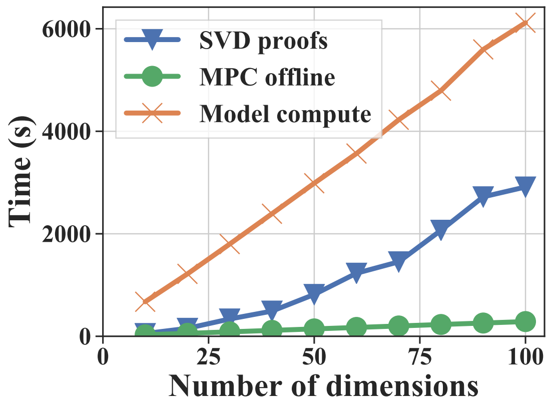


Figure 5.2: Helen's scaling as we increase the number of dimensions. The number of parties is fixed to be 4, and the number of samples per party is 100,000.

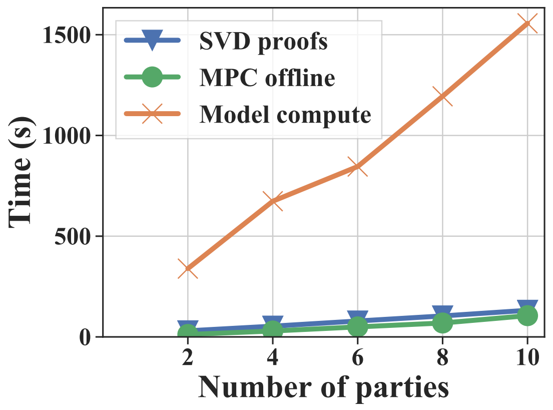


Figure 5.3: Helen's two phases as we increase the number of parties. The dimension is set to be 10, and the number of samples per party is $100,000$.

Samples per party	4000	6000	8000	10K	40K	100K	200K	400K	800K	1M
sklearn L2 error	8928.32	8933.64	8932.97	8929.10	8974.15	8981.24	8984.64	8982.88	8981.11	8980.35
Helen L2 error	8839.96	8828.18	8839.56	8837.59	8844.31	8876.00	8901.84	8907.38	8904.11	8900.37
sklearn MAE	58.07	58.04	58.10	58.05	58.34	58.48	58.55	58.58	58.56	58.57
Helen MAE	57.44	57.46	57.44	57.47	57.63	58.25	58.38	58.36	58.37	58.40

Table 5.2: Select errors for gas sensor (due to space), comparing Helen with a baseline that uses sklearn to train on all plaintext data. L2 error is the squared norm; MAE is the mean average error. Errors are calculated after post-processing.

Samples per party	1000	2000	4000	6000	8000	10K	20K	40K	60K	80K	100K
sklearn L2 error	92.43	91.67	90.98	90.9	90.76	90.72	90.63	90.57	90.55	90.56	90.55
Helen L2 error	93.68	91.8	91.01	90.91	90.72	90.73	90.67	90.57	90.54	90.57	90.55
sklearn MAE	6.86	6.81	6.77	6.78	6.79	6.81	6.80	6.79	6.79	6.80	6.80
Helen MAE	6.92	6.82	6.77	6.78	6.79	6.81	6.80	6.79	6.80	6.80	6.80

Table 5.3: Errors for song prediction, comparing Helen with a baseline that uses sklearn to train on all plaintext data. L2 error is the squared norm; MAE is the mean average error. Errors are calculated after post-processing.

5.7.3 Synthetic datasets

We want to answer two questions about Helen's scalability using synthetic datasets: how does Helen scale as we vary the number of features and how does it scale as we vary the number of parties? Note that we are not varying the number of input samples because that will be explored in Section 5.7.4 in comparison to the secure SGD baseline.

Figure 5.2 shows a breakdown of Helen's cryptographic computation as we scale the number of dimensions. The plaintext SVD computation is not included in the graph. The SVD proofs phase is dominated by the matrix multiplication proofs, which scales in d^2. The MPC offline phase and the model compute phase are both dominated by the linear scaling in

d, which corresponds to the MPC conversion proofs.

Figure 5.3 shows the same three phases as we increase the number of parties. The SVD proofs phase scales linearly in the number of parties m. The MPC offline phase scales quadratic in m, but its effects are not very visible for a small number of parties. The model compute phase is dominated by the linear scaling in m because the quadratic scaling factor isn't very visible for a small number of parties.

Finally, we also ran a microbenchmark to understand Helen's network and compute costs. The experiment used 4 servers and a synthetic dataset with 50 features and 100K samples per party. We found that the network costs account for approximately 2% of the input preparation phase and 22% of Helen's model compute phase.

5.7.4 Real world datasets

We evaluate on two different real world datasets: gas sensor data [75] and the million song dataset [26, 75]. The gas sensor dataset records 16 sensor readings when mixing two types of gases. Since the two gases are mixed with random concentration levels, the two regression variables are independent and we can simply run two different regression problems (one for each gas type). For the purpose of benchmarking, we ran an experiment using the ethylene data in the first dataset. The million song dataset is used for predicting a song's published year using 90 features. Since regression problems produce real values, the year can be calculated by rounding the regressed value.

For SGD, we set the batch size to be the same size as the dimension of the dataset. The number of iterations is equal to the total number of sample points divided by the batch size. Unfortunately, we had to extrapolate the runtimes for a majority of the baseline online phases because the circuits were too big to compile on our EC2 instances.

Figure 5.4 and Figure 5.5 compare Helen to the baseline on the two datasets. Note that Helen's input preparation graph combines the three phases that are run during the offline: plaintext SVD computation, SVD proofs, and MPC offline generation. We can see that Helen's input preparation phase scales very slowly with the number of samples. The scaling actually comes from the plaintext SVD calculation because both the SVD proofs and the MPC offline generation do not scale with the number of samples. Helen's model compute phase also stays constant because we fixed the number of iterations to a conservative estimate. SGD, on the other hand, does scale linearly with the number of samples in both the offline and the online phases.

For the gas sensor dataset, Helen's total runtime (input preparation plus model compute) is able to achieve a 21.5x performance gain over the baseline's total runtime (offline plus online) when the number of samples is 1000. When the number of samples per party reaches 1 million, Helen is able to improve over the baseline by 20689x. For the song prediction dataset, Helen is able to have a 9.1x performance gain over the baseline when the number of samples is 1000. When the number of samples per party reaches 100K, Helen improves over the baseline by 911x. Even if we compare Helen to the baseline's offline phase only, we find that Helen still has close to constant scaling while the baseline scales linearly with the

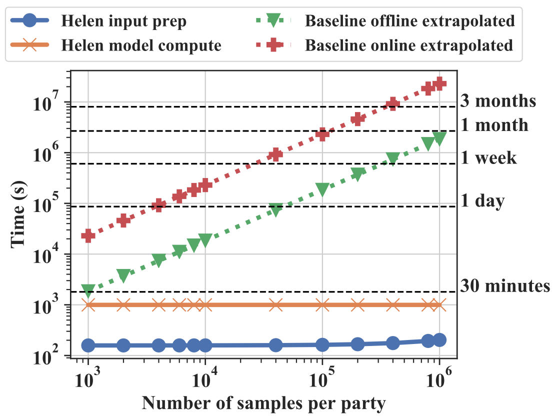


Figure 5.4: Helen and baseline performance on the gas sensor data. The gas sensor data contained over 4 million data points; we partitioned into 4 partitions with varying number of sample points per partition to simulate the varying number of samples per party. The number of parties is 4, and the number of dimensions is 16.

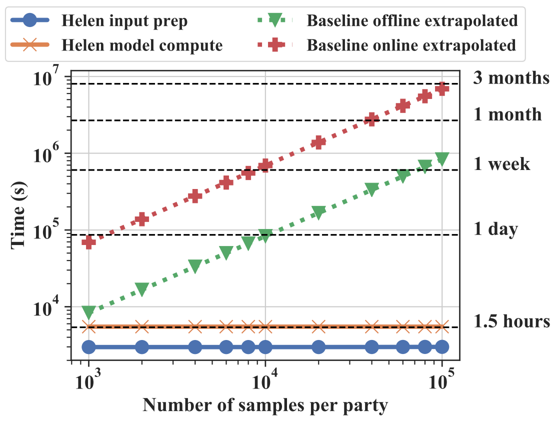


Figure 5.5: Helen and baseline performance on the song prediction data, as we vary the number of samples per party. The number of parties is 4, and the number of dimensions is 90.

Figure 5.6: Helen comparison with SGD

number of samples. The performance improvement compared to the baseline offline phase is up to 1540x for the gas sensor dataset and up to 98x for the song prediction dataset.

In table 5.2 and table 5.3, we evaluate Helen's test errors on the two datasets. We compare the L2 and mean average error for Helen to the errors obtained from a model trained using sklearn (a standard Python library for machine learning) on the plaintext data. We did not directly use the SGD baseline because its online phase does not compile for larger instances, and using sklearn on the plaintext data is a conservative estimate. We can see that Helen achieves similar errors compared to the sklearn baseline.

Work	Functionality	n-party?	Malicious security?	Practical?
Nikolaenko et al. [167]	ridge regression	no	no	–
Hall et al. [112]	linear regression	**yes**	no	–
Gascon et al. [92]	linear regression	no	no	–
Cock et al. [55]	linear regression	no	no	–
Giacomelli et al. [97]	ridge regression	no	no	–
Alexandru et al. [10]	quadratic opt.	no	no	–
SecureML [162]	linear, logistic, deep learning	no	no	–
Shokri&Shmatikov [193]	deep learning	not MPC (heuristic)	no	–
Semi-honest MPC [23]	any function	**yes**	no	–
Malicious MPC [69, 103, 31, 204]	any function	**yes**	**yes**	no
Our proposal, Helen: regularized linear models		**yes**	**yes**	**yes**

Figure 5.7: **Insufficiency of existing cryptographic approaches.** "n-party" refers to whether the $n(>2)$ organizations can perform the computation with *equal trust* (thus not including the two non-colluding servers model). We answer the practicality question only for maliciously-secure systems. We note that a few works that we marked as not coopetitive and not maliciously secure discuss at a high level how one might extend their work to such a setting, but they did not flesh out designs or evaluate their proposals.

5.8 Related work

We organize the related work section into related coopetitive systems and attacks.

5.8.1 Coopetitive systems

Coopetitive training systems In Figure 5.7, we compare Helen to prior coopetitive training systems [167, 112, 93, 55, 97, 10, 162, 193]. The main takeaway is that, excluding *generic* maliciously secure MPC, prior training systems do not provide malicious security. Furthermore, most of them also assume that the training process requires outsourcing to two non-colluding servers. At the same time, and as a result of choosing a weaker security model, some of these systems provide richer functionality than Helen, such as support for neural networks. As part of our future work, we are exploring how to apply Helen's techniques to logistic regression and neural networks.

Other coopetitive systems Other than coopetitive training systems, there are prior works on building coopetitive systems for applications like machine learning prediction and

SQL analytics. Coopetitive prediction systems [35, 187, 185, 147, 98, 127] typically consist of two parties, where one party holds a model and the other party holds an input. The two parties jointly compute a prediction without revealing the input or the model to the other party. Coopetitive analytics systems [20, 165, 33, 59, 28] allow multiple parties to run SQL queries over all parties' data. These computation frameworks do not directly translate to Helen's training workloads. Most of these works also do not address the malicious setting. Recent work has also explored secure learning and analytics using separate compute nodes and blockchains [89, 90]. The setup is different from that of Helen where we assume that the data providers are malicious and are also performing and verifying the computation.

Trusted hardware based systems The related work presented in the previous two sections all utilize purely software based solutions. Another possible approach is to use trusted hardware [155, 60], and there are various secure distributed systems that could be extended to the coopetitive setting [189, 116, 222]. However, these hardware mechanisms require additional trust and are prone to side-channel leakages [136, 203, 141].

5.8.2 Attacks on machine learning

Machine learning attacks can be categorized into data poisoning, model leakage, parameter stealing, and adversarial learning. As mentioned in §5.2.1, Helen tackles the problem of cryptographically running the training algorithm without sharing datasets amongst the parties involved, while defenses against these attacks are *orthogonal* and complementary to our goal for Helen. Often, these machine learning attacks can be separately addressed outside of Helen. We briefly discuss two relevant attacks related to the training stage and some methods for mitigating them.

Poisoning Data poisoning allows an attacker to inject poisoned inputs into a dataset before training [124, 53]. Generally, malicious MPC does not prevent an attacker from choosing incorrect initial inputs because there is no way to enforce this requirement. Nevertheless, there are some ways of mitigating arbitrary poisoning of data that would complement Helen's training approach. Before training, one can check that the inputs are confined within certain intervals. The training process itself can also execute *cross validation*, a process that can identify parties that do not contribute useful data. After training, it is possible to further post process the model via techniques like fine tuning and parameter pruning [148].

Model leakage Model leakage [192, 44] is an attack launched by an adversary who tries to infer information about the training data from the model itself. Again, malicious MPC does not prevent an attacker from learning the final result. In our coopetitive model, we also assume that all parties want to cooperate and have agreed to release the final model to everyone.

5.8.3 Differential privacy

One way to alleviate model leakage is through the use of differential privacy [121, 5, 79]. For example, one way to add differential privacy is to add carefully chosen noise directly to the output model [121]. Each party's noise can be chosen directly using MPC, and the final result can then be added to the final model before releasing. In Helen, differential privacy would be added after the model is computed, but before the model release phase. However, there are more complex techniques for differential privacy that involve modification to the training algorithm, and integrating this into Helen is an interesting future direction to explore.

5.9 ADMM derivations

Ridge regression solves a similar problem as LASSO, except with L2 regularization. Given dataset $(X, \mathbf{y})$ where X is the feature matrix and $\mathbf{y}$ is the prediction vector, ridge regression optimizes $\arg\min_{\mathbf{w}} \frac{1}{2}\|X\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_2$. Splitting the weights into $\mathbf{w}$ and $\mathbf{z}$, we have

$$\begin{aligned}&\text{minimize } \frac{1}{2}\|X\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{z}\|_2\\ &\text{subject to } \mathbf{w} - \mathbf{z} = 0\end{aligned}$$

We first find the augmented Lagrangian

$$\begin{aligned}\mathcal{L}(\mathbf{w}, \mathbf{z}, \mathbf{v}) = &\frac{1}{2}\|X\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{z}\|_2\\ &+ \mathbf{v}^T(\mathbf{w} - \mathbf{z}) + \frac{\rho}{2}\|\mathbf{w} - \mathbf{z}\|_2^2\end{aligned}$$

where $\mathbf{w}$ and $\mathbf{z}$ are the primal weight vectors, and $\mathbf{v}$ is the dual weight vector. To simply the equations, we replace $\mathbf{v}$ with the scaled dual variable $\mathbf{u}$ where $\mathbf{u} = (1/\rho)\mathbf{v}$. The update equations come out to

$$\begin{aligned}\mathbf{w}^{k+1} &= \arg\min_{\mathbf{w}} \frac{1}{2}\|X\mathbf{w} - \mathbf{y}\|_2^2 + (\rho/2)\|\mathbf{w} - \mathbf{z}^k + \mathbf{u}^k\|_2^2)\\ \mathbf{z}^{k+1} &= \arg\min_{\mathbf{z}} \lambda\|\mathbf{z}\|_2^2 + (\rho/2)(\mathbf{w}^{k+1} + \mathbf{z} + \mathbf{u}^k)\\ \mathbf{u}^{k+1} &= \mathbf{u}^k + \mathbf{w}^{k+1} + \mathbf{z}^{k+1}\end{aligned}$$

Since our loss function is decomposable based on data blocks, we can apply the generic global variable consensus ADMM algorithm and find

$$
\begin{aligned}
\mathbf{w}_i^{k+1} &= \arg\min_{\mathbf{w}_i} \frac{1}{2}\|X_i\mathbf{w} - \mathbf{y}\|_2^2 + (\rho/2)\|\mathbf{w}_i - \mathbf{z}^k + \mathbf{u}^k\|_2^2) \\
\mathbf{z}^{k+1} &= \arg\min_{\mathbf{z}} \lambda\|\mathbf{z}\|_2^2 + (m\rho/2)\|\mathbf{z} - \bar{\mathbf{w}}^{k+1} - \bar{\mathbf{u}}^k\|_2^2 \\
\mathbf{u}^{k+1} &= \mathbf{u}^k + \mathbf{w}^{k+1} - \mathbf{z}^{k+1}
\end{aligned}
$$

Thus, the $\mathbf{w}$ update is

$$
\begin{aligned}
\mathbf{w}_i^{k+1} &= (X_i^T X_i + \rho I)^{-1}(X_i^T \mathbf{y}_i + \rho(\mathbf{z}^k - \mathbf{u}_i^k)) \\
&\quad + (\rho/2)\|\mathbf{w}_i - \mathbf{z}^k + \mathbf{u}_i^k\|_2^2 \\
\mathbf{z}^{k+1} &= \frac{\rho}{2\lambda/m + \rho}(\bar{\mathbf{w}}^{k+1} + \bar{\mathbf{u}}^k) \\
\mathbf{u}_i^{k+1} &= \mathbf{u}_i^k + \mathbf{x}_i^{k+1} - \mathbf{z}^{k+1}
\end{aligned}
$$

Therefore, the $\mathbf{w}_i$ update is the same as LASSO and can be computed using the same techniques. The $\mathbf{z}$ update is actually linear and does not require comparisons, though MPC is still required for reducing the scaling factors accumulated during fixed point integer multiplications.

5.10 Security proofs

5.10.1 Definitions

We first define the MPC model. The full definitions are taken from [62, 43], so please refer to those papers for more details.

Real world model. Let π be an n-party protocol. The protocol is executed on an open broadcast network with static, active, and rushing adversary $\mathcal{A}$ (rushing means that the adversaries can send their messages after seeing all of the honest parties' messages). The number of adversaries can be a majority of the participants. Let κ be the security parameter. Each party P_i has public input x_i^p and secret input x_i^s, as well as public output y_i^p and secret output y_i^s. The adversary $\mathcal{A}$ receives the public input and output of all parties.

Let $\mathbf{x} = (x_1^s, x_1^p, \dots, x_n^s, x_n^p)$ be the parties' input, and let $\mathbf{r} = (r_1, \dots, r_n, r_{\mathcal{A}})$ be the parties' and the adversary's private input randomness. Let $C \subset M$ be the corrupted parties, and let $a \in \{0,1\}^*$ be the adversary's auxiliary input. Let $H \subset M$ be the honest parties. Therefore, we have that $H + C = M$.

By $\text{ADVR}_{\pi,\mathcal{A}}(\kappa, \mathbf{x}, C, a, \mathbf{r})$ and $\text{EXEC}_{\pi,\mathcal{A}}(\kappa, \mathbf{x}, C, a, \mathbf{r})_i$ we denote the output of the adversary $\mathcal{A}$ and the output of party P_i, respectively, after a real world execution of π with the given input under attack from $\mathcal{A}$. Let

$$\begin{aligned}\mathrm{EXEC}_{\pi,\mathcal{A}}(\kappa,\mathbf{x},C,a,\mathbf{r}) =&(\mathrm{ADVR}_{\pi,\mathcal{A}}(k,\mathbf{x},C,a,\mathbf{r}), && (5.8)\\ &\mathrm{EXEC}_{\pi,\mathcal{A}}(\kappa,\mathbf{x},C,a,\mathbf{r})_1, && (5.9)\\ &\dots, && (5.10)\\ &\mathrm{EXEC}_{\pi,\mathcal{A}}(\kappa,\mathbf{x},C,a,\mathbf{r})_n) && (5.11)\end{aligned}$$

This is simply the union of the different parties' and the adversary's real world output distribution. Denote by $\mathrm{EXEC}_{\pi,\mathcal{A}}(\kappa,\mathbf{x},C,a)$ the random variable $\mathrm{EXEC}_{\pi,\mathcal{A}}(k,\mathbf{x},C,a,\mathbf{r})$, where $\mathbf{r}$ is chosen uniformly random. We define the distribution ensemble with security parameter κ and index $(\mathbf{x},C,a)$ by

$$\mathrm{EXEC}_{\pi,\mathcal{A}} = \{\mathrm{EXEC}_{\pi,\mathcal{A}}(\kappa,\mathbf{x},C,a,\mathbf{r})_i\}_{\kappa\in M,\mathbf{x}\in(\{0,1\}^*)^{2n},a\in\{0,1\}^*,i\in H} \quad (5.12)$$

Ideal world model. Let $f : M\times(\{0,1\}^*)^{2n}\times\{0,1\}^* \to (\{0,1\}^*)^{2n}$ be a probabilistic n-party function computable in probabilistic polynomial time (PPT). The inputs and outputs are defined as $(y_1^s,y_1^p,\dots,y_n^s,y_n^p) \leftarrow f(\kappa,x_1^s,x_1^p,\dots,x_n^s,x_n^p,r)$, where κ is the security parameter and r is the random input. In the ideal world, the parties send their inputs to a trusted third party T that chooses a uniformly random r, computes f on these inputs and returns (y_i^s,y_i^p) to P_i.

