Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning

Wenting Zheng, *UC Berkeley/CMU;* Ryan Deng, Weikeng Chen, and Raluca Ada Popa, *UC Berkeley;* Aurojit Panda, *New York University;* Ion Stoica, *UC Berkeley*

https://www.usenix.org/conference/usenixsecurity21/presentation/zheng

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

Cerebro: A Platform for Multi-Party Cryptographic Collaborative Learning

Wenting Zheng[1,2], Ryan Deng[1], Weikeng Chen[1], Raluca Ada Popa[1], Aurojit Panda[3], Ion Stoica[1]
[1]*UC Berkeley,* [2]*CMU,* [3]*NYU*

Abstract

Many organizations need large amounts of high quality data for their applications, and one way to acquire such data is to combine datasets from multiple parties. Since these organizations often own sensitive data that cannot be shared in the clear with others due to policy regulation and business competition, there is increased interest in utilizing secure multi-party computation (MPC). MPC allows multiple parties to jointly compute a function without revealing their inputs to each other. We present Cerebro, an *end-to-end* collaborative learning platform that enables parties to compute learning tasks without sharing plaintext data. By taking an end-to-end approach to the system design, Cerebro allows multiple parties with complex economic relationships to safely collaborate on machine learning computation through the use of release policies and auditing, while also enabling users to achieve good performance without manually navigating the complex performance tradeoffs between MPC protocols.

1 Introduction

Recently, there has been increased interest in collaborative machine learning [1, 2], where multiple organizations run a training or a prediction task over data collectively owned by all of them. Collaboration is often advantageous for these organizations because it enables them to train models on larger datasets than what is available to any one organization, leading to higher quality models [3]. However, potential participants often own sensitive data that cannot be shared due to privacy concerns, regulatory policies [4, 5], and/or business competition. For example, many banks wish to detect money laundering by training models on customer transaction data, but they are unwilling to share plaintext customer data with each other because they are also business competitors.

Enabling these use cases requires more than a traditional centralized machine learning system, since such a platform will require a single trusted centralized party to see all of the other parties' plaintext data. Instead, the goal is to develop techniques through which participants can collaboratively compute on their sensitive data *without revealing* this data to other participants. A promising approach is to use *secure multi-party computation* (MPC) [16, 17], a cryptographic technique that allows P parties to compute a function f on their private inputs $\{x_1,...,x_P\}$ in such a way that the participants only learn $f(x_1,...,x_P)$ and nothing else about each other's inputs.

While there is a vast amount of prior work on MPC for collaborative learning, none take *an end-to-end approach*, which is essential for addressing two major obstacles encountered by these organizations. The first obstacle is the tussle between generality and performance. Many recent papers on MPC for collaborative learning [6–15] focus on hand-tuning MPC for specific learning tasks. While these protocols are highly optimized, this approach is not generalizable for a real world deployment because every new application would require extensive development by experts. On the other hand, there exist *generic* MPC protocols [16–19] that can execute arbitrary programs. However, there are many such protocols (most of which are further divided into sub-protocols [32, 33]), 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. Take the anti-money laundering use case as an example: while MPC guarantees that nothing other than the final model is revealed, this privacy property is also problematic because the banks 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 banks may worry that releasing a jointly trained model will not increase accuracy over their own models, but instead help their competitors. They may also have privacy concerns, such as whether the model itself contains too much information about their sensitive data [34–39] or whether the model is poisoned with backdoors [40].

In this paper, we present Cerebro, a platform for multi-party

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

Table 1: Comparison with prior work in categories on properties necessary for collaborative learning. There are a number of works in specialized MPC protocols [6–15], generic MPC [16–19], and MPC compilers [20–31]. Since the work space is so broad, we use "✓/✗" to indicate that only some systems in this category support that feature.

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 1.

To address the first challenge, Cerebro develops a compiler that can automatically compile any program written in our Python-like domain specific language (DSL) into an optimized MPC protocol. While there is prior work on MPC compilers [20–24, 26–29, 29–31], none provides a *holistic* tool chain for the machine learning setting, which is the focus of our work. Beyond the compiler, the ML APIs provided by Cerebro abstract away the complexities of MPC from users, while keeping information needed for our compiler to do further optimizations. Our compiler also uses novel physical planning and considers the deployment environment to further choose the best MPC algorithms for executing machine learning workloads. We note here that Cerebro's goal is to provide a generic platform where users can write *arbitrary* learning programs, and not to compete with hand-optimized protocols (we compare Cerebro's performance against such protocols in §7.5).

We address the second challenge by introducing a set of mechanisms for organizations to ensure that their incentives and constraints are met before the result of a learning task is released, and also for 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.

We implemented and evaluated Cerebro on common learning tasks—decision tree prediction, linear regression training, and logistic regression training. Our evaluation (§7) shows that our compiler generates optimized secure computation plans that are 1-2 orders of magnitude faster than an incorrect choice of a state-of-the-art generic MPC protocol (that is also un-optimized for machine learning), which is what a user might use without our system. Even with these performance gains, we want to remark that secure computation is not yet practical for all learning tasks. Nonetheless, we believe that a careful choice of protocols is practical for a number of useful learning tasks as our evaluation shows. Moreover, cryptographers have been improving MPC techniques at an impressive pace, and we believe that new MPC tools can be incorporated into the Cerebro compiler.

