

ACCO Algebraic Computation with Comparison

Xiaoqi Duan
duanxq17@mails.tsinghua.edu.cn
Tsinghua University
China

Vipul Goyal
vipul@cmu.edu
Carnegie Mellon University & NTT
Research
USA

Hanjun Li
hanjul@cs.washington.edu
University of Washington
USA

Rafail Ostrovsky
rafail@cs.ucla.edu
University of California, Los Angeles
USA

Antigoni Polychroniadou
antigonipoly@gmail.com
J.P. Morgan AI Research
USA

Yifan Song
yifans2@andrew.cmu.edu
Carnegie Mellon University
USA

ABSTRACT

We propose ACCO: the first maliciously secure multiparty computation engine *in the honest majority setting*, which also supports secure and *efficient* comparison and integer truncation. Our system is also the first to achieve *information theoretic* security.

We use ACCO to build an information theoretic privacy preserving machine learning system where a set of parties collaboratively train regression models in the presence of a malicious adversary.

We report an implementation of our system and compare the performance against Helen, the work of Zheng, Popa, Gonzalez and Stoica (SP'19) which provided multiparty regression models secure against malicious adversaries. Our system offers a significant speedup over Helen.

CCS CONCEPTS

• **Security and privacy** → **Information-theoretic techniques; Privacy-preserving protocols**; • **Theory of computation** → **Communication complexity; Cryptographic protocols**.

KEYWORDS

Multiparty Computation, Information-theoretic Cryptography, Communication Complexity, Privacy-preserving Machine Learning

ACM Reference Format:

Xiaoqi Duan, Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. 2021. ACCO: Algebraic Computation with Comparison. In *Proceedings of the 2021 Cloud Computing Security Workshop CCSW '21*, November 15, 2021, Virtual Event, Republic of Korea. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3474123.3486757>

1 INTRODUCTION

Financial institutions are incorporating machine learning algorithms to drive decisions in tasks such as fraud detection, loan approvals and prediction of defaults. Such algorithms perform better when being exposed to more and more data. However, financial

institutions refuse to share data as it results in a myriad of privacy related problems. Hence, there is a need to train models in a privacy-preserving manner.

Secure Multi-Party Computation (MPC)[2, 7, 17, 30] is a framework where multiple parties collaborate to compute a common function of their interest without revealing their private inputs to other parties. That said, financial institutions could collaboratively train a machine learning model on their private data such that no information about the data is revealed apart from the trained model. While it is possible to perform privacy preserving machine learning using generic maliciously secure MPC protocols for *boolean circuits*, the performance of such a system will be prohibitively low. On the other hand, there are maliciously secure MPC protocols that support efficient multiplication and addition operations for *arithmetic circuits*, but they do not efficiently support essential operations, namely comparison and truncation, required by machine learning algorithms. Helen [31] is one notable work that uses maliciously secure MPC for training machine learning models. To aim for the best of both worlds, they use MPC for boolean circuits to perform non-linear operations such as comparison, while performing efficient arithmetic operations on field elements outside of MPC. We notice a few places for improvement in this approach:

- Helen uses a maliciously secure MPC for *dishonest majority*, while in many scenarios, *honest majority* suffices. Honest majority MPC protocols are usually more efficient than dishonest majority ones.¹
- Helen uses zero-knowledge proofs to connect steps in and out of the MPC protocol. Performing all operations in the same MPC protocol can eliminate the overhead of zero-knowledge proofs.
- Given the above, Helen offers security in the computational setting (even in the online phase). The methods typically used in information-theoretic secure protocols tend to be computationally much more efficient than the cryptographic machinery required for the computational security. We also strike for *information theoretic* security.

To improve the above points, we propose ACCO: the first maliciously secure multiparty computation engine *in the honest majority setting* with information theoretic security, which also *natively* supports secure comparison and integer truncation.

¹We would like to point out that guaranteed output delivery and fairness are only possible in the honest majority setting.



This work is licensed under a Creative Commons Attribution International 4.0 License.

CCSW '21, November 15, 2021, Virtual Event, Republic of Korea.

© 2021 Copyright is held by the owner/author(s).

ACM ISBN 978-1-4503-8653-1/21/11.

<https://doi.org/10.1145/3474123.3486757>

1.1 Our Contribution

In this work, we build ACCO: a secure multiparty computation engine for arithmetic circuits in the honest majority setting which also supports secure and *efficient* comparison and integer truncation with information theoretic security against a fully malicious and static adversary.

Our construction uses elements in a finite field to represent real numbers. Let ℓ denote the length of a field element. The communication complexity per multiplication gate is $O(n \cdot \ell)$ bits. The communication complexity of our comparison protocol and truncation protocol are both $O(n^2 \cdot \ell + n \cdot \ell^2)$ bits. See Section 2 for a detailed comparison to prior works.

We next apply the ACCO engine to build a privacy preserving machine learning system. We consider a set of n parties, each holding a private dataset. The parties collaboratively train a LASSO regression model using the Alternating Direction Method of Multipliers (ADMM) algorithm. Running the training in ACCO ensures that no information about each private dataset is leaked beyond the final model. We compare the performance of our system against Helen [31], and report 73x speedup (for the million song dataset with 90 features).

2 TECHNICAL OVERVIEW

Our starting point is the best-known efficient information-theoretical MPC protocol in [18] for computing arithmetic circuits. It offers security-with-abort against a fully malicious adversary in the honest majority setting. Informally, it means that the protocol guarantees the privacy of the inputs of honest parties except the information which can be deduced from the output, while allowing a premature abort due to malicious behaviors of the adversary. The protocol in [18] proceeds in a gate-by-gate fashion supporting arithmetic circuits with addition and multiplication gates.

To run machine learning algorithms, which computes on rational values, we use field elements in a finite field $\mathbb{Z}_q$ to represent *fixed-point values*. Concretely, for a fixed-point value $\tilde{x}$ with k bits in the integer part and f bits in the decimal part, we use the field element $x := 2^f \cdot \tilde{x} \bmod q$ in $\mathbb{Z}_q$ to represent it. Note that after each multiplication, the number of bits in the decimal part doubles. We therefore need to introduce a *truncation* protocol to apply truncation to every multiplication result in [18]. Machine learning algorithms often consist of not only additions and multiplications, but also *comparisons*. Hence we also extend [18] to support secure comparisons.

Therefore, starting with the efficient protocol of [18] our goal is to build an MPC protocol with secure-with-abort which supports comparison and truncation operations. In particular, *achieving malicious security is our primary goal*. In [18], to achieve malicious security efficiently, all parties first run a lightweight protocol which does not guarantee the correctness against malicious behaviors but provide perfect secrecy, and then check the correctness of the computation using a verification protocol in [20] at the end. It means that the intermediate values which are shared among parties may not be correct. This requires the comparison and truncation protocols to satisfy that:

- They provide perfect secrecy against malicious behaviors even if the input values are incorrect.

- They can be easily verified after the computation.

Our comparison and truncation protocols are based on previous works [5, 10, 24]. However, the original protocols do not satisfy the above properties. We modify these protocols to achieve the above properties and improve the concrete efficiency of the comparison protocol as well.

We summarize our main contributions as follows:

- (1) Our first building block is a bit-decomposition protocol which transforms a degree- t Shamir sharing $[x]_t$ to a batch of degree- t Shamir sharings of the bit decomposition of x . This allows us to perform bit operations for comparison and truncation. We abstract the bit-decomposition protocol as a functionality and prove that it satisfies the above properties. See more discussion in Section 2.1.
- (2) Our second building block is a bit-wise comparison protocol which compares a shared value with bit decomposition and a public value. It is achieved by securely computing the circuit for bit-wise comparison in [15]. Our protocol only uses addition and multiplication operations. Note that addition and multiplication protocols are guaranteed to have the above properties due to the underlying MPC protocol from [18]. See more discussion in Section 2.2.
- (3) For comparison, our protocol is based on the comparison protocol in [24]. We replace the bit-wise comparison protocol in [24] by our second building block. This has the following two advantages:
 - The bit-wise comparison protocol in [24] only achieves semi-honest security. By using our second building block, the comparison protocol can fit in the MPC protocol from [18].
 - Let ℓ be the length of a field element in $\mathbb{Z}_q$. The bit-wise comparison protocol in [24] requires to compute 17ℓ multiplications with 5 rounds. On the other hand, our second building block only requires to compute 2ℓ multiplications with $\log \ell$ rounds. In the experiment, we use 127-bit Mersenne field. Compared with [24], our protocol improves the communication complexity by a factor of 8.5 at a cost of 2 additional rounds.
- (4) For truncation, our protocol is based on the truncation protocol in [5]. The original protocol only achieves semi-honest security. We modify the protocol in [5] to satisfy the above properties. See more discussion in Section 2.3.
- (5) Finally, we extend the original MPC protocol in [18] to also support inner-product operations. See more discussion in Section 2.5

2.1 Bit Decomposition

In [18], every field element is *secret shared*. The main difficulty of performing truncation and comparison over these shared values comes from the fact that we don't have access to their bit representations. Let $[x]_t$ denote a degree- t secret sharing of some value x in the field $\mathbb{Z}_q$. Let $\ell = |x|$ be the bit-length of x , ideally we want ℓ degree- t sharings for each of x 's bit:

$$[x_\ell]_t, [x_{\ell-1}]_t, \dots, [x_1]_t$$

where $(x_\ell x_{\ell-1} \dots x_1)_2$ is the binary representation of x . While there are protocols to achieve directly this, these protocols either have

a large overhead in the communication complexity or require the number of rounds to be proportional to the length of a field element. Our idea is to prepare a random sharing $[r]_t$ with bit decomposition $[r_\ell]_t, [r_{\ell-1}]_t, \dots, [r_1]_t$, and transform $[x]_t$ to

$$x \oplus r \text{ and } [r]_t, [r_{\ell-1}]_t, \dots, [r_1]_t,$$

which is much more efficient than preparing the bit decomposition of x . Note that $x \oplus r$ can be obtained by reconstructing $[x \oplus r]_t := [x]_t \oplus [r]_t$. We will show that it is sufficient for comparison and truncation.

To prepare a random sharing $[r]_t$ with bit decomposition $[r_\ell]_t, [r_{\ell-1}]_t, \dots, [r_1]_t$, we rely on a semi-honest protocol from [10], which allows all parties to prepare random bit sharings. To obtain malicious security, we use the multiplication verification protocol from [20] (which is used in the design of the MPC protocol in [18]) to check its correctness. More details can be found in Section 5.1. After preparing ℓ random bit sharings $[r_\ell]_t, [r_{\ell-1}]_t, \dots, [r_1]_t$, all parties can locally compute

$$[r]_t := \sum_{i=1}^{\ell} 2^{i-1} [r_i]_t$$

Note that $r = \sum_{i=1}^{\ell} 2^{i-1} r_i \pmod{q}$, and $(r_\ell r_{\ell-1} \dots r_1)_2$ is the correct bit representation of r only when $\sum_{i=1}^{\ell} 2^{i-1} r_i < q$. To ensure this, we choose to use a Mersenne field (i.e. $q = 2^\ell - 1$ for some ℓ), in which we have $\sum_{i=1}^{\ell} 2^{i-1} r_i \leq q$. In particular, $\sum_{i=1}^{\ell} 2^{i-1} r_i = q$ only when $r_1 = r_2 = \dots = r_\ell = 1$, which happens with probability $1/2^\ell$. When ℓ is large enough, the procedure succeeds with overwhelming probability.

Using $[r]_t$ we compute $[x \oplus r]_t = [x]_t \oplus [r]_t$, and reconstructs $x \oplus r$ publicly to every party. When r is uniform, $x \oplus r$ is also uniform and independent of x . Since $x \oplus r$ is the only information corrupted parties learn from the above process, the secrecy of x is protected. In Section 5.1, we present the ideal functionality $\mathcal{F}_{\text{decompose}}$ for the above process, and prove the security of our construction.

2.2 Bit-wise Comparisons

Instead of comparing two secret shared values, we first consider a simpler scenario: Given a sharing $[x]_t$ with its bit decomposition $[x_\ell]_t, [x_{\ell-1}]_t, \dots, [x_1]_t$ and a public value r , we want to securely compute $[r < x]_t$, a sharing of the comparison result. Our idea is to securely compute a circuit for the comparison. We choose the circuit designed in [15] due to its simplicity and efficiency. Let $(r_\ell \dots r_1)_2$ be the bit decomposition of r . Then the comparison result of $r < x$ is equal to $r_j < x_j$ where j is the largest index such that $r_j = x_j$, or 0 if such a j does not exist. For all $i \in [\ell]$, let c_i denote the comparison result of $r_i < x_i$ and b_i denote whether $r_i = x_i$. Then, $\prod_{k=i-1}^{\ell} b_k$ represents whether the first $\ell - i$ bits of r and x are equal. We can represent the comparison result of $r < x$ as follows:

$$(r < x) = c_\ell \oplus b_\ell c_{\ell-1} \oplus \dots \oplus \left(\prod_{k=2}^{\ell} b_k \right) c_1$$

To see this, note that if $r = x$, then $c_i = 0$ for all $i \in [\ell]$ and therefore the final result is 0. If $r < x$, let j be the largest index such that $r_j = x_j$. For $i > j$, we have $c_i = 0$ and therefore $\prod_{k=i-1}^{\ell} b_k c_i = 0$.

For $i < j$, we have $b_j = 0$ and therefore $\prod_{k=i-1}^{\ell} b_k c_i = 0$. Thus, $(r < x) = \prod_{k=j-1}^{\ell} b_k c_j = c_j$.

The circuit in [15] follows from a straightforward divide-and-conquer technique to compute the comparison result. As a result, the depth of the circuit is $\log \ell$ and the number of multiplications is 2ℓ . Our protocol uses the secure multiplication protocol in [18] to compute this circuit. The protocol servers as a building block for secure comparison and truncation.

2.3 Secure Comparison and Truncation

For secure comparison, we borrow the idea from [24] to transform the problem of comparing two secret shared values into the following:

- All parties first transform the problem of comparing two values to extracting *the least significant bit* of one value.
- Relying on $\mathcal{F}_{\text{decompose}}$, the least significant bit can be obtained by computing a bit-wise comparison.

More details can be found in Section 5.2.