The active static ideal world adversary $\mathcal{A}_I$ sees all x_i^p values, as well as x_i^s for all corrupted parties. The adversary then substitutes the values (x_i^s,x_i^p) for the corrupted parties by values of his choice $(x_i^{s'},x_i^{p'})$. We set $(x_i^{s'},x_i^{p'}) = (x_i^s,x_i^p)$ for the honest parties. The ideal function f is evaluated on $(\kappa,x_1^{s'},x_1^{p'},\dots,x_n^{s'},x_n^{p'},r)$ via an oracle call. Each party receives output (y_i^s,y_i^p), and the adversary sees y_i^p for all parties as well as y_i^s for all corrupted parties.

Similar to the real world execution, we define $\mathrm{ADVR}_{\pi,\mathcal{A}_i}(\kappa,\mathbf{x},C,a,\mathbf{r})$ and $\mathrm{IDEAL}_{\pi,\mathcal{A}_i}(\kappa,\mathbf{x},C,a,\mathbf{r})_i$ we denote the output of the adversary $\mathcal{A}_i$ and the output of party P_i, respectively, after an ideal world execution with the given input under attack from $\mathcal{A}_i$. Let

$$\begin{aligned}\mathrm{IDEAL}_{f,\mathcal{A}_I}(\kappa,\mathbf{x},C,a,\mathbf{r}) =& (\mathrm{ADVR}_{f,\mathcal{A}_I}(\kappa,\mathbf{x},C,a,\mathbf{r}), && (5.13)\\ &\mathrm{IDEAL}_{f,\mathcal{A}_I}(\kappa,\mathbf{x},C,a,\mathbf{r})_1, && (5.14)\\ &\mathrm{IDEAL}_{f,\mathcal{A}_I}(\kappa,\mathbf{x},C,a,\mathbf{r})_2, && (5.15)\\ &\dots, && (5.16)\\ &\mathrm{IDEAL}_{f,\mathcal{A}_I}(\kappa,\mathbf{x},C,a,\mathbf{r})_n) && (5.17)\end{aligned}$$

denote the collective output distribution of the parties and the adversary. Define a distribution ensemble by

$$\mathrm{IDEAL}_{f,\mathcal{A}_I} = \{\mathrm{IDEAL}_{f,\mathcal{A}_I}(\kappa,\mathbf{x},C,a)_i\}_{\kappa\in N,\mathbf{x}\in(\{0,1\}^*)^{2n},a\in\{0,1\}^*,i\in H} \quad (5.18)$$

Hybrid model. In the $(g_1,\dots,g_l)$-hybrid model, the execution of a protocol π proceeds in the real-life model, except that the parties have access to a trusted party T for evaluating the n-party functions $g_1, \ldots, g_l$. These ideal evaluations proceed as in the ideal world model. The distribution ensemble is

$$\mathrm{EXEC}^{g_1,\ldots,g_l}_{\pi,\mathcal{A}} = \{\mathrm{EXEC}^{g_1,\ldots,g_l}_{\pi,\mathcal{A}}(\kappa, \mathbf{x}, C, a)_i\}_{\kappa\in N, \mathbf{x}\in(\{0,1\}^*)^{2n}, a\in\{0,1\}^*, i\in H} \tag{5.19}$$

Security can be defined by requiring a real world execution or a $(g_1, \ldots, g_l)$-hybrid execution of a protocol π for computing an ideal functionality f to reveal no more information to an adversary than does an ideal execution of f. We can define the real world model by the ()-hybrid model.

Definition 1. *Let f be an n-party function, let π be an n-party protocol We say that π securely evaluates f in the $(g_1, \ldots, g_l)$-hybrid model if for any active static $(g_1, \ldots, g_l)$-hybrid adversary $\mathcal{A}$, which corrupts only subsets of C, there exists a static active ideal-model adversary $\mathcal{S}$ such that $IDEAL_{f,\mathcal{S}} \approx_c EXEC^{g_1,\ldots,g_l}_{\pi,\mathcal{A}}$.*

Finally, we utilize the modular composition operation that was defined in [43]. The modular composition theorem (informally) states that if π Γ-securely evaluates f in the $(g_1, \ldots, g_l)$-hybrid model and π_{g_i} Γ-securely evaluates g_i in the $(g_1, \ldots, g_{i-1}, g_{i+1}, \ldots, g_l)$-hybrid model, then the protocol π', which follows protocol π except with oracle calls to g_i replaced by executions of the protocol π_{g_i}, Γ-securely evaluates f in the $(g_1, \ldots, g_{i-1}, g_{i+1}, \ldots, g_l)$-hybrid model.

Next, we describe some essential lemmas and existing protocols that we use.

Theorem 2 (Schwartz-Zippel)**.** *Let $P \in F[x_1, x_2, \ldots, x_n]$ be a non-zero polynomial of total degree $d > 0$ over a field F. Let S be a finite subset of F and let $r_1, r_2, \ldots, r_n$ be selected at random independently and uniformly from S. Then $\Pr[P(r_1, r_2, \ldots, r_n) = 0] \le \dfrac{d}{|S|}$.*

Lemma 1 (Smudging lemma)**.** *Let $B_1 = B_1(\kappa)$, and $B_2 = B_2(\kappa)$ be positive integers and let $\epsilon_1 \in [-B_1, B_1]$ be a fixed integer. Let $e_2 \in_R [-B_2, B_2]$ be chosen uniformly at random. Then, the distribution of e_2 is statistically indistinguishable from that of $e_2 + e_1$ as long as $B_1/B_2 = neg(\kappa)$.*

Lemma 1 is used for arguing statistical indistinguishability between two distributions.

Next, we list three existing zero-knowledge proofs that serve as building blocks in our system. They are all Σ protocols [67], which assume that the verifier is honest. However, they can be transformed into full zero knowledge, as we explain in detail later. Since this is taken from existing literature, we will not re-derive the simulators here, and instead assume access to simulators for all three protocols.

Protocol 1 (Paillier proof of plaintext knowledge)**.** *A protocol for proving plaintext knowledge of a Paillier ciphertext $\texttt{Enc}_{PK}(a)$ [62].*

Protocol 2 (Paillier multiplication proof)**.** *A protocol for: given $\texttt{Enc}_{PK}(\alpha)$, $\texttt{Enc}_{PK}(a)$, $\texttt{Enc}_{PK}(b)$, prove that $\texttt{Enc}_{PK}(b)$ indeed encrypts $\alpha \cdot a$ and that the prover has plaintext knowledge of $\texttt{Enc}_{PK}(\alpha)$ [62].*

Protocol 3 (Encryption interval proof)**.** *An efficient interval proof that proves an encrypted value lies within an interval.*

Note that in order to construct Protocol 3, we combine two existing protocols together. The first protocol is an interval proof for a committed value [36]. The second protocol is an additional proof that proves the equality of plaintexts under a commitment and an encryption [68].

To turn our honest verifier proofs into full zero knowledge (as well as non-malleable and concurrent), we utilize an existing transformation [91]. Informally, the transformation does two things. First, the random challenge in a Σ protocol is first generated by the verifier giving a challenge, then the prover proves an OR protocol given this challenge. The OR protocol consists of the actual statement to be proved, plus a signature proof. A simulator can simply simulate the second protocol in the OR protocol, instead of the main proof. Second, the witness for the proof is extracted via encrypting it under a public key generated from the common reference string. During the simulation, the simulator is able to generate the encryption key parameters, and can thus decrypt the encryption and extract an adversary's input. This means no rewinding is needed either for simulation or extraction. We denote the parameter generation functionality as f_{crs}. In Helen's design, we transform a Σ protocol S using this method, then use the simulator $S_{zk}(S)$ and extractor $E_{zk}(S)$ in our proofs for simulation and extraction of the adversary's secrets.

Finally, we assume that we have access to a threshold encryption scheme that can provably decrypt a ciphertext for our threshold structure. The scheme is described in [83], and we do not provide further proofs for this protocol. In our MPC simulation, we assume that we have a simulator S_{dec} for the decryption protocol.

5.10.2 Proofs

Theorem 3. *Protocol described in Section 5.3.1 is an honest verifier zero knowledge proof of plaintext knowledge for a committed matrix* $\texttt{Enc}_{PK}(\mathbf{X})$.

Proof sketch. The Paillier ciphertext proof of plaintext knowledge is a Σ protocol. Correctness, soundness, and simulation arguments are given in [62]. Using S_{zk} and E_{zk}, we can achieve full zero knowledge in the concurrent setting by simply proving knowledge for each element in the matrix using the ciphertext proof of plaintext knowledge. □

Theorem 4. *Gadget 1, with applied transformation from [91], is a zero knowledge argument for proving the following: given a public committed* $\texttt{Enc}_{PK}(\mathbf{X})$*, an encrypted* $\texttt{Enc}_{PK}(\mathbf{Y})$*, and* $\texttt{Enc}_{PK}(\mathbf{Z})$ *prove that the prover knows* $\mathbf{X}$ *and that* $Z = XY$ *under standard cryptographic assumptions.*

Proof sketch. To prove this theorem, we first prove the security of the honest verifier version of this protocol. The argument itself contains several parts. The first is a proof of plaintext knowledge of the matrix $\mathbf{X}$, which is applied straightforwardly from Theorem 3. This allows us to extract the content of the commitment. The second is a matrix multiplication proof, which consists of a reduction and ciphertext multiplication proofs.

Completeness is straightforward to see since we simply follow the computation of matrix multiplication, except that $\mathbf{Y}$ is encrypted. If $\texttt{Enc}_{\text{PK}}(\mathbf{Z}) = \mathbf{X}\texttt{Enc}_{\text{PK}}(\mathbf{Y})$, then $\mathbf{t}\texttt{Enc}_{\text{PK}}(\mathbf{Z}) = \mathbf{tX}\texttt{Enc}_{\text{PK}}(\mathbf{Y})$. Soundness can be proved in two steps. The first step utilizes the Schwartz-Zippel lemma Theorem 2. Given a random vector $\mathbf{t} \in [0, 2^l]$ where $l = |n| - 2|p| - \log M$, we will verify that $\mathbf{tZ} = \mathbf{tXY}$. This step reduces the problem to verifying matrix-vector multiplication instead of matrix-matrix multiplication. Using the lemma, we can view this as a multivariable polynomial equality testing problem. Therefore, an inequality will correctly pass with probability $d/|S|$. Since the commitments to $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ are homomorphic, the prover and the verifier can calculate $\texttt{Enc}_{\text{PK}}(\mathbf{tX})$ and $\texttt{Enc}_{\text{PK}}(\mathbf{tZ})$ independently. The second step of the soundness argument comes from the fact that after this transformation, we execute individual ciphertext multiplication proofs, which are Σ protocols themselves. These Σ-protocols are used for the individual products of a dot product (these are proved independently), as well as for proving the summation of these individual products. The summation itself can be computed by the verifier directly via the homomorphic properties of the ciphertexts. Then the summation proof is another Σ-protocol. Σ-protocols satisfy the special soundness property [67], which means that a cheating verifier can cheat with probability 2^{-t}, where t indicates the length of the challenge. Therefore, the probability of cheating is overall negligible via union bound. To simulate the matrix multiplication proofs, we first use the randomness on the verifier input tape to construct $\mathbf{t}$. Assuming that the simulator for Protocol 2 is S_{mult}, we then use the challenges for the ciphertext multiplication protocols and feed each into the simulator S_{mult} that simulates the Σ-protocol for ciphertext multiplication. Finally, to make this entire argument full zero knowledge, we apply the protocol transformation from [91]. □

Theorem 5. *Section 5.3.2, with applied transformation from [91], is a zero knowledge argument for proving the following: given committed matrices $\texttt{Enc}_{PK}(\mathbf{X})$, $\texttt{Enc}_{PK}(\mathbf{Y})$, and $\texttt{Enc}_{PK}(\mathbf{Z})$, prove that $\mathbf{XY} = \mathbf{Z}$ and that the prover knows the committed values X and Y under standard cryptographic assumptions.*

Proof sketch. The proof is a straightforward combination of Theorem 3 and Theorem 4. □

Theorem 6. *Gadget 3 has a simulator S_a such that S_a's distribution is statistically indistinguishable from Gadget 3's real world execution.*

Proof sketch. First, we must construct such a simulator S_a. To do so, we modify a similar simulator from [62]. Let M denote all of the parties in the protocol. The simulator runs the following:

1. Let s be the smallest index of an honest party and let $H^{'}$ be the set of the remaining honest parties. For each honest party in $H^{'}$, generate r_i and $\texttt{Enc}_{\text{PK}}(r_i)$ correctly. For party s, choose $r_s^{'}$ uniformly random from $[0, 2^{|p|+\kappa}]$, and let $\texttt{Enc}_{\text{PK}}(r_s) = \texttt{Enc}_{\text{PK}}(r_s^{'} - a)$.

2. Hand the values $\texttt{Enc}_{\text{PK}}(r_i)_{i \in H}$ to the adversary and receive from the adversary $\texttt{Enc}_{\text{PK}}(r_i)$.

3. Run the augmented proofs of knowledge $(S_{zk}(P_{pok}))$ from the adversaries and simulate the proof for party s (since the simulator does not know the plaintext value of r_s). If any proof fails from the adversary, abort. Otherwise, continue and use the augmented extractor $(E_{zk}(P_{pok}))$to extract the adversary's inputs r_i.

4. Compute $e = \sum_{i \neq s}(r_i) + r_s^{'} = \sum_i r_i + a$.

5. Simulate a call to decrypt using S_{dec}. Note that the decrypted value is exactly e due to the relation described in the previous step.

6. Simulator computes the shares as indicated in the original protocol except for party s, which sets its value to $a_s^{'} = (a_s - a)\mathsf{mod}p$.

We now prove that the simulator's distribution is statistically indistinguishable from the real world execution's distribution.

Note that other than $\texttt{Enc}_{\text{PK}}(r_s)$, the rest of S_a's simulation follows exactly from the real world execution A, and thus is distributed exactly the same as the execution.

In simulation step 1, $\texttt{Enc}_{\text{PK}}(r_s)$ encrypts r_s. Given a plaintext that is within $[0, p)$, using Lemma 1 we know that r_s and $r_s^{'}$ are statistically indistinguishable from each other. This means that both r_s's and $r_s^{'}$'s distributions are statistically close to being uniformly drawn from the interval $[0, 2^{|p|+\kappa}]$. Since we always blind encryptions, this means that $\texttt{Enc}_{\text{PK}}(r_s)$ is a random encryption of a statistically indistinguishable uniformly random element from $[0, 2^{|p|+\kappa}]$. Therefore, $\texttt{Enc}_{\text{PK}}(d_s)$'s distribution in the simulator is statistically indistinguishable from the corresponding distribution in the execution.

The distributions of the real world proofs and the simulated proofs follow straightforwardly from [91].

Finally, we know that $a_s^{'} = a_s - a\mathsf{mod}p$. The real world execution's share is a_s. Since r_s and $r_s^{'}$ are statistically indistinguishable from the uniformly random distribution and $a < p$, the r_s's and $r_s^{'}$'s distributions after applying modulo p are also statistically indistinguishable. Therefore, the $a_s^{'}$ and a_s distributions are also statistically indistinguishable.

□

Next, we first define f_{crs} and f_{SPDZ}, two ideal functionalities.

1. f_{crs}: an ideal functionality that generates a common reference string, as well as secret inputs to the parties. As mentioned before, this functionality is used for the augmented proofs so that the extractor can extract the adversary's inputs by simple decryption.

2. f_{SPDZ}: an ideal functionality that computes the ADMM consensus phase using SPDZ.

Theorem 7. *f_{ADMM} in the (f_{crs}, f_{SPDZ})-hybrid model under standard cryptographic assumptions, against a malicious adversary who can statically corrupt up to $m-1$ out of m parties.*

Proof sketch. To prove Helen's security, we first start the proof by constructing a simulator for Helen's two phases: input preparation and model compute. Next, we prove that the simulator's distribution is indistinguishable from the real world execution's distribution. Thus, we prove security in the $(f_{\text{crs}}, f_{\text{SPDZ}})$-hybrid model.

First, we construct a simulator S that first simulates the input preparation phase, followed by the model compute phase.

1. S simulates f_{crs} by generating the public key PK_{crs} and a corresponding secret key SK_{crs}. These parameters are used for the interactive proof transformations so that we are able to extract the secrets from the proofs of knowledge.

2. S next generates the threshold encryption parameters. The public key PK is handed to every party. The secret key shares $[\text{SK}]_i$ are handed to each party as well. Discard $[\text{SK}]_i$ for the honest parties.

3. Next, S starts simulating the input preparation phase. It receives matrix inputs, as well as interval proofs of knowledge from the adversary $\mathcal{A}$. It also generates dummy inputs for the honest parties, e.g., encrypting vectors and matrices of 0.

4. If the proofs of knowledge from $\mathcal{A}$ pass, then S extracts the inputs using the augmented extractors from Theorem 3. Otherwise, abort.

5. S hands the inputs from the adversary to the ideal functionality f_{ADMM}, which will output the final weights $\mathbf{w}_{\text{final}}$ to the simulator.

6. The first two steps in the input preparation phase utilize matrix plaintext multiplication proofs. For each honest party, S simply proves using its dummy inputs and simulates the appropriate proofs. S should also receive proofs from the adversary. If the proofs pass, continue. Otherwise, abort the simulation.

7. The simulator continues to step 3 of the input preparation phase. This step has two different proofs. The first proof is a matrix multiplication proof between $\mathbf{V}^{\mathbf{T}}$ and $\mathbf{V}$. This can be simulated like the previous steps. The next step is an element-wise interval proof with respect to the identity matrix. The simulator can again simulate this step using the simulator for the interval proof. Next, S verifies the proofs from the malicious parties. If the proofs pass, move on. Otherwise, abort the simulation.

8. Something similar can be done to prove and verify the step 4 of input preparation phase from each party, since the proofs utilized are similar to those used in step 3.

9. S now begins simulating the model compute phase.

10. Initialize the encrypted weights $(\mathbf{w}, \mathbf{z}, \mathbf{u})$ to be the zero vector.

11. for i in $admm\_iters$:

a) The first step in the iteration is the local compute phase. S simulates the honest parties by correctly executing the matrix multiplications using the dummy input matrices and the encrypted weight and produces both the encrypted results and the multiplication proofs from Protocol 2. S also receives a set of encrypted results and multiplication proofs from the adversary for the corrupted parties. S can verify that $\mathcal{A}$ has indeed executed the matrix multiplication proofs correctly. If any of the proofs doesn't pass, the simulator aborts. If not all of the ciphertexts are distinct, the simulator also aborts.

b) Now, S needs to simulate the additive sharing protocol for each party. This can be done by invoking the simulator from Theorem 6.

c) After the secret sharing process is complete, all parties need to publish encryptions of their secret shares. S publishes encryptions of these shares for the honest parties and the appropriate interval proofs of knowledge. The malicious parties also publish encrypted shares and their interval proofs of knowledge. If the interval proofs of knowledge do not pass for the adversary, then abort the computation. Otherwise, S extracts all of the encrypted shares from the adversary. Here, S also calculates the *expected* shares from the adversary. This can be calculated because S knows the randomness r_i's used by $\mathcal{A}$.

d) S needs to now simulate a call to the f_{SPDZ} oracle. The simulator first picks a random $\alpha \in \mathbb{Z}_p$, which serves as the global MAC key. Then it splits α into random shares and gives one share to each party. If S is generating shares for the input values, it will simply generate MACs $\gamma(a)_i$ for the shares it receives from running S_a. Otherwise, S then generates random SPDZ shares and MAC shares $\gamma(a)_i$. In both cases, the values shares and the MAC shares satisfy the SPDZ invariant: $\alpha(\sum_i a_i) - (\sum_i \gamma(a)_i)$.

e) Each party publishes encryptions of all SPDZ input and output shares, as well as their MAC shares. The simulator S will simulate the honest parties' output by releasing those encryptions and interval proofs of knowledge. S also receives the appropriate encryptions and interval proofs of knowledge from the adversary. Run the extractor to extract the contents of the adversary. Again, keep track of the shares distributed to the adversary, as well as the extracted shares that were committed by the adversary.

f) If the iteration is the last iteration, then S simulates a call to SPDZ by splitting the $\mathbf{w}_{\text{final}}$ shares into random shares, as well as creating the corresponding MAC shares to satisfy the relation with the global key α.