2 Background

Machine learning. Machine learning pipelines consist of two types of tasks: training and prediction. Training takes in a dataset and uses a training algorithm to produce a model. Prediction (or inference) takes in a model and a feature vector, and runs a prediction algorithm to make a prediction.

Secure multi-party computation (MPC). In MPC, P parties compute a function f over their private inputs $x_{i\in[1\ldots P]}$, without revealing x_i to any other parties. In this paper, we consider that the final result is released in plaintext to every party.

There are two main MPC paradigms for generic computations: *arithmetic MPC* [16,18] and *boolean MPC* (based on garbled circuits). In arithmetic MPC, data is represented as finite field elements, and the main operations are addition and multiplication (called "gates"). In boolean MPC, data is represented as boolean values, and the main operations are XOR and AND.

One interesting commonality in these two frameworks is that they can often be 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 more expensive to compute. Similarly, in boolean circuits, XOR is fast to compute while AND is much slower. The preprocessing phase for these frameworks pre-computes a majority part of executing multiplication/AND gates. And 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 MPC protocol releases the output in plaintext to all the parties.

3 Overview of Cerebro

3.1 Threat model

We consider P parties who want to compute a learning function on their sensitive data. The parties are unwilling or unable to

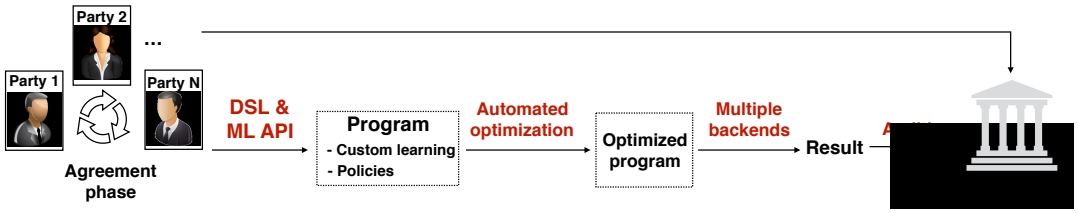


Figure 1: The Cerebro workflow.

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 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 can 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 honest party's data beyond the final result. In the malicious setting, the adversary can cause compromised participants to deviate from the protocol. The misbehavior includes altering the computation and using inconsistent inputs. Cerebro can support both settings by using 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 §7, the semi-honest protocol can be 61-3300× faster than the malicious counterpart, so the participants may not wish to sacrifice performance for malicious security.

Recent work has described many attacks for machine learning. One category is data poisoning [40] 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 [39, 41, 42] or steals model parameters from prediction results [35–38]. 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 mitigate 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 §5).

3.2 System workflow

Cerebro's pipeline consists of multiple components, as shown in Figure 1. In 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, 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. Parties need to agree on the computation (including the learning task and any compute policies) to run and agree 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 configuration, such as the number of parties and how much data each party should contribute.

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 can 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 §4.2). Next, this optimized program is input to the *physical planning* stage (see §4.3) 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 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 §5.2).

4 Programming Model and Compiler

In this section we describe Cerebro's programming model. Similar to prior work [20–24, 26–29, 29–31], and based on the DSL implementation by SCALE-MAMBA (see §6), users specify programs that Cerebro can execute using a domain-specific language (§4.1), which is then used as input to the Cerebro compiler (Figure 3). The Cerebro compiler implements two logical optimization passes, which *reduce the amount of computation expressed in MPC* while preserving security guaran-

```
# set_params() initializes parameters
# for an MPC execution, such as fixed-point
# parameters (p, f, k) and
# the number of parties (num_parties)
Params.set_params
    (p=64, f=32, k=64, num_parties=2)
# Decision tree prediction
# Reads in the tree model from party 0
tree = p_fix_mat.read_input(tree_size, 4, 0)
# Party 1 provides the features
x = p_fix_array.read_input(dim, 1)
...
for i in range(LEVELS-1):
    # Store user information in variables
    # like index and split
    ...
    cond = (x[index] < split)
    # This is a fused operation
    root = secret_index_if
        (cond, tree, left_child, right_child)
# Reveal prediction results
reveal_to_all(root[1], "Prediction")
```

Figure 2: A sample program written in Cerebro's DSL

tees. Finally, the Cerebro physical planner (§4.3) takes the logical plan generated by the compiler, and uses information about the physical deployment to instantiate and execute the plan.

4.1 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 analytics, and generic functions and are useful for expressing training and inference algorithms. Figure 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 security level indicates which parties can access the raw value of the variable. Cerebro currently supports three security levels:

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

Our current implementation restricts that private variables are 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 §4.2.1. 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 §4.2.1). Security annotations also play an important role in enabling several of the optimizations employed by Cerebro.

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 2: Rules for defining a function's execution mode

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), a set of 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 §4.2.2) provide Cerebro with more opportunities to optimize complex code patterns.