For secure truncation, our starting point is the idea from [5]. The original protocol also transforms the input sharing $[x]_t$ to $x \oplus r$ and the bit decomposition of $[r]_t$, i.e., $[r_\ell]_t, [r_{\ell-1}]_t, \dots, [r_1]_t$. However the correctness of the protocol requires $x \oplus r < q$. In [5], this is achieved by restricting the ranges of both x and r . For r , we can control its range by using fewer random bit sharings. For x , however, it would require our protocol to perform a range check before using the protocol in [5] to do truncation. This is due to the fact that in [18], the multiplication protocol is only secure up to an additive attack. Without the final verification step, the multiplication protocol does not guarantee the correctness. *That means, even if the correct value of the machine learning algorithm guarantees that it is bounded by a certain range, the value which is shared during the computation can still be out of this range due to the malicious behaviors of corrupted parties.* In addition to the security issue, the approach in [5] inherently requires to use a large field which has κ extra bits in each field element, where κ is the security parameter. These extra bits are used to avoid leaking information about x from $x \oplus r$.

To avoid the expensive range check for each truncation operation and the overhead in communicating field elements, we modify the protocol in [5] by using a uniformly random r , and computing $x \oplus r < q$ using the previously described bit-wise comparison protocol. Finally, we can compute the truncation result using the comparison result of $x \oplus r < q$. More details can be found in Section 5.2.

2.4 Security of Our Extensions

To evaluate a circuit with addition gates, multiplication gates, comparison gates, and truncation gates, we combine the secure-without-abort MPC protocol in [18] with our secure comparison and truncation protocols. We make use of the functionality $\mathcal{F}_{\text{mult}}$ and its instantiation from [18] to evaluate multiplications. Our comparison and truncation protocols are composed of local computations, $\mathcal{F}_{\text{decompose}}$, and $\mathcal{F}_{\text{mult}}$. Since $\mathcal{F}_{\text{decompose}}$ is always correct, we only need to verify multiplications computed by $\mathcal{F}_{\text{mult}}$, which is achieved by using the functionality $\mathcal{F}_{\text{multVerify}}$ and its instantiation in [20]

(which is used in the design of the MPC protocol in [18]). The security of our protocol follows from the fact that both $\mathcal{F}_{\text{decompose}}$ and $\mathcal{F}_{\text{mult}}$ do not leak any information about the inputs. We refer the readers to Section 5.4 for more details.

2.5 Supporting Inner-product Operations

We note that machine learning algorithms usually require a lot of inner-product operations. An inner-product operation is to compute the summation of the coordinate-wise multiplications between two vectors. Although an inner-product operation can be computed by a batch of multiplication operations, the communication complexity will grow linearly with the dimension of the two input vectors.

It has been observed in many previous works (e.g., [8, 18, 20]) that the multiplication protocol in [9] can be extended to compute an inner-product operation *with the same communication complexity as a multiplication operation*. We make use of the functionality $\mathcal{F}_{\text{inner}}$ and its instantiation from [18] to evaluate inner-product operations. As $\mathcal{F}_{\text{mult}}$, $\mathcal{F}_{\text{inner}}$ does not guarantee the correctness of the results, we need to check the correctness of inner-products computed by $\mathcal{F}_{\text{inner}}$. We show that by following the techniques in [20], we can modify the protocol for $\mathcal{F}_{\text{multVerify}}$ in [20] to support the verification of inner-products. We refer the readers to Section 5.3 for more details.

3 RELATED WORK

Many prior works have considered the problem of privacy-preserving machine learning (PPML). Generally speaking, securely training a machine learning model is more difficult than inference using a given model. Any system that achieves training usually is also capable of inference. We therefore focus on comparing with systems that also achieves training.

SecureML[23] uses secure two-party computation (2PC) for training linear regression, logistic regression, and neural network models. They implemented the stochastic gradient descent method for training, and proposed several low-degree polynomial approximation for the non-polynomial functions required in those models. When computing non-arithmetic operations on secret shared values, they used the ABY framework [11] to switch from arithmetic sharing to Yao sharing and back. Chameleon [26] is a similar work that focuses on the 2PC setting and also requires conversion between arithmetic sharing and Yao sharing. The ABY framework is extended to work in the 3-party setting by ABY3 [22]. In contrast, in our ACCO system we extended our algebraic MPC protocol to directly perform non-arithmetic operations on Shamir's secret sharing, and avoid expensive conversion to other secret sharing schemes.

SecureNN[29] builds a specialized MPC protocols for 3 and 4 parties to perform neural network training and inference. Blaze [25] is a similar work that builds an MPC protocol for 3 parties over the ring $\mathbb{Z}_2$ for training linear regression, logistic regression, and neural network models. Similar to our work, SecureNN [29] and Blaze [25] proposed custom comparison and truncation protocols to avoid converting between arithmetic sharing and Yao sharing. SecureML [23], Chameleon [26], and SecureNN [29] provides semi-honest security against one corruption. Blaze [25] and ABY3 [22] provides malicious security against one corruption. In contrast,

our MPC protocol works for any number of parties and provides malicious security for honest majority.

Helen [31] builds a system by combining an MPC protocol with custom zero-knowledge proofs for training LASSO regression models. Helen [31] optimizes its running time by using MPC only on parts of its training algorithm, and connecting different runs of MPC protocols with zero-knowledge proofs to guarantee security. In contrast, our work uses a much more efficient MPC protocol and runs the whole training algorithm within a single run of MPC. We therefore avoid computing expensive zero-knowledge proofs. Helen [31] provides computational security against any number of corruption while our work provides information theoretical security in the case of honest majority.

Differential Privacy. Our MPC protocol guarantees that an adversary does not gain any information about each party's private inputs beyond the trained model. However, an adversary can still try to learn information from the model. Differential privacy [14] is exactly the notion to counter such attacks. Works on adding differential privacy to machine learning include [1] and [21]. Many works also consider combining MPC with differential privacy: [28], [6], and [13]. In our setting we assume that the final trained model is revealed to everyone. Differential privacy could be used by adding noise before model release.

4 PRELIMINARIES

4.1 Model

In the problem of Privacy Preserving Machine Learning, we consider a set of parties $\mathcal{P} = \{P_1, P_2, \dots, P_n\}$ where each party holds a set of private data as samples for a machine learning problem. The parties want to collaboratively train a model that fits the union of their datasets without leaking any information (beyond the model itself) about each of their private dataset.

This problem can be a natural application for the cryptography frame work of secure multi-party computation (MPC). In this work, we consider the *client-server model*. In the client-server model, the clients provide inputs to the functionality and receive outputs, and the servers participate in the computation but do not have inputs or get outputs. Note that, each party may have different roles in the computation, and if every party plays a single client and a single server, then we get the standard MPC model. For all clients and servers, we assume that every two of them are connected via a secure (private and authentic) synchronous channel so that they can directly send messages to each other

Let c denote the number of clients and $n = 2t - 1$ denote the number of servers. An adversary $\mathcal{A}$ can corrupt at most c clients and t servers, provide inputs to corrupted clients, and receive all messages sent to corrupted clients and servers. Corrupted clients and servers can deviate from the protocol arbitrarily. In this work, we focus on static adversaries, which means that the corrupted clients and servers should be fixed before the computation. We refer the readers to Appendix A for the security definition. Let κ be the security parameter. We assume that errors which happen with negligible probability in κ are acceptable.

One convenient result from [16] says:

THEOREM 4.1 (LEMMA 5.2 [16]). *Let π be a protocol computing a c -client circuit C using $n = 2t + 1$ parties. Then, if π is secure against any adversary controlling exactly t parties, then π is secure against any adversary controlling at most t parties.*

This theorem allows us to only consider the case where the adversary controls exactly t parties, which we assume in the following.

4.2 Data Presentation

We assume inputs and outputs to the machine learning problem, and all intermediate values are in the range $(-2^k, 2^k)$. We will use fixed point rational numbers with f bits in the decimal part to approximate the inputs and the intermediate values. Therefore, the inputs and the intermediate values are approximated to values in the set $\mathbb{Q}_{(k,f)} = \{\tilde{x} \in \mathbb{Q} \mid \tilde{x} = x \cdot 2^{-f}, x \in \mathbb{Z}, -2^k \leq x < 2^k\}$.

A fixed point number $\tilde{x}$ is represented by an element x in a finite field $\mathbb{Z}_q$ where q is a Mersenne prime of the form $q = 2^\ell - 1$ for some ℓ . The representation is given by $x = \tilde{x} \cdot 2^f \bmod q$. We choose a field large enough such that $\ell > k + 2f + 1$. This guarantees that the multiplication operation will not lead to overflow in $\mathbb{Z}_q$. This is because we will have $2f$ bits for the decimal part after the multiplication and k bits for the integer part (since we assume that the result is in the range $(-2^k, 2^k)$). After each multiplication operation, the last f bits in the decimal part will be truncated to ensure that the result is in the set $\mathbb{Q}_{(k,f)}$.

In the following, we assume that ℓ is greater than the security parameter κ .

4.3 Secret Sharing

In this work, we will use the standard Shamir Secret Sharing Scheme [27]. Let n be the number of parties and $\mathbb{G}$ be a finite field of size $|\mathbb{G}| \geq n + 1$. Let $\alpha_1, \dots, \alpha_n$ be n distinct non-zero elements in $\mathbb{G}$.

A degree- d Shamir sharing of $x \in \mathbb{G}$ is a vector $(x_1, \dots, x_n)$ which satisfies that, there exists a polynomial $f(\cdot) \in \mathbb{G}[X]$ of degree at most d such that $f(0) = x$ and $f(\alpha_i) = x_i$ for $i \in \{1, \dots, n\}$. Each party P_i holds a share x_i and the whole sharing is denoted by $[x]_d$.

Properties of the Shamir Secret Sharing Scheme. In the following, we will utilize two properties of the Shamir secret sharing scheme.

- Linear Homomorphism:

$$[x]_d, [y]_d, [x + y]_d = [x]_d + [y]_d$$

- Multiplying two degree- d sharings yields a degree- $2d$ sharing. The secret value of the new sharing is the product of the original two secrets.

$$[x]_d, [y]_d, [x \cdot y]_{2d} = [x]_d \cdot [y]_d$$

4.4 Building Blocks from [18, 20]

The MPC protocol in [18] is based on Shamir secret sharing [27]. At a high level, all parties will compute a degree- t sharing for each wire in the circuit. Since the Shamir secret sharing scheme is linear homomorphic, addition gates can be evaluated locally. For each multiplication gate, all parties will invoke the multiplication protocol in [18] to compute the result. The protocol in [18] also makes use of several building blocks in [20]. We briefly describe the building blocks from [18, 20] which will be used in our construction.

- The functionality $\mathcal{F}_{\text{mult}}$ allows all parties to evaluate a multiplication gate. It takes two input sharings $[x]_t, [y]_t$ and outputs $[z]_t := [x \cdot y]_t$. An instantiation of $\mathcal{F}_{\text{mult}}$ can be found in [18]. The communication complexity is $O(n)$ field elements.
- The functionality $\mathcal{F}_{\text{multVerify}}$ allows all parties to verify the correctness of all previous multiplications. An instantiation of $\mathcal{F}_{\text{multVerify}}$ can be found in [20] (Protocol 17 in Section 5.4 of [19]). The communication complexity is $O(n^2 \cdot \log N \cdot \kappa)$ bits, where N is the number of multiplications and κ is the security parameter.
- The functionality $\mathcal{F}_{\text{rand}}$ allows all parties to prepare a random degree- t Shamir sharing. An instantiation of $\mathcal{F}_{\text{rand}}$ can be found in [9, 20] (Protocol 2 in Section 3.3 of [19]). The amortized communication cost per sharing is $O(n)$ field elements.
- The functionality $\mathcal{F}_{\text{coin}}$ allows all parties to generate a random element. An instantiation of $\mathcal{F}_{\text{coin}}$ can be found in [20] (Protocol 6 in Section 3.5 of [19]). The communication complexity of the instantiation is $O(n^2)$ field elements.

The description of these functionalities can be found in Appendix B, and we refer the readers to [18, 20] for more details of the protocols.

5 CONSTRUCTIONS

In this section, we describe the construction of our extensions and modifications to [18]. First, we introduce useful building blocks to support comparison and truncation operations. Then, we show how to combine all the components together into ACCO, a secure multiparty computation engine for arithmetic circuits which also supports secure comparison and integer truncation.

5.1 Useful Building Blocks

In this part, we will introduce two functionalities which will be used for comparison and truncation operations.

- The first functionality $\mathcal{F}_{\text{bit}}$ allows all parties to prepare a degree- t Shamir sharing of a random bit.
- The second functionality $\mathcal{F}_{\text{decompose}}$ allows all parties to do bit-decomposition of an input sharing $[x]_t$. However, instead of obtaining a batch of sharings of the bit decomposition of x , $\mathcal{F}_{\text{decompose}}$ randomly samples r , and outputs $x - r$ with a batch of sharings of the bit decomposition of r . We will show that this is sufficient for comparison and truncation operations.

5.1.1 Preparing random bit sharings. In this part, we describe a protocol from [10] which allows all parties to prepare random bit sharings. The original protocol is only secure in the semi-honest setting. To obtain malicious security, we rely on $\mathcal{F}_{\text{multVerify}}$ to check the correctness of the computation. The functionality $\mathcal{F}_{\text{bit}}$ appears in Functionality 1. The description of BRTs appears in Protocol 2. The communication complexity of BRTs is $O(N \cdot n \cdot \ell - n^2 \cdot \log N \cdot \ell)$ bits, where ℓ is the length of an element in $\mathbb{Z}_q$.

At a high-level, the protocol in [10] transforms a non-zero random sharing $[r]_t$ into a random bit sharing $[b]_t$ where $b \in \{0, 1\}$. The first step is to compute $r^2 \bmod q$ and open the result to all parties. Then, each party can locally compute r' such that $(r')^2 \equiv r^2$

mod q and $0 < r' < q/2$. Note that if $0 < r < q/2$, we have $r = r'$. Otherwise, we have $r \equiv -r' \pmod{q}$. Therefore, r/r' is either 1 or

1. All parties can obtain a random bit sharing $[b]_t$ by computing $[b]_t := ([r]_t / r' - 1)/2$.

Functionality 1: $\mathcal{F}_{\text{bit}}$

- (1) $\mathcal{F}_{\text{bit}}$ receives from the adversary the set of shares $\{r_i\}_{i \in C}$.
- (2) $\mathcal{F}_{\text{bit}}$ randomly samples a bit $b \in \{0, 1\}$. Based on the secret b and the t shares $\{r_i\}_{i \in C}$ of corrupted parties, $\mathcal{F}_{\text{bit}}$ reconstructs the whole sharing $[b]_t$.
- (3) $\mathcal{F}_{\text{bit}}$ asks the adversary whether it should continue.
 - If the adversary replies abort, $\mathcal{F}_{\text{bit}}$ sends abort to honest parties.
 - If the adversary replies continue, $\mathcal{F}_{\text{bit}}$ distributes the shares of $[b]_t$ to honest parties.