12. After the iterations, S needs to simulate the MPC checks described in Section 5.5.5, where we need to prove modular equality for a set of encrypted values and shares. S runs the following in parallel, for each equality equation that needs to be proven:

- In the original protocol, we have $\texttt{Enc}_{\text{PK}}(a), \texttt{Enc}_{\text{PK}}(b_i), \texttt{Enc}_{\text{PK}}(c_i)$ where we want to prove that $a \equiv \sum_i b_i \mathsf{mod} p$ and $\alpha(\sum_i b_i) \equiv \sum_i c_i \mathsf{mod} p$. First, S simulates the proof of the first equality. S follows the protocol by verifying the interval proofs of knowledge from every party. If a malicious party's proof fails, then abort.
- S computes the cipher $\texttt{Enc}_{\text{PK}}(a - \sum_i b_i)$ by directly operating on the known ciphertexts.
- S follows the protocol described in Section 5.5.5 by following the protocol exactly. S picks random s_i's and generates interval proofs of knowledge. S also receives $\texttt{Enc}_{\text{PK}}(s_i)$ from the adversary and the corresponding interval proofs of knowledge. Extract the $\texttt{Enc}_{\text{PK}}(s_i)$ values from the adversary if the proofs are verified.
- Before S releases the decrypted value, it needs to know what value to release. To do so, S needs to compare an encrypted value a and its secret shares b_i (each party P_i retains b_i). We want to make sure that $a \equiv \sum_i b_i \mathsf{mod} p$. While the simulator does not know a, it does know some information when a was first split into shares. More specifically, S knows the adversary's generated randomness during the additive secret sharing, the decrypted number e, as well as the shares committed by the adversary. The equation $a + \sum_i r_i \equiv e \mathsf{mod} p$ holds because everyone multiplies the published $\texttt{Enc}_{\text{PK}}(r_i)$ with $\texttt{Enc}_{\text{PK}}(a)$ to get $\texttt{Enc}_{\text{PK}}(a + \sum_i r_i)$, and the decryption process is simulated and verified. This means that $a \equiv e - \sum_i r_i \mathsf{mod} p$. Let's assume that an adversary alters one of its input shares b_j to $b_j^{'}$ for parties in set $\mathcal{A}$. Then the difference between a and the b_i shares is simply

$$\begin{aligned}
a - \sum_{i \not\subset \mathcal{A}} b_i - \sum_{i \subset \mathcal{A}} b_i^{'} \mathsf{mod} p &\equiv e - \sum_i r_i - \sum_{i \not\subset \mathcal{A}} b_i - \sum_{i \subset \mathcal{A}} b_i^{'} \mathsf{mod} p \\
&\equiv (e - r_0) + \sum_{i \neq 0} (-r_i) - \sum_{i \not\subset \mathcal{A}} b_i - \sum_{i \subset \mathcal{A}} b_i^{'} \mathsf{mod} p \\
&\equiv b_0 + \sum_{i \neq 0} b_i - \sum_{i \not\subset \mathcal{A}} b_i - \sum_{i \subset \mathcal{A}} b_i^{'} \mathsf{mod} p \\
&\equiv \sum_i b_i - \sum_{i \not\subset \mathcal{A}} b_i - \sum_{i \subset \mathcal{A}} b_i^{'} \mathsf{mod} p \\
&\equiv \sum_{i \subset \mathcal{A}} (b_i - b_i^{'}) \mathsf{mod} p
\end{aligned}$$

  Hence, the difference modulo p is simply the the difference in the changes in the adversary's shares, and is completely independent from the honest parties' values. Let this value be v.
- The next step is simple: S simply simulates S_{dec} and releases the value $v + \sum_i (s_i p)$.
- S can follow a similar protocol for checking the SPDZ shares b_i and the MACs c_i.

13. Finally, if the previous step executes successfully, then the parties will release their plaintext shares of $\mathbf{w}_{\text{final}}$ by decommitting to the encrypted ciphers of those shares and publishing their plaintext shares.

We now prove that the distribution of the simulator is statistically indistinguishable from the distribution of the real world execution. To do so, we construct hybrid distributions.
Hybrid 1 This is the real world execution.
Hybrid 2 Same as hybrid 1, except replace the proofs from the input preparation phase with simulators.

Hybrid 1 and 2 are indistinguishable because of the properties of the zero-knowledge proofs we utilize (see Theorem 3 and Theorem 4).
Hybrid 3 Same as the previous hybrid, except the rest of the proofs are run with the simulated proofs instead, and the secret sharing is replaced by the simulator S_a. However, step 12 is still run with the real world execution.

Hybrid 2 and 3 are statistically indistinguishable because of the properties of the zero-knowledge proofs (Theorem 3, Theorem 4), and the fact that the secret sharing is also simulatable (Theorem 6).
Hybrid 4 Same as the previous hybrid, except swap out the real world execution with step 12 described by the simulator.

Hybrid 3 and 4 are statistically indistinguishable. The abort probabilities in step 12 are based on the release values, so we just need to argue that the release value distributions are statistically indistinguishable.

Since the simulator is always able to extract the adversary's values, it will always be able to calculate the correct answer in step 12. The real world execution, on the other hand, could potentially have a different answer if any of the zero-knowledge proofs fails to correctly detect wrong behavior. Therefore, if the zero-knowledge proofs are working correctly, then:

1. If the execution does not abort, then the decrypted values must be both divisible by p. The two values are statistically indistinguishable because of Lemma 1.

2. If the execution does abort, then S's output value would be v. If the proofs pass, then from the argument made in step 12 where we know that the decrypted value modulo p is exactly the same as v. Furthermore, if we subtract v from the two values, the new values have the same distribution by the argument in the prior step.

If any proof does fail to detect a malicious adversary cheating, then the released answer could be potentially different. However, this will happen with negligible probability because of the properties of the zero-knowledge proofs we are using. Therefore, the hybrids are statistically indistinguishable.
Hybrid 5 First, define $\boxplus$ to be ciphertext addition, $\boxminus$ to be ciphertext subtraction, and $\boxdot$ to be ciphertext multiplication. Replace the input encryptions of the honest parties with encryptions where $\texttt{Enc}_{\text{PK}}(x_i)$ is transformed into $\texttt{Enc}_{\text{PK}}(0)$ is transformed into $\text{Blind}(\texttt{Enc}_{\text{PK}}(0) \boxdot \texttt{Enc}_{\text{PK}}(b)) \boxplus (\texttt{Enc}_{\text{PK}}(x_i) \boxdot (\texttt{Enc}_{\text{PK}}(1) \boxminus \texttt{Enc}_{\text{PK}}(b)))$, where $b = 0$. Hybrid 4

and 5 are computationally indistinguishable because the inputs used by the honest parties have not changed from the previous hybrid. With the guarantees of the encryption algorithm, the encrypted ciphers from the two hybrids are also indistinguishable.

Hybrid 6 Replace the input encryptions of the honest parties with encryptions of 0 (so same as the simulator), except with additional randomizers such that the encryption of an input $\texttt{Enc}_{\text{PK}}(0)$ is transformed into $\text{Blind}(\texttt{Enc}_{\text{PK}}(0) \boxdot \texttt{Enc}_{\text{PK}}(b)) \boxplus (\texttt{Enc}_{\text{PK}}(x_i) \boxdot (\texttt{Enc}_{\text{PK}}(1) \boxminus \texttt{Enc}_{\text{PK}}(b)))$, where $b = 1$.

Hybrid 5 and 6 are indistinguishable because one could use a distinguisher D to break the underlying encryption scheme. Since the encrypted ciphertexts are randomized and only differ by the value of b (whether it is 0 or 1), if one were to build such a distinguisher D, then D can also distinguish whether $b = 0$ or $b = 1$. This breaks the semantic encryption scheme.

Hybrid 7 This is the simulator's distribution.

Hybrid 6 and 7 are indistinguishable because the inputs are distributed exactly the same (they are all 0's).

This completes our proof.

□

5.11 Conclusion

In this chapter, we proposed Helen, a coopetitive system for training linear models. Compared to prior work, Helen assumes a stronger threat model by defending against *malicious* participants. This means that each party only needs to trust itself. Compared to a baseline implemented with a state-of-the-art malicious framework, Helen is able to achieve up to five orders of magnitude of performance improvement. Given the lack of efficient maliciously secure training protocols, we hope that our work on Helen will lead to further work on efficient systems with such strong security guarantees.

Chapter 6

Cerebro

While a system like Helen is able to train regularized linear models very efficiently, it does not address two major obstacles faced by collaborative learning systems to be deployed in the real world. The first obstacle is the tussle between generality and performance. Many recent papers on MPC for collaborative learning [162, 97, 219, 146, 127, 186, 202, 168, 112, 93, 55, 10] focus on hand-tuning MPC for specific learning tasks. While these protocols are highly optimized, this approach is not easily generalizable for a real world deployment. On the other hand, there exist *generic* MPC protocols [24, 103, 215, 69, 211] that can execute arbitrary programs. However, there are many such protocols (most of which can be divided into sub-protocols [130, 132]), and choosing the right combination of tools as well as optimizations that result in an efficient secure execution is a difficult and daunting task for users without a deep understanding of MPC.

The second obstacle lies in the tussle between privacy and transparency. The platform needs to ensure that it addresses the organizations' incentives and constraints for participating in the collaborative learning process. For example, in the secure collaborative training scenario, while MPC guarantees that nothing other than the final model is revealed, this privacy property is also problematic because the parties effectively lose some control over the computation. They cannot observe the inputs or the computation's intermediate outputs before seeing the final result. In this case, some parties might worry that releasing a jointly trained model will not increase accuracy over their own models, but instead help their competitors. They might also have privacy concerns, such as whether the model itself contains too much information about their sensitive data [44, 200, 87, 17, 86, 212, 194] or whether the model is poisoned with backdoors [53].

This chapter presents Cerebro, a platform for multi-party cryptographic collaborative learning using MPC. Cerebro's goal is to address the above two obstacles via a holistic design of an *end-to-end* learning platform, as illustrated in Figure 6.1. To address the aforementioned challenges, Cerebro first exposes a Python-like domain specific language (DSL) and a machine learning API to the users. Cerebro can then automatically compile an optimized secure n-party protocol for any program written by the user in this DSL.

The second challenge is addressed by introducing a set of mechanisms that allows or-

System	n-party	DSL & API	Policies	Automated optimization	Multiple backends	Auditing
Specialized ML protocols	✓/ ✗	✗	✗	✗	✓/ ✗	✗
Generic MPC	✓	✗	✗	✗	✗	✗
MPC compilers	✓/ ✗	✓	✗	✓	✓/ ✗	✗
Cerebro	✓	✓	✓	✓	✓	✓

Table 6.1: Comparison with prior work in categories that are necessary in a collaborative learning platform. There is a lot of prior work in specialized MPC protocols [162, 97, 219, 146, 127, 186, 202, 168, 112, 93, 55, 10], generic MPC [24, 103, 215, 69, 211], and MPC compilers [210, 145, 196, 183, 46, 41, 11, 217, 164, 85, 74, 85, 207]. Since the work space is so broad, we use "✓/ ✗" to indicate that only some systems in this category support that feature.

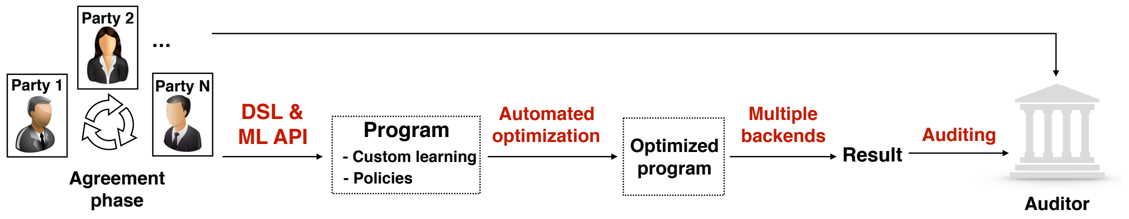


Fig. 6.1: The Cerebro workflow.

ganizations to ensure that their incentives and constraints are met before the result of a learning task is available, and also enables participants to identify the source of malicious and ill-formed input data. Our insight is that we can leverage cryptographic primitives to enable this functionality without leaking additional data in the process. Based on this observation we define two important mechanisms: *compute policies* and *cryptographic auditing*. Compute policies allow parties to provide code that controls when and how the result of a learning task is released; while cryptographic auditing allows parties to backtrack and audit the inputs used during private computation, thus holding all parties accountable for their actions.

6.1 Overview of Cerebro

6.1.1 Threat model

We consider P parties who want to compute a learning function on their sensitive data. The parties are unwilling or unable to share the plaintext data with each other, but want to release the result of the function (e.g., a model or a prediction) according to some user-defined policies. We assume that the parties come together in an *agreement* phase during which they decide on the learning task to run, the results they want to disclose to each other, and the policies they want to implement. We assume this agreement is enforced by an external mechanism, e.g., through a legal agreement.

Cerebro allows the parties to choose what threat model applies to their use case by supporting both semi-honest and malicious settings. In the semi-honest setting, Cerebro is able to protect against an adversary who does not deviate from protocol execution. This adversary can compromise up to $P - 1$ of the parties and analyze the data these parties receive in the computation, in hopes of learning more information about the P-th honest party's data beyond the function result. In the malicious setting, the adversary can not only compromise a majority of the parties, but also cause participants to deviate from the protocol. The misbehavior includes altering the computation and using inconsistent inputs. Cerebro is able to support both settings by utilizing different generic cryptographic backends. We believe that it is useful to support a flexible threat model because different organizations' use cases result in different assumptions about the adversary. Moreover, as we show in Section 6.5, the semi-honest protocol can be $61 - 3300\times$ faster than the malicious counterpart, so the participants may not wish to sacrifice performance.

Recent work has described many attacks that target machine learning. One category is data poisoning [53], attacks where the parties inject malicious data into the training process. Another category is attacks on the released result, where an attacker learns about the training dataset from the model [194, 195, 149] or steals model parameters from prediction results [200, 87, 17, 86, 212]. By definition, MPC does not protect against such attacks, and Cerebro similarly cannot make formal guarantees about maliciously constructed inputs or leakage from the result. However, we try to address these issues via an end-to-end design of the system, where Cerebro provides a platform for users to program compute policies and add cryptographic auditing (explained in Section 6.3).

6.1.2 System workflow

Cerebro's pipeline is composed of multiple components, as illustrated in Figure 6.1. For the rest of this section, we provide an overview of a user's workflow using Cerebro.

Agreement phase This phase is executed before running Cerebro. During the agreement phase, the potential participants come together and agree to participate in the computation. We assume that the number of participants is on the order of tens of parties instead of millions of parties. Parties need to agree on the computation (including the learning task and any compute policies) to run and on the threat model. Parties should also establish a public key infrastructure (PKI) to identify the participants.

Programming model Users make use of Cerebro's Python-like domain-specific language (DSL) to write their programs. Users can easily express custom learning tasks as well as policies using our DSL and APIs. Cerebro also allows users to specify the computation configuration parameters, such as the number of parties, how much data is contributed by each party, etc.

Compute policies Cerebro supports user-defined compute policies via our DSL to handle concerns arising from the complex economic relationships among the parties. Compute policies can be generic logic for how results are obtained, or special *release policies* such that the result of a computation is only revealed if the policy conditions are satisfied.

Cryptographic compiler Cerebro's cryptographic compiler is able to generate an efficient secure execution plan from a given program written in the Cerebro DSL. Our compiler first applies *logical optimization* directly on the program written in our DSL (see Section 6.2.2). Next, this optimized program is input to the *physical planning* stage (see Section 6.2.3) to generate an efficient physical execution plan.

Secure computation In this phase, Cerebro executes the secure computation using the compiler's physical plan. When it finishes, the parties can jointly release the plaintext result.

Cryptographic auditing Even after the result is released, the learning life cycle is not finished. Cerebro gives the parties the ability to audit each other's inputs with a third-party auditor in a post-processing phase (see Section 6.3.2).

6.2 Programming and Execution Model

In this section we describe Cerebro's programming and execution model. Users specify programs that Cerebro can execute using a domain-specific language (Section 6.2.1, which is then used as input to the Cerebro compiler (Figure 6.3). The Cerebro compiler implements two logical optimization passes, which *reduce the amount of computation expressed in MPC* while preserving security guarantees. Finally, the Cerebro physical planner (Section 6.2.3) takes the logical plan generated by the compiler, and uses information about the physical deployment to instantiate and execute the plan.

6.2.1 Cerebro DSL

In Cerebro, users express training and inference algorithms, compute policies, and auditing functions using a Python-like domain specific language (DSL). Our DSL supports a variety of numerical data types that are commonly used in machine learning, data collections, and generic functions that are useful for easily expressing training and inference algorithms. Figure 6.2 shows an example program.

Data types Each variable in a Cerebro program is automatically tagged with a type (integer, fixed-point, etc.) and a security level, the latter indicating which parties can access the raw value of the variable. Cerebro currently supports three security levels:

- *Public*: the value is visible to all parties

```
# Set the fixed-point parameters
Params.set_params(p=64, f=32, k=64, num_parties=2)
# Decision tree prediction: party 0 inputs the model
tree = p_fix_mat.read_input(tree_size, 4, 0)
# Party 1 inputs the features
x = p_fix_array.read_input(dim, 1)
...
for i in range(LEVELS-1):
    # Load node information: index, split
    ...
    cond = (x[index] < split)
    # This is a fused operation
    root = secret_index_if(cond, tree, left_child, right_child)
reveal_to_all(root[1], "Prediction")
```

Fig. 6.2: A sample program written in Cerebro's DSL

- *Private*: the value is visible to a single party
- *Secret*: the value is hidden from all parties

Our current implementation restricts private variables to being owned and visible to a single party, and we represent a private value visible to the party i as `private(i)`. The security level of variables is automatically upgraded based on type inference rules, described in Section 6.2.2.1. Programs can explicitly downgrade security levels by calling `reveal`.

Functions Our DSL provides a set of mathematical and logical operators to process tagged data. Each operator can accept inputs with any security tag, and the output tag is determined using a set of type inference rules (explained more in Section 6.2.2.1). Security annotations also play an important role in enabling several of the optimizations employed by Cerebro.

Cerebro provides a variety of basic operators over data types including arithmetic operations and comparisons. Users can compose these basic operators to implement user-defined learning algorithms. Cerebro also provides a set of higher-level mathematical operators common to machine learning tasks (e.g., linear algebra operators, sigmoid), functions for efficiently indexing into arrays or matrices, a set of branching operators, and a set of more complex fused operators. Fused operators (explained in Section 6.2.2.2) provide Cerebro with more opportunities to optimize complex code patterns.

6.2.2 Logical optimization

Given a program written in the Cerebro DSL, the Cerebro compiler is responsible for generating a logical execution plan that minimizes runtime. In Cerebro we develop and implement two logical optimization mechanisms that are particularly useful for machine learning tasks: the first is *program splitting*, where a program Q is split into two portions

Q_1 and Q_2 such that Q_1 can be executed in plaintext, while Q_2 is executed using secure computation. The second optimization is operator fusion, where the compiler tries to detect pre-defined compound code patterns in Q_2 and transforms them to more efficient fused operations.

6.2.2.1 Program splitting

Program splitting is a type of logical optimization that delegates part of the secure computation to one party which computes *locally in plaintext*. We can illustrate this optimization by applying to sorting. If a program needs to sort training samples from all parties (e.g., in decision tree training), then parties can instead pre-sort their data. This allows the MPC computation to merely merge pre-sorted data, providing a significant speedup over a case where it executes the entire sort algorithm in the secure computation.

In the semi-honest setting, Cerebro can *automatically identify* opportunities for local computation within the code. As explained in Section 6.2, users write their programs using Cerebro's API's, and the compiler automatically tags their data using Cerebro's secure types. Cerebro uses a set of rules (see table 6.2) to infer a function's security level. If a function only has public input, then the output should also be public since it can be inferred from inputs. This type of computation can be executed in plaintext by any party. Similarly, if a function only takes input from a single party i, party i can compute this function locally in plaintext. However, if a function's input includes private data from different parties or secret data, then the function needs to be executed using MPC, and the output will also be tagged as secret.

However, in a malicious setting the criteria for secure local plaintext execution are more complex because a compromised participant can arbitrarily deviate from the protocol and substitute inconsistent/false data and/or compute a different function. Therefore, we cannot assume that a party will compute correct values locally. Using the same sort example: we can no longer trust the parties to correctly pre-sort their inputs. Therefore, secure computation must add an extra step to ensure that the input from each party is sorted.

In general, automatically finding efficient opportunities for local plaintext computation in the malicious setting is challenging. In Cerebro, we approach this problem by designing pre-defined APIs with this optimization in mind. If a user uses our API, Cerebro will apply program splitting appropriately while guaranteeing security in the malicious threat model. For example, our `sort` API will automatically group the inputs into private inputs from each party, followed by a local plaintext sort in plaintext at each party. However, since a malicious party can still try to input unsorted data into the secure computation, the global sort function will first check that the inputs from each party are sorted.