4.2 Logical optimization

Given a program written in the Cerebro DSL, the Cerebro compiler is responsible for generating a logical execution plan that minimizes runtime. Our programming model also allows us to easily apply logical optimizations. More specifically, Cerebro includes two new optimizations 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.

4.2.1 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. In this way, MPC only needs to merge pre-sorted data, providing a significant speedup over the naive solution in which it executes the entire sorting algorithm in the secure computation.

In the semi-honest setting, Cerebro can *automatically identify* opportunities for local computation within the code. As explained in §4, users write their programs using Cerebro's API, and the compiler automatically tags their data using Cerebro's secure types. Cerebro uses a set of rules (see Table 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. Thus, we cannot assume that a party will compute correct values locally. As in the sorting example, we cannot 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 sorting in plaintext at each party. However, since a malicious party can still try to input unsorted data into the secure computation, the global sorting function will first check that the inputs from each party are sorted.

This optimization allows Cerebro to automatically generate an efficient MPC protocol that has similar benefits to prior *specialized* work. For example, in [13], 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 ridge regression. We note that program splitting is compatible with cryptographic auditing mentioned in §5.2.2 by committing to the precomputed local data instead of the original input data.

4.2.2 Fused operations

Recognizing compound code patterns is crucial in MPC, since many compound operations that are cheap in plaintext 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 a 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 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 need to be executed. Therefore, conditionally accessing an index can require several scans through the same array.

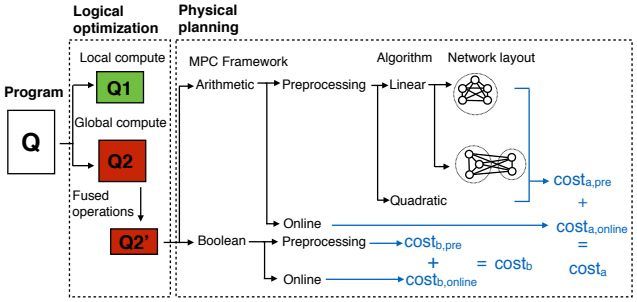


Figure 3: Cerebro architecture, showing choices we can make under the semi-honest threat model.

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 [44] operations in BLAS [45] and MKL [46], and fused operations generated by systems such as Weld [47]; 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.

4.3 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 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 (§4.3.2). To choose between these implementation options, Cerebro uses a set of cost models (§4.3.3) 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 (Appendix A.3) computation among available nodes—this choice can significantly impact performance in the wide area network.

[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 [43] because ORAM needs to keep a non-trivial amount of state.

4.3.1 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).

4.3.2 Vectorization

Cerebro supports compilation to two main MPC backends: arithmetic [18] and boolean [19]. 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 especially useful 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 describe 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 [6], 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 $A^{(i)} \cdot \mathbf{b}^{(j)}$.

4.3.3 Cost models

In this section, we provide two examples of the different cost models in Cerebro (see Appendix A.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* [18, 33]. We describe the high-level protocols in Appendix A.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:

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

The per-party cost model for quadratic preprocessing is:

$$c + N_m(P-1)(f(|C_q|) + g(B,|C_q|)) \tag{2}$$

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:

$$\begin{aligned} &c + f_1(|C_q|)(n + m(P-1)) + f_2(|C_q|)m(P-1) \\ &+ g(B,|C_q|)(m+n)(P-1) \end{aligned} \tag{3}$$

Comparing this cost model to Equation (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.

4.3.4 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

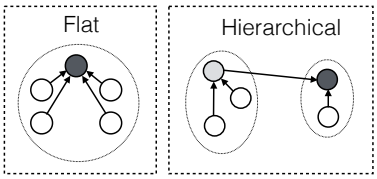


Figure 4: Communication pattern for a single multiplication triple. Shaded nodes are coordinators.

a more detailed analysis (including a walkthrough for the case of two parties), see Appendix A.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. There, 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:

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

We solve this optimization problem in cvxpy [48] by transforming it into a linear program, described in Appendix A.3. As an example, a setting with five regions is solved in roughly 100 milliseconds on a standard laptop computer.

5 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 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.

```
1 def release_policy
      (prediction_fn, test_data, weights, tau):
2     score = prediction_fn(data, weights)
3     return (score > tau)
4 # Make a call to release_policy
5 if_release = release_policy(
      lr_prediction, vdata, weights, min_score)
6 # Set weights to 0 if if_release is false
7 final_weights = release(if_release, weights)
8 return final_weights
```

Figure 5: Example validation-based release policy.

5.1 Compute policies

5.1.1 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` and calls 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 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.

5.1.2 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 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.

5.1.3 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 [39] 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 [49] 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 [50–52] 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 [34] 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. .

5.2 Cryptographic auditing

In the malicious setting, Cerebro can use a maliciously secure MPC protocol to protect against deviations during the compute phase. However, even such an MPC protocol cannot protect against any attack that happens *before* the computation begins; namely, an adversarial party can inject carefully crafted malicious input into the secure computation in order to launch an attack on the computed result.