LEMMA 5.1. *The protocol BITS securely computes $\mathcal{F}_{\text{bit}}$ with abort in the $(\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{mult}}, \mathcal{F}_{\text{multVerify}})$ -hybrid model in the presence of a fully malicious adversary controlling t corrupted parties.*

We refer the readers to Appendix D.1 for the proof of Lemma 5.1. Recall that the field size of $\mathbb{Z}_q$ is $q = 2^\ell - 1$. Therefore, a field element r can be represented using ℓ bits. Given ℓ random bit sharings $[r_1]_t, \dots, [r_\ell]_t$, we can map them to a random sharing $[r]_t = \sum_{i=1}^{\ell} 2^{i-1} [r_i]_t$. In particular, $(r_\ell - r_1)_2$ can be seen as the binary representation of r when $r < q$. Since we choose q to be a Mersenne prime, we always have $r \leq q$ and the probability that $r = q$ is negligible. Therefore, with overwhelming probability, $([r_\ell]_t, \dots, [r_1]_t)$ is a bit decomposition of $[r]_t$.

5.1.2 Bit Decomposition. We note that comparison and truncation are very difficult to compute by only using additions and multiplications. Therefore, we want to transform the input sharing $[x]_t$ to a batch of degree- t Shamir sharings of the bit decomposition of x so that we can perform bit operations to compute comparison and truncation. However, it is unclear how to do it efficiently. Our idea is to prepare a random sharing $[r]_t$ with bit decomposition, and transform $[x]_t$ to $x - r$ and a batch of degree- t Shamir sharings of the bit decomposition of r . We will show that it is sufficient for comparison and truncation.

The functionality $\mathcal{F}_{\text{decompose}}$ appears in Functionality 3. The protocol DECOMPOSE (Protocol 4) makes use of $\mathcal{F}_{\text{bit}}$ to prepare a random degree- t Shamir sharing with bit decomposition. The communication complexity of DECOMPOSE is $O(n^2 \cdot \ell)$ bits plus ℓ calls of $\mathcal{F}_{\text{bit}}$, where ℓ is the length of an element in $\mathbb{Z}_q$. In our main protocol, all random bit sharings will be prepared together.

LEMMA 5.2. *The protocol DECOMPOSE securely computes $\mathcal{F}_{\text{decompose}}$ with abort in the $\mathcal{F}_{\text{bit}}$ -hybrid model in the presence of a fully malicious adversary controlling t corrupted parties.*

We refer the readers to Appendix D.2 for the proof of Lemma 5.2.

5.2 Secure Comparison and Truncation

5.2.1 Bit wise Comparison. We first consider a simple scenario: Given a sharing $[x]_t$ with the bit decomposition $([x_\ell]_t, \dots, [x_1]_t)$,

Protocol 2: BITS

- (1) Let N denote the number of random bit sharings all parties want to generate. All parties invoke $\mathcal{F}_{\text{rand}}$ to prepare N random degree- t Shamir sharings:

$$[r_1]_t, [r_2]_t, \dots, [r_N]_t$$

- (2) For all $i \in [N]$, all parties invoke $\mathcal{F}_{\text{mult}}$ to compute $[r_i^2]_t$.
- (3) All parties invoke $\mathcal{F}_{\text{multVerify}}$ to check the following multiplication tuples:

$$([r_1]_t, [r_1]_t, [r_1^2]_t), \dots, ([r_N]_t, [r_N]_t, [r_N^2]_t)$$

- (4) In this step, all parties reconstruct the sharings $[r_1^2]_t, [r_2^2]_t, \dots, [r_N^2]_t$.

- (a) For all $i \in [N]$, all parties send their shares of $[r_i^2]_t$ to the first party P_1 .

- (b) For all $i \in [N]$, P_1 checks whether the shares of $[r_i^2]_t$ are consistent. If not, P_1 aborts. Otherwise, P_1 reconstructs the secret r_i^2 and sends it to all other parties.

- (c) All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random element $\lambda \in \mathbb{Z}_q$. All parties locally compute:

$$\begin{aligned} [x]_t &= [r_1^2]_t - [r_2^2]_t \cdot \lambda - [r_N^2]_t \cdot \lambda^{N-1} \\ x &= r_1^2 - r_2^2 \cdot \lambda - r_N^2 \cdot \lambda^{N-1} \end{aligned}$$

- (d) All parties send their shares of $[x]_t$ to other parties to reconstruct x . If a party P_j receives an inconsistent sharing or the reconstruction result is different from the value P_j computed in the last step, P_j aborts.

- (5) For all $i \in [N]$, each party locally computes r'_i such that $(r'_i)^2 \equiv r_i^2 \pmod{q}$ and $0 \leq r'_i < q/2$. If $r_i^2 = 0$, all parties abort.

- (6) For all $i \in [N]$, all parties locally compute

$$[b_i]_t = ([r_i]_t / r'_i - 1)/2$$

All parties output $[b_1]_t, \dots, [b_N]_t$.

i.e., $x = (x_\ell - x_1)_2$, and a public field element r , we want to securely compute a sharing of the comparison result of $r < x$.

To this end, suppose $(r_\ell - r_1)_2$ is the binary representation of r . Then the comparison result of $r < x$ is equal to $r_j < x_j$ where j is the largest index such that $x_j = r_j$, or 0 if such a j does not exist. For all $i \in [\ell]$, let c_i denote the comparison result of $r_i < x_i$ and b_i denote whether $x_i = r_i$. Then the comparison result of $r < x$ can be computed by

$$c = c_\ell - b_\ell c_{\ell-1} - \left(\prod_{k=2}^{\ell} b_k \right) c_1$$

Note that for all $i \in [\ell]$, c_i can be computed by $c_i := x_i(1 - r_i)$, and b_i can be computed by $b_i := 1 - r_i - x_i - 2r_i x_i$. Therefore, all parties first locally compute $\{[b_i]_t, [c_i]_t\}_{i=1}^{\ell}$ using r and $\{[x_i]_t\}_{i=1}^{\ell}$. To compute $[c]_t$, we use the circuit designed in [15]. The main

Functionality 3: $\mathcal{F}_{\text{decompose}}$

- (1) Let $[x]_t$ denote the input sharing. $\mathcal{F}_{\text{decompose}}$ receives from honest parties their shares of $[x]_t$. Then $\mathcal{F}_{\text{decompose}}$ reconstructs the secret x . $\mathcal{F}_{\text{decompose}}$ further computes the shares of $[x]_t$ held by corrupted parties, and sends these shares to the adversary.
- (2) $\mathcal{F}_{\text{decompose}}$ samples a random element $r \in \mathbb{Z}_q$ and sends $x \oplus r$ to the adversary.
- (3) Let ℓ denote the length of an element in $\mathbb{Z}_q$. For all $i \in [\ell]$, $\mathcal{F}_{\text{decompose}}$ receives from the adversary a set of shares $\{r_{i,j}\}_{j \in C}$, where C is the set of corrupted parties. Let $(r_\ell r_{\ell-1} \dots r_1)_2$ denote the bit decomposition of r . For all $i \in [\ell]$, based on the secret r_i and the shares $\{r_{i,j}\}_{j \in C}$ held by corrupted parties, $\mathcal{F}_{\text{decompose}}$ reconstructs the whole sharing $[r_i]_t$.
- (4) $\mathcal{F}_{\text{decompose}}$ asks the adversary whether it should continue.
 - If the adversary replies abort, $\mathcal{F}_{\text{decompose}}$ sends abort to honest parties.
 - If the adversary replies continue, $\mathcal{F}_{\text{decompose}}$ sends $x \oplus r$ and distributes the shares of $[r_\ell]_t, [r_{\ell-1}]_t, \dots, [r_1]_t$ to honest parties.

Protocol 4: DECOMPOSE

- (1) Let $[x]_t$ denote the input sharing, and ℓ denote the length of an element in $\mathbb{Z}_q$. All parties invoke $\mathcal{F}_{\text{bit}}$ to prepare ℓ random bit sharings:

$$[r_\ell]_t, [r_{\ell-1}]_t, \dots, [r_1]_t$$

- Then all parties locally compute $[r]_t := \sum_{i=1}^{\ell} 2^{i-1} \cdot [r_i]_t$.
- (2) All parties locally compute $[x \oplus r]_t := [x]_t \oplus [r]_t$. They send their shares of $[x \oplus r]_t$ to other parties to reconstruct $x \oplus r$. If a party P_j receives an inconsistent sharing, P_j aborts.
- (3) All parties take $x \oplus r$ and $[r_\ell]_t, [r_{\ell-1}]_t, \dots, [r_1]_t$ as output.

observation in [15] is that

$$\begin{aligned} c &= c_\ell \oplus b_\ell c_{\ell-1} \oplus \left(\prod_{k=2}^{\ell} b_k \right) c_1 \\ &= (c_\ell \oplus b_\ell c_{\ell-1}) \oplus b_\ell b_{\ell-1} (c_{\ell-2} \oplus b_{\ell-2} c_{\ell-3}) \\ &\quad \oplus \left(\prod_{k=3}^{\ell} b_k \right) (c_2 \oplus b_2 c_1) \end{aligned}$$

By setting $C_i = c_{2i} \oplus b_{2i} c_{2i-1}$ and $B_i = b_{2i} b_{2i-1}$ for all $i \in [\ell/2]$, we have

$$c = C_{\ell/2} \oplus B_{\ell/2} C_{\ell/2-1} \oplus \left(\prod_{k=2}^{\ell/2} B_k \right) C_1$$

This reduces the size of the circuit by half.

Thus all parties invoke $\mathcal{F}_{\text{mult}}$ to compute $[C_i]_t, [B_i]_t$ for all $i \in [\ell/2]$ and reduce the size of the circuit by half. This step requires ℓ multiplication operations. After repeating these steps $\log \ell$ times, all parties will obtain the sharing $[c]_t$. The total number of multiplications is bounded by 2ℓ . The description of **BitComp** appears in Protocol 5. The communication complexity of **BitComp** is $O(\ell^2 \cdot n)$ bits, where ℓ is the length of an element in $\mathbb{Z}_q$.

Protocol 5: BitComp

- (1) Let $[x]_t$ denote the input sharing and $([x_\ell]_t, \dots, [x_1]_t)$ denote the bit decomposition. Let $(r_\ell \dots r_1)_2$ be the binary representation of r . All parties locally compute the following sharings ($i \in \{1, \dots, \ell\}$):

$$[c_i]_t = [x_i]_t \oplus (1 - r_i) \cdot [b_i]_t = 1 - r_i \oplus [x_i]_t \oplus 2 \cdot r_i \cdot [x_i]_t$$
- (2) All parties set $\ell' = \ell$. While $\ell' > 1$, all parties execute the following steps:
 - (a) The goal is to compute $[c]_t = [c_{\ell'}]_t \oplus b_{\ell'} c_{\ell'-1}$

$$\prod_{i=2}^{\ell'} b_i \cdot c_1)_t.$$
 - (b) For all $i \in \{1, \dots, \lceil \ell'/2 \rceil\}$, all parties invoke $\mathcal{F}_{\text{mult}}$ to compute $[b_{2i} \cdot c_{2i-1}]_t$ and $[b_{2i} \cdot b_{2i-1}]_t$.
 - (c) For all $i \in \{1, \dots, \lceil \ell'/2 \rceil\}$, all parties locally compute $[c_i]_t := [c_{2i}]_t \oplus [b_{2i} \cdot c_{2i-1}]_t$ and $[b_i]_t := [b_{2i} \cdot b_{2i-1}]_t$.
 - If ℓ' is even, then set $\ell' := \ell'/2$.
 - If ℓ' is odd, then set $\ell' := (\ell' - 1)/2$ and $[c_{\ell'}]_t := [c_{2\ell'}]_t \oplus [b_{\ell'}]_t := [b_{2\ell'}]_t$.
- (3) All parties take $[c]_t := [c_1]_t$ as output.

5.2.2 Main Comparison Protocol. Now, we consider a general scenario: Given two sharings $[x]_t, [y]_t$ representing two fixed point values $\tilde{x}, \tilde{y} \in \mathbb{Q}_{\langle k, f \rangle}$, we want to securely compute the comparison result of $\tilde{x} < \tilde{y}$. Recall that a fixed point value in $\mathbb{Q}_{\langle k, f \rangle}$ has f bits in the decimal part and k bits in the integer part. Therefore, $2^k f^{-1} - 2 \leq 2^f (\tilde{x} - \tilde{y}) \leq 2^k f^{-1} - 2$.

Let $d = x - y \bmod q$. Recall that the field size of $\mathbb{Z}_q$ is $q = 2^{\ell} - 1$ where $\ell > k - 2f - 1$. Therefore, if $\tilde{x} \geq \tilde{y}$, we have $d \leq 2^k f^{-1} - 2 < q/2$. If $\tilde{x} < \tilde{y}$, we have $d \geq q - 2^k f^{-1} - 2 > q/2$. Thus, it is sufficient to compare d and $q/2$. We define comparison over $\mathbb{Z}_q$ as follows.

Definition 5.3 Less Than). For two elements $x, y \in \mathbb{Z}_q$, we say x is less than y , denoted by $x < y$, if $d = x - y \bmod q$ satisfies $d > q/2$.

As observed in [24], if $d < q/2$, then $2d < q$, which means that the least significant bit of $2d \bmod q$ is 0. If $d > q/2$, then $2d \equiv 2d - q \bmod q$. Since q is an odd prime, the least significant bit of $2d - q$ is 1. Thus, it is sufficient to compute the least significant bit of $2d \bmod q$. Let $z = 2d \bmod q$. In [24], to compute the least significant bit of z , all parties first prepare a random sharing $[r]_t$ with bit decomposition $([r_\ell]_t, \dots, [r_1]_t)$. Then all parties open the sharing $[c]_t = [z \oplus r]_t$. There are two cases.

- If $z \oplus r < q$, the least significant bit of z can be computed by $c_1 \oplus r_1$, where $\oplus$ denotes the XOR operation and c_1 is the least significant bit of c .