Using this optimization allows Cerebro to automatically generate an efficient MPC protocol that has similar benefits to prior *specialized* work. For example, in [168], one of the techniques is to have the parties pre-compute the covariance matrix locally, then sum up these matrices using linearly homomorphic encryption. While Cerebro's underlying cryptography is quite different – hence resulting in a very different overall protocol – we are able to automatically discover the same local computation splitting as is used by a specialized system written for

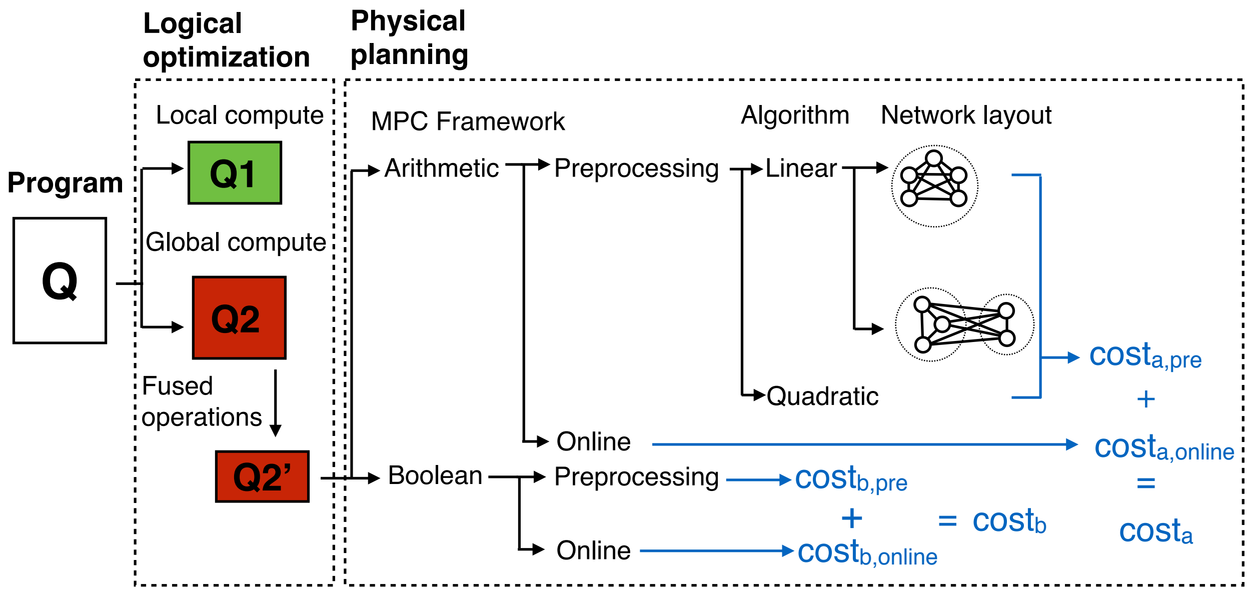


Fig. 6.3: Cerebro architecture, showing choices we can make under the semi-honest threat model.

ridge regression. We note that program splitting is compatible with cryptographic auditing mentioned in Section 6.3.2.1 by committing to the precomputed local data instead of the original input data.

6.2.2.2 Fused operations

Recognizing compound code patterns is crucial in MPC, since many compound operations that are cheap in plain text incur significant performance penalties when executed securely. For example, plaintext array indexing under the RAM model has a constant cost. In MPC, while array indexing using a public index has constant cost, array indexing using a *secret variable* takes time that is proportional to the length of the array. This is because when executing secure computation, the structure of the function cannot depend on any private or secret value, otherwise a party may infer the value from the structure of the computation. Therefore, it is impossible to index an array using a secret value in constant time.

In Cerebro, as is common, we index arrays by linearly scanning the entire array, which is an $O(n)$ operation. [1] Next, consider a compound code pattern that occurs in programs like decision tree prediction (see Figure 6.2): an if/else statement that wraps around multiple secret accesses to the same array. In a circuit-based MPC, all branches of an if/else statement

[1]Cerebro can be augmented to use oblivious RAM (ORAM) for secret indexing, which has $O(polylogn)$ overhead for an array of size n. Prior work has shown that for smaller arrays, linear scanning is faster [209] because ORAM needs to keep a non-trivial amount of state [101, 104].

need to be executed. Therefore, conditionally accessing an index can require several scans through the same array.

For this scenario, Cerebro will combine the operators into a single fused operation `secret_index_if` that can be used to represent such conditional access and minimizes the number of array scans required during computation. Fused operators in Cerebro play the same role as level 2 and level 3 [78] operations in BLAS [29] and MKL [117], and fused operations generated by systems such as Weld [174], i.e., they provide optimized implementations of frequently recurring complex code patterns. Since operator fusion only happens on code expressed in MPC and preserves the functionality, it works for both the semi-honest and the malicious settings.

6.2.3 Physical planning

Once a logical plan has been generated, Cerebro determines an efficient physical instantiation of the computation, which can then be executed using one of Cerebro's MPC backends. We call this step physical planning (illustrated in Figure 6.3 on the right side) and describe it in this section. When converting logical plans into physical implementations, Cerebro must decide whether to use operations provided by existing boolean and arithmetic MPC protocols or to use our special vectorized primitives (Section 6.2.3.2). To choose between these implementation options, Cerebro uses a set of cost models (Section 6.2.3.3) to predict the performance of different implementation choices and picks the best among these choices. Finally, once a physical implementation has been selected, Cerebro decides where to place (Section 6.2.3.4) computation among available nodes – this choice can significantly impact performance in the wide area setting.

6.2.3.1 Notation

Let P denote the number of parties, and let $\mathcal{P}_i$ denote the i-th party. We use N to represent the total number of gates in a circuit. N_m is the number of **m**ultiplication gates in an arithmetic circuit; N_a is the number of **A**ND gates in a boolean circuit. $B_{(\cdot)}$ represents network bandwidth parameters and $l_{(\cdot)}$ represents latency parameters. For a given type of encryption algorithm $C_{(\cdot)}$, we use $|C_{(\cdot)}|$ to represent the number of bytes in a single ciphertext. We use c to capture any constant cost in a cost model, like an initialization cost. The rest of the cost can be categorized as *compute* (represented using f_i functions) and *network* costs (represented using g_i functions).

6.2.3.2 Vectorization

Cerebro supports compilation to two main MPC backends: arithmetic [69] and boolean [211]. Both backends consist of two phases: preprocessing and online. During the preprocessing phase, random elements are computed and can be used later during the online phase. Preprocessing is also especially interesting because it can be executed before the parties' private inputs are available.

In arithmetic MPC preprocessing, parties need to compute multiplication triples, which are used to speed up multiplication operations during the online phase. However, many common machine learning tasks contain matrix multiplication, which is especially costly because of the large number of multiplication operations. In this section, we introduce an optimization for arithmetic MPC preprocessing that allows us to vectorize multiplication triple generation. This idea was introduced in prior work for the semi-honest two party setting [162], and here we generalize the algorithm to the n-party semi-honest setting.

The two-party vectorized protocol happens in the preprocessing phase where it computes random matrix multiplication triples such that each $\mathcal{P}_i$ holds $A_j^{(i)}, B_j^{(i)}, C_j^{(i)}$ where $\sum_i (A_j^{(i)} \cdot B_j^{(i)}) = \sum_i C_j^{(i)}$. For the sake of a simpler analysis, we assume that B_j is a vector $\mathbf{b}$, and that the relation is $\mathbf{c} = A\mathbf{b}$. To generalize this to the multi-party setting, we can apply the two-party protocol in a pairwise fashion to generate the triples. To compute the triple, it suffices for each party to first sample random $A^{(i)}$ and $\mathbf{b}^{(i)}$, then use the two-party protocol to compute the pairwise products $A^{(i)} \cdot \mathbf{b}^{(j)}$.

6.2.3.3 Cost models

In this section, we provide two examples of the different cost models in Cerebro (see Section 6.8.1 for more).

Preprocessing planning As previously stated, Cerebro's MPC backends consist of preprocessing and online phases. Semi-honest arithmetic MPC has two different preprocessing protocols: *linear preprocessing* and *quadratic preprocessing* [69, 132]. We describe the high-level protocols in Section 6.8.2. These two methods can behave quite differently under different setups, and we illustrate this by presenting their cost models. We define C_l to be the encryption algorithm used in linear preprocessing, and C_q to be encryption algorithm used in quadratic preprocessing. The per-party cost model for linear preprocessing is given by:

$$\begin{aligned} &c + N_m(f_1(|C_l|) \\ &\quad + \frac{1}{P}[f_2(|C_l|)(P-1) + f_3(|C_l|) + g(B, |C_l|)(P-1)]) \end{aligned} \tag{6.1}$$

The per-party cost model for quadratic preprocessing is:

$$c + N_m(P-1)(f(|C_q|) + g(B, |C_q|)) \tag{6.2}$$

In terms of the scaling in the number of parties, linear preprocessing is much better than quadratic preprocessing. However, since $|C_q| < |C_l|$, quadratic preprocessing's encryption algorithm uses less computation and consumes less bandwidth.

Cost of vectorization The cost model to preprocess a matrix-vector multiplication for $(m, n) \times (n, 1)$ is:

$$\begin{aligned} c + f_1(|C_q|)(n + m(P-1)) + f_2(|C_q|)m(P-1) \\ + g(B, |C_q|)(m+n)(P-1) \end{aligned} \tag{6.3}$$

Comparing this cost model to eq. (6.2) (where we replace N_m with mn), the triple generation load is reduced from mn to m or $m+n$. We note that vectorization not only speeds up triple generation, but also introduces another planning opportunity if a program has a mix of matrix multiplication and regular multiplication.

6.2.3.4 Layout optimization

In the wide area network setting, different physical layouts can significantly impact the performance of a protocol. In this section, we give an example of *layout optimization*, where Cerebro plans an alternative communication pattern for parties that span multiple regions.

In the semi-honest setting, linear preprocessing requires a set of coordinators that aggregate data from all parties. The coordinators can be trivially load balanced among all parties by evenly distributing the workload. However, this only works when the pairwise communication costs are similar, and no longer works when the parties are located in different regions.

We make the observation that the underlying algorithm requires coordinators to perform an aggregation operation. Therefore, we introduce *two-level hierarchical layout*, where the coordination happens at both the intra-region and the inter-region levels. Each triple is still assigned to a single global coordinator, and is also additionally assigned a *regional coordinator* that is in charge of partially aggregating every party's data from a single region and sending the result to the global coordinator.

Assumptions We assume that the regions are defined by network bandwidth. The regions can be manually determined based on location, or automatically identified by measuring pairwise bandwidth and running a clustering algorithm. For a more detailed analysis (including a walkthrough of the derivation for two parties, please see Section 6.8.3).

Given k regions, let B_{ij} denote the bandwidth between regions i and j and let B_i denote the bandwidth within region i. Let n_i denote the number of triples assigned to each party in region i and $\mathcal{P}_i$ be the number of parties in region i. As before, we have $\sum_{i=1}^{k} n_i \cdot P_i = N_m$. The cost function can now be formulated as $C = L'_1 + L'_2 + L'_3$ where the constants are analogous to those in the previous example of two regions. We generalize the constants as follows:

$$\begin{aligned} L'_1 &= \max\left(\textstyle\sum_{j \neq i} \left(\frac{n_j \cdot P_j (P_i - 1)}{P_i B_i}\right)\right) i = 1, 2, ...k, \\ L'_2 &= \max\left(\frac{n_i \cdot (P_i - 1)}{B_i}\right) i = 1, 2, ..., k, \\ L'_3 &= \max\left(\textstyle\sum_{j \neq i} \left(\frac{n_j \cdot P_j}{B_{ij}}\right)\right) i = 1, 2, ..., k \end{aligned}$$

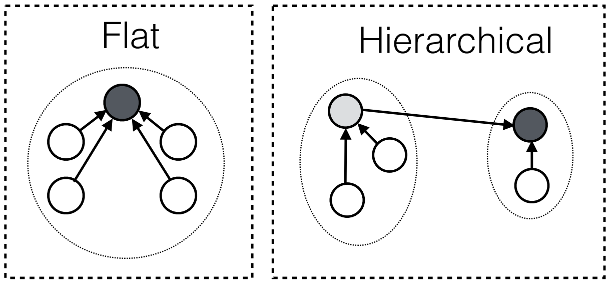


Fig. 6.4: The arrows show the aggregation communication pattern for a single multiplication triple. The shaded nodes represent coordinators.

We can transform the optimization problem into a linear program by moving the max into the constraints.

$$\begin{aligned}
&\min(L_1' + L_2' + L_3') \text{ s.t.} \\
&\textstyle\sum_{i=1}^{k} n_i \cdot P_i \geq N_m \\
&L_1' \geq \textstyle\sum_{j \neq i} \left(\frac{n_j \cdot P_j (P_i - 1)}{P_i B_i} \right) i = 1, 2, ...k, \\
&L_2' \geq \frac{n_i \cdot (P_i - 1)}{B_i} i = 1, 2, ..., k, \\
&L_3' \geq \textstyle\sum_{j \neq i} \frac{n_j \cdot P_j}{B_{ij}}, i = 1, 2, ..., k
\end{aligned}$$

We loosen the first constraint to be an inequality rather than an exact equality to make it easier to find feasible solutions since we require the n_i's to be integral. We solve this optimization problem in cvxpy [76, 7]. As an example, a setting with five regions is solved in roughly 100 milliseconds on a standard laptop computer.

6.3 Policies and auditing

In the collaborative learning setting, an end-to-end platform needs to take into account the incentives and constraints of the participants. This is critical when competing parties want to cooperate to train a model together. For example, the participants may be concerned about each other's behavior during training, as well as the costs and benefits of releasing the final model to other parties. A party may want to make sure that the economic benefits accrued by its competitors do not greatly outweigh its own benefits. Thus, a collaborative learning platform needs to allow participants to specify their incentives and constraints and also needs to ensure that both are met.

Cerebro addresses this problem by introducing the notion of user-defined *compute policies* and a framework for enabling *cryptographic auditing*. Compute policies are executed as part of the secure computation and are useful for integrating extra pre-computation and post-computation checks before the result is released. Auditing is executed at a later time after the result is released and can make parties accountable for their inputs to the original

```
def release_policy(prediction_fn, test_data, weights, tau):
    score = prediction_fn(data, weights)
    return (score > tau)
# Make a call to release_policy
if_release = release_policy(lr_prediction, vdata, weights, min_score)
# Set weights to 0 if if_release if false
final_weights = release(if_release, weights)
return final_weights
```

Fig. 6.5: Example validation-based release policy

secure computation. In the rest of this section, we give an overview of how users can use our system to encode policies and audit cryptographically.

6.3.1 Compute policies

6.3.1.1 Overview

We first make the observation that secure computation can enable *user-defined compute policies* that can be used to dictate how the result of a computation is released. In fact, MPC's security guarantees means that it can also be used to *conditionally release the computation result*. This simple property is very powerful because users can Cerebro provides an easy way for users to write an arbitrary release policy by first writing as a function that returns a boolean value `if_release`. call our `release` API on this boolean value and the result of the learning task. If `if_release` is true, then `release` will return the real result; otherwise it will return 0 values, thus un-releasing the result. Figure 6.5 shows an example policy written in Cerebro.

We assume that policy *functions* are public, and that all participants must agree on them during the agreement phase. This workflow allows participants to verify that each other's policy conforms to some constraints before choosing to input private data and dedicate resources for the secure computation. However, the constants/inputs for these policies can be kept private using MPC (e.g., a training accuracy threshold).

Since our DSL is generic, the participants can program any type of policy. We focus on two major categories of policies – validation-based policies and privacy policies – and how they can be encoded in our DSL.

6.3.1.2 Validation-based policy

In training, model accuracy can be a good metric of economic gains/losses experienced by a participant since it is usually the objective that a party seeks to improve via collaborative learning. In a single party environment, the metric is commonly computed by measuring the prediction accuracy on the trained model using a held-back dataset. When constructing validation-based policies in Cerebro, each party provides a test dataset in addition to their training dataset and provides a prediction function. We now describe some examples:

Threshold-based validation In this policy, party i wants to ensure that collaborative training gives better accuracy than what it can obtain from its local model. The policy takes in the model w, a test dataset $X_{t,i}$, as well as a minimum accuracy threshold τ_i. This policy runs prediction on $X_{t,i}$ and obtains an accuracy score. If this score is greater than τ_i, then the policy returns true. See example code in Figure 6.5.

Accuracy comparison with other parties In this policy, party i's decision to release depends on how much its competitors' test accuracy scores improve. Therefore, the inputs to this policy are: the model w, every party's test dataset $X_{t,j}$, every party's local accuracy scores a_j, and a percentage x. The policy runs prediction on every party's test dataset and obtains accuracy scores b_j. Then it checks b_j against a_j, and will only return true if $b_j - a_j < x(b_i - a_i)$ for all $j \neq i$.

Cross validation Since the parties cannot see each other's training data, it is difficult to know whether a party has contributed enough to the training process. All parties may agree to implement a policy such that if a party does not contribute enough to training, then it also does not receive the final model. Such a party can be found by running *cross validation*, a common statistical technique for assessing model quality. In this setting, Cerebro treats the different parties as different partitions of the overall training dataset and takes out a different party every round. The training is executed on the leftover $P - 1$ parties' data, and an accuracy is obtained using everyone's test data. At the end of P rounds, the policy can find the round that results in the highest test accuracy. The party that is *not* included in this round is identified as a party that contributed the least to collaborative training.

6.3.1.3 Privacy policy

For training tasks, the secure computation needs to compute and release the model in plaintext to the appropriate participants. Since the model is trained on everyone's private input, it must also embed *some* information about this private input. Recent attacks [194] have shown that it is possible to infer information about the training data from the model itself. Even when parties do not actively misbehave (applicable in the semi-honest setting), it is still possible to have *unintended leakage* embedded in the model. Therefore, parties may wish to include *privacy checks* to ensure that the final model is not embedding too much information about the training dataset. We list some possible example policies that can be used to prevent leakage from the model.

Differential privacy Differential privacy [80] is a common technique for providing some privacy guarantees in the scenario where a result has to be released to a semi-trusted party.

There are differential privacy techniques [48, 213, 122] for machine learning training, where some amount of noise is added to the model before release. For example, one method requires sampling from a public distribution and adding this noise directly to the weights. This can be implemented in Cerebro by implementing the appropriate sampling algorithm and adding the noise to the model before releasing it.

Model memorization Another possible method for dealing with leakage is to measure the amount of training data memorization that may have occurred in a model. One particular method [44] proposes injecting some randomness into the training dataset and measuring how much this randomness is reflected in the final model. This technique can be implemented by altering the training dataset X_i and programming the measurement function as a release policy.

6.3.2 Cryptographic auditing

In the malicious setting, Cerebro can use a maliciously secure MPC protocol to protect against deviations during the computation. However, even such an MPC protocol cannot protect against any attack that happens *before* the computation begins; namely, a party can inject carefully crafted malicious input into the secure computation. This is a big problem for machine learning training. For example, prior work has shown that a party can inject malicious training data that causes the released model to provide incorrect prediction results for any input with an embedded backdoor [53]. Furthermore, Cerebro's compute policies may be insufficient to detect such attacks since either the policy writer has to be aware of the chosen backdoor – which is unlikely – or the policies have to exhaustively check the input domain – which is infeasible.

In Cerebro, we propose an auditing framework that aims to hold all parties accountable for their original inputs *even after the result has been released*. Using the previous attack as an example: one way to construct malicious samples is to embed a pattern into non-malicious samples to alter the prediction results. If a poisoned model is triggered, the victim can backtrack the examples that triggered the attack and identify the backdoor. The victim can then use our auditing procedure to audit other parties' input data and identify who input the malicious training samples.

6.3.2.1 Auditing framework

When auditing a computation in Cerebro, we need to ensure that the audit procedure has access to the same inputs as were used in the original computation. Otherwise, we run the risk of having a malicious participant provide a sanitized input during the audit, thus avoiding detection. Cerebro enforces that a participant cannot modify input data after the fact using *cryptographic commitments* [38, 179], a cryptographic tool that ties a user to their input values without revealing the actual input. A participant commits to its input data by producing a randomized value that has two properties: *binding* and *hiding*. Informally, binding means that a party who produces a commitment from its malicious dataset will not be able to produce an alternate sanitized version later and claim that the commitment matches this new dataset. At the same time, hiding ensures that the commitments do not reveal information about the inputs.