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 [40]. If multiple self-driving car companies wish to collaboratively train a model for better object detection, a malicious participant can embed a specific backdoor pattern into non-malicious training samples and also change the corresponding prediction labels. If there are enough poisoned training samples, then the trained model will associate the backdoor pattern with a specific prediction label. If this poisoned model is deployed in a real world application by the victim in their self driving cars, the same adversary can attack the poisoned model by embedding the backdoor pattern—perhaps detecting a stop sign as a speed limit sign—thus triggering a malicious behavior that could cause a crash.

5.2.1 Overview

The previously proposed 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. Therefore, we propose an auditing framework that instead aims to hold all parties accountable for their original inputs even after the result has been released. Auditing allows parties to execute an *auditing function* on the same inputs that were used during the compute phase—Cerebro guarantees that no party can maliciously substitute an alternative sanitized input during auditing without being detected as cheating. Using the previous attack as an example: if a poisoned model is triggered during inference, the victim can request an auditing phase. During the auditing phase, all parties must first agree on a public auditing function, then undergo an audit on their input training data. If the auditing function is correctly constructed, the auditing phase should be able to identify the parties that input the malicious training samples.

We note that Cerebro's aim is to provide a *framework* for

auditing instead of specific auditing functions. Therefore, we rely on the participants to formulate auditing functions for specific attacks that they wish to protect against. In the above example, the self-driving car companies will need to design an auditing function that finds similarities in the malicious samples that trigger a misprediction and the training samples from each party. If an auditing function is not correctly formulated, then the auditing process cannot detect wrongdoing. The goal of auditing is to ensure that either the auditing function is successfully executed to completion, or the participant who causes an abort during auditing is identified (addressed in more detail in §5.2.2).

Finally, the type of threat that Cerebro is attempting to address is one where the result of the computation is attacked by constructing malicious input to the computation. Consequently, we assume that the attacker wants to get the result of the computation, and therefore do not address aborts during the compute time.

5.2.2 Auditing framework design

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 allowing a malicious participant to provide sanitized inputs during the audit, thus avoiding detection. Cerebro enforces that the same input from the compute phase is used in the auditing phase as well by using *cryptographic commitments* [53, 54], 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 showing whether the commitment matches 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, an arithmetic MPC that uses information theoretic MACs to check for protocol correctness cannot distinguish which party incorrectly triggered a MAC check failure. 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. During auditing, we require all parties to be online, and any party who is not online or aborts is identified as malicious.

Auditing workflow. 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.
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.

Using the same training example from above, we can see that any $\mathcal{P}_i$ who cheats by substituting input can only avoid detection via a badly formulated auditing function. A cheating party will be detected and identified by the auditor if it attempts to substitute an alternative copy of the input or if it attempts to abort during auditing. We provide a security argument for

the auditing process in the full version.

5.2.3 Commitment schemes

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

SIS-based commitment. Based on the short integer solution (SIS) problem in lattices, there is a class of collision-resistant hash functions [55–60], from which we can instantiate commitment schemes that are efficient in MPC. This has been used in zero-knowledge proof systems [61, 62].

Pedersen commitment. In this section, we additionally provide a way of batch checking commitments in MPC using a homomorphic commitment such as Pedersen [54]. We utilize the fact that our arithmetic framework is reactive to construct such a scheme.

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 at least 256 bits.

Tradeoffs. While SIS-based commitments work with our standard benchmarking prime field of 170 bits, Pedersen commitments need a minimum prime field size of 256 bits. Thus, while Pedersen commitments are more efficient because they enable triple batching, the larger bit size means offline generation can be more expensive. Of course, if the application already needs a larger field size (e.g., more precision for fixed-point representation), then Pedersen commitments would not have extra overhead. 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 Implementation

We implemented Cerebro's compiler on top of SCALE-MAMBA [26], 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 [20]. 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.

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 [26] and AG-MPC [20]. Additionally, we implemented Cerebro's semi-honest cryptographic backend by modifying the two backends to support semi-honest security.

7 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 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.

We also do not experimentally compare performance against a traditional centralized machine learning system. Such a system can greatly outperform an MPC-based system because it can operate directly on the parties' plaintext training data, but is also *insecure* under our definition because it requires a centralized party that sees all of the plaintext training data. Due to the lack of security, the applications we are tackling cannot be realized with a centralized learning system.