- If $z - r \geq q$, we have $c = z - r - q$. Therefore, the least significant bit of z can be computed by $1 - c_1 \oplus r_1$.

Note that, if $z - r < q$, we must have $c = z - r \geq r$. Otherwise, $c = z - r - q < r$. Thus, to compare $z - r$ and q , it is sufficient to compute the comparison result of $c < r$. Since c has been reconstructed and all parties hold $[r]_t$ and $([r_\ell]_t, \dots, [r_1]_t)$, we can use **BitComp** to compute the comparison result. The description of **LessThan** appears in Protocol 6. The communication complexity of **LessThan** is $O(n^2 \cdot \ell - n \cdot \ell^2)$ bits plus ℓ calls of $\mathcal{F}_{\text{bit}}$ (invoked in $\mathcal{F}_{\text{decompose}}$), where ℓ is the length of an element in $\mathbb{Z}_q$. Recall that in our main protocol, all random bit sharings will be prepared together.

Protocol 6: LessThan

- (1) Let $[x]_t, [y]_t$ denote the input sharings.
- (2) All parties locally compute $[z]_t = 2([x]_t - [y]_t)$. Then, all parties invoke $\mathcal{F}_{\text{decompose}}$ on $[z]_t$ and receive $c := z - r$ and $[r_\ell]_t, [r_{\ell-1}]_t, \dots, [r_1]_t$.
- (3) All parties locally compute $[r]_t := \sum_{i=1}^{\ell} 2^{i-1} \cdot [r_i]_t$. Then, all parties invoke **BitComp** on $[r]_t, ([r_\ell]_t, \dots, [r_1]_t)$ and c . Let $[w]_t$ denote the output.
- (4) Let c_1 denote the least significant bit of c . All parties invoke $\mathcal{F}_{\text{mult}}$ enough times to compute

$$[b]_t = [w(1 - c_1 \oplus r_1) - (1 - w)(c_1 \oplus r_1)]_t,$$

where $\oplus$ denotes the XOR operation. Here $[c_1 \oplus r_1]_t$ can be computed by

$$[c_1 \oplus r_1]_t := c_1 - [r_1]_t - 2c_1[r_1]_t$$

All parties take $[b]_t$ as output.

5.2.3 Computing Truncation. Let x and y be the representation of the fixed point values $\tilde{x}$ and $\tilde{y}$ respectively. It's clear that $x \cdot y$ does not correspond to a valid floating point number in $\mathbb{Q}_{(k,f)}$ since the multiplication result of $\tilde{x}$ and $\tilde{y}$ has $2f$ bits in the decimal part. Therefore, We want to compute $z = \lfloor x \cdot y / 2^f \rfloor$. We follow the idea in [5], which allows 1 bit error in the least significant bit of z . Concretely, we will compute $z = \lfloor x \cdot y / 2^f \rfloor - b$, where $b \in \{0, 1\}$ denotes the bit error in the least significant bit of the final result. The probability that $b = 1$ is $(2^{2f} \cdot \tilde{x} \cdot \tilde{y} \bmod 2^f) / 2^f$. We will use $\tilde{z}$ to approximate the multiplication result of $\tilde{x} \cdot \tilde{y}$. We define truncation over $\mathbb{Z}_q$ as follows.

Definition 5.4 Truncation. Let f be an integer. For an element $x \in \mathbb{Z}_q$, the result of truncating f bits on x is defined by the following random algorithm:

- (1) Set $y := x \cdot 2^{\ell-1} \bmod q$, where ℓ satisfies that $q = 2^\ell - 1$.
- (2) Let $y' = \lfloor y / 2^f \rfloor$ and $y'' = y \bmod 2^f$. Randomly sample $b \in \{0, 1\}$ such that $\Pr[b = 1] = y'' / 2^f$.
- (3) Output $x' = y' - b \cdot 2^{\ell-f-1}$.

We first show that if x represents a fixed point value $\tilde{x}$ which has k bits in the integer part and $2f$ bits in the decimal part, x' defined in Definition 5.4 is the result we want. In this case, $2^{2f} \cdot \tilde{x} \in (2^k - 2^f, 2^k + 2^f)$ and $x = 2^{2f} \cdot \tilde{x} \bmod q$. Note that the element that

represents the fixed point value $\tilde{x}$ after truncating the last f bits is $\lfloor \frac{2^{2f} \cdot \tilde{x}}{2} \rfloor \bmod q$.

We first compute $y = 2^{2f} \cdot \tilde{x} \cdot 2^{\ell-1}$. Since $\ell > k - 2f - 1$, we have $0 < y < q$ and therefore $y = 2^{2f} \cdot \tilde{x} \cdot 2^{\ell-1} = x \cdot 2^{\ell-1} \bmod q$. Let $y' = \lfloor y / 2^f \rfloor$ denote the result of truncating f bits on y . Then we have

$$y' \cdot 2^{\ell-f-1} = \lfloor \frac{y}{2^f} \rfloor \cdot 2^{\ell-f-1} = \lfloor \frac{y \cdot 2^{\ell-1}}{2^f} \rfloor = \lfloor \frac{2^{2f} \cdot \tilde{x}}{2} \rfloor$$

Therefore, the element $(y' \cdot 2^{\ell-f-1} \bmod q)$ represents the fixed point value $\tilde{x}$ after truncating the last f bits. Since $y = 2^{2f} \cdot \tilde{x} \cdot 2^{\ell-1}$, we have $(2^{2f} \cdot \tilde{x}) \bmod 2^f = y \bmod 2^f$. Therefore, the bit error $b \in \{0, 1\}$ satisfies that $\Pr[b = 1] = (y \bmod 2^f) / 2^f = (2^{2f} \cdot \tilde{x} \bmod 2^f) / 2^f$. Thus, $x' = y' - b \cdot 2^{\ell-f-1}$ is the result we want.

Now, we focus on computing the truncation result of y with the bit error. All parties first prepare a random sharing $[r]_t$ with bit decomposition $([r_\ell]_t, \dots, [r_1]_t)$. The first step is to compute and open the sharing $[c]_t := \lfloor y / 2^f \rfloor$. Let $y' = \lfloor y / 2^f \rfloor$, $y'' = y \bmod 2^f$, and $r' = \lfloor r / 2^f \rfloor$, $r'' = r \bmod 2^f$. There are two cases:

- If $y - r < q$, then $c = y - r$. Therefore $\lfloor c / 2^f \rfloor = \lfloor (y - r) / 2^f \rfloor = y' - r' - \lfloor (y'' - r'') / 2^f \rfloor$. We have

$$y' - \lfloor (y'' - r'') / 2^f \rfloor = \lfloor c / 2^f \rfloor - r' \\ = \lfloor c / 2^f \rfloor - \sum_{i=f+1}^{\ell} 2^{i-f-1} r_i$$

- If $y - r \geq q$, then $c = y - r - q$. Similarly, we have

$$y' - \lfloor (y'' - r'') / 2^f \rfloor = \lfloor (c + q) / 2^f \rfloor - r' \\ = \lfloor (c + q) / 2^f \rfloor - \sum_{i=f+1}^{\ell} 2^{i-f-1} r_i$$

To learn whether $y - r < q$ or not, it is sufficient to compute the comparison result of $c < r$. This is because if $y - r < q$, we must have $c = y - r \geq r$; and otherwise, we have $c = y - r - q < r$. Now, we analyze the bit error $\lfloor (y'' - r'') / 2^f \rfloor$. Note that the distribution of r'' is statistically close to the uniform distribution in $\{0, 1, \dots, 2^f - 1\}$. Therefore

$$\Pr[\lfloor (y'' - r'') / 2^f \rfloor = 1] = \Pr[r'' \geq 2^f - y''] \approx y'' / 2^f$$

Thus, the distribution of $\lfloor (y'' - r'') / 2^f \rfloor$ is statistically close to the bit error $b \in \{0, 1\}$ generated according to Definition 5.4. The description of **TRUNC** appears in Protocol 7. The communication complexity of **TRUNC** is $O(n^2 \cdot \ell - n \cdot \ell^2)$ bits plus ℓ calls of $\mathcal{F}_{\text{bit}}$ (invoked in $\mathcal{F}_{\text{decompose}}$), where ℓ is the length of an element in $\mathbb{Z}_q$. Recall that in our main protocol, all random bit sharings will be prepared together.

5.3 Supporting Inner-product Operations with Efficient Verification

Our main protocol will make use of the following two functionalities:

- The functionality $\mathcal{F}_{\text{inner}}$ allows all parties to evaluate an inner-product gate. It takes two vector of degree- t Shamir sharings $[x]_t = ([x_1]_t, [x_2]_t, \dots, [x_k]_t)$ and $[y]_t = ([y_1]_t, [y_2]_t, \dots, [y_k]_t)$ and outputs $[z]_t = [\odot y]_t$, where $\odot$ is the

Protocol 7: TRUNC

- (1) Let $[x]_t$ denote the sharing to be truncated, f be the number of bits to truncate.
- (2) All parties locally compute $[y]_t = [x]_t \cdot 2^{\ell-1}$. Then, all parties invoke $\mathcal{F}_{\text{decompose}}$ on $[y]_t$ and receive $c := y \cdot r$ and $[r_\ell]_t, [r_{\ell-1}]_t, \dots, [r_1]_t$.
- (3) All parties locally compute

$$[x']_t = \lfloor \frac{c}{2^f} \rfloor \sum_{i=f}^{\ell} 2^{i-f-1} \cdot [r_i]_t \cdot 2^{\ell-f-1}$$

$$[x'']_t = \lfloor \frac{c}{2^f} q \rfloor \sum_{i=f}^{\ell} 2^{i-f-1} \cdot [r_i]_t \cdot 2^{\ell-f-1}$$

- (4) All parties locally compute $[r]_t := \sum_{i=1}^{\ell} 2^{i-1} \cdot [r_i]_t$. Then all parties invoke BitComp on $[r]_t, ([r_\ell]_t, \dots, [r_1]_t)$ and c . Let $[w]_t$ denote the output.
- (5) All parties invoke $\mathcal{F}_{\text{mult}}$ enough times to compute

$$[z]_t = [x' \cdot (1 - w) - x'' \cdot w]_t$$

All parties take $[z]_t$ as output.

inner-product operation. An instantiation of $\mathcal{F}_{\text{inner}}$ can be found in [18] (See Remark 1 in Section 4.2 of [18]). The communication complexity of the instantiation is $O(n)$ elements.

- The functionality $\mathcal{F}_{\text{innerVerify}}$ allows all parties to verify a batch of inner-product tuples (which may have various dimensions) computed by $\mathcal{F}_{\text{inner}}$. In Appendix C, we discuss how to extend the instantiation of $\mathcal{F}_{\text{multVerify}}$ in [20] to support the verification of inner-product tuples. The communication complexity is $O(n^2 \cdot \log \ell \cdot \kappa)$ bits, where ℓ is the sum of the dimensions of all inner-product tuples.

Note that ℓ is the number of multiplications we need to do if each inner-product operation is computed by a batch of multiplication operations. And using $\mathcal{F}_{\text{multVerify}}$ to verify ℓ multiplications has the same communication overhead $O(n^2 \cdot \log \ell \cdot \kappa)$. Therefore, using $\mathcal{F}_{\text{inner}}$ to evaluate inner-product operations and $\mathcal{F}_{\text{innerVerify}}$ to verify inner-product tuples is strictly better than using $\mathcal{F}_{\text{mult}}$ and $\mathcal{F}_{\text{multVerify}}$. In Appendix C, we present the full descriptions of these two functionalities and discuss how to realize $\mathcal{F}_{\text{innerVerify}}$ using the techniques from [20].

Note that a multiplication operation is a special case of an inner-product operation when the dimension of the input vectors is 1. Therefore, we will use $\mathcal{F}_{\text{inner}}$ to also evaluate multiplication gates. Furthermore, since a multiplication tuple is a special case of an inner-product tuple when the dimension of the input vectors is 1, we will also use $\mathcal{F}_{\text{innerVerify}}$ to check the correctness of the multiplication tuples together with the inner-product tuples.

5.4 ACCO: MPC for Arithmetic Circuits with Comparison

In this part, we are ready to introduce our MPC protocol for an arithmetic circuit with comparison and truncation gates. We focus on

the Mersenne field $\mathbb{Z}_q$. The comparison and truncation operations over $\mathbb{Z}_q$ are defined in Definition 5.3 and Definition 5.4. Recall that we are in the client-server model. In particular, all the inputs belong to the clients, and only the clients receive the outputs. Without loss of generality, we assume that the circuit is consist of addition gates, inner-product gates, comparison gates, and truncation gates (since a multiplication gate can be seen as a special case of an inner-product gate when the inputs are vectors of dimension 1). The functionality $\mathcal{F}_{\text{ACCO}}$ appears in Functionality 8.

Functionality 8: $\mathcal{F}_{\text{ACCO}}$

- (1) $\mathcal{F}_{\text{ACCO}}$ receives from all clients their inputs, which are elements in $\mathbb{Z}_q$.
- (2) $\mathcal{F}_{\text{ACCO}}$ evaluates the circuit as follows:
 - For an addition gate with inputs x, y , $\mathcal{F}_{\text{ACCO}}$ computes $z := x + y$ as output.
 - For an inner-product gate with inputs x, y , $\mathcal{F}_{\text{ACCO}}$ computes $z := x \odot y$ as output, where $\odot$ is the inner-product operation.
 - For a comparison gate with inputs x, y , $\mathcal{F}_{\text{ACCO}}$ checks whether $x < y$ based on Definition 5.3. If not, $\mathcal{F}_{\text{ACCO}}$ takes $b := 1$ as output. Otherwise, $\mathcal{F}_{\text{ACCO}}$ takes $b := 0$ as output.
 - For a truncation gate with inputs $x \in \mathbb{Z}_q, f \in \mathbb{N}^*$, $\mathcal{F}_{\text{ACCO}}$ computes the result of truncating f bits on x based on Definition 5.4, denoted by x' . $\mathcal{F}_{\text{ACCO}}$ takes x' as output.
- (3) $\mathcal{F}_{\text{ACCO}}$ first sends the output of corrupted clients to the adversary.
 - If the adversary replies continue, $\mathcal{F}_{\text{ACCO}}$ distributes the output to honest clients.
 - If the adversary replies abort, $\mathcal{F}_{\text{ACCO}}$ sends abort to honest clients.