Auditing API In order to abstract away the cryptographic complexity and to provide users with an intuitive workflow, we design the following API:

- `c, m = commit(X)`: returns `c`, the actual commitment, as well as `m`, the metadata used in generation of the commitment. `c` is automatically published to every other party, while `m` is a private output to the owner of `X`.
- `audit(X, c, m)`: this function returns a boolean value based on whether the commitment matches with the input data `X`.

Handling malicious aborts A serious concern during auditing is that a participant might cause the secure computation to abort since maliciously secure MPC generally does not protect against parties aborting computation. There are two types of aborts: a malicious party can refuse to proceed with the computation or can maliciously alter its input to MPC so that the computation will fail. The first type of abort is easy to catch, but the second is sometimes impossible to detect. For example, arithmetic MPC uses secret sharing (where each party's data is randomly shared with all other parties) and information theoretic MACs on those secret shares. If, at any time during the computation, a malicious party alters its share of another party's data, then the MAC check will fail, and the protocol cannot distinguish who triggered the abort. Therefore, a party can maliciously fail during the auditing phase and make it impossible to run an auditing function to track accountability.

To resolve this challenge, we introduce a *third-party auditor* into our auditing workflow. We do not believe this is an onerous requirement, since audit processes often already involve third-party arbitrators, e.g., courts, who help decide when to audit and how to use audit results. We *do not* require the third-party to be completely honest, but instead assume that it is honest-but-curious, does not collude with any of the participants, and does not try to abort the computation. Under this assumption, we enable the auditor to audit a party without forcing the party to release its data. This means that the auditor will not see any party's data in plaintext, since we still require the auditor to run the auditing process using MPC.

Auditing protocol Let $\mathcal{A}$ denote a separate auditor entity, and let $\mathcal{P}_i$ denote the parties running the collaborative computation. We construct the following auditing protocol.

1. Using the established PKI, $\mathcal{P}_i$'s have public keys corresponding to every participant in the secure computation. $\mathcal{P}_i$'s agree on the same unique number qid for this computation.

2. $\mathcal{P}_i$ computes a commitment of its data. Let the commitment be $\mathbf{c}_i$. $\mathcal{P}_i$ hashes the commitments $h_i = \mathsf{hash}(\mathbf{c}_i)$ and generates a signature $\sigma_i = \mathsf{sign}(\mathsf{qid}, h_i)$ using its secret key. $\mathcal{P}_i$ publishes $(\mathbf{c}_i, \sigma_i)$ to $P_{j \neq i}$.

3. All $\mathcal{P}_i$'s run the secure computation, which encodes the original learning task and a preprocessing stage that checks that $\mathcal{P}_i$'s input data indeed commits to the public commitments received by every party from $\mathcal{P}_i$. If the check fails, then the computation aborts. Note that we won't know who is cheating in this stage, but the parties also won't get any result since the computation will abort before any part of the learning task is executed.

4. During auditing, $\mathcal{P}_i$ will publish its signed commitments, along with the $(\mathbf{c}_j, \sigma_j)$ received from $\mathcal{P}_j$, to $\mathcal{A}$. $\mathcal{A}$ checks that all commitments received from $\mathcal{P}_j$ about $\mathcal{P}_i$ match. If they do not, then $\mathcal{P}_i$ is detected as malicious.

5. $\mathcal{A}$ runs a two-party secure computation with each $\mathcal{P}_i$ separately. $\mathcal{P}_i$ inputs its data, and $\mathcal{A}$ checks the data against the corresponding commitment. If there is a match, continue with the auditing function. If this computation aborts, $\mathcal{P}_i$ is also detected as malicious. Since the auditing is in secure computation, $\mathcal{A}$ will not directly see $\mathcal{P}_i$'s input data.

Our auditing protocol is generic enough to be implemented with any commitment and MPC design. In practice, there are ways of constructing efficient commitments that can also be easily verified in MPC. Due to space constraints, we elaborate on optimizations below.

6.3.2.2 Commitment schemes

In this section, we describe some commitment schemes that integrate well with MPC, and how to efficiently check these commitments.

Ajtai's subset sum hash function Ajtai's subset sum hash function [9, 102, 25, 42] is a collision-resistant hash function from lattice hardness problems. We instantiate a type of cryptographic commitment using this hash function. For $p, d, m \in \mathbb{N}$, where p is a prime, the hash function $H_M : \{0,1\}^m \to \mathbb{Z}_p^d$ maps an m-bit string x to $\sum_{i=1}^{m} x_i M(i)$, where $M(i)$ is the i-th column of a random matrix $M \in \mathbb{Z}_p^{d \times m}$. Choosing appropriate parameters of d and m to achieve sufficient bit security [42], we can instantiate such a collision-resistant hash function. By hashing a value with an appropriately chosen random value, we can make the hash a randomized commitment scheme.

Pedersen commitment Verifying a commitment inside MPC can be expensive, especially if we have to verify every data point separately because that would scale linearly with the input data size. In this section, we provide a way of batch checking commitments in MPC using a homomorphic commitment such as Pedersen [179]. We utilize the fact that our arithmetic framework is reactive to construct such a scheme.

Input 1	**Input 2**	**Output**	**Compute**
public	public	public	local at all
public	private(i)	private(i)	local at i
private(i)	private(i)	private(i)	local at i
private(i)	private(j)	secret	global
any	secret	secret	global

Table 6.2: Rules for defining a function's execution mode

Denote $\mathsf{com}(x; r)$ as the Pedersen commitment. The protocol is as follows:

1. As before, each $\mathcal{P}_i$ commits and publishes its commitments.
2. $\mathcal{P}_i$'s start a SPDZ computation and inputs both its input data $\mathbf{x}_i$, as well as the randomness used $\mathbf{r}_i$ for the commitments.
3. Everyone releases a random number s from SPDZ.
4. Each $\mathcal{P}_i$ computes $\tilde{c}_j = \sum_k s^k \otimes \mathbf{c}_j[k]$ for every $\mathcal{P}_j$.
5. $\mathcal{P}_i$'s input s as well as $\tilde{c}_j$ into the same SPDZ computation computed in Step 2.
6. The secure computation calculates $\tilde{x}_i = \sum_k s^k \cdot \mathbf{x}_i[k]$ and $\tilde{r}_i = \sum_k s^k \cdot \mathbf{r}_i[k]$. Then it checks that $\mathsf{com}(\tilde{x}_i; \tilde{r}_i) = \tilde{c}_i$.

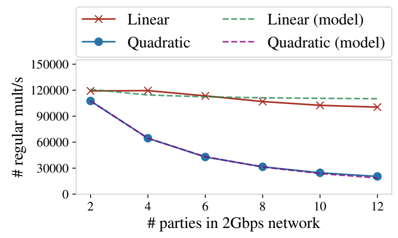
For elliptic curve groups, the prime modulus will need to be on the order of 256 bits.

Tradeoffs While subset-sum can work with our standard benchmarking prime field of 170 bits, Pedersen commitments need the prime field to be at least 256 bits security. Therefore, while Pedersen commitments are generally more efficient in terms of the number of triples used, the larger bit size means offline generation can be more costly. Although if one needs more precision for example, then the 256-bit prime field may seem more desirable as well.

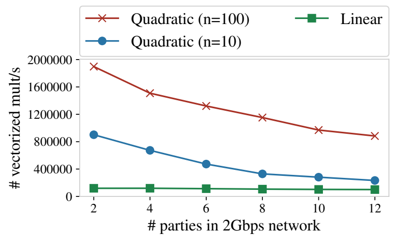
Additionally, Pedersen commitments require a reactive framework such as SPDZ in order for the batching to work properly in the secure computation phase. Cerebro's planner takes these circumstances into account, and chooses the best plan accordingly.

6.4 Implementation

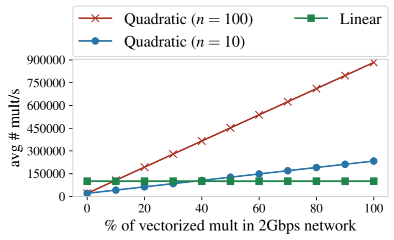
We implemented Cerebro's compiler on top of SCALE-MAMBA [11], an open source framework for arithmetic MPC. Our DSL is inspired by and quite similar to that of SCALE-MAMBA, though we have the notion of private types. In order to support both arithmetic and boolean MPC, we added a boolean circuit generator based on EMP-toolkit [210]. Both of these circuit generators are plugged into our DSL so that a user can write one program that can be compiled into different secure computation representations.



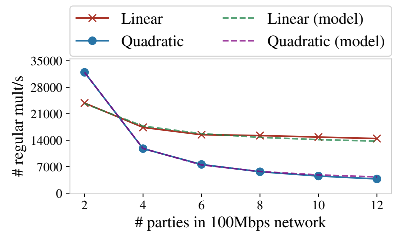
(a) Throughput of preprocessing (2Gbps network).



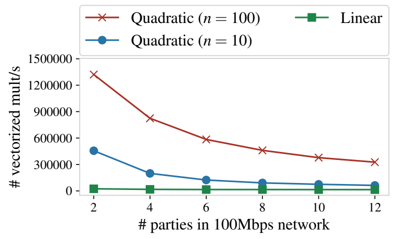
(b) Throughput of vectorized preprocessing (2Gbps network).



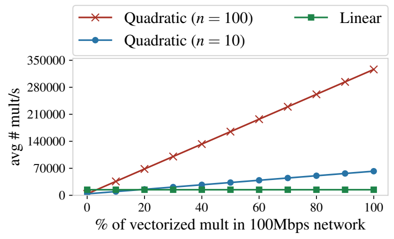
(c) Throughput when varying % of vectorized multiplications ($P = 12$, 2Gbps network).



(d) Throughput of preprocessing (100Mbps network).



(e) Throughput of preprocessing vectorized multiplications (100Mbps network).



(f) Throughput when varying % of vectorized multiplications ($P = 12$, 100Mbps network).

Fig. 6.6: Choosing linear vs. quadratic protocol for preprocessing arithmetic circuits.

Cerebro uses different cryptographic backends that support both semi-honest and malicious security. We implemented Cerebro's malicious cryptographic backend by using the two existing state-of-the-art malicious frameworks – SPDZ [11] and AG-MPC [210]. Additionally, we implemented Cerebro's semi-honest cryptographic backend by modifying the two backends to support semi-honest security.

6.5 Evaluation

We evaluate the effectiveness of Cerebro's cryptographic compiler in terms of the performance gained using our techniques. We use the two generic secure multiparty frameworks that Cerebro uses as a baseline for evaluation, in both semi-honest and malicious settings. We do not compare against plaintext learning systems because they cannot be used for collaborative learning on sensitive data. Instead, we compare to what users would be doing today without our system, which is choosing a generic MPC framework and implementing a learning task using it. Our goal is to show that, without Cerebro's compiler, users can experience *orders of magnitude* worse performance if they choose the wrong framework and/or do not have our optimizations.

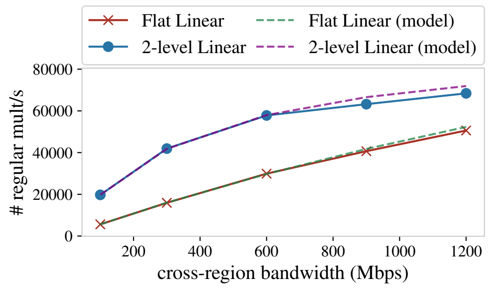


Fig. 6.7: Flat vs. two-level linear protocol for 9-party vs. 3-party bipartite network layout with varied cross-region total bandwidth (2Gbps intra-region per-party bandwidth).

6.5.1 Evaluation setup

Our experiments were run on EC2 using r4.8xlarge instances. Each instance has 32 virtual CPUs and 244GB of memory. In order to benchmark in a controlled environment, we use `tc` and `ifb` to fix network conditions. Unless stated otherwise, we limit each instance to 2Gbps of upload bandwidth and 2Gbps of download bandwidth. We also adjust latency so the round trip time (RTT) is 80ms between any two instances. According to [18], this is roughly the RTT between the east-coast servers and west-coast servers of EC2 in the U.S.

6.5.2 Compiler evaluation

We evaluate Cerebro's compiler by answering these questions:

1. Are Cerebro's cost models accurate?
2. How do Cerebro's logical optimizations impact performance?
3. For realistic setups, does Cerebro's physical planning improve performance?

To answer these questions, we run a series of microbenchmarks as well as end-to-end application-level benchmarks. We first curve fit our cost models and extrapolate against experimental results. We then evaluate different planning points to show Cerebro's gain in performance. We focus our evaluation on planning in the semi-honest setting, but our planning also supports the malicious setting.

6.5.2.1 Microbenchmarks

Cost models Our first microbenchmark compares the two methods for semi-honest arithmetic MPC preprocessing (see Section 6.2.3.3): linear and quadratic preprocessing.

For both of the following experiments, we fit the constants of our cost model to the first four points of the graph and then extrapolate the results for the remaining two points. The dotted lines of the graph indicate the cost model's predictions and we can see that it closely matches with the experimental results. Figure 6.6a shows the preprocessing throughput of the linear and the quadratic protocols on high-bandwidth network. When the number of parties is small, the two protocols have similar throughput. However, as the number of parties increases, the quadratic protocol becomes slower than the linear protocol, mainly due to the increased communication.

Figure 6.6d compares the same protocols when the network is slow and becomes the bottleneck. When the number of parties is small, the quadratic protocol is faster than the linear protocol because it uses smaller ciphertexts, but it performs worse than the linear protocol as the number of parties increases.

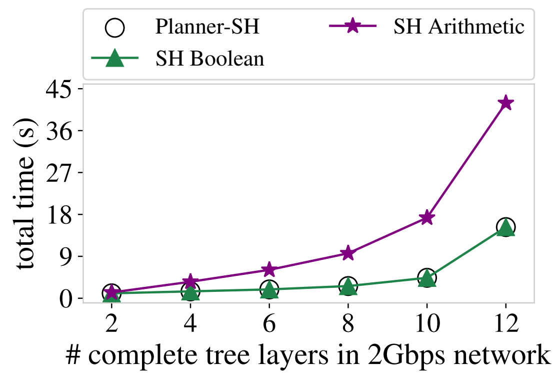
Vectorization Figures 6.6b and 6.6e show the preprocessing throughput of a single matrix-vector multiplication – where the matrices are of sizes $(m \times n)$ and $(n \times 1)$ – under different network conditions. We test with a fixed $m = 128$ and vary n in our experiments. On a high-bandwidth network, when there are two parties and $n = 100$, the quadratic protocol achieves a 16× speedup over the linear protocol. Even when the number of parties is increased to 12, these two protocols still have an 8.8× gap. On a slower network, the matrix-vector technique has a larger performance gain since it mainly saves communication, with up to 55× speed up.

Next, we evaluate the two protocols when there is a mix of matrix multiplication and regular multiplication. The results are shown in Figures 6.6c and 6.6f. The planning decision will be different based on the percentage of multiplication gates that can be substituted with matrix-vector multiplications, the shape of such matrices, the number of parties, and the network bandwidth. For example, in 2Gbps network with 12 parties and $n = 10$, if 40% of the multiplication gates can be vectorized, then Cerebro will pick quadratic. If the network bandwidth drops to 100Mbps, then 20% of such computation is enough for the compiler to pick quadratic.

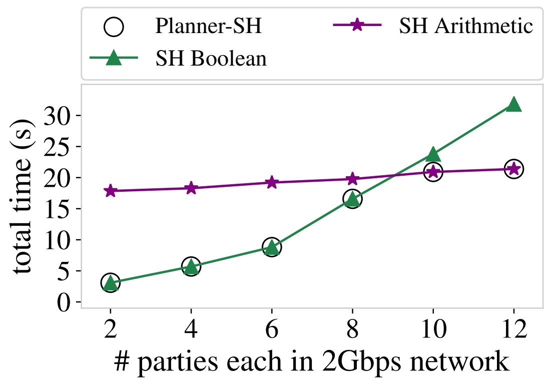
Layout planning We evaluate the hierarchical layout preprocessing against a flat one for 12 parties across two regions: 9 are located in one region, and 3 are located in the other. Each party has 2Gbps bandwidth for intra-region communication, and we vary the *total* cross-region bandwidth shared by parties in the same region. Figure 6.7 shows the throughput comparison as well as our fitted cost models. Similar to before, we fit the constants of our cost model to the first three points of the graph and then extrapolate the results. The flat layout throughput scales linearly to the cross-region total bandwidth. To evaluate the hierarchical layout, we need to first determine the workload of each coordinator using cvxpy. From the graph we can see that the hierarchical layout achieves a speed up of 4× to 4.5× over the flat layout.

6.5.2.2 Machine learning applications

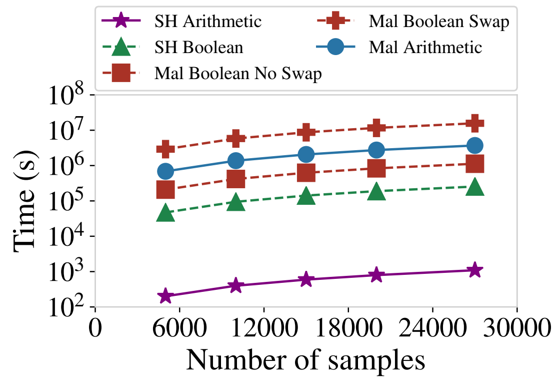
In this section we evaluate Cerebro using decision tree prediction, logistic regression training via SGD, and linear regression training via ADMM [37, 10, 219]. We estimate the



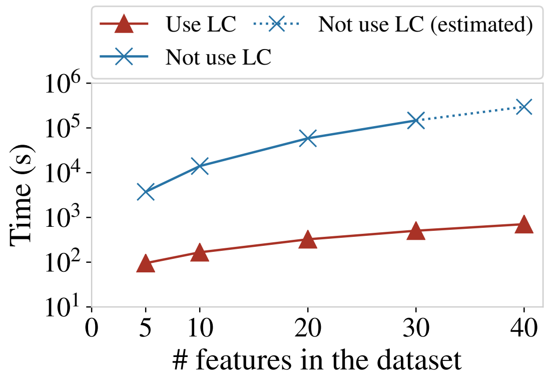
(a) Decision tree prediction time for different tree sizes (2-party).



(b) Decision tree prediction time for different number of parties (10-layer tree).



(c) Time for logistic regression training in different dataset size (time in log-scale).



(d) ADMM with/without local compute, 10^4 training samples per party (time in log-scale).

Fig. 6.8: Experiments on machine-learning applications (2Gbps network).

network cost for the preprocessing phase of the arithmetic protocol using the throughput gathered in the previous benchmarks.

Decision tree prediction We implement decision tree prediction using Cerebro's DSL, which evaluates a complete h layer binary decision tree, where the i^{th} layer has 2^{i-1} nodes. We evaluate a scenario where there are P parties, one of which has the input feature vector and all P parties secret-share a model. If $P = 2$, we assume that we are doing a two-party secure prediction, where one party has the feature vector and the other has the model.

We show the prediction performance in the 2-party semi-honest setting in Figure 6.8a. In this experiment, we varied the number of layers in the decision tree. We fit the data points involving $3, 6, 9$ layers and then extrapolate the cost model to estimate the performance of our graphed points. Cerebro always picks the protocol that has the lower estimated cost from our model. In the 2-party scenario, Cerebro always chooses to use a boolean protocol since evaluating the decision tree requires many comparisons and data selection. In a 12-layer tree, the semi-honest boolean protocol takes 7.5× less time than the semi-honest arithmetic protocol. In Figure 6.8b, we vary the number of parties, and plot the inference runtime for a 10 layer tree. We observe that the total execution time for the boolean protocol grows linearly with number of parties, and sublinearly for the arithmetic protocol. Therefore, with 9 or more parties, Cerebro chooses to use the arithmetic protocol.

As noted previously, Cerebro also supports the malicious setting, we exclude those results here for brevity.

Logistic regression We implemented and evaluated Cerebro on logistic regression training using SGD. In this experiment, we evaluated training in both the semi-honest and the malicious settings to show a difference in the performance for different variants of the protocols. For the semi-honest and malicious boolean protocols, we ran logistic regression for one iteration of SGD and extrapolated the remaining results. First, we compare the performance between the semi-honest boolean and semi-honest arithmetic protocol in Figure 6.8c. As expected, the arithmetic protocol significantly outperforms the boolean protocol in this case, both because it is better suited for this task and because it enables vectorization. Using these results we see that for a 27000 record training set the arithmetic protocol is 67× faster than the boolean protocol, taking an hour instead of three days.