7.1 Evaluation setup

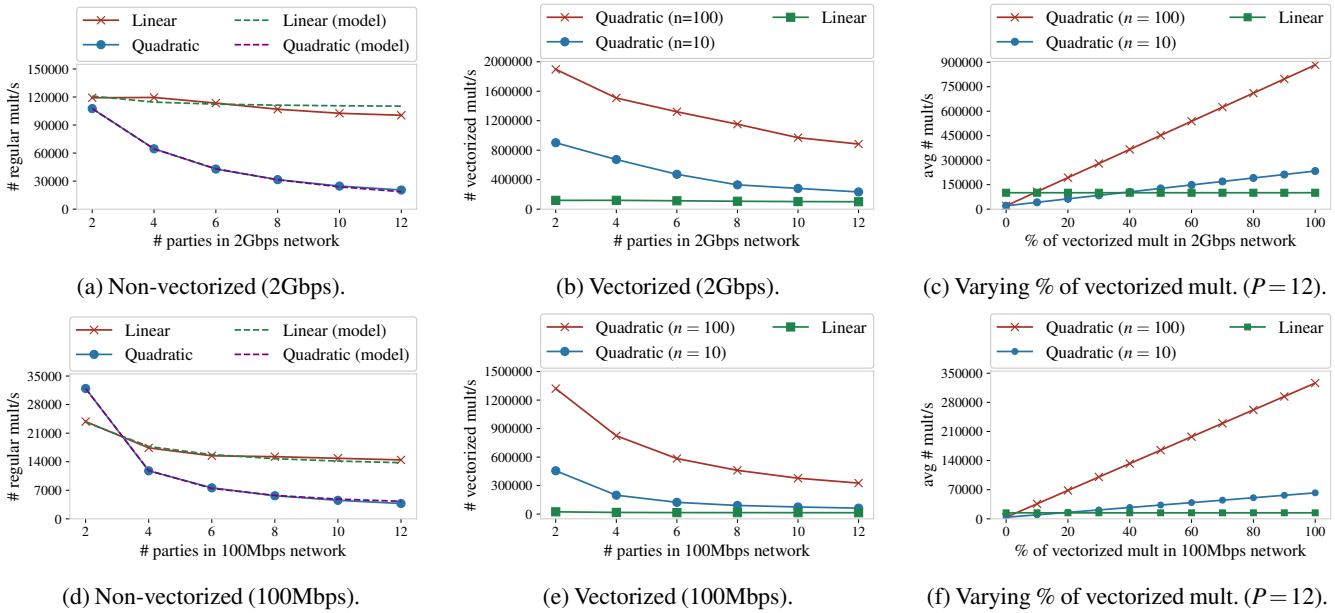
Our experiments were run on EC2 using r4.8xlarge instances. Each instance has 32 virtual CPUs and 244GB of memory. In order to benchmark in a controlled environment, we use `tc` and `ifb` to fix network conditions. Unless stated otherwise, we limit each instance to 2Gbps of upload bandwidth and 2Gbps of download bandwidth. We also adjust latency so the round trip time (RTT) is 80ms between any two instances. According to [63], this is roughly the RTT between the east-coast servers and west-coast servers of EC2 in the U.S.

7.2 Compiler evaluation

We evaluate Cerebro's compiler by answering these questions:

1. Are Cerebro's cost models accurate?
2. How do 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.



(a) Non-vectorized (2Gbps).

(b) Vectorized (2Gbps).

(c) Varying % of vectorized mult. ($P = 12$).

(d) Non-vectorized (100Mbps).

(e) Vectorized (100Mbps).

(f) Varying % of vectorized mult. ($P = 12$).

Figure 6: Choosing linear vs. quadratic protocol for arithmetic MPC preprocessing; y-axis shows triple generation throughput.

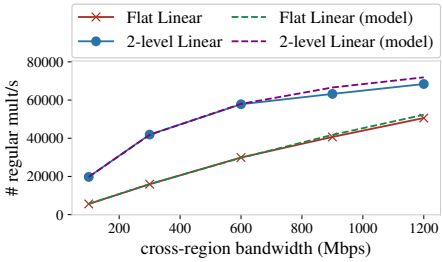


Figure 7: Flat vs. two-level linear protocol for 9-party vs. 3-party bipartite network layout with varied cross-region total bandwidth (2Gbps intra-region per-party bandwidth).

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, though a number of optimizations would be unavailable.

7.2.1 Microbenchmarks

Cost models. Our first microbenchmark compares the two methods for semi-honest arithmetic MPC preprocessing (see §4.3.3): 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 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 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 6b and 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 $16\times$ speedup over the linear protocol. Even when the number of parties is increased to 12, these two protocols still have an $8.8\times$ gap. On a slower network, the matrix-vector technique has a larger performance gain since it mainly saves communication, with up to a $55\times$ 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 6c and 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 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 $4\times$ to $4.5\times$ over the flat layout.

7.2.2 Machine learning applications

In this section we evaluate Cerebro using decision tree prediction, logistic regression training via SGD, and linear regression training via ADMM [8, 15, 64]. We estimate the 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 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 $7.5\times$ less time than the semi-honest arithmetic protocol. In Figure 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, and we exclude those results for brevity.

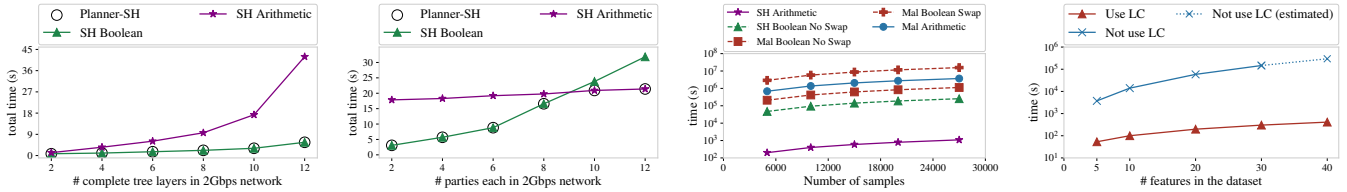
Logistic regression. We implemented and evaluated Cerebro on logistic regression training using SGD. In this experiment, we evaluated training in both the semi-honest and the malicious settings to show a difference in the performance for different variants of the protocols. For the semi-honest and malicious 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 8c. We run one epoch over the dataset in these experiments with a batch size of 128. 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 $67\times$ 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 8c, if the machines have enough memory, then the malicious boolean protocol is $3\times$ faster than the malicious arithmetic protocol, but if swap space is used instead, then the malicious boolean protocol is $4\times$ slower than the malicious arithmetic protocol. Overall, the malicious boolean protocol is up to $61\times$ slower than its semi-honest counterpart and the malicious arithmetic protocol is up to $3300\times$ 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 §4. We evaluate these benefits in Figure 8d and find that the use of local computation allows Cerebro to improve ADMM performance by up to $700\times$ when training a 40-feature model using 10000 records per party for 6 parties. 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.