Our protocol for $\mathcal{F}_{\text{ACCO}}$ is based on the MPC protocol in [18]. In addition to addition gates and inner-product gates (recall that a multiplication gate can be seen a special case of an inner-product gate), we use LESS THAN for comparison gates and TRUNC for truncation gates. Note that in LESS THAN and TRUNC , they only contain local computations except invocations of $\mathcal{F}_{\text{decompose}}$ and $\mathcal{F}_{\text{inner}}$ (recall that we have replaced $\mathcal{F}_{\text{mult}}$ by $\mathcal{F}_{\text{inner}}$). For $\mathcal{F}_{\text{decompose}}$, an adversary cannot affect the correctness of the output. Therefore, it is sufficient to check the correctness of the inner-product tuples computed by $\mathcal{F}_{\text{inner}}$. Thus, the verification of the whole protocol is the check of all inner-product tuples computed by $\mathcal{F}_{\text{inner}}$ throughout the protocol. The description of ACCO appears in Protocol 9.

THEOREM 5.5. *Let c be the number of clients and $n = 2t - 1$ be the number of parties. The protocol ACCO securely computes $\mathcal{F}_{\text{ACCO}}$ with abort in the $\{\mathcal{F}_{\text{inner}}, \mathcal{F}_{\text{decompose}}, \mathcal{F}_{\text{innerVerify}}\}$ -hybrid model in the presence of a fully malicious adversary controlling up to c clients and t parties.*

We refer the readers to Appendix D.3 for the proof of Theorem 5.5.

Protocol 9: ACCO**(1) Input Phase:**

For *each* client with input x , client randomly samples a degree- t sharing $[x]_t$ and distributes the shares to all parties.

(2) Computation Phase:

All parties start with holding a degree- t sharing for each input gate. The circuit is evaluated in a predetermined topological order.

- For each addition gate with input sharings $[x]_t, [y]_t$, all parties locally compute $[z]_t = [x]_t + [y]_t$.
- For each inner-product gate with input sharings $[x]_t, [y]_t$, all parties invoke $\mathcal{F}_{\text{inner}}$ on $[x]_t, [y]_t$ to compute $[z]_t$ where $z = x \cdot y$.
- For each comparison gate with input sharings $[x]_t, [y]_t$, all parties invoke LESS_THAN on $[x]_t, [y]_t$ to compute the comparison result of $x < y$.
- For each truncation gate with inputs $[x]_t, f \in \mathbb{N}^*$, all parties invoke TRUNC on $[x]_t, f$ to compute the result of truncating f bits of x .

(3) Verification Phase:

All parties invoke $\mathcal{F}_{\text{innerVerify}}$ to check the correctness of the inner-products.

(4) Output Phase:

For each output gate, let $[x]_t$ be the sharing associated with this gate and client be the client who should receive this output. All parties send their shares of $[x]_t$ to client. Client checks whether the shares of $[x]_t$ is consistent. If not, client aborts. Otherwise, client reconstructs the result x .

Analysis of the Communication Complexity of ACCO. Let c, c_{ip}, c_L, c_T, c_O denote the number of input gates, inner-product gates, comparison gates, truncation gates, and output gates. Let $|C|$ denote the circuit size. Recall that ℓ is the length of an element in $\mathbb{Z}_q$.

We first count the number of calls of $\mathcal{F}_{\text{bit}}$. Recall that we need ℓ calls of $\mathcal{F}_{\text{bit}}$ in LESS_THAN and TRUNC . Therefore, we need $(c_L + c_T) \cdot \ell$ calls of $\mathcal{F}_{\text{bit}}$. When we use our protocol BRTs to instantiate $\mathcal{F}_{\text{bit}}$, the communication complexity is $O((c_L + c_T) \cdot n \cdot \ell^2 + n^2 \cdot \log((c_L + c_T) \cdot \ell)) = O((c_L + c_T) \cdot n \cdot \ell^2 + n^2 \cdot \log |C| \cdot \ell)$ bits.

Now we compute the sum of dimensions of inner-product tuples we need to verify in the verification phase. For the inner-product gates, the sum of dimensions are bounded by the circuit size $|C|$. For a comparison gate or a truncation gate, we need to invoke $O(\ell)$ times of $\mathcal{F}_{\text{inner}}$ and each invocation is used to compute a multiplication. Therefore, the sum of dimensions of inner-product tuples we need to verify is $O(|C| + (c_L + c_T) \cdot \ell)$, which requires $O(n^2 \cdot \log(|C| + (c_L + c_T) \cdot \ell) \cdot \ell) = O(n^2 \cdot \log |C| \cdot \ell)$ bits.

In summary, the communication complexity of ACCO is

$$O((c + c_O + c_{ip}) \cdot n \cdot \ell + (c_L + c_T) \cdot (n^2 \cdot \ell + n \cdot \ell^2) + n^2 \cdot \log |C| \cdot \ell)$$

6 EXPERIMENTAL EVALUATION

We implement our system ACCO, and apply it to two machine learning tasks. We choose the same regression training tasks as used in [31], and use their reported performance as a baseline for comparison. Recall that our protocol is information theoretically secure. The security requires the local random tape of each party to be independent and uniformly distributed. However, we note that local random tapes can always be prepared before the computation. To better expose the running time of our protocol, we choose to use pseudo-random generators (PRG) to prepare the local random tape of each party in the implementation. We derive and implement the Alternating Direction Method of Multipliers (ADMM) algorithm for LASSO regression similarly to [31]. Details of the derivation and our implementations can be found in Appendix E. We also benchmark the LESS_THAN and the TRUNC protocols individually to show their concrete efficiency.

6.1 Experiment Setup

First, we use the same setup as [31] for comparison. We run 5 parties each on an independent r4i.xlarge instance (32 cores with 2.3GHz and 244GB RAM) on Amazon AWS. The instances are located in two different regions (i.e. a WAN configuration): 3 in N. Virginia, and 2 in Oregon.

Next, we extend our experiment to include more parties. We run 5, 9, 13, 17 parties using the same configuration as before, but place all instances in the same region (i.e. a LAN configuration). Throughout our experiments, we use the 127-bit Mersenne field. For each training experiment, we report L2 errors and Mean Absolute Errors (MAE) to match the results in [31]. Later, we also use those same AWS machines in the LAN configuration to benchmark the LESS_THAN and TRUNC protocols individually.

Our training benchmark consists of two real world datasets (the same as [31]) from UCI [12]: the gas sensor dataset, and the million song dataset. The gas sensor dataset contains recordings of 16 chemical sensors when exposed to two gas mixtures at different concentrations. The dataset contains two sets of gas mixtures: Ethylene and Methane in air, and Ethylene and CO in air. To match the experiment in [31], we use the Ethylene and CO mixture, and we only run LASSO regression to predict the concentration of Ethylene. We split the dataset to have 4/5 training samples and 1/5 test samples. Test samples are uniformly drawn from the dataset. For n participants, the circuit for training this dataset consists of $16(n-1)$ truncations, 32 comparisons, and $16(n-2)$ multiplication / inner-product gates (besides those included by computing truncations or comparisons). Details on how the circuits are derived can be found in Appendix E.2.

The million song dataset is a subset of [3], prepared specifically for predicting the release year of a song from its audio features. Each sample contains 90 features. The dataset contains 463715 training samples, and 51630 test samples. We preprocess the dataset simply by centering each variable around its mean from the training set. Since the release year of a song is always an integer, we round regression results to their nearest integers as prediction results. For n participants, the circuit for training this dataset consists of $90(n$

1) truncations, 180 comparisons, and $90(n-2)$ multiplication / inner-product gates (besides those included by computing truncations or comparisons).

6.2 Benchmark Results

To justify our choice of using a 127-bit field, we first compare training errors of ACCO (using fixed point values) against running Algorithm 16 directly on plain text data (using real values). Table 1 shows that running ACCO with restricted precision (i.e. 127-bit field) differs from training over plain text by less than 0.01 for the gas sensor dataset. A similar comparison is also run on the million song dataset, where no difference is shown when rounded to 2 decimal places, and its table is therefore omitted.

Training Error. We next compare training errors of ACCO against the standard training algorithm in a popular python package sklearn. We compare the L2 error, which is the average of squared error for each prediction, and the MAE error, which is the average of absolute error for each prediction. Table 2 shows the comparison for the gas sensor dataset and the million song dataset respectively. As our results match those of sklearn closely, we confirm the effectiveness of ACCO. To provide a baseline for comparison, we recreate the corresponding tables (merged into Table 3) from [31] that shows similar results for their proposed system, Helen. To save space, we select only several representative columns from each original table. Note that while Helen works in the dishonest majority setting, ACCO works in the honest majority setting. We test with 5 parties (instead of 4 in Helen) to represent a natural scenario where the honest majority is around half of the participants.²

Running Time and Communication Size. In Table 4, we show the total running time (including both communication time and computation time) and the concrete communication size of ACCO for 5, 9, 13, and 17 parties in the LAN setting to demonstrate its efficiency. We get almost identical errors as in the WAN cases, so we omit them to save space.

In Table 5, we show a comparison of total running time (including both communication time and computation time) to Helen in the WAN setting, because it's the only system that achieves also malicious security (albeit in the dishonest majority setting). When the problem size is large (i.e. each sample having large number of features), we achieve an $\sim 73\times$ speedup.

Each of our experiments consists of two phases. In the offline phase, every party locally generates i, b_i from its private dataset as its input to Algorithm 16. In all cases, the generation processes take less than 0.5 second, and is insignificant compared to the total running time. In the online phase, parties run our implementation of ACCO. Since i, b_i only scale with the number of features in each sample, and not with the number of samples used for training, the running times of our experiments for the same dataset stays the same.

Comparison and Truncation. In Table 6, we show the total running time (including computation and communication times) of

computing only comparison and truncation in two scenarios. We use 5 parties in the LAN configuration for this benchmark.

First, we compute 10,000 comparison/ truncation (using LESS THAN and TRUNC) in sequence (i.e. only start computing the next after the previous is finished). This corresponds to the extreme case where each circuit layer contains only a single comparison / truncation. Note that although we can't start computing the comparison/ truncation before the previous one is finished, we *can* prepare all the random bit sharings that are needed to compute the whole circuit. We therefore break up the running time into first preparing random bit sharings (in the preparation phase), and next computing comparison/ truncation in sequence (in the online phase). The results are shown in the first column of Table 6.

Next, we compute 10,000 comparison/ truncation in parallel. This corresponds to the extreme case where all of the comparison / truncation are in the same circuit layer. We similarly break up the running time into the preparation phase and the online phase. The results are shown in the second column of Table 6. Since the underlying multiplication protocol (for $\mathcal{F}_{\text{inner}}$ and $\mathcal{F}_{\text{mult}}$) is capable of computing all multiplications in a single layer in parallel, we can observe that our LESS THAN and TRUNC protocols benefit greatly from parallelly computing comparisons and truncation in the same circuit layer, whenever possible.

Note that since the LESS THAN and the TRUNC protocols require the same number of random bit sharings, their preparation phase are identical. Hence the running time of their preparation phases are the same, and is shown in the first row of Table 6.

ACKNOWLEDGMENTS

V. Goyal, Y. Song—Supported in part by the NSF award 1916939, DARPA SIEVE program, a gift from Ripple, a DoE NETL award, a JP Morgan Faculty Fellowship, a PNC center for financial services innovation award, and a Cylab seed funding award.

X. Duan, H. Li—Work done in part while at CMU and supported by grants of Vipul Goyal mentioned above.

R. Ostrovsky—Supported in part by DARPA under Cooperative Agreement HR0011-20-2-0025, NSF grant CNS-2001096, US-Israel BSF grant 2015782, Google Faculty Award, JP Morgan Faculty Award, IBM Faculty Research Award, Xerox Faculty Research Award, OKAWA Foundation Research Award, B. John Garrick Foundation Award, Teradata Research Award, Lockheed-Martin Research Award and Sunday Group. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, the Department of Defense, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes not withstanding any copyright annotation therein.

A. Polychroniadou—This paper was prepared in part for information purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment

²Note the differences between Table 2 and Table 3 even in sklearn training results. This is because some hyper-parameters used for training are not provided in [31], and are hard to match exactly.

Samples / party	2k	10k	40k	100k	400k
ACCO L2	8491.65	8361.08	8371.01	8369.30	8368.38
plain text L2	8491.66	8361.08	8371.02	8369.32	8368.37
ACCO MAE	57.76	57.69	58.05	58.02	57.96
plain text MAE	57.76	57.69	58.05	58.02	57.95

Table 1: This table shows the L2 and MAE errors when 5 parties train on the gas sensor dataset. The columns show the cases when the parties are given different number of random samples such as 2000, 10000 etc. The rows compare ACCO and running Algorithm 16 on plain text data. This table shows that running ACCO with restricted precision has minimal effect.

dataset	gas sensor					million song					
Samples / party	2k	10k	40k	100k	400k	2k	10k	20k	40k	60k	80k
ACCO L2	8491.65	8361.08	8371.01	8369.30	8368.38	91.95	90.97	90.72	90.76	90.76	90.73
sklearn L2	8431.92	8260.47	8292.85	8296.13	8294.01	91.76	90.95	90.75	90.74	90.73	90.69
ACCO MAE	57.76	57.69	58.05	58.02	57.96	6.88	6.86	6.85	6.86	6.88	6.88
sklearn MAE	56.78	56.58	56.74	56.70	56.66	6.90	6.88	6.87	6.88	6.88	6.88

Table 2: This table shows the L2 and MAE errors when 5 parties train on the gas sensor dataset, and the million song dataset. The columns show the cases when the parties are given different number of random samples. The rows compare ACCO and the standard sklearn package on plain text data.

dataset	gas sensor					million song					
Samples / party	2k	10k	40k	100k	400k	2k	10k	20k	40k	60k	80k
Helen L2	8841.33	8837.59	8844.31	8876.00	8907.38	91.8	90.73	90.67	90.57	90.54	90.57
sklearn L2	8937.01	8929.10	8974.15	8981.24	8982.88	91.67	90.72	90.63	90.57	90.55	90.56
Helen MAE	57.23	57.47	57.63	58.25	58.36	6.82	6.81	6.80	6.79	6.80	6.80
sklearn MAE	57.89	58.05	58.34	58.48	58.58	6.81	6.81	6.80	6.79	6.79	6.80

Table 3: This is Table II and III (several columns omitted to save space) from [31], as a comparison baseline for Table 2. The table shows errors of Helen for the gas sensor dataset, and the million song dataset.

dataset	5 parties	9 parties	13 parties	17 parties
gas (time)	3 0s	6 3s	10 0s	14 4s
gas (size)	100MB	273MB	528MB	866MB
song (time)	8 0s	16 5s	28 6s	42 5s
song (size)	0.55GB	1.50GB	2.90GB	4.76GB

Table 4: This table shows the total running time and communication size of ACCO on two datasets and increasing the number of parties in the LAN setting.

dataset	gas (16 features)	song (90 features)
ACCO	65s	71s
Helen	~ 20m	~ 1 5h
Speedup	~ 19x	~ 73x

Table 5: This table shows the total running time including both offline and online phases, and including both communication time and computation time) comparison between ACCO and Helen [31] in the WAN setting.

	in sequence	in parallel
preparation phase	5.5s	5.5s
COMPARISON, online phase	47.2s	1.2s
TRUNCATION, online phase	42.5s	2.0s

Table 6: This table shows the total running time of computing 10,000 comparisons and truncations with 5 parties in the LAN configuration. The first column shows computing them in sequence, and the second shows computing them in parallel. The preparation time for both protocols are the same, and is shown in the first row.

financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful. 2020 JPMorgan Chase & Co. All rights reserved.