However, in the malicious setting, the arithmetic protocol does not always perform better. The amount of memory used by the malicious boolean protocol is linear in the number of parties and the number of gates. As a result, we run out of memory when trying to benchmark larger circuits. We estimate the malicious boolean protocol on machines with enough memory as well as on the original machines with swap space to use as additional memory. As shown in Figure 6.8c, if the machines have enough memory, then the malicious boolean protocol is 3× faster than the malicious arithmetic protocol, but if swap space is used instead, then the malicious boolean protocol is 4× *slower* than the malicious arithmetic protocol. Overall, the malicious boolean protocol is up to 61× slower than its semi-honest counterpart and the malicious arithmetic protocol is up to 3300× slower than its semi-honest counterpart, indicating a significant tradeoff between performance and security.

ADMM We evaluate ADMM in the semi-honest setting to show Cerebro's automated planning of local computation. Cerebro automatically detects that the parties can locally compute much of the ADMM algorithm, thus minimizing the number of MPC operations required as described previously in Section 6.2. We evaluate these benefits in Figure 6.8d and find that the use of local computation allows Cerebro to improve ADMM performance by up to 400× when training a 40-feature model using 10000 records. We estimate the preprocessing and run the online phase for the first four data points, but estimate the fifth. Beyond this we also find that the use of arithmetic circuits is beneficial here for the same reasons as in the case of logistic regression, i.e., it allows vectorization and is better suited to expressing matrix operations.

# **Training samples**	D.P. time (s)	Validation time (s)
1000	0.038	9.52
5000	0.038	47.6
15000	0.038	143
25000	0.038	238
27000	0.038	257

Table 6.3: Time for applying the privacy policy or the validation policy to a logistic regression model.

6.5.3 Policy evaluation

We evaluate the performance of Cerebro's release policies in the semi-honest setting. Specifically, we evaluate logistic regression that uses both differential privacy and the threshold-based validation policies. Our differential privacy policy is output perturbation-based [48, 213], which simply requires each party to locally sample noise. The secure computation will sum every party's noise and add the noise to the weights. As table 6.3 shows, the time for adding this noise is independent of the number of training samples, and is insigificant compared to the training time.

The threshold-based validation policy requires the model to achieve a sufficient level of accuracy in order to be released. To see how much time is needed for validation, we split the dataset with 30000 records into a training set of 27000 records and a validation set composed of the remaining 3000 records. We train the model using a subset of the training set and validate the trained model using part of the validation set which is 10% the size of the used training set. From table 6.3, we can see that the validation time grows linearly to the used validation set. Compared with training in logistic regression, the time taken by validation is equivalent to training another 10% of the training samples, which matches the training behavior of logistic regression.

6.5.4 Auditing evaluation

Next, we present the overheads from enabling auditing support for logistic regression. There are two main costs in this case. The first cost is producing and signing a commitment, which takes 24.4 seconds, of which 8 milliseconds are spent generating a signature for user input (which is a 27000×23 matrix in our case). The second cost is spent on the commitment protocol described in Section 6.3.2.2. Checking the commitment within MPC using a non-batching commitment scheme such as subset sum takes $424,113$ seconds while checking the commitment using a batching commitment scheme such as Pedersen commitments takes roughly 8040 seconds. The speedup is roughly 53x, which only grows as the number of samples increases as the batched commitment scheme scales better with respect to the number of samples. Overall we find that enabling auditing has reasonable overhead.

6.6 Related work

Related plaintext systems There is a large body of prior work on distributed linear algebra systems [182, 34, 96] and machine learning training/prediction [51, 126, 158, 50, 4, 63]. While some of these systems are general and can be adapted to the distributed setting, they do not provide security guarantees and cannot be used in the collaborative machine learning setting when the input data is sensitive. Some of these systems provide interesting linear algebra optimizations that are similar to Cerebro's optimizations at a very high level, but Cerebro additionally must consider the effects of optimizing a *cryptographic* protocol. This means that Cerebro has different rules for transformation and a very different cost model. The idea of "physical planning" is similar to prior systems and database work [125, 205, 151, 88, 123, 208, 137]. The main difference is that we instantiate this idea to the MPC setting and work closely with the underlying cryptography.

MPC compilers Cerebro draws inspiration and ideas from a body of work on MPC compilers [210, 145, 196, 183, 46, 120, 41, 11, 217, 164, 85, 74, 85, 207]. Compared to prior work, Cerebro's compiler differs in two important aspects. First, we provide n-party compilation supporting two MPC frameworks under different threat models. There is prior work providing n-party compilation supporting a single framework [210, 145, 196, 11, 217, 164, 85] and two-party compilation supporting multiple frameworks [74, 46, 120]. Second, Cerebro adds optimization in *both* the logical and the physical layers, which allows us to consider a multitude of factors like computation type, network setup, and others. Conclave [207] is a recent system that is similar to Cerebro because it handles multiple frameworks and does optimization. However, it is designed for SQL, and does not consider physical planning or release policies. Finally, Cerebro itself is an end-to-end platform for collaborative learning and supports policies and auditing.

Secure learning systems There are two main approaches to building secure learning systems: utilizing hardware enclaves or cryptography. There is prior work that utilize hardware enclaves to execute generic computation, analytics, or machine learning [22, 116, 221, 115, 170]. Compared to Cerebro, the threat model is quite different. While hardware enclaves support arbitrary functionality, the parties have to put trust in the hardware manufacturer. We have also seen that enclaves are prone to leakages [214, 39, 109, 172, 136].

There has been much prior work on secure learning using cryptography, both in training and prediction [168, 112, 93, 55, 97, 10, 162, 223, 219, 146, 127, 186, 202, 52, 8, 33]. However, these prior works are lacking in several aspects. First, they mostly focus on optimizing specific training/prediction algorithms and models and do not consider supporting an interface for programming generic models. Second, they do not automatically navigate the tradeoffs of different physical setups. Finally, these frameworks also do not take into account the incentive-driven nature of secure collaborative learning, while Cerebro supports policies and auditing.

Other related work A recent paper by Frankle et. al. [84] leverages SNARKs, commitments, and MPC for accountability. However, the objective is to make the government more accountable to the public, so the setting and the design are both quite different from ours. Other papers [118, 119, 64, 21] explore identifying cheating parties in maliciously secure MPC. However, these papers are either highly theoretical in nature, or require proof that each party behaved honestly during the *entire protocol execution*, which can be quite expensive. Cerebro is mainly concerned with holding the users accountable for their input data, and our scheme both works with multiple MPC frameworks and does not need to require proof of honest behavior for the entire protocol execution. With regards to the logical optimizations that Cerebro performs, there has been work [183] that also performs partitioning of computation into local and secure modes. However, Cerebro does not require the user to specify the mode of computation for every single operation and instead automatically partitions the source code into local and secure components.

6.7 Security proofs

6.7.1 Extended vectorized triple generation

In SecureML [162], the authors provided a way to generate matrix multiplication triples instead of scalar multiplication triples. However, the design was only for two parties. In Cerebro, we generalize their design by applying the two-party protocol to the multi-party setting where each $\mathcal{P}_i$ runs the two-party protocol in a pairwise fashion with all other parties $\mathcal{P}_j$. The protocol is split into two parts: an "offline" phase that generates independent matrix multiplication triples without having access to any party's private input, as well as an "online" phase that actually runs the computation with every party's private input and outputs a result to everyone.

There are P parties participating in the secure computation, and we denote each party as $\mathcal{P}_{i\in[0,\ldots,P)}$. We also define a semihonest adversary A who is able to statically compromise up to $P-1$ out of P parties, but does not deviate from the cryptographic protocol. Since the adversary is semihonest adversary's view is therefore the corrupted parties' inputs, the messages that it receives from the honest parties, as well as the final output.

We first define the ideal functionalities $F_{offline}$ and F_{online}, where $F_{offline}$ generates the vectorized random multiplication triples and F_{online} is the online phase that computes on secret shared private values by utilizing the multiplication triples. We will prove using the hybrid $(F_{offline}, F_{online})$.

Proof sketch. We are in the semihonest setting, so we do not need extraction of the adversary's inputs. Instead, the simulator is given those inputs directly can pass those inputs to the ideal functionality. The simulator S_{online} then receives a final output from the ideal functionality. S_{online} will generate a view for the adversary given the corrupted parties' inputs as well as the final output. The adversary's inputs are the corrupted parties' private outputs from $F_{offline}$ and the private inputs to the secure computation. Therefore, the simulate the adversary's view, S_{online} simply generates random shares for the honest parties and follows the specified protocol. Before combining the final shares, the simulator adjusts the honest parties' shares so that the result matches the output produced by the ideal functionality. □

Since the online phase relies on the offline phase producing the correct multiplication triples. The ideal functionality takes no inputs, and generates random shares of matrix multiplication triples so that for each triple $(A \cdot B = C)$, each party $\mathcal{P}_i$ has shares $A^{(i)}, B^{(i)}, C^{(i)}$. There are different protocols for generating multiplication triples, and here we will give a proof of the homomorphic encryption based method. In this protocol, $\mathcal{P}_i$ each generates a different public/private Paillier key pair. For the sake of a simpler analysis, we assume that B_j is a vector $\mathbf{b}$, and that the relation is $\mathbf{c} = A\mathbf{b}$. To generalize this to the multi-party setting, we can apply the two-party protocol in a pairwise fashion to generate the triples. To compute the triple, it suffices for each party to first sample random $A^{(i)}$ and $\mathbf{b}^{(i)}$, and $\mathbf{c} = \sum_i \sum_j A^{(i)} \cdot \mathbf{b}^{(j)}$. Since each $\mathcal{P}_i$ can compute $A^{(i)} \cdot \mathbf{b}^{(i)}$ locally, we use can adapt the two-party cross product to compute shares of $A^{(i)} \cdot \mathbf{b}^{(j)}$ between $\mathcal{P}_i$ and $\mathcal{P}_j$. In the two-party protocol, $\mathcal{P}_i$ individually generates a public/private key pair. The sender then encrypts its share and sends it to the receiver party. The receiver executes encrypted encryption and subtracts a random mask to the resulting encrypted vector. The encrypted vector is sent back to the sender and it can decrypt this masked vector. The decrypted vector is the sender's new share, and the mask vector is the receiver's new share. At the end of the protocol, each party adds up all of the shares.

Proof sketch. The simulator $S_{offline}$ begins by calling the ideal functionality and receiving the final output shares for the corrupted parties. If the honest party is sender in the pairwise protocol, the simulator simply encrypts 0 using the honest party's public key and sends it to the receiver. The receiver will execute the protocol correctly and sets is share to be the random mask vector. If the honest party is the receiver, the simulator then computes the encrypted product with 0 and adds a random mask vector $\mathbf{r}$. Since the simulator knows the final private output that each corrupted adversary is supposed to receive, it simply adjusts one of the shares of the simulated honest parties so that the final summed private share is indeed equal to the private output. Since the other shares are random, this adjusted share is also random.

Clearly, the real world protocol *correctly* implements the ideal functionality. In terms of the privacy argument, the simulated view is computationally indistinguishable from the real world view. If the receiver is corrupt, it only receives an encrypted cipher. Any privacy leakage from the receiver's side will mean that the receiver can distinguish the ciphertext, breaking the computational assumption of the underlying encryption. If the sender is corrupt, it can decrypt the received cipher, but the simulated random vector mask is drawn from a uniformly random distribution, which is the same as the real world protocol's view. □

6.7.2 Hierarchical triple generation

In Section 6.2.3.4, we describe a hierarchical design for triple generation for linear preprocessing. To generate a P-party Beaver triple $(\{a_i\}, \{b_i\}, \{c_i\})$ in which $(\sum a_i) \cdot (\sum b_i) = (\sum c_i)$, the original linear preprocessing protocol works as follows:

1. All parties run a distributed key generation to generate a SWHE (somewhat homomorphic encryption) public key PK; each party $\mathcal{P}_i$ has a share of the SWHE private key SK_i.

2. Each party $\mathcal{P}_i$ samples random a_i, b_i, and f_i locally and encrypt them under the SWHE public key PK; $\mathcal{P}_i$ sends $\mathsf{Enc}_{PK}(a_i)$, $\mathsf{Enc}_{PK}(b_i)$, $\mathsf{Enc}_{PK}(f_i)$ to a coordinator.

3. The coordinator, which could be one of the P parties, aggregates the ciphertexts of a, b, and f by running the SWHE addition algorithm, thereby obtaining $\mathsf{Enc}_{PK}(a)$, $\mathsf{Enc}_{PK}(b)$, and $\mathsf{Enc}_{PK}(f)$; then, the coordinator computes a new ciphertext $\mathsf{Enc}_{PK}(a*b+f)$, denoted by $\mathsf{Enc}_{PK}(c+f)$, and sends $\mathsf{Enc}_{PK}(c+f)$ to all parties.

4. All parties run an existing distributed decryption protocol to decrypt $\mathsf{Enc}_{PK}(c+f)$ together, so that all parties see $c+f$ in plaintext. $\mathcal{P}_1$ sets $c_1 = c+f-f_1$, and other parties $\mathcal{P}_i$ $(i \neq 1)$ set $c_i = -f_i$. Together with a_i and b_i sampled previously, $(\{a_i\}, \{b_i\}, \{c_i\})$ forms a Beaver triple.

In the strawman flat design, Cerebro distributes the work of the coordinator to each party *evenly*, which means that each party, throughout the MPC, coordinates the same amount of triples. This is ill-suited for the *cross-region* setting in which parties are separated into a few physical regions, where intra-region communication is fast, but the inter-region communication is slow.

Now, in the hierarchical design, as we mention in Section 6.2.3.4, a coordinator is split into (1) many regional coordinators, one per region and (2) a global coordinator that communicates with the regional coordinators. Cerebro still distributes the work of the coordinator to different parties, but the workload is now computed by solving an optimization problem related to the network layout and the bandwidth, as discussed in Section 6.2.3.4 and Section 6.8.3. The detailed changes to the protocol above are as follows:

- On step 2, each party $\mathcal{P}_i$ firstly sends $\mathsf{Enc}_{PK}(a_i)$ and $\mathsf{Enc}_{PK}(b_i)$ to the regional coordinator in $\mathcal{P}_i$'s region. Each regional coordinator aggregates the ciphertexts via the SWHE additions algorithm and sends the aggregated results to the global coordinator.

- On step 3, the global coordinator sends $\mathsf{Enc}_{PK}(c)$ to regional coordinators. Each regional coordinator then forwards this ciphertext to the parties in the region.

- On step 4, the regional coordinators also help aggregate some information broadcast in the distributed decryption protocol.

We now outline that these change preserves the security guarantees of the original protocol in the semi-honest setting.

Proof sketch. The insight of the proof is that, in this modified protocol, the parties' view is, in principle, *a strict subset* of the view in the original SPDZ protocol (which is maliciously secure). In the maliciously secure SPDZ protocol, each party sees $\mathsf{Enc}_{PK}(a_i)$, $\mathsf{Enc}_{PK}(b_i)$, $\mathsf{Enc}_{PK}(f_i)$ of one another and also sees the information for the distributed decryption protocol sent by one another, without aggregation. Our hierarchical protocol aggregates more ciphertexts, so that each party sees conceptually less information than the original SPDZ. This informal description is the high-level idea of the proof below.

We now state this argument more formally. Note that the original, maliciously secure SPDZ is also secure against semi-honest adversaries. Therefore, there exists a simulator $\mathcal{S}_i$ for each party $\mathcal{P}_i$, such that the following two distributions are computationally indistinguishable:

$$\left\{\mathcal{S}_i(1^\lambda, f_i(1^\lambda)), f(1^\lambda)\right\}_\lambda \overset{\mathsf{c}}{\approx} \left\{\mathsf{view}_i(1^\lambda), \mathsf{output}_i(1^\lambda)\right\}_\lambda ,$$

where λ is the security parameter, $f_i(1^\lambda)$ is the output of $\mathcal{P}_i$ in a triple generation functionality (i.e., (a_i, b_i, c_i)), $f(1^\lambda)$ denotes all parties' output, $\mathsf{view}_i(1^\lambda)$ denotes the view of $\mathcal{P}_i$, and $\mathsf{output}_i(1^\lambda)$ denotes the output of $\mathcal{P}_i$ in the protocol; $\overset{\mathsf{c}}{\approx}$ denotes computational indistinguishability.

It is not hard to construct another simulator $\mathcal{S}_i^H$ for our hierarchical design, which, as described above, surgically aggregates some of the ciphertexts following the specific hierarchical parameters, which can be computed efficiently. This will, therefore, be computationally indistinguishable with the view in the hierarchical setting, denoted by $\mathsf{view}_i^H(1^\lambda)$:

$$\left\{\mathcal{S}_i^H(1^\lambda, f_i(1^\lambda)), f(1^\lambda)\right\}_\lambda \overset{\mathsf{c}}{\approx} \left\{\mathsf{view}_i^H(1^\lambda), \mathsf{output}_i(1^\lambda)\right\}_\lambda .$$

This concludes the security proof sketch. □

6.7.3 Auditing

In Cerebro, the auditing protocol is used to identify wrong doings by parties who run the secure computation with malicious datasets. By injection malicious data into the computation, the malicious parties are able to affect the result of the computation. Therefore, the security goal of the auditing protocol is to make sure to produce a *superset* of the cheating parties produced by the result of the auditing function. Note that Cerebro's auditing protocol is trying to protect the honest parties from an attack where the malicious parties *want* to finish the secure computation, but also wants to attack the released result itself.

Proof sketch. We will argue for the security of this protocol via two dimensions: privacy and correctness. We make use of existing cryptographic constructs such as commitments, MPC, and PKI.

Privacy is easy to see because we make blackbox usage of cryptographic commitments as well as MPC. Any privacy leakage in the protocol will result in breaking these cryptographic primitives. In terms of correctness, note that the security guarantee we provide is that

1. Any malicious party who can be caught by the auditing function is either caught by the auditing function, or is caught aborting and will automatically identified as cheating

2. Any honest party cannot be falsely caught as cheating

Cerebro is able to provide this guarantee again because of the underlying cryptographic primitives that are used. First, cryptographic commitments are binding, so the cheating party with bounded computation power cannot find an alternative sanitized version of the input. This holds during both the secure computation phase, as well as the auditing phase so as to make sure that all parties are using only one input dataset. Therefore, the question is whether a malicious party can somehow trick the auditor into using a different commitment during the auditing phase, and thus inputting an alternative dataset during this phase. This cannot be true because each party is supposed to sign this commitment (assuming the existence of a public key infrastructure) and passes it to every other party. During the auditing process, every party will then pass the received commitments to the auditor $\mathcal{A}$. Inconsistency cannot occur during the secure computation phase because the computation itself checks that $\mathcal{P}_i$'s input data matches the commitments from every party, which means that there is only one copy of each party's data. During auditing, the auditor will also receive these commitments and check that they are consistent. Even if $P-1$ parties collude together during this phase to construct an alternative commitment, the honest party will give the original commitment to the auditor, who will then verify that each party's input in this phase matches the commitments. Therefore, each party is guaranteed to always use the same dataset, and cannot swap out the original dataset used in the secure computation for a new one. Furthermore, a malicious party cannot falsely accuse of an honest party of being malicious by making changes to its commitments, since each party's commitment is signed by that party's private key, and each session has a unique qid which prevents replay attacks. $\square$

6.8 Physical planning

6.8.1 More cost models

Boolean MPC preprocessing cost For boolean MPC, there are two phases within the preprocessing phase. The first phase is very similar to the preprocessing generation phase for arithmetic MPC, except that this step now generates AND triples instead of multiplication triples. For this phase, Cerebro only provides one method, which is similar to the quadratic preprocessing, and has the same cost model as eq. (6.2). The second phase is a circuit generation phase, where each party creates a copy of the final circuit and sends it to a *single* party. This "evaluator" party will be in charge of executing the circuit during the online phase.

Therefore, the cost model for the boolean MPC preprocessing phase is:

$$c + N_a(P-1)(f_1(\lambda) + g_1(B,\lambda)) + g_2(B,\lambda)N(P-1) \tag{6.4}$$

where λ is the security parameter, g_1 refers to the cost of preprocessing AND gates, and g_2 refers to the cost for a single evaluator to receive $P-1$ copies of the garbled circuit.

Online execution cost The online phases for arithmetic and boolean MPC have quite different behaviors, which in turn result in different cost models. Arithmetic MPC requires interaction (hence network round trips) among all parties for multiplication gates throughout the entire computation. The number of round trips is proportional to the *depth* of the circuit. Boolean MPC, on the other hand, is able to evaluate the online phase in a *constant* number of rounds. Therefore, arithmetic MPC's online phase can be modeled as $c + g(l)R$, where R indicates the number of communication rounds. We do not consider the compute cost because it should be very insignificant compared to the cost of the round trips.

Boolean MPC's online phase is modeled as $c + f(\lambda)N$, where λ is the security parameter. This captures the compute cost of evaluating the entire boolean circuit. There is interaction at the beginning of the protocol because the evaluator needs to receive encrypted inputs, and at the end of the protocol because the evaluator needs to publish the output.