7.3 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 [50, 51], 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 shows, the time for adding this noise is independent of the number of training samples, and is insigificant compared



(a) Decision tree prediction varying tree sizes.

(b) Decision tree prediction varying number of parties.

(c) Logistic regression training in different dataset size.

(d) ADMM with/without local compute.

Figure 8: Experiments on machine-learning applications (2Gbps network).

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, 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.

7.4 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 §5.2.3. Checking the commitment within MPC using a non-batching commitment scheme such as subset sum takes approximately 4.5 days while checking the commitment using a batching commitment scheme such as Pedersen commitments takes approximately 2.23 hours. The speedup is roughly $53\times$, 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.

7.5 Comparison with hand-tuned protocols

We compare with three hand-tuned protocols: SecureML's logistic regression [6], EzPC's decision tree [24], and secure ridge regression [13] (see Tables 3 to 5). Since [6] and [13] are not open sourced, we compare to the reported numbers; we ran EzPC since they provide an open source repository. These works also only support two parties who are semi-honest whereas Cerebro supports an arbitrary number of parties under different threat models. Compared to SecureML, Cerebro has

# Training samples	# Training features	Cerebro time (s)	Secure Ridge Regression time (s)
1000000	10	51.23	80
1000000	15	247.88	180
1000000	20	767.89	330

Table 3: Comparison with Secure Ridge Regression [13].

10–92$\times$ performance overhead. Cerebro performs better in the WAN setting than the LAN setting due to better batching. Compared to EzPC, Cerebro has an overhead of $3\times$. Compared to ridge regression, our compiler discovers similar insights as the hand-tuned protocol, except we can *automatically* split a program into plaintext precomputation and MPC. Cerebro is $2.5\times$ slower on a dataset with 20 features and 1 million samples, and $2.5\times$ faster for a dataset with 10 features and 1 million samples. We also tested Cerebro's performance with and without automatic optimization on a dataset with 20000 samples and 10 features, and Cerebro with precomputation is $25\times$ faster. We did not test larger circuits for the baseline because it could not run.

7.6 Discussion on automatic optimization

Based on these evaluation results, we believe that automatic compilation and optimization of MPC protocols has a lot of potential. Compared to hand-tuned MPC protocols, Cerebro's performance comes close or even exceeds that of protocols specifically tailored to a particular threat model and application. Though Cerebro cannot always compile a protocol that is as efficient as a hand-tuned version (which is true for regular compilers as well), our compiler can generalize to any learning task, hence obviating the need for users to consult an expert for every new functionality. For experts who wish to hand-optimize a learning task, Cerebro's compiled program can also act as a starting point upon which more efficient MPC protocols can be built. We hope that Cerebro can also act as a standard platform for researchers to continue to improve automatic MPC optimization. One area for research is how an MPC compiler handles memory's impact on performance. Cerebro could easily be extended to model memory usage directly for MPC backends, or work with runtime cost models with memory size as an input parameter.

# Training samples	# Training features	Network type	Cerebro time (s)	SecureML time (s)
10000	100	LAN	825.17	8.9
10000	500	LAN	2563.39	63.37
10000	100	WAN	3941.28	12.59
10000	500	WAN	10345	950.2

Table 4: Comparison with SecureML's logistic regression [6].

# Nodes	# Dims	Cerebro time (s)	EzPC time (s)
3095	13	7.15	3.67
2048	64	7.22	3.41

Table 5: Comparison with EzPC [24].

# Training samples	D.P. time (s)	Validation time (s)
1000	1.192	14.19
5000	1.192	48.66
15000	1.192	140.34
25000	1.192	238.01
27000	1.192	257.05

Table 6: Time for applying policies to logistic regression.

8 Related work

Related plaintext systems. There is a large body of prior work on distributed linear algebra systems [65–67] and machine learning training/prediction [68–73]. 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 on sensitive data. 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 [74–80]. 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 from a body of work on MPC compilers [20–29, 29–31, 81]. 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 [20–22, 26–29] and two-party compilation supporting multiple frameworks [24, 30, 81]. 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 [31] 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 is prior work that uses hardware enclaves to execute generic computation, analytics, or machine learning [82–86]. Compared with 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 [87–90].

There has been much work on secure learning using cryptography, both in training and prediction [6–15, 91–93]. However, these prior works are insufficient 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. [94] 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 [95–97] 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 [23] 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.