REFERENCES

- [1] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 308–318. <https://doi.org/10.1145/2976749.2978318>

research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument,

- [2] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, May 2-4, 1988, Chicago, Illinois, USA. 1–10. <https://doi.org/10.1145/62212.62213>
- [3] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. 2011. The Million Song Dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*.
- [4] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. 2011. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Found. Trends Mach. Learn.* 3, 1 (Jan. 2011), 1–122. <https://doi.org/10.1561/22000000016>
- [5] Octavian Catrina and Sebastiaan De Hoogh. 2010. Improved Primitives for Secure Multiparty Integer Computation. In *Proceedings of the 7th International Conference on Security and Cryptography for Networks (SCN 2010)*. Springer-Verlag, Berlin, Heidelberg, 182–199.
- [6] T-H Hubert Chan, Elaine Shi, and Dawn Song. 2012. Privacy-preserving stream aggregation with fault tolerance. In *International Conference on Financial Cryptography and Data Security*. Springer, 200–214.
- [7] David Chaum, Claude Crépeau, and Ivan Damgård. 1987. Multiparty Unconditionally Secure Protocols (Abstract). In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques*, Santa Barbara, California, USA, August 16-20, 1987, *Proceedings*. 462. https://doi.org/10.1007/3-540-48184-2_43
- [8] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *Annual International Cryptology Conference*. Springer, 34–64.
- [9] Ivan Damgård and Jesper Buus Nielsen. 2007. Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference*. Springer, 572–590.
- [10] Ivan Damgård, Matthias Fitz, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. 2006. Unconditionally Secure Constant-Rounds Multi-Party Computation for Equality, Comparison, Bits and Exponentiation. In *Proceedings of the Third Conference on Theory of Cryptography (TCC 2006)*. Springer-Verlag, Berlin, Heidelberg, 285–304. https://doi.org/10.1007/11681878_15
- [11] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*.
- [12] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. (2017). <http://archive.ics.uci.edu/ml>
- [13] Cynthia Dwork, Krishnamurthy Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our Data, Ourselves: Privacy via Distributed Noise Generation. In *Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques (EUROCRYPT 2006)*. Springer-Verlag, Berlin, Heidelberg, 486–503. https://doi.org/10.1007/11761679_29
- [14] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. *Theory of Cryptography* Vol. 3876, 265–284. https://doi.org/10.1007/11681878_14
- [15] Juan Garay, Berry Schoenmakers, and José Villegas. 2007. Practical and Secure Solutions for Integer Comparison. In *Public Key Cryptography - PKC 2007*, Tatsuki Okamoto and Xiaoyun Wang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 330–342.
- [16] Daniel Genkin, Yuval Ishai, Manoj M. Prabhakaran, Amit Sahai, and Eran Tromer. 2014. Circuits Resilient to Additive Attacks with Applications to Secure Computation. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing (STOC '14)*. ACM, New York, NY, USA, 495–504. <https://doi.org/10.1145/2591796.2591861>
- [17] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or a Completeness Theorem for Protocols with Honest Majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, 1987, New York, New York, USA. 218–229. <https://doi.org/10.1145/28395.28420>
- [18] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychoriadou, and Yifan Song. 2021. ATLAS: Efficient and Scalable MPC in the Honest Majority Setting. In *Advances in Cryptology - CRYPTO 2021*, Tal Malkin and Chris Peikert (Eds.). Springer International Publishing, Cham, 244–274.
- [19] Vipul Goyal and Yifan Song. 2020. Malicious Security Comes Free in Honest-Majority MPC. *Cryptology ePrint Archive*, Report 2020/134. (2020). <https://eprint.iacr.org/2020/134>.
- [20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. 2020. Guaranteed Output Delivery Comes Free in Honest Majority MPC. In *Advances in Cryptology - CRYPTO 2020*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer International Publishing, Cham, 618–646.
- [21] R. Iyengar, J. P. Near, D. Song, O. Thakkar, A. Thakurta, and L. Wang. 2019. Towards Practical Differentially Private Convex Optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. 299–316.
- [22] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*. Association for Computing Machinery, New York, NY, USA, 35–52. <https://doi.org/10.1145/3243734.3243760>
- [23] P. Mohassel and Y. Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *2017 IEEE Symposium on Security and Privacy (SP)*. 19–38.
- [24] Takashi Nishide and Kazuo Ohta. 2007. Multiparty Computation for Interval, Equality, and Comparison without Bit-Decomposition Protocol. In *Proceedings of the 10th International Conference on Practice and Theory in Public-Key Cryptography (PKC 2007)*. Springer-Verlag, Berlin, Heidelberg, 343–360.
- [25] Arpita Patra and Ajith Suresh. 2020. BLAZE: Blazing Fast Privacy-Preserving Machine Learning. *CoRR abs/2005.09042* (2020). [arXiv:2005.09042](https://arxiv.org/abs/2005.09042) <https://arxiv.org/abs/2005.09042>
- [26] M. Sadegh Riaz, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *Proceedings of the 2018 Asia Conference on Computer and Communications Security (ASIACCS 2018)*. Association for Computing Machinery, New York, NY, USA, 707–721. <https://doi.org/10.1145/3196494.3196522>
- [27] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (Nov. 1979), 612–613. <https://doi.org/10.1145/359168.359176>
- [28] Elaine Shi, T-H. Hubert Chan, Eleanor Gilbert Rieffel, Richard Chow, and Dawn Song. 2011. Privacy-Preserving Aggregation of Time-Series Data. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society. <https://www.ndss-symposium.org/ndss2011/privacy-preserving-aggregation-of-time-series-data>
- [29] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: Efficient and Private Neural Network Training. In *Privacy Enhancing Technologies Symposium (PETS 2019)*. <https://www.microsoft.com/en-us/research/publication/securenn-efficient-and-private-neural-network-training/>
- [30] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*. 162–167. <https://doi.org/10.1109/SFCS.1986.25>
- [31] Wenting Zheng, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2019. Helen: Maliciously secure cooperative learning for linear models. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 724–738.

A SECURITY DEFINITION

Let c denote the number of clients and $n = 2t - 1$ denote the number of servers. Let $\mathcal{F}$ be a secure function evaluation functionality. An adversary $\mathcal{A}$ can corrupt at most c clients and t servers, provide inputs to corrupted clients, and receive all messages sent to corrupted clients and servers. Corrupted clients and servers can deviate from the protocol arbitrarily.

Real World Execution. In the real world, the adversary $\mathcal{A}$ controlling corrupted clients and servers interacts with honest clients and servers. At the end of the protocol, the output of the real world execution includes the inputs and outputs of honest clients and servers, and the view of the adversary.

Ideal World Execution. In the ideal world, a simulator Sim simulates honest clients and servers and interacts with the adversary $\mathcal{A}$. Furthermore, Sim has a one-time access to $\mathcal{F}$, which includes providing inputs of corrupted clients and servers to $\mathcal{F}$, receiving the outputs of corrupted clients and servers, and sending instructions specified in $\mathcal{F}$ (e.g., asking $\mathcal{F}$ to abort). The output of the ideal world execution includes the inputs and outputs of honest clients and servers, and the view of the adversary.

We say that a protocol π securely computes $\mathcal{F}$ if there exists a simulator Sim , such that for all adversary $\mathcal{A}$, the distribution of the output of the real world execution is *statistically close* to the distribution of the output of the ideal world execution. If π allows a premature abort, then we say π securely computes $\mathcal{F}$ with abort. We refer the readers to [16] for a formal definition.

B BUILDING BLOCKS FROM [18, 20]

Multiplicating Two degree- t Shamir Sharings. The functionality $\mathcal{F}_{\text{mult}}$ allows all parties to evaluate a multiplication gate. It takes two input sharings $[x]_t, [y]_t$ and outputs $[z]_t := [x \cdot y]_t$. The description of $\mathcal{F}_{\text{mult}}$ appears in Functionality 10. An instantiation of $\mathcal{F}_{\text{mult}}$ can be found in [18].

Functionality 10: $\mathcal{F}_{\text{mult}}$

- (1) Let $[x]_t, [y]_t$ denote the input sharings. $\mathcal{F}_{\text{mult}}$ receives from honest parties their shares of $[x]_t, [y]_t$. Then $\mathcal{F}_{\text{mult}}$ reconstructs the secrets x, y . $\mathcal{F}_{\text{mult}}$ further computes the shares of $[x]_t, [y]_t$ held by corrupted parties, and sends these shares to the adversary.
- (2) $\mathcal{F}_{\text{mult}}$ receives from the adversary a value d and a set of shares $\{z_i\}_{i \in C}$.
- (3) $\mathcal{F}_{\text{mult}}$ computes $x \cdot y \cdot d$. Based on the secret $z := x \cdot y \cdot d$ and the t shares $\{z_i\}_{i \in C}$, $\mathcal{F}_{\text{mult}}$ reconstructs the whole sharing $[z]_t$ and distributes the shares of $[z]_t$ to honest parties.

Verifying Multiplication Results. Since $\mathcal{F}_{\text{mult}}$ does not guarantee the correctness of the multiplications, all parties need to verify the multiplications computed by $\mathcal{F}_{\text{mult}}$ at the end of the protocol. The functionality $\mathcal{F}_{\text{multVerify}}$ takes N multiplication tuples as input and outputs to all parties a single bit b indicating whether all multiplication tuples are correct. The description of $\mathcal{F}_{\text{multVerify}}$ appears in Functionality 11. An instantiation of $\mathcal{F}_{\text{multVerify}}$ can be found in [20] (Protocol 17 in Section 5.4 of [19]).

Functionality 11: $\mathcal{F}_{\text{multVerify}}$

- (1) Let N denote the number of multiplication tuples. The multiplication tuples are denoted by $([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), \dots, ([x^{(N)}]_t, [y^{(N)}]_t, [z^{(N)}]_t)$.
- (2) For all $i \in [N]$, $\mathcal{F}_{\text{multVerify}}$ receives from honest parties their shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$. Then $\mathcal{F}_{\text{multVerify}}$ reconstructs the secrets $x^{(i)}, y^{(i)}, z^{(i)}$. $\mathcal{F}_{\text{multVerify}}$ further computes the shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$ held by corrupted parties and sends these shares to the adversary.
- (3) For all $i \in [N]$, $\mathcal{F}_{\text{multVerify}}$ computes $d^{(i)} = z^{(i)} - x^{(i)} \cdot y^{(i)}$ and sends $d^{(i)}$ to the adversary.
- (4) Finally, let $b \in \{\text{abort}, \text{accept}\}$ denote whether there exists $i \in [N]$ such that $d^{(i)} \neq 0$. $\mathcal{F}_{\text{multVerify}}$ sends b to the adversary and waits for its response.
 - If the adversary replies continue, $\mathcal{F}_{\text{multVerify}}$ sends b to honest parties.
 - If the adversary replies abort, $\mathcal{F}_{\text{multVerify}}$ sends abort to honest parties.

Preparing Random degree- t Shamir Sharings. The functionality $\mathcal{F}_{\text{rand}}$ allows all parties to prepare a random degree- t Shamir sharing $[r]_t$. The description of $\mathcal{F}_{\text{rand}}$ appears in Functionality 12. An instantiation of $\mathcal{F}_{\text{rand}}$ can be found in [9, 20] (Protocol 2 in Section 3.3 of [19]).

Functionality 12: $\mathcal{F}_{\text{rand}}$

- (1) $\mathcal{F}_{\text{rand}}$ receives from the adversary the set of shares $\{r_i\}_{i \in C}$.
- (2) $\mathcal{F}_{\text{rand}}$ randomly samples r . Based on the secret r and the t shares $\{r_i\}_{i \in C}$ of corrupted parties, $\mathcal{F}_{\text{rand}}$ reconstructs the whole sharing $[r]_t$ and distributes the shares of $[r]_t$ to honest parties.

Generating Random Coins. The functionality $\mathcal{F}_{\text{coin}}$ allows all parties to generate a random field element in $\mathbb{K}$. Recall that $\mathbb{K}$ is an extension field of $\mathbb{F}$ such that $|\mathbb{K}| \geq 2^\kappa$, where κ is the security parameter. The description of $\mathcal{F}_{\text{coin}}$ appears in Functionality 13. An instantiation of $\mathcal{F}_{\text{coin}}$ can be found in [20] (Protocol 6 in Section 3.5 of [19]).

Functionality 13: $\mathcal{F}_{\text{coin}}$

- (1) $\mathcal{F}_{\text{coin}}$ samples a random field element r in $\mathbb{K}$.
- (2) $\mathcal{F}_{\text{coin}}$ sends r to the adversary.
 - If the adversary replies continue, $\mathcal{F}_{\text{coin}}$ sends r to honest parties.
 - If the adversary replies abort, $\mathcal{F}_{\text{coin}}$ sends abort to honest parties.

C EVALUATION AND VERIFICATION OF INNER-PRODUCT OPERATIONS

In the following, for a vector $\mathbf{x} = (x_1, x_2, \dots, x_k)$, we use $[\mathbf{x}]_t$ to denote a vector of degree- t Shamir sharings $([x_1]_t, [x_2]_t, \dots, [x_k]_t)$. We first present the functionality $\mathcal{F}_{\text{inner}}$, which takes two vectors of degree- t sharings $[\mathbf{x}]_t, [\mathbf{y}]_t$ as input, and outputs $[z]_t = [\mathbf{x} \odot \mathbf{y}]_t$. The description of $\mathcal{F}_{\text{inner}}$ appears in Functionality 14.