6.8.2 Linear vs. quadratic preprocessing

Without diving into the cryptography, we describe these two methods at a very high level. Both methods are *constant round*, which means that they only need 1 - 2 roundtrips. In linear preprocessing, each party independently generates data for each triple and sends this data to a set of *coordinators*. The coordinators then aggregate this data, compute on it, and send the results back to each party. A similar pattern repeats for a second round. Since the triples can be generated independently, we distribute the coordination across all parties. In quadratic preprocessing, each party interacts with every other party in constant round to compute the triples.

6.8.3 Extended description of layout optimization

In this section, we give an extended analysis of the layout optimization problem. For an easier analysis, we assume that there are at most two regions (see Figure 6.4). In order to explain our cost model, we first define some preliminary notation as follows. The two regions are denoted as L and R. P_L parties are located in region L, and P_R parties are located in region R. We assume that each party has roughly the same computation power, that each party has a fixed inbound and outbound bandwidth limit for in-continent data transfer, and that between the two regions there is another inbound and outbound bandwidth limit shared by all the parties. Let n_L be the number of triples that a single global coordinator in L handles; n_R is similarly defined for region R. Hence we have the following relation $n_L \cdot P_L + n_R \cdot P_R = N_m$. The cost (*i.e.*, the wall-clock time) for preprocessing arithmetic circuits is:

$$\begin{aligned} T =& g_1(B_1)(L_1 + L_2) + g_2(B_2)L_3 \\ &+ f_1(|p|)L_4 + f_2(|p|)(L_1 + L_3) \end{aligned} \tag{6.5}$$

B_1 is the intra-region bandwidth per party, while B_2 is the *total* inter-region bandwidth between the two regions. Therefore, the g_1 and g_2 terms capture the network cost. The f_1 and f_2 terms correspond to the compute cost, where f_1 captures ciphertext multiplication, and f_2 captures the other ciphertext operations. L_1 - L_4 are scaling factors that are functions of $n_L,\ n_R,\ P_L,$ and P_R:

$$\begin{aligned} &L_1 = \max\left(n_R \cdot \tfrac{P_R \cdot (P_L - 1)}{P_L}, n_L \cdot \tfrac{P_L \cdot (P_R - 1)}{P_R}\right), \\ &L_2 = \max\left(n_L \cdot (P_L - 1),\ n_R \cdot (P_R - 1)\right), \\ &L_3 = \max\left(n_L \cdot P_L,\ n_R \cdot P_R\right),\ L_4 = \max\left(n_L,\ n_R\right). \end{aligned}$$

The intra-region communication cost is captured by the g_1 term. Because of hierarchical planning, each node needs to act as both an intra-region coordinator and an inter-region coordinator. Without loss of generality, we analyze region L. The intra-region coordination load is $n_L \cdot (P_L - 1)$, because each node receives from every other node in the region. The inter-region coordination load can be derived by first summing the total number of triples that need to be partially aggregated within L, which is equal to the total number of triples handled by region R: $n_R \cdot P_R$. Since there are P_L parties, each party handles $n_R \cdot P_R/P_L$ triples. Finally, since each party only needs to receive from the other parties in L, the cost per party is $nR \cdot P_R(P_L - 1)/P_L$. The g_2 term captures the inter-region communication cost. Since we are doing partial aggregation, we found that the best way to capture this cost is to sum up the total number of triples per region (see L_3) and scale that according to the total inter-region bandwidth B_2. The f_1 term captures the ciphertext multiplication cost. Since that happens only once per triple at the intra-region coordinator, we have the scaling in L_4. Finally, the rest of the ciphertext cost is attributed to ciphertext addition. This can be similarly derived using the logic for deriving g_1, so we omit this due to space constraints.

6.9 Conclusion

Cerebro is a secure collaborative learning platform that allows users to program custom learning tasks without expertise in cryptography. Compared to Helen, Cerebro is much more flexible and usable, but is a first but important step towards making secure collaborative learning programmable and highly efficient.

Chapter 7

Conclusion

The systems I built for my dissertation are part of a broader vision of bringing the capabilities of advanced cryptography to the masses. By incorporating the latest theoretical advances into practical systems, I've worked towards enabling people to better collaborate and communicate across trust domains with few compromises in data privacy and security.

For future work, I would like to build upon my thesis work to continue to develop systems to enable users who have no expertise in cryptography to efficiently address their applications' needs and constraints while enjoying the benefits of provable and verifiable security. My graduate work has laid out some initial steps towards achieving these goals, and here I list several possible future research directions.

Tailored MPC protocols for learning. Helen is the first tailored cryptographic protocol for machine learning training, but the functionality is only limited to regularized linear models. An interesting direction is to build more efficient tailored protocol for training more complex models such as logistic regression. Inference is also an interesting use case, but the challenge here is that the application often needs to be more real time. It would be interesting to look into secure prediction for recurrent neural networks and LSTMs, which are different from convolutional neural networks (CNNs) and are used for speech models. An additional direction is to explore alternative threat models for MPC, such as the covert threat model.

Better systems tools for MPC. MPC research over the past decades has seen dramatic improvement in asymptotic performance from a theoretic perspective. However, there is a rich set of systems tools that can help make MPC more efficient and more usable. One existing problem is that many MPC implementations assume that the protocol only operates on a single machine at each party. Therefore, an interesting direction of research is to look into how to parallelize and distribute various MPC protocols within each trust domain to take advantage of their local cluster compute resources. Another problem is that there exist many generic MPC protocols, but no one protocol is best suited to all applications. I want to build an extensible platform that can automatically utilize new MPC backends. Cryptographers should be able to easily integrate their new MPC frameworks into this platform, and it should automatically optimize and plan high-level programs using the new backends.

Theoretical guarantees for incentivizing collaboration. Most prior work in secure collaborative computation assumes that the parties are ready to collaborate, but it is still unclear what actions need to be taken to incentivize these parties to collaborate in the first place. I want to use mechanism design to define economic policies and provide theoretical guarantees for these policies. Combining game theory with MPC also leads to an interesting threat model: what if we assume that the participants are all rational actors with utility functions, instead of a setting where an external adversary can compromise a subset of the parties?

Automated synthesis and verification of secure protocols. Cerebro shows some initial opportunities for automated compilation and optimization of MPC. Using this as a starting point, I want to explore how to automatically generate an efficient secure computation program such that its performance is close to a hand-tuned protocol. I want to focus on two specific kinds of secure protocols: oblivious computation and MPC. I think there are very interesting opportunities for applying program synthesis techniques for optimization by exploring random transformations of the program while ensuring that the transformations are still provably secure. Moreover, many cryptographic systems claim provable security, but that aspect only comes from the system design. Real world implementations of cryptography often have bugs. I want to explore program verification techniques that can help generate provable secure implementations of protocols. I also hope to extend verification to the design level and build tools that aid theoreticians in creating and checking cryptographic proofs for their protocols.

Attacks and defenses for collaborative learning. When there are multiple competing parties that compute a joint function together and release the result to everyone at the end, the parties have incentives to cheat the process. Therefore, it would be useful to think about concrete attacks that are possible in the collaborative computation setting. Understanding concrete attacks can also lead to better defenses.

Bibliography

[1] *25th*. Austin, TX, Aug. 2016.

[2] *36th*. San Jose, CA, May 2015.

[3] Daniel J. Abadi, Samuel R. Madden, and Miguel Ferreira. “Integrating Compression and Execution in Column-Oriented Database Systems”. In: *ACM International Conference on Management of Data (SIGMOD)*. 2006.

[4] Martín Abadi et al. “Tensorflow: A system for large-scale machine learning”. In: *OSDI 2016*. 2016, pp. 265–283.

[5] Martin Abadi et al. “Deep learning with differential privacy”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 308–318.

[6] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. “Succinct: Enabling Queries on Compressed Data”. In: *12th*. Oakland, CA, May 2015.

[7] Akshay Agrawal et al. “A Rewriting System for Convex Optimization Problems”. In: *Journal of Control and Decision* 5.1 (2018), pp. 42–60.

[8] Google AI. *Federated Learning: Collaborative Machine Learning without Centralized Training Data*. https://ai.googleblog.com/2017/04/federated-learning-collaborative.html.

[9] Miklós Ajtai. “Generating hard instances of lattice problems”. In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM. 1996, pp. 99–108.

[10] Andreea B. Alexandru et al. “Cloud-based Quadratic Optimization with Partially Homomorphic Encryption”. In: *arXiv preprint arXiv:1809.02267* (2018).

[11] A Aly et al. *SCALE and MAMBA*. https://github.com/KULeuven-COSIC/SCALE-MAMBA.

[12] AMPlab, University of California, Berkeley. *Big Data Benchmark*. https://amplab.cs.berkeley.edu/benchmark/.

[13] Arvind Arasu and Raghav Kaushik. “Oblivious Query Processing”. In: *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*. 2014, pp. 26–37. DOI: 10.5441/002/icdt.2014.07. URL: http://dx.doi.org/10.5441/002/icdt.2014.07.

[14] Arvind Arasu et al. “Orthogonal Security with Cipherbase”. In: *6th*. Asilomar, CA, Jan. 2013.

[15] Arvind Arasu et al. “Secure database-as-a-service with cipherbase”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 1033–1036.

[16] M. Armbrust et al. “Spark SQL: Relational Data Processing in Spark”. In: *2015*. Melbourne, Australia, May 2015.

[17] Giuseppe Ateniese et al. “Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers”. In: *IJSN* (2015).

[18] *AWS Inter-Region ping*. https://www.cloudping.co. Accessed: 2019-09-16.

[19] Sumeet Bajaj and Radu Sion. “TrustedDB: a trusted hardware based database with privacy and data confidentiality”. In: *2011*. Athens, Greece, June 2011, pp. 205–216.

[20] Johes Bater et al. “SMCQL: secure querying for federated databases”. In: *Proceedings of the VLDB Endowment* 10.6 (2017), pp. 673–684.

[21] Carsten Baum, Bernardo David, and Rafael Dowsley. “Insured MPC: Efficient Secure Multiparty Computation with Punishable Abort.” In: *IACR Cryptology ePrint Archive* 2018 (2018), p. 942.

[22] Andrew Baumann, Marcus Peinado, and Galen Hunt. “Shielding applications from an untrusted cloud with haven”. In: *ACM Transactions on Computer Systems (TOCS)* 33.3 (2015), p. 8.

[23] Assaf Ben-David, Noam Nisan, and Benny Pinkas. *FairplayMP: a system for secure multi-party computation*. www.cs.huji.ac.il/project/Fairplay/FairplayMP.html. 2008.

[24] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. “Completeness theorems for non-cryptographic fault-tolerant distributed computation”. In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM. 1988, pp. 1–10.

[25] Eli Ben-Sasson et al. “Scalable zero knowledge via cycles of elliptic curves”. In: *Algorithmica* 79.4 (2017), pp. 1102–1160.

[26] Thierry Bertin-Mahieux et al. “The Million Song Dataset.” In: *Ismir*. Vol. 2. 9. 2011, p. 10.

[27] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. “Dictionary-based order-preserving string compression for main memory column stores”. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM. 2009, pp. 283–296.

[28] Andrea Bittau et al. “Prochlo: Strong privacy for analytics in the crowd”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM. 2017, pp. 441–459.

[29] L. Susan Blackford and Myricom. “An updated set of basic linear algebra subprograms (BLAS)”. In: *TOMS*. 2002.

[30] Matt Blaze. “A Cryptographic File System for Unix”. In: *1st ACM Conference on Communications and Computing Security*. Nov. 1993, pp. 9–16.

[31] Dan Bogdanov, Sven Laur, and Jan Willemson. “Sharemind: A Framework for Fast Privacy-Preserving Computations”. In: *Computer Security - ESORICS 2008: 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*. Ed. by Sushil Jajodia and Javier Lopez. 2008.

[32] Alexandra Boldyreva et al. “Order-Preserving Symmetric Encryption”. In: *28th*. 2009.

[33] Keith Bonawitz et al. “Practical Secure Aggregation for Privacy-Preserving Machine Learning”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. 2017.

[34] George Bosilca et al. “Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA”. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE. 2011, pp. 1432–1441.

[35] Raphael Bost et al. “Machine Learning Classification Over Encrypted Data”. In: *Network and Distributed System Security Symposium (NDSS)*. 2015.

[36] Fabrice Boudot. “Efficient proofs that a committed number lies in an interval”. In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2000, pp. 431–444.

[37] Stephen Boyd et al. “Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers”. In: *Foundations and Trends in Machine Learning, Vol. 3, No. 1*. 2010.

[38] Gilles Brassard, David Chaum, and Claude Crépeau. “Minimum Disclosure Proofs of Knowledge”. In: *J. Comput. Syst. Sci.* 37.2 (Oct. 1988), pp. 156–189. ISSN: 0022-0000. DOI: 10.1016/0022-0000(88)90005-0. URL: http://dx.doi.org/10.1016/0022-0000(88)90005-0.

[39] Ferdinand Brasser et al. “Software Grand Exposure: SGX Cache Attacks Are Practical”. In: *CoRR* abs/1702.07521 (2017).

[40] *bristolcrypto/SPDZ-2: Multiparty computation with SPDZ, MASCOT, and Overdrive offline phases*. https://github.com/bristolcrypto/SPDZ-2. Accessed: 2018-10-31.

[41] Niklas Büscher et al. “HyCC: Compilation of Hybrid Protocols for Practical Secure Computation”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada: ACM, 2018, pp. 847–861. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243786. URL: http://doi.acm.org/10.1145/3243734.3243786.

[42] Jan Camenisch, Stephan Krenn, and Victor Shoup. “A Framework for Practical Universally Composable Zero-Knowledge Protocols”. In: *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*. 2011, pp. 449–467. DOI: 10.1007/978-3-642-25385-0_24. URL: https://doi.org/10.1007/978-3-642-25385-0%5C_24.

[43] Ran Canetti. “Security and composition of cryptographic protocols: a tutorial (part I)”. In: *ACM SIGACT News* 37.3 (2006), pp. 67–92.

[44] Nicholas Carlini et al. “The Secret Sharer: Measuring Unintended Neural Network Memorization & Extracting Secrets”. In: *arXiv preprint arXiv:1802.08232* (2018).

[45] Kiran Challapalli. “The Internet of Things: A time series data challenge”. In: *IBM Big data and Analytics Hub*. 2014.

[46] Nishanth Chandran et al. “EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning”. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*. 2019, pp. 496–511. DOI: 10.1109/EuroSP.2019.00043. URL: https://doi.org/10.1109/EuroSP.2019.00043.

[47] Fay Chang et al. “Bigtable: A Distributed Storage System for Structured Data”. In: *7th*. Seattle, WA, Nov. 2006.

[48] Kamalika Chaudhuri and Claire Monteleoni. “Privacy-preserving logistic regression”. In: *Advances in neural information processing systems*. 2009, pp. 289–296.

[49] Hongmei Chen and Yaoxin Xiang. “The Study of Credit Scoring Model Based on Group Lasso”. In: *Procedia Computer Science* 122 (2017). 5th International Conference on Information Technology and Quantitative Management, ITQM 2017, pp. 677–684. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2017.11.423. URL: http://www.sciencedirect.com/science/article/pii/S1877050917326716.

[50] Tianqi Chen and Carlos Guestrin. “Xgboost: A scalable tree boosting system”. In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM. 2016, pp. 785–794.

[51] Tianqi Chen et al. “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems”. In: *arXiv preprint arXiv:1512.01274* (2015).

[52] Valerie Chen, Valerio Pastro, and Mariana Raykova. “Secure Computation for Machine Learning With SPDZ”. In: *arXiv preprint arXiv:1901.00329* (2019).

[53] Xinyun Chen et al. "Targeted backdoor attacks on deep learning systems using data poisoning". In: *arXiv preprint arXiv:1712.05526* (2017).

[54] Richard Cleve. "Limits on the security of coin flips when half the processors are faulty". In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. ACM. 1986, pp. 364–369.

[55] Martine de Cock et al. "Fast, Privacy Preserving Linear Regression over Distributed Datasets Based on Pre-Distributed Data". In: *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security (AISec)*. 2015.

[56] *Conviva*. http://www.conviva.com/.

[57] James C. Corbett et al. "Spanner: GoogleÕs Globally-distributed Database". In: *10th*. Hollywood, CA, Oct. 2012.

[58] Thomas H. Cormen et al. "Sorting Networks". In: *Introduction to algorithms*. MIT Press, 2001. Chap. 27.

[59] Henry Corrigan-Gibbs and Dan Boneh. "Prio: Private, Robust, and Scalable Computation of Aggregate Statistics". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017.

[60] Victor Costan and Srinivas Devadas. "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86.

[61] Victor Costan, Ilia Lebedev, and Srinivas Devadas. "Sanctum: Minimal Hardware Extensions for Strong Software Isolation". In: *25th*. Austin, TX, Aug. 2016.

[62] Ronald Cramer, Ivan Damgård, and Jesper Nielsen. "Multiparty computation from threshold homomorphic encryption". In: *EUROCRYPT 2001* (2001), pp. 280–300.

[63] Daniel Crankshaw et al. "Clipper: A low-latency online prediction serving system". In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 613–627.

[64] Robert K. Cunningham, Benjamin Fuller, and Sophia Yakoubov. "Catching MPC Cheaters: Identification and Openability". In: *Information Theoretic Security - 10th International Conference, ICITS 2017, Hong Kong, China, November 29 - December 2, 2017, Proceedings*. 2017, pp. 110–134.

[65] Gina M. D'Angelo, D. Chandrasekhra Rao, and Chi Charles Gu. "Combining least absolute shrinkage and selection operator (LASSO) and principal-components analysis for detection of gene-gene interactions in genome-wide association studies". In: *BMC proceedings*. 2009.

[66] Ivan Damgård. "Efficient concurrent zero-knowledge in the auxiliary string model". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2000, pp. 418–430.

[67] Ivan Damgård. "On Σ-protocols". In: *Lecture Notes, University of Aarhus, Department for Computer Science* (2002).

[68] Ivan Damgård and Mads Jurik. "Client/server tradeoffs for online elections". In: *International Workshop on Public Key Cryptography*. Springer. 2002, pp. 125–140.

[69] Ivan Damgård et al. "Multiparty computation from somewhat homomorphic encryption". In: *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 643–662.

[70] Daniel J. Abadi and Samuel R. Madden and Nabil Hachem. "Column-Stores vs. Row-Stores: How Different Are They Really?" In: *ACM International Conference on Management of Data (SIGMOD)*. 2008.

[71] Gionatan Danti. "Linux compressors comparison on CentOS 6.5 x86-64: lzo vs lz4 vs gzip vs bzip2 vs lzma". In: *Hardware and software benchmark and analysis*. 2014.

[72] Giuseppe DeCandia et al. "Dynamo: AmazonÕs Highly Available Key-value Store". In: *21st*. Stevenson, WA, Oct. 2007.

[73] Harry S. Delugach and Thomas H. Hinke. "Wizard: A Database Inference Analysis and Detection System". In: *IEEE Transactions on Knowledge and Data Engineering* 8.1 (Feb. 1996), pp. 56–66. ISSN: 1041-4347. DOI: 10.1109/69.485629. URL: http://dx.doi.org/10.1109/69.485629.

[74] Daniel Demmler, Thomas Schneider, and Michael Zohner. "ABY: A Framework for Efficient Mixed-Protocol Secure Two-Party Computation". In: *NDSS'15*.

[75] Dua Dheeru and Efi Karra Taniskidou. *UCI Machine Learning Repository*. 2017. URL: http://archive.ics.uci.edu/ml.

[76] Steven Diamond and Stephen Boyd. "CVXPY: A Python-Embedded Modeling Language for Convex Optimization". In: *Journal of Machine Learning Research* 17.83 (2016), pp. 1–5.

[77] Tien Tuan Anh Dinh et al. "M2R: Enabling Stronger Privacy in MapReduce Computation". In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015.

[78] Jack J. Dongarra et al. "A set of level 3 basic linear algebra subprograms". In: *ACM Trans. Math. Softw.* 16 (1990), pp. 1–17.

[79] John C Duchi, Michael I Jordan, and Martin J Wainwright. "Local privacy, data processing inequalities, and statistical minimax rates". In: *arXiv preprint arXiv:1302.3203* (2013).

[80] Cynthia Dwork. "Differential privacy". In: *Encyclopedia of Cryptography and Security* (2011), pp. 338–340.

[81] Jonathan Ellis. *Lightweight transactions in Cassandra 2.0*. http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0.

[82] Sebastian Faust et al. "On the non-malleability of the Fiat-Shamir transform". In: *International Conference on Cryptology in India*. Springer. 2012, pp. 60–79.

[83] Pierre-Alain Fouque, Guillaume Poupard, and Jacques Stern. "Sharing decryption in the context of voting or lotteries". In: *International Conference on Financial Cryptography*. Springer. 2000, pp. 90–104.