9 Conclusion

Cerebro is a secure collaborative learning platform that allows users to program custom learning tasks without expertise in cryptography. We have open sourced our software at https://github.com/mc2-project/cerebro and we hope that Cerebro will help enable new and rich learning applications.

10 Acknowledgment

We thank the anonymous reviewers for their valuable reviews and feedback, and we thank the SCALE-MAMBA authors for the invaluable help with their platform. This research was supported by the NSF CISE Expeditions Award CCF-1730628, NSF Career 1943347, as well as gifts from the

Sloan Foundation, Bakar, Okawa, Amazon Web Services, Ant Group, Capital One, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

References

[1] TensorFlow. Federated learning. https://www.tensorflow.org/federated/federated_learning.

[2] Santanu Bhattacharya. The new dawn of AI: Federated learning, 2019. https://towardsdatascience.com/the-new-dawn-of-ai-federated-learning-8ccd9ed7fc3a.

[3] Alon Halevy, Peter Norvig, and Fernando Pereira. The unreasonable effectiveness of data. In *IEEE Intelligent Systems '09*.

[4] GDPR. Official Journal of the European Union '16.

[5] California Consumer Privacy Act (CCPA) 2018. https://oag.ca.gov/privacy/ccpa, 2018.

[6] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *S&P '17*.

[7] Irene Giacomelli, Somesh Jha, Marc Joye, C. David Page, and Kyonghwan Yoon. Privacy-preserving ridge regression with only linearly-homomorphic encryption. In *ACNS '18*.

[8] Wenting Zheng, Raluca Ada Popa, Joseph Gonzalez, and Ion Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *S&P '19*.

[9] Jian Liu, Mika Juuti, Yao Lu, and N Asokan. Oblivious neural network predictions via MiniONN transformations. In *CCS '17*.

[10] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *SEC '18*.

[11] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In *SEC '19*.

[12] Anselme Tueno, Florian Kerschbaum, and Stefan Katzenbeisser. Private evaluation of decision trees using sublinear cost. In *PETS '19*.

[13] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *S&P '13*.

[14] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Privacy-preserving distributed linear regression on high-dimensional data. In *PETS '17*.

[15] Andreea B. Alexandru, Konstantinos Gatsis, Yasser Shoukry, Sanjit A. Seshia, Paulo Tabuada, and George J. Pappas. Cloud-based quadratic optimization with partially homomorphic encryption. In *IEEE Transactions on Automatic Control '20*.

[16] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC '88*.

[17] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *STOC '87*.

[18] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO '12*.

[19] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS '17*.

[20] Emp-toolkit: Efficient multiparty computation toolkit. https://github.com/emp-toolkit.

[21] Chang Liu, Xiao Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *S&P '15*.

[22] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *S&P '15*.

[23] Aseem Rastogi, Matthew A Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *S&P '14*.

[24] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable, efficient, and scalable secure two-party computation for machine learning. In *EuroS&P '19*.

[25] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In *CCS '18*.

[26] SCALE-MAMBA. https://github.com/KULeuven-COSIC/SCALE-MAMBA.

[27] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: A general-purpose compiler for private distributed computation. In *CCS '13*.

[28] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *EuroS&P '16*.

[29] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C compiler for secure two-party computations. In *CC '14*.

[30] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY: A framework for efficient mixed-protocol secure two-party computation. In *NDSS '15*.

[31] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: Secure multi-party computation on big data. In *EuroSys '19*.

[32] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *CCS '16*.

[33] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT '18*.

[34] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *SEC '19*.

[35] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction APIs. In *SEC '16*.

[36] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *SEC '14*.

[37] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *CCS'15*.

[38] Xi Wu, Matthew Fredrikson, Somesh Jha, and Jeffrey F. Naughton. A methodology for formalizing model-inversion attacks. In *CSF '16*.

[39] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership inference attacks against machine learning models. In *S&P'17*.

[40] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. In *ArXiv 1712.05526*.

[41] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine learning models that remember too much. In *CCS '17*.

[42] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A. Gunter, and Kai Chen. Understanding membership inferences on well-generalized learning models. In *ArXiv 1802.04889*.

[43] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *CCS '15*.

[44] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A set of level 3 basic linear algebra subprograms. In *ACM Transactions on Mathematical Software '90*.

[45] BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/.

[46] Intel Math Kernel Library. https://software.intel.com/en-us/mkl.

[47] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei A. Zaharia. Evaluating end-to-end optimization for data analytics applications in Weld. In *VLDB '18*.

[48] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. In *Journal of Machine Learning Research '16*.

[49] Cynthia Dwork. Differential privacy. In *Encyclopedia of Cryptography and Security '11*.

[50] Kamalika Chaudhuri and Claire Monteleoni. Privacy-preserving logistic regression. In *NeurIPS '09*.

[51] Xi Wu, Fengan Li, Arun Kumar, Kamalika Chaudhuri, Somesh Jha, and Jeffrey Naughton. Bolt-on differential privacy for scalable stochastic gradient descent-based analytics. In *SIGMOD '17*.

[52] Roger Iyengar, Joseph P Near, Dawn Song, Om Thakkar, Abhradeep Thakurta, and Lun Wang. Towards practical differentially private convex optimization. In *S&P '19*.