As $\mathcal{F}_{\text{mult}}$, the functionality $\mathcal{F}_{\text{inner}}$ does not guarantee the correctness of the final results. Therefore, we need to verify the correctness of inner-product tuples computed by $\mathcal{F}_{\text{inner}}$. We describe the functionality $\mathcal{F}_{\text{innerVerify}}$ which is used to verify a batch of inner-product tuples computed by $\mathcal{F}_{\text{inner}}$ in Functionality 15. Note that a multiplication tuple can be seen as an inner-product tuple where the dimension of the input vectors is 1.

To securely compute $\mathcal{F}_{\text{innerVerify}}$, we adapt the construction for $\mathcal{F}_{\text{multVerify}}$ (Functionality 11) in [20]. We first introduce the idea of verifying a batch of multiplication tuples in [20]. At a high level, the verification is done as follows:

- (1) Suppose $\{([x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t)\}_{i=1}^N$ are the multiplication tuples.

Functionality 14: $\mathcal{F}_{\text{inner}}$

- (1) Let $[x]_t, [y]_t$ denote the input sharings. $\mathcal{F}_{\text{inner}}$ receives from honest parties their shares of $[x]_t, [y]_t$. Then $\mathcal{F}_{\text{inner}}$ reconstructs the secrets x, y . $\mathcal{F}_{\text{inner}}$ further computes the shares of $[x]_t, [y]_t$ held by corrupted parties, and sends these shares to the adversary.
- (2) $\mathcal{F}_{\text{inner}}$ receives from the adversary a value d and a set of shares $\{z_i\}_{i \in C}$.
- (3) $\mathcal{F}_{\text{inner}}$ computes $x \odot y \odot d$. Based on the secret $z := x \odot y \odot d$ and the t shares $\{z_i\}_{i \in C}$, $\mathcal{F}_{\text{inner}}$ reconstructs the whole sharing $[z]_t$ and distributes the shares of $[z]_t$ to honest parties.

Functionality 15: $\mathcal{F}_{\text{innerVerify}}$

- (1) Let N denote the number of inner-product tuples. The inner-product tuples are denoted by $([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), \dots, ([x^{(N)}]_t, [y^{(N)}]_t, [z^{(N)}]_t)$. The dimensions of different tuples can vary.
- (2) For all $i \in [N]$, $\mathcal{F}_{\text{innerVerify}}$ receives from honest parties their shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$. Then $\mathcal{F}_{\text{innerVerify}}$ reconstructs the secrets $x^{(i)}, y^{(i)}, z^{(i)}$. $\mathcal{F}_{\text{innerVerify}}$ further computes the shares of $[x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t$ held by corrupted parties and sends these shares to the adversary.
- (3) For all $i \in [N]$, $\mathcal{F}_{\text{innerVerify}}$ computes $d^{(i)} = z^{(i)} - x^{(i)} \odot y^{(i)}$ and sends $d^{(i)}$ to the adversary.
- (4) Finally, let $b \in \{\text{abort}, \text{accept}\}$ denote whether there exists $i \in [N]$ such that $d^{(i)} \neq 0$. $\mathcal{F}_{\text{innerVerify}}$ sends b to the adversary and waits for its response.
 - If the adversary replies continue, $\mathcal{F}_{\text{innerVerify}}$ sends b to honest parties.
 - If the adversary replies abort, $\mathcal{F}_{\text{innerVerify}}$ sends abort to honest parties.

- (2) All parties first transform these N multiplication tuples into a single inner-product tuple, denoted by $([x]_t, [y]_t, [z]_t)$. It satisfies that, if there exists $i \in [N]$ such that $([x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t)$ is an incorrect multiplication tuple, then $([x]_t, [y]_t, [z]_t)$ is an incorrect inner-product tuple with overwhelming probability.
- (3) All parties repeatedly reduce the dimension of the inner-product tuple $([x]_t, [y]_t, [z]_t)$ until the dimension becomes 1.
- (4) All parties check the final multiplication tuple (note that an inner-product tuple with dimension 1 is just a multiplication tuple).

To support the verification of inner-product tuples, we only need to change the second step by transforming N inner-product tuples into a single inner-product tuple. This can be done by following a similar approach in [20].

In [20], the transformation is done by the following steps (Protocol 13 in Section 5.1 of [19]):

- (1) Suppose $([x^{(1)}]_t, [y^{(1)}]_t, [z^{(1)}]_t), \dots, ([x^{(N)}]_t, [y^{(N)}]_t, [z^{(N)}]_t)$ are the multiplication tuples.
- (2) All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random field element r .
- (3) All parties set

$$[x]_t = ([x^{(1)}]_t, r \cdot [x^{(2)}]_t, \dots, r^{N-1} \cdot [x^{(N)}]_t)$$

$$[y]_t = ([y^{(1)}]_t, [y^{(2)}]_t, \dots, [y^{(N)}]_t)$$

$$[z]_t = \sum_{i=1}^N r^{i-1} \cdot [z^{(i)}]_t,$$

and output $([x]_t, [y]_t, [z]_t)$.

To transforming N inner-product tuples into a single inner-product tuple, we make the changes as follows:

- (1) Suppose $\{([x^{(i)}]_t, [y^{(i)}]_t, [z^{(i)}]_t)\}_{i=1}^N$ are the multiplication tuples.
- (2) All parties invoke $\mathcal{F}_{\text{coin}}$ to generate a random field element r .
- (3) All parties set

$$[x]_t = ([x^{(1)}]_t, r \cdot [x^{(2)}]_t, \dots, r^{N-1} \cdot [x^{(N)}]_t)$$

$$[y]_t = ([y^{(1)}]_t, [y^{(2)}]_t, \dots, [y^{(N)}]_t)$$

$$[z]_t = \sum_{i=1}^N r^{i-1} \cdot [z^{(i)}]_t,$$

and output $([x]_t, [y]_t, [z]_t)$.

With these changes, we obtain an efficient verification for a batch of inner-product tuples. The security can be shown in a similar way to that in [20]. We refer the readers to Lemma 8 (Section 5.1) and Lemma 11 (Section 5.4) in [19] for more details. The communication complexity of the new protocol is $O(n^2 \cdot \log \ell \cdot \kappa)$ bits, where n is the number of parties, ℓ is the summation of the dimensions of all inner-product tuples, and κ is the security parameter.

D PROOF OF SECURITY**D.1 Proof of Lemma 5.1**

LEMMA 5.1. *The protocol BRTS securely computes $\mathcal{F}_{\text{bit}}$ with abort in the $\{\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{mult}}, \mathcal{F}_{\text{multVerify}}\}$ -hybrid model in the presence of a fully malicious adversary controlling t corrupted parties.*

PROOF. Let $\mathcal{A}$ denote the adversary. We will construct a simulator $\mathcal{S}$ to simulate the behaviors of honest parties. Recall that C denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

Simulation for BRTS. We describe the strategy of $\mathcal{S}$ step by step.

- (1) In the first step, $\mathcal{S}$ emulates $\mathcal{F}_{\text{rand}}$ and receives the shares of $[r_1]_t, \dots, [r_N]_t$ held by corrupted parties.
- (2) In the second step $\mathcal{S}$ emulates $\mathcal{F}_{\text{mult}}$ and receives the shares of $[r_1^2]_t, \dots, [r_N^2]_t$ held by corrupted parties and the differences $d_1, \dots, d_N$.
- (3) $\mathcal{S}$ emulates $\mathcal{F}_{\text{multVerify}}$ in this step. Concretely, for all $i \in [N]$, $\mathcal{S}$ sends the shares of $([r_i]_t, [r_i]_t, [r_i^2]_t)$ of corrupted parties and the difference d_i to the adversary. If there exists

- $i \in [N]$ such that $d_i = 0$, S sets $b = \text{abort}$. Otherwise S sets $b = \text{accept}$. Then S follows the rest of steps in $\mathcal{F}_{\text{multVerify}}$.
- (4) For all $i \in [N]$, S randomly samples a non-zero element r and sets $r_i^2 := r^2$. Based on the secret r_i^2 and the shares of $[r_i^2]_t$ held by corrupted parties, S computes the shares of honest parties. Then S honestly follows this step.
 - (5) S follows this step in BTRs . If some party aborts, S sends abort to $\mathcal{F}_{\text{bit}}$ and aborts.
 - (6) For all $i \in [N]$, S computes the shares of $[b_i]_t$ held by corrupted parties. Then S sends the shares of $[b_i]_t$ of corrupted parties to $\mathcal{F}_{\text{bit}}$.

Analysis of the security. Note that the first three steps do not require honest parties to send any message to corrupted parties. In Step 4, the real world execution uses a uniformly random r_i (sampled by $\mathcal{F}_{\text{rand}}$) while the ideal world execution uses a uniformly random non-zero r_i . Since the field size of $\mathbb{Z}_q$ is $q = 2^\ell - 1 \geq 2^k$, the distributions of r_i in both executions are statistically close. Then the shares of $[r_i^2]_t$ held by honest parties are determined by the secret r_i^2 and the shares of corrupted parties. Thus, the distributions of the shares of $[r_i^2]_t$ held by honest parties in both executions are statistically close.

Now we argue that, with overwhelming probability, all parties obtain the correct reconstructions in Step 4, or at least one party aborts. We use $r_1^2, \dots, r_N^2$ to denote the secrets of $[r_1^2]_t, \dots, [r_N^2]_t$, and $\tilde{r}_1^2, \dots, \tilde{r}_N^2$ to denote the reconstruction results output by P_1 . Assume that there exists $i \in [N]$ such that $r_i^2 \neq \tilde{r}_i^2$. We show that the probability that all parties accept the reconstructions in Step 4.(d) is negligible. Consider the following two polynomials:

$$\begin{aligned} f(\lambda) &= r_1^2 + r_2^2 \cdot \lambda + \dots + r_N^2 \cdot \lambda^{N-1} \\ \tilde{f}(\lambda) &= \tilde{r}_1^2 + \tilde{r}_2^2 \cdot \lambda + \dots + \tilde{r}_N^2 \cdot \lambda^{N-1} \end{aligned}$$

Then $f(\cdot)$ and $\tilde{f}(\cdot)$ are two different polynomials. The number of λ such that $f(\lambda) = \tilde{f}(\lambda)$ is bounded by $N - 1$. Since λ is randomly sampled from $\mathbb{Z}_q$, the probability that $f(\lambda) = \tilde{f}(\lambda)$ is at most $\frac{N-1}{2^k-1}$. Thus, with overwhelming probability, $f(\lambda) \neq \tilde{f}(\lambda)$. Note that in Step 4.(c), we have $[x]_t = [f(\lambda)]_t$ and $x = \tilde{f}(\lambda)$. If $f(\lambda) \neq \tilde{f}(\lambda)$, an honest party will either receive an inconsistent sharing $[x]_t$ or reject the reconstruction results. In either case, this honest party will abort.

Finally given $r_i^2 = 0$, $r_i = r'_i$ and $r_i = -r'_i$ happen with equal probability. Therefore, b_i is a uniformly random bit in both executions. Thus, the distributions of the real world execution and the ideal world execution are statistically close.

D.2 Proof of Lemma 5.2

LEMMA 5.2. *The protocol DECOMPOSE securely computes $\mathcal{F}_{\text{decompose}}$ with abort in the $\mathcal{F}_{\text{bit}}$ -hybrid model in the presence of a fully malicious adversary controlling t corrupted parties.*

PROOF. Let $\mathcal{A}$ denote the adversary. We will construct a simulator S to simulate the behaviors of honest parties. Recall that C denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

Simulation for DECOMPOSE. At the beginning of the protocol, S receives the shares of $[x]_t$ held by corrupted parties and the value $x = r$. We describe the strategy of S step by step.

- (1) In the first step, S emulates $\mathcal{F}_{\text{bit}}$ and receives the shares of $[r_\ell]_t, [r_{\ell-1}]_t, \dots, [r_1]_t$ held by corrupted parties. Then, S computes the shares of $[r]_t$ held by corrupted parties.
- (2) In the second step, S computes the shares of $[x - r]_t$ held by corrupted parties. Based on the secret $x = r$ received from $\mathcal{F}_{\text{decompose}}$ and the shares of corrupted parties, S reconstructs the whole sharing. S follows the rest of this step. If some party aborts, S sends abort to $\mathcal{F}_{\text{decompose}}$ and aborts.
- (3) Finally, S sends the shares of $[r_\ell]_t, [r_{\ell-1}]_t, \dots, [r_1]_t$ to $\mathcal{F}_{\text{decompose}}$, and then sends continue to $\mathcal{F}_{\text{decompose}}$.

Analysis of the security. Note that the only difference between the real world execution and the ideal world execution is that, in the ideal world, $r = (r_\ell r_{\ell-1} \dots r_1)_2$ is randomly sampled by $\mathcal{F}_{\text{decompose}}$, while in the real world, $r = (r_\ell r_{\ell-1} \dots r_1)_2$ is generated by first preparing ℓ random bits $r_\ell, r_{\ell-1}, \dots, r_1$, and then computes $r = \sum_{i=1}^{\ell} 2^{i-1} \cdot r_i$. Since $\mathbb{Z}_q$ is a Mersenne field, we always have $r \leq q = 2^\ell - 1$, and

$$r = \sum_{i=1}^{\ell} 2^{i-1} \cdot r_i = (r_\ell r_{\ell-1} \dots r_1)_2$$

holds for all $r < q$, which happens with overwhelming probability. Therefore, the distributions of $r = (r_\ell r_{\ell-1} \dots r_1)_2$ in both executions are statistically close, which implies that the distributions of both executions are statistically close.

D.3 Proof of Theorem 5.5

THEOREM 5.5. *Let c be the number of clients and $n = 2t - 1$ be the number of parties. The protocol ACCO securely computes $\mathcal{F}_{\text{ACCO}}$ with abort in the $\{\mathcal{F}_{\text{inner}}, \mathcal{F}_{\text{decompose}}, \mathcal{F}_{\text{innerVerify}}\}$ -hybrid model in the presence of a fully malicious adversary controlling up to c clients and t parties.*

PROOF. According to Theorem 4.1, we assume that the adversary controls exactly t parties.

