SoK: Secure Computation over Secret Shares

Tamir Tassa
The Open University, Israel
tamirta@openu.ac.il

Arthur Zamarin
The Open University, Israel
arthurzam@gmail.com

Abstract

Secure multiparty computation (MPC) enables mutually distrustful parties to jointly compute functions over private data without revealing their inputs. A central paradigm in MPC is the secret-sharing-based model, where secret sharing underpins the efficient realization of arithmetic, comparison, numerical, and Boolean operations on shares of private inputs. In this paper, we systematize protocols for these operations, with particular attention to two foundational contributions [6, 21] that devised secure multiplication and comparison. Our survey provides a unified, self-contained exposition that highlights the composability, performance trade-offs, and implementation choices of these protocols. We further demonstrate how they support practical privacy-preserving systems, including recommender systems, distributed optimization platforms, and e-voting infrastructures. By clarifying the protocol landscape and connecting it to deployed and emerging applications, we identify concrete avenues for improving efficiency, scalability, and integration into real-world MPC frameworks. Our goal is to bridge theory and practice, equipping both researchers and practitioners with a deeper understanding of secret-sharing-based MPC as a foundation for privacy technologies.

Keywords: secure multiparty computation; secret sharing; privacy-preserving computation; secure numerical protocols; threshold cryptography; secure comparison; protocol efficiency

1 Introduction

Secure multiparty computation (MPC) enables a set of mutually distrusting parties to compute joint functions over private inputs while ensuring both correctness and privacy. A particularly elegant and practical approach to MPC is based on secret sharing, where each private value is split into shares distributed among the parties. Computation then proceeds directly on these shares, without revealing or reconstructing the underlying data at intermediate stages.

This SoK focuses on secure computation over secret-shared values, a setting that combines information-theoretic security in the honest-majority model with high efficiency. We give a detailed, self-contained treatment of two core building blocks—secure polynomial evaluation via multiplication and secure comparison, based on protocols devised by [6] and [21]. Along the way, we introduce concrete optimizations (e.g., reduced online complexity or more efficient alternative constructions) and demonstrate how these primitives enable other secure computations, such as equality testing, conditional branching, field inversion, integer division, computation of square roots, bit extraction, Boolean evaluation, and set-membership testing.

Secret-sharing-based MPC now underpins a wide range of privacy-preserving applications. In machine learning, it enables secure training and inference [18, 30]; in secure voting, it protects ballot privacy while supporting verifiability without reliance on a central authority [8, 29]; and in federated analytics, it allows the computation of aggregate statistics (e.g., means, medians, regression coefficients) without disclosing individual records [4, 20]. Further applications include genomic data analysis [19, 25], collaborative filtering [28, 16], distributed constraint optimization [27, 15], and privacy-preserving finance [1, 2]. We conclude this SoK with a detailed account of recent systems that implement MPC over secret shares in three domains: recommender systems, distributed optimization platforms, and e-voting infrastructures.

Our aim is to consolidate and systematize the foundations and applications of secret-sharing-based MPC, highlighting both common principles and domain-specific adaptations. By unifying these developments into a coherent framework, we not only clarify the state of the art but also chart directions for future research, providing a reference point for advancing the theory and practice of secure computation.

2 Preliminaries

We begin with a crash course on secret sharing (Section 2.1), followed by an overview of MPC based on secret shares (Section 2.2).

2.1 Secret Sharing

Secret sharing schemes [23] are protocols that enable distributing a secret scalar S among a set of parties, $P_1, \ldots, P_n$. Each party, P_i, $i \in [n]$,[1] gets a random value $[[S]]_i$, called *a share*, so that some subsets of those shares enable the reconstruction of S, while each of the other subsets of shares reveals no information on S. In its most basic form, called *Threshold Secret Sharing*, there is a threshold value $t \leq n$, and then a subset of shares enables the reconstruction of S iff its size is at least t.

Shamir's t-out-of-n threshold secret sharing scheme [23] operates over a finite field $\mathbb{Z}_p$, where $p > n$ is a prime sufficiently large so that all possible secrets may be represented in $\mathbb{Z}_p$. It has two procedures