[53] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. In *Journal of Computer and System Sciences '88*.

[54] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO'91*.

[55] Miklós Ajtai. Generating hard instances of lattice problems. In *STOC '96*.

[56] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Collision-free hashing from lattice problems. In *IACR ePrint 1996/9*.

[57] Jin-Yi Cai and A.P. Nerurkar. An improved worst-case to average-case connection for lattice problems. In *FOCS '97*.

[58] Daniele Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions from worst-case complexity assumptions. In *FOCS '02*.

[59] Daniele Micciancio and Oded Regev. Worst-case to average-case reductions based on Gaussian measures. In *FOCS '04*.

[60] Chris Peikert and Alon Rosen. Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. In *TCC '06*.

[61] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO '14*.

[62] Jan Camenisch, Stephan Krenn, and Victor Shoup. A framework for practical universally composable zero-knowledge protocols. In *ASIACRYPT '11*.

[63] AWS inter-region ping. https://www.cloudping.co. Accessed: 2019-09-16.

[64] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. In *Foundations and Trends in Machine Learning '10*.

[65] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. MadLINQ: Large-scale distributed matrix computation for the cloud. In *EuroSys '12*.

[66] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, et al. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *IPDPSW '11*.

[67] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *ICDE '11*.

[68] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *NeurIPS Workshop on Machine Learning Systems '15*.

[69] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *MM '14*.

[70] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in Apache Spark. In *Journal of Machine Learning Research '16*.

[71] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD '16*.

[72] Google Brain Team. TensorFlow: A system for large-scale machine learning. In *OSDI '16*.

[73] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *NSDI '17*.

[74] Matthias Jarke and Jürgen Koch. Query optimization in database systems. In *ACM Computer Survey '84*.

[75] Stratis Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD '02*.

[76] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An acquisitional query processing system for sensor networks. In *ACM Transactions of Database Systems '05*.

[77] Andrew Friedley and Andrew Lumsdaine. Communication optimization beyond MPI. In *IPDPSW '11*.

[78] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU communication management and optimization. In *PLDI '11*.

[79] Minjie Wang, Chien chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *EuroSys '18*.

[80] David J. Kuck, Robert H. Kuhn, David A. Padua, Bruce Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *POPL '81*.

[81] Muhammad Ishaq, Ana L. Milanova, and Vassilis Zikas. Efficient MPC via program analysis: A framework for efficient optimal mixing. In *CCS '19*.

[82] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. *OSDI '14*.

[83] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *OSDI '15*.

[84] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI'17*.

[85] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. In *ArXiv 1803.05961*.

[86] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *SEC '16*.

[87] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P '15*.

[88] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT '17*.

[89] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in MapReduce. In *CCS '15*.

[90] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *S&P '19*.

[91] Jan Henrik Ziegeldorf, Jan Metzke, and Klaus Wehrle. SHIELD: A framework for efficient and secure machine learning classification in constrained environments. In *ACSAC '18*.

[92] Valerie Chen, Valerio Pastro, and Mariana Raykova. Secure computation for machine learning with SPDZ. In *PPML '18*.

[93] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *CCS '17*.

[94] Jonathan Frankle, Sunoo Park, Daniel Shaar, Shafi Goldwasser, and Daniel Weitzner. Practical accountability of secret processes. In *USENIX Security'18*.

[95] Yuval Ishai, Rafail Ostrovsky, and Hakan Seyalioglu. Identifying cheaters without an honest majority. In *TCC '12*.

[96] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multiparty computation with identifiable abort. In *CRYPTO '14*.

[97] Carsten Baum, Bernardo David, and Rafael Dowsley. Insured MPC: Efficient secure computation with financial penalties. In *FC'20*.

A More details on physical planning

A.1 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 Equation (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:

$$c+N_a(P-1)(f_1(\lambda)+g_1(B,\lambda))+g_2(B,\lambda)N(P-1) \qquad (4)$$

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.

A.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.

A.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 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:

$$\begin{aligned} T = {} & g_1(B_1)(L_1+L_2)+g_2(B_2)L_3 \\ & +f_1(|p|)L_4+f_2(|p|)(L_1+L_3) \end{aligned} \tag{5}$$

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:

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

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.

Finally, for the k region case, we can transform the optimization problem described in into a linear program by moving the max into the constraints as follows:

$$\begin{aligned} \min(L'_1+L'_2+L'_3) \text{ s.t.} \quad & \textstyle\sum_{i=1}^{k} n_i \cdot P_i \geq N_m \\ & L'_1 \geq \textstyle\sum_{j \neq i} \left(\frac{n_j \cdot P_j (P_i-1)}{P_i B_i}\right) && i = 1,2,...,k, \\ & L'_2 \geq \tfrac{n_i \cdot (P_i-1)}{B_i} && i = 1,2,...,k, \\ & L'_3 \geq \textstyle\sum_{j \neq i} \frac{n_j \cdot P_j}{B_{ij}} && i = 1,2,...,k \end{aligned}$$

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. Therefore, the equations above formulate the linear program we solve to obtain the optimal assignment of triple generation tasks.