Let $\mathcal{A}$ denote the adversary. We will construct a simulator S to simulate the behaviors of honest parties and honest clients. Recall that C denotes the set of corrupted parties and $\mathcal{H}$ denotes the set of honest parties.

Simulation for ACCO. We describe the strategy of S phase by phase.

- Simulation for Input Phase:

For an input x belongs to an honest client, S randomly samples t elements as the shares of $[x]_t$ of corrupted parties. Then S sends these shares to corrupted parties.

For an input x belongs to a corrupted client, S receives from the adversary the shares held by honest parties. Note that, S learns $t - 1$ shares of $[x]_t$. S reconstructs the whole sharing $[x]_t$ and sends x as the input of this corrupted client to $\mathcal{F}_{\text{ACCO}}$.

Note that for each input sharing, S learns the shares held by corrupted parties.

- Simulation for Computation Phase:

In the computation phase, $\mathcal{S}$ will compute the shares of each sharing held by corrupted parties. Note that this already holds for the sharing of each input gate.

- For each addition gate with input sharings $[x]_t, [y]_t$, $\mathcal{S}$ computes the shares of $[z]_t = [x]_t + [y]_t$ held by corrupted parties.
- For each inner-product gate with input sharings $[x]_t, [y]_t$, $\mathcal{S}$ emulates $\mathcal{F}_{\text{inner}}$ and receives the shares of $[z]_t$ held by corrupted parties and the difference d .
- For each comparison gate with input sharings $[x]_t, [y]_t$, $\mathcal{S}$ simulates **LESTHAN**. Note that **LESTHAN** only contains local computation and invocations of $\mathcal{F}_{\text{decompose}}$ and $\mathcal{F}_{\text{inner}}$. For local computation, $\mathcal{S}$ follows the protocol and computes the shares of corrupted parties. $\mathcal{S}$ emulates $\mathcal{F}_{\text{inner}}$ in the same way as that for each inner-product gate. For $\mathcal{F}_{\text{decompose}}$, $\mathcal{S}$ sends to the adversary the shares of $[x]_t$ of corrupted parties computed by itself. Then $\mathcal{S}$ samples a random element in $\mathbb{Z}_q$ as $x \cdot r$ and sends it to the adversary. For all $i \in [\ell]$, $\mathcal{S}$ receives the shares of $[r_i]_t$ of corrupted parties. Finally, if $\mathcal{S}$ receives abort, $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{ACCO}}$ and aborts.
- For each truncation gate with input $[x]_t, f \in \mathbb{N}^*$, $\mathcal{S}$ simulates **TRUNC**. As **LESTHAN**, **TRUNC** only contains local computation and invocations of $\mathcal{F}_{\text{decompose}}$ and $\mathcal{F}_{\text{inner}}$. For local computation, $\mathcal{S}$ follows the protocol and computes the shares of corrupted parties. $\mathcal{S}$ emulates $\mathcal{F}_{\text{decompose}}$ and $\mathcal{F}_{\text{inner}}$ in the same way as that for **LESTHAN**.

Note that for each inner-product tuple $([x]_t, [y]_t, [z]_t)$ computed by $\mathcal{F}_{\text{inner}}$, $\mathcal{S}$ learns the shares held by corrupted parties and the difference $d = z - x \odot y$.

- Simulation for Verification Phase:

$\mathcal{S}$ emulates the functionality $\mathcal{F}_{\text{innerVerify}}$. Recall that in the simulation of the computation phase, $\mathcal{S}$ has computed the shares of each inner-product tuple held by corrupted parties and the difference. $\mathcal{S}$ directly sends these shares and differences to the adversary as in $\mathcal{F}_{\text{innerVerify}}$. If there exists a non-zero difference, $\mathcal{S}$ sets $b = \text{abort}$. Otherwise, $\mathcal{S}$ sets $b = \text{accept}$. Then $\mathcal{S}$ sends b to the adversary.

- If $b = \text{accept}$ and the adversary replies continue, $\mathcal{S}$ moves to the next phase.
- Otherwise, $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{ACCO}}$ and aborts.

- Simulation for Output Phase:

For each output gate with $[x]_t$ associated with it, if the receiver is an honest client, $\mathcal{S}$ receives from the adversary the shares held by corrupted parties. Then $\mathcal{S}$ checks whether the shares are the same as the ones computed by $\mathcal{S}$. If true, $\mathcal{S}$ accepts the output. Otherwise, $\mathcal{S}$ rejects the output.

If the receiver is a corrupted client, $\mathcal{S}$ receives the result x from $\mathcal{F}_{\text{ACCO}}$. Then, based on the shares of $[x]_t$ held by corrupted parties and the secret x , $\mathcal{S}$ reconstructs the shares held by honest parties, and sends these shares to the adversary.

Finally, if $\mathcal{S}$ rejects any output of honest clients, $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{ACCO}}$. Otherwise, $\mathcal{S}$ sends continue to $\mathcal{F}_{\text{ACCO}}$.

Hybrid Arguments. Now we show that $\mathcal{S}$ perfectly simulates the behaviors of honest parties with overwhelming probability. Consider the following hybrids.

Hybrid₀: The execution in the real world.

Hybrid₁: In this hybrid, $\mathcal{S}$ computes the input of corrupted clients and sends them to $\mathcal{F}_{\text{ACCO}}$. The distribution of **Hybrid₁** is identical to **Hybrid₀**.

Hybrid₂: In this hybrid, $\mathcal{S}$ simulates the output phase as described above. Note that the output phase is executed only when the computation is correct. For **TRUNC**, when the computation is correct, the distribution of the secret of the output sharing is statistically close to the truncation result defined in Definition 5.4. Therefore, the distribution of the outputs in **Hybrid₁** is statistically close to the distribution of the outputs in **Hybrid₂** (i.e., computed by $\mathcal{F}_{\text{ACCO}}$).

For an output gate with $[x]_t$ associated with it, if the receiver is an honest client, the shares that corrupted parties should hold are determined by the shares of honest parties. Therefore, if corrupted parties send different shares from the ones computed by $\mathcal{S}$, the shares of $[x]_t$ will be inconsistent and the client will reject the output. If the receiver is a corrupted client, the shares of honest parties are determined by the output x and the shares held by corrupted parties. Therefore, the shares prepared by $\mathcal{S}$ are identical to the real shares held by honest parties.

Therefore, the distribution of **Hybrid₂** is statistically close to the distribution of **Hybrid₁**.

Hybrid₃: In this hybrid, $\mathcal{S}$ computes the difference of each inner-product tuple in the computation phase. Then $\mathcal{S}$ simulates the verification phase. Note that $\mathcal{F}_{\text{innerVerify}}$ simply checks whether there is an incorrect inner-product tuple, which is equivalent to checking whether there is a non-zero difference.

Therefore, the distribution of **Hybrid₃** is identical to the distribution of **Hybrid₂**.

Hybrid₄: In this hybrid, $\mathcal{S}$ simulates the computation phase. For $\mathcal{F}_{\text{decompose}}$, $\mathcal{S}$ receives the shares of corrupted parties for $[r_\ell]_t, [r_{\ell-1}]_t, \dots, [r_1]_t$, which are sufficient for $\mathcal{S}$ to compute the shares of corrupted parties in the rest of steps. The value $x \cdot r$ is uniformly random in **Hybrid₃** since r is randomly chosen by $\mathcal{F}_{\text{decompose}}$. Therefore, $\mathcal{S}$ perfectly emulates $\mathcal{F}_{\text{decompose}}$. For $\mathcal{F}_{\text{inner}}$, $\mathcal{S}$ receives the shares of corrupted parties and the difference when emulating $\mathcal{F}_{\text{inner}}$. These are sufficient to simulating the verification phase and the output phase. Thus, the distribution of **Hybrid₄** is identical to the distribution of **Hybrid₃**.

Hybrid₅: In this hybrid, $\mathcal{S}$ simulates the input phase. The only difference between **Hybrid₄** and **Hybrid₅** is that, in **Hybrid₄**, $\mathcal{S}$ uses the real input of honest clients to generate the input sharings, while in **Hybrid₅**, $\mathcal{S}$ simply samples random elements as the shares of corrupted parties. Note that the distributions of the shares of corrupted parties in both hybrids are the same. Therefore, the distribution of **Hybrid₅** is the same as **Hybrid₄**.

Note that **Hybrid₅** is the execution in the ideal world, and the distribution of **Hybrid₅** is identical to the distribution of **Hybrid₀**, the execution in the real world.

E DERIVING AND IMPLEMENTING ADMM

In the following, we first give some preliminaries on the training algorithm we use: Alternating Direction Method of Multipliers (ADMM), and the general regression problem we consider: least absolute shrinkage and selection operator (LASSO). We then describe our experiment setup and report our performances on two public datasets.

E.1 ADMM training for LASSO regression

Discussions of Alternating direction Method of Multipliers (ADMM) in distributed machine learning can be found in [4]. We briefly re-create what's needed in our application. In this section, we only deal with real values in $\mathbb{R}$.

ADMM considers optimization problems of the following form:

$$\min_{x,z} f(x) + g(z) \quad \text{subject to} \quad x - Bz = c$$

The augmented Lagrangian (for a parameter $\rho > 0$) of the above problem is given by

$$L_\rho(x, z, u) = f(x) + g(z) + u^T(x - Bz - c) + \frac{\rho}{2} \|x - Bz - c\|_2^2$$

The ADMM algorithm solves the optimization problem by repeating: for $k = 1, 2, 3$,

$$\begin{aligned} x^k &= \arg \min_x L_\rho(x, z^{k-1}, u^{k-1}) \\ z^k &= \arg \min_z L_\rho(x^k, z, u^{k-1}) \\ u^k &= u^{k-1} + \rho(x^k - Bz^k - c) \end{aligned}$$

LASSO regression is an optimization problem defined as:

$$\min_w \frac{1}{2} \|Y - Xw\|_2^2 + \lambda \|w\|_1$$

Where $Y \in \mathbb{R}^{m \times 1}$ are m labels, $X \in \mathbb{R}^{m \times d}$ are m samples each containing d features, and $w \in \mathbb{R}^{d \times 1}$ is the desired model weight. Roughly, LASSO regression solves the least square problem while penalizing non-sparse solutions.

We can rewrite the problem as:

$$\min_{w,z} \frac{1}{2} \|Y - Xw\|_2^2 + \lambda \|z\|_1 \quad \text{subject to} \quad z - w = 0$$

and derive the following ADMM steps: for $k = 1, 2, 3$,

$$\begin{aligned} w^k &= (X^T X + \rho I)^{-1} (X^T Y + \rho(z^{k-1} - u^{k-1})) \\ z^k &= S_{\lambda/\rho}(w^k - u^{k-1}) \\ u^k &= u^{k-1} + w^k - z^k \end{aligned}$$

Where $S_\kappa : \mathbb{R} \rightarrow \mathbb{R}, \kappa > 0$ is the soft-threshold function:

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

In our application, each of the total n parties holds part of the entire dataset. The optimization problem therefore becomes:

$$\min_{w_1, \dots, w_n, z} \frac{1}{2} \sum_{i=1}^n \|Y_i - X_i w_i\|_2^2 + \lambda \|z\|_1 \quad \text{subject to} \quad z - w_i = 0$$

Where $\{X_i, Y_i\}$ represents the partial dataset held by party i . We can derive the following ADMM steps: for $k = 1, 2, 3$,

$$w_i^k = (X_i^T X_i + \rho I)^{-1} (X_i^T Y_i + \rho(z^{k-1} - u_i^{k-1})) \quad (1)$$

$$z^k = S_{\lambda/(n\rho)}\left(\frac{1}{n} \sum_{i=1}^n (w_i^k - u_i^{k-1})\right) \quad (2)$$

$$u_i^k = u_i^{k-1} + w_i^k - z^k \quad (3)$$

E.2 Implementing the ADMM steps

We implement the above iteration steps using $\mathcal{F}_{\text{ACCO}}$. We first note that in the derived ADMM steps from the previous subsection, the terms $(X_i^T X_i + \rho I)^{-1}$ and $X_i^T Y_i$ are fixed through out all iterations. We write $\tilde{w}_i = (X_i^T X_i + \rho I)^{-1}$ and $\tilde{b}_i = X_i^T Y_i$ as short-hands. Since $\tilde{w}_i, \tilde{b}_i$ depend only on the private dataset of party i , we do not need to include the calculation of $\tilde{w}_i, \tilde{b}_i$ into the MPC circuit. Instead, we let $\tilde{w}_i, \tilde{b}_i$ be the secret inputs from party i . $\tilde{w}_i, \tilde{b}_i$ can be seen as a compression of the private dataset from party i . Before input $\tilde{w}_i, \tilde{b}_i$ into $\mathcal{F}_{\text{ACCO}}$, we truncate each real number to the nearest fixed point value, and represent it as a field element. Algorithm 16 summarizes the algorithm we use in our experiments, where we set $t = 10$ (i.e. 10 iterations of ADMM). Since we are primarily concerned with realizing machine learning algorithms using our MPC protocol, we do not perform any hyper-parameter (e.g. λ, ρ) tuning in our training experiments.

Algorithm 16: ADMM-LASSO

Public Parameter: λ, ρ ; Initially: $w_i^0, u_i^0, z^0 = \vec{0}$.

Input: For $i = 1, \dots, n$, $\tilde{w}_i, \tilde{b}_i$.

For $k = 1, 2, \dots, t$

- (1) $w_i^k = \tilde{w}_i(\tilde{b}_i + \rho(z^{k-1} - u_i^{k-1}))$
- (2) $z^k = S_{\lambda/(n\rho)}\left(\frac{1}{n} \sum_{i=1}^n (w_i^k - u_i^{k-1})\right)$
- (3) $u_i^k = u_i^{k-1} + w_i^k - z^k$

We next describe the arithmetic circuit augmented with comparison and truncation gates that realizes Algorithm 16. Step 1 consists of vector addition and matrix multiplication, which translate into inner product and addition gates of field elements. Additionally, we add one truncation gate after each inner product to maintain the fixed point precision. Step 2 requires us to compute the soft-threshold function. We translate this function into comparison (abbreviated as $LT(x, y)$) and multiplication gates by noting:

$$\begin{aligned} S_\kappa(x) &= (LT(\kappa, x) - LT(\kappa, x)) \cdot \kappa \\ &\quad (LT(\kappa, x) - LT(\kappa, x)) \cdot x \end{aligned}$$

Step 3 only consists of vector addition. We translate those into addition gates of field elements directly.