[84] Jonathan Frankle et al. "Practical accountability of secret processes". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 657–674.

[85] Martin Franz et al. "CBMC-GC: an ANSI C compiler for secure two-party computations". In: *International Conference on Compiler Construction*. Springer. 2014, pp. 244–249.

[86] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. "Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. 2015.

[87] Matthew Fredrikson et al. "Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing". In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 2014.

[88] Andrew Friedley and Andrew Lumsdaine. "Communication Optimization Beyond MPI". In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* (2011), pp. 2018–2021.

[89] David Froelicher et al. "Drynx: Decentralized, Secure, Verifiable System for Statistical Queries and Machine Learning on Distributed Datasets". In: *arXiv preprint arXiv:1902.03785* (2019).

[90] David Froelicher et al. "Unlynx: a decentralized system for privacy-conscious data sharing". In: *Proceedings on Privacy Enhancing Technologies* 2017.4 (2017), pp. 232–250.

[91] Juan A Garay, Philip MacKenzie, and Ke Yang. "Strengthening zero-knowledge protocols using signatures". In: *Eurocrypt*. Vol. 2656. Springer. 2003, pp. 177–194.

[92] A. Gascon et al. "Secure linear regression on vertically partitioned datasets". In: *Crypto ePrint Archive*. 2016.

[93] Adrià Gascón et al. *Privacy-Preserving Distributed Linear Regression on High-Dimensional Data*. Cryptology ePrint Archive, Report 2016/892. 2016.

[94] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *41st*. Bethesda, MD, May 2009, pp. 169–178.

[95] Craig Gentry, Shai Halevi, and Nigel P. Smart. *Homomorphic Evaluation of the AES Circuit*. Cryptology ePrint Archive, Report 2012/099. June 2012.

[96] Amol Ghoting et al. "SystemML: Declarative machine learning on MapReduce". In: *2011 IEEE 27th International Conference on Data Engineering*. IEEE. 2011, pp. 231–242.

[97] Irene Giacomelli et al. *Privacy-Preserving Ridge Regression with only Linearly-Homomorphic Encryption*. Cryptology ePrint Archive, Report 2017/979. `https://eprint.iacr.org/2017/979`. 2017.

[98] Ran Gilad-Bachrach et al. "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy". In: *International Conference on Machine Learning*. 2016, pp. 201–210.

[99] Eu-Jin Goh et al. "SiRiUS: Securing Remote Untrusted Storage". In: *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium*. Internet Society (ISOC), Feb. 2003, pp. 131–145.

[100] Oded Goldreich. *Modern cryptography, probabilistic proofs and pseudorandomness*. Vol. 1. Springer Science & Business Media, 1998.

[101] Oded Goldreich. "Towards a Theory of Software Protection and Simulation by Oblivious RAMs". In: *STOC'87*. 1987.

[102] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. "Collision-Free Hashing from Lattice Problems." In: *IACR Cryptology ePrint Archive* 1996 (1996), p. 9.

[103] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to play any mental game". In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM. 1987, pp. 218–229.

[104] Oded Goldreich and Rafail Ostrovsky. "Software Protection and Simulation on Oblivious RAMs". In: *JACM'96*. 1996.

[105] Oded Goldreich and Rafail Ostrovsky. "Software Protection and Simulation on Oblivious RAMs". In: *J. ACM* 43.3 (May 1996), pp. 431–473. ISSN: 0004-5411.

[106] Gene H Golub and Charles F Van Loan. *Matrix computations*. Vol. 3. JHU Press, 2012.

[107] Joseph E. Gonzalez et al. "GraphX: Graph Processing in a Distributed Dataflow Framework". In: *11th*. Broomfield, CO, Oct. 2014.

[108] Google Research. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. https://www.tensorflow.org/. 2015.

[109] Johannes Götzfried et al. "Cache Attacks on Intel SGX". In: *Proceedings of the 10th European Workshop on Systems Security*. EuroSec'17. 2017.

[110] Jens Groth. "Homomorphic Trapdoor Commitments to Group Elements." In: *IACR Cryptology ePrint Archive* 2009 (2009), p. 7.

[111] Alon Halevy, Peter Norvig, and Fernando Pereira. "The Unreasonable Effectiveness of Data". In: *IEEE Intelligent Systems* 24.2 (Mar. 2009), pp. 8–12. ISSN: 1541-1672. DOI: 10.1109/MIS.2009.36. URL: http://dx.doi.org/10.1109/MIS.2009.36.

[112] Rob Hall, Stephen E. Fienberg, and Yuval Nardi. "Secure Multiple Linear Regression Based on Homomorphic Encryption". In: *Journal of Official Statistics*. 2011.

[113] Thomas H. Hinke. "Inference aggregation detection in database management systems". In: *IEEE Symposium on Security and Privacy*. 1988, pp. 96–106.

[114] Allison L Holloway et al. "How to barter bits for chronons: compression and bandwidth trade offs for database scans". In: *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM. 2007, pp. 389–400.

[115] Tyler Hunt et al. "Chiron: Privacy-preserving Machine Learning as a Service". In: *CoRR* abs/1803.05961 (2018). arXiv: 1803.05961. URL: http://arxiv.org/abs/1803.05961.

[116] Tyler Hunt et al. "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data." In: *OSDI*. 2016, pp. 533–549.

[117] Intel. *Intel Math Kernel Library*. https://software.intel.com/en-us/mkl.

[118] Yuval Ishai, Rafail Ostrovsky, and Hakan Seyalioglu. "Identifying cheaters without an honest majority". In: *Theory of Cryptography Conference*. Springer. 2012, pp. 21–38.

[119] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. "Secure multi-party computation with identifiable abort". In: *Annual Cryptology Conference*. Springer. 2014, pp. 369–386.

[120] Muhammad Ishaq, Ana L. Milanova, and Vassilis Zikas. "Efficient MPC via Program Analysis: A Framework for Efficient Optimal Mixing". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro et al. ACM, 2019, pp. 1539–1556. DOI: 10.1145/3319535.3339818. URL: https://doi.org/10.1145/3319535.3339818.

[121] Roger Iyengar et al. "Towards Practical Differentially Private Convex Optimization". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[122] Roger Iyengar et al. "Towards practical differentially private convex optimization". In: *Towards Practical Differentially Private Convex Optimization*. IEEE. 2019, p. 0.

[123] Thomas B. Jablin et al. "Automatic CPU-GPU communication management and optimization". In: *PLDI*. 2011.

[124] Matthew Jagielski et al. "Manipulating machine learning: Poisoning attacks and countermeasures for regression learning". In: *arXiv preprint arXiv:1804.00308* (2018).

[125] Matthias Jarke and Jürgen Koch. "Query Optimization in Database Systems". In: *ACM Comput. Surv.* 16 (1984), pp. 111–152.

[126] Yangqing Jia et al. "Caffe: Convolutional architecture for fast feature embedding". In: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 2014, pp. 675–678.

[127] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "Gazelle: A Low Latency Framework for Secure Neural Network Inference". In: *CoRR* abs/1801.05507 (2018).

[128] M. Kallahalla et al. "Plutus: Scalable secure file sharing on untrusted storage". In: *2nd USENIX conference on File and Storage Technologies (FAST '03)*. San Francisco, CA, Apr. 2003.

[129] David Kaplan, Jeremy Powell, and Tom Woller. *AMD Memory Encryption*. White paper. Apr. 2016.

[130] Marcel Keller, Emmanuela Orsini, and Peter Scholl. “MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer”. In: *CCS’16*.

[131] Marcel Keller, Emmanuela Orsini, and Peter Scholl. “MASCOT: faster malicious arithmetic secure computation with oblivious transfer”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 830–842.

[132] Marcel Keller, Valerio Pastro, and Dragos Rotaru. “Overdrive: making SPDZ great again”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2018, pp. 158–189.

[133] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. “Blowfish: Dynamic storage-performance tradeoff in data stores”. In: *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 2016, pp. 485–500.

[134] Andrew C Kidd et al. “Survival prediction in mesothelioma using a scalable Lasso regression model: instructions for use and initial performance using clinical predictors”. In: *BMJ Open Respiratory Research* 5.1 (2018). DOI: 10.1136/bmjresp-2017-000240. eprint: https://bmjopenrespres.bmj.com/content/5/1/e000240.full.pdf. URL: https://bmjopenrespres.bmj.com/content/5/1/e000240.

[135] Les King. “Why Compression is a Must in the Big Data Era”. In: *IBM Big Data and Analytics Hub*. 2013.

[136] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *arXiv preprint arXiv:1801.01203* (2018).

[137] David J. Kuck et al. “Dependence Graphs and Compiler Optimizations”. In: *POPL*. 1981.

[138] A. Lakshman and P. Malik. “Cassandra: A Decentralized Structured Storage System”. In: *ACM SIGOPS Operating Systems Review, 44(2):35Ð40*. 2010.

[139] Andrew Lamb et al. “The Vertica Analytic Database: C-store 7 Years Later”. In: *Proceedings of the VLDB Endowment* 5.12 (2012), pp. 1790–1801.

[140] Dayeol Lee et al. “Keystone: A framework for architecting tees”. In: *arXiv preprint arXiv:1907.10119* (2019).

[141] Sangho Lee et al. “Inferring fine-grained control flow inside SGX enclaves with branch shadowing”. In: *26th USENIX Security Symposium, USENIX Security*. 2017, pp. 16–18.

[142] T. Leighton. “Tight Bounds on the Complexity of Parallel Sorting”. In: *IEEE Transactions on Computers* C-34.4 (Apr. 1985), pp. 344–354. ISSN: 0018-9340. DOI: 10.1109/TC.1985.5009385.

[143] Chang Liu et al. "GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. 2015, pp. 87–101.

[144] Chang Liu et al. "ObliVM: A Programming Framework for Secure Computation". In: *36th*. San Jose, CA, May 2015.

[145] Chang Liu et al. "Oblivm: A programming framework for secure computation". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 359–376.

[146] Jian Liu et al. "Oblivious Neural Network Predictions via MiniONN Transformations". In: *CCS'17*.

[147] Jian Liu et al. "Oblivious neural network predictions via minionn transformations". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 619–631.

[148] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. "Fine-Pruning: Defending Against Backdooring Attacks on Deep Neural Networks". In: *arXiv preprint arXiv:1805.12185* (2018).

[149] Yunhui Long et al. "Understanding Membership Inferences on Well-Generalized Learning Models". In: *CoRR* abs/1802.04889 (2018). arXiv: `1802.04889`. URL: `http://arxiv.org/abs/1802.04889`.

[150] Ben Lorica. "The re-emergence of time-series". In: *O'Reilly Radar*. 2013.

[151] Samuel Madden et al. "TinyDB: an acquisitional query processing system for sensor networks". In: *ACM Trans. Database Syst.* 30 (2005), pp. 122–173.

[152] Umesh Maheshwari, Radek Vingralek, and William Shapiro. "How to Build a Trusted Database System on Untrusted Storage". In: *4th*. San Diego, CA, Oct. 2000.

[153] David Mazières and Dennis Shasha. *Building secure file systems out of Byzantine storage*. Tech. rep. TR2002–826. NYU Department of Computer Science, May 2002.

[154] Frank McKeen et al. "Innovative Instructions and Software Model for Isolated Execution". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. 2013.

[155] Frank McKeen et al. "Innovative instructions and software model for isolated execution." In: *HASP@ ISCA* 10 (2013).

[156] *MemCached*. `http://www.memcached`.

[157] X. Meng et al. "MLlib: Machine Learning in Apache Spark". In: *Journal of Machine Learning Research*. 2016.

[158] Xiangrui Meng et al. "Mllib: Machine learning in apache spark". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1235–1241.

[159] Microsoft. *Always Encrypted Database Engine*. https://msdn.microsoft.com/en-us/library/mt163865.aspx.

[160] David L Mills. "Internet time synchronization: the network time protocol". In: *IEEE Transactions on communications* 39.10 (1991), pp. 1482–1493.

[161] Pratyush Mishra et al. "Delphi: A Cryptographic Inference Service for Neural Networks". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020.

[162] Payman Mohassel and Yupeng Zhang. "SecureML: A System for Scalable Privacy-Preserving Machine Learning." In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 396.

[163] *MongoDB*. http://www.mongodb.org.

[164] Benjamin Mood et al. "Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation". In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, pp. 112–127.

[165] Arjun Narayan and Andreas Haeberlen. "DJoin: Differentially Private Join Queries over Distributed Databases". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. 2012.

[166] Kartik Nayak et al. "GraphSC: Parallel Secure Computation Made Easy". In: *36th*. San Jose, CA, May 2015.

[167] Valeria Nikolaenko et al. "Privacy-preserving ridge regression on hundreds of millions of records". In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, pp. 334–348.

[168] Valeria Nikolaenko et al. "Privacy-preserving ridge regression on hundreds of millions of records". In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, pp. 334–348.

[169] Olga Ohrimenko et al. "Oblivious Multi-Party Machine Learning on Trusted Processors". In: *25th*. Austin, TX, Aug. 2016.

[170] Olga Ohrimenko et al. "Oblivious multi-party machine learning on trusted processors". In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 619–636.

[171] Olga Ohrimenko et al. "Observing and Preventing Leakage in MapReduce". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 1570–1581.

[172] Olga Ohrimenko et al. "Observing and Preventing Leakage in MapReduce". In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. 2015.

[173] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *EUROCRYPT*. 1999, pp. 223–238.

[174] Shoumik Palkar et al. “Evaluating End-to-End Optimization for Data Analytics Applications in Weld”. In: *PVLDB* 11 (2018), pp. 1002–1015.

[175] Aurojit Panda et al. “SCL: Simplifying Distributed SDN Control Planes”. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 329–345.

[176] Charalampos Papachristou, Carole Ober, and Mark Abney. “A LASSO penalized regression approach for genome-wide association analyses using related individuals: application to the Genetic Analysis Workshop 19 simulated data”. In: *BMC Proceedings* 10.7 (Oct. 2016), p. 53. ISSN: 1753-6561. DOI: 10.1186/s12919-016-0034-9. URL: https://doi.org/10.1186/s12919-016-0034-9.

[177] Antonis Papadimitriou et al. “Big Data Analytics over Encrypted Datasets with Seabed”. In: *12th*. Savannah, GA, Nov. 2016.

[178] Vasilis Pappas et al. “Blind Seer: A Scalable Private DBMS”. In: *35th*. San Jose, CA, May 2014.

[179] Torben P. Pedersen. “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing”. In: *Advances in Cryptology - CRYPTO ’91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*. 1991, pp. 129–140. DOI: 10.1007/3-540-46766-1_9. URL: https://doi.org/10.1007/3-540-46766-1%5C_9.

[180] Raluca Ada Popa, Frank H. Li, and Nickolai Zeldovich. “An Ideal-Security Protocol for Order-Preserving Encoding”. In: *34th*. San Francisco, CA, May 2013, pp. 463–477.

[181] Raluca Ada Popa et al. “CryptDB: Protecting Confidentiality with Encrypted Query Processing”. In: *23rd*. Cascais, Portugal, Oct. 2011, pp. 85–100.

[182] Zhengping Qian et al. “MadLINQ: Large-scale Distributed Matrix Computation for the Cloud”. In: *EuroSys’12*. 2012.

[183] Aseem Rastogi, Matthew A Hammer, and Michael Hicks. “Wysteria: A programming language for generic, mixed-mode multiparty computations”. In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 655–670.

[184] *Redis*. http://www.redis.io.

[185] M. Sadegh Riazi et al. *Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications*. Cryptology ePrint Archive, Report 2017/1164. https://eprint.iacr.org/2017/1164. 2017.

[186] M. Sadegh Riazi et al. “XONN: XNOR-based Oblivious Deep Neural Network Inference”. In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pp. 1501–1518. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/riazi.

[187] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. "DeepSecure: Scalable Provably-Secure Deep Learning". In: *CoRR* abs/1705.08963 (2017).

[188] Sanjiban Roy et al. *Stock Market Forecasting Using LASSO Linear Regression Model*. Jan. 2015.

[189] Felix Schuster et al. "VC3: Trustworthy data analytics in the cloud using SGX". In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 38–54.

[190] *Scotiabank's chief risk officer on the state of anti–money laundering*. https://www.mckinsey.com/business-functions/risk/our-insights/scotiabanks-chief-risk-officer-on-the-state-of-anti-money-laundering. Oct. 2019.

[191] Ming-Wei Shih et al. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". In: *In Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*. 2017.

[192] Vitaly Shmatikov and Congzheng Song. "What Are Machine Learning Models Hiding?" In: ().

[193] Reza Shokri and Vitaly Shmatikov. "Privacy-Preserving Deep Learning". In: *CCS*. 2015.

[194] Reza Shokri et al. "Membership inference attacks against machine learning models". In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 3–18.

[195] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. "Machine learning models that remember too much". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 587–601.

[196] Ebrahim M Songhori et al. "Tinygarble: Highly compressed and scalable sequential garbled circuits". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 411–428.

[197] Emil Stefanov and Elaine Shi. "Multi-cloud oblivious storage". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 247–258.

[198] Emil Stefanov, Elaine Shi, and Dawn Song. "Towards practical oblivious RAM". In: *arXiv preprint arXiv:1106.3652* (2011).

[199] Emil Stefanov et al. "Path ORAM: an extremely simple oblivious RAM protocol". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 299–310.

[200] Florian Tramèr et al. "Stealing machine learning models via prediction apis". In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 601–618.

[201] Stephen Tu et al. "Processing Analytical Queries over Encrypted Data". In: *39th*. Riva del Garda, Italy, Aug. 2013, pp. 289–300.

[202] Anselme Tueno, Florian Kerschbaum, and Stefan Katzenbeisser. "Private evaluation of decision trees using sublinear cost". In: *Proceedings on Privacy Enhancing Technologies* 2019.1 (2019), pp. 266–286.

[203] Jo Van Bulck et al. "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution". In: *Proceedings of the 27th USENIX Security Symposium. USENIX Association*. 2018.

[204] *VIFF, the Virtual Ideal Functionality Framework*. http://viff.dk/. 2015.

[205] Stratis Viglas and Jeffrey F. Naughton. "Rate-based query optimization for streaming information sources". In: *SIGMOD Conference*. 2002.

[206] Radek Vingralek. "GnatDb: A Small-footprint, Secure Database System". In: *28th*. Hong Kong, China, Aug. 2002.

[207] Nikolaj Volgushev et al. "Conclave: secure multi-party computation on big data". In: *European Conference on Computer Systems*. 2019.

[208] Minjie Wang, Chien-chin Huang, and Jinyang Li. "Supporting Very Large Models using Automatic Dataflow Graph Partitioning". In: *EuroSys*. 2018.

[209] Xiao Wang, Hubert Chan, and Elaine Shi. "Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound". In: *CCS'15*. 2015.

[210] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. *EMP-toolkit: Efficient MultiParty computation toolkit*. https://github.com/emp-toolkit.

[211] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. "Global-scale secure multiparty computation". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 39–56.

[212] Xi Wu et al. "A Methodology for Formalizing Model-Inversion Attacks". In: *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 2016.

[213] Xi Wu et al. "Bolt-on differential privacy for scalable stochastic gradient descent-based analytics". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM. 2017, pp. 1307–1322.

[214] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-channel attacks: Deterministic side channels for untrusted operating systems". In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 640–656.

[215] Andrew C Yao. "Protocols for secure computations". In: *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*. IEEE. 1982, pp. 160–164.

[216] M. Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *9th*. San Jose, CA, Apr. 2012.

[217] Yihua Zhang, Aaron Steele, and Marina Blanton. "PICCO: a general-purpose compiler for private distributed computation". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 813–826.

[218] Wenting Zheng et al. "Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning". In: *Under submission* ().

[219] Wenting Zheng et al. "Helen: Maliciously Secure Coopetitive Learning for Linear Models". In: *S&P'19*. 2019.

[220] Wenting Zheng et al. "MiniCrypt: Reconciling encryption and compression for big data stores". In: *Proceedings of the 12th European Conference on Computer Systems*. ACM. 2017, pp. 191–204.

[221] Wenting Zheng et al. "Opaque: An Oblivious and Encrypted Distributed Analytics Platform". In: *14th*. Boston, MA, 2017.

[222] Wenting Zheng et al. "Opaque: An Oblivious and Encrypted Distributed Analytics Platform." In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 283–298.

[223] Jan Henrik Ziegeldorf, Jan Metzke, and Klaus Wehrle. "SHIELD: A Framework for Efficient and Secure Machine Learning Classification in Constrained Environments". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC '18. San Juan, PR, USA: ACM, 2018, pp. 355–370. ISBN: 978-1-4503-6569-7. DOI: 10.1145/3274694.3274716. URL: http://doi.acm.org/10.1145/3274694.3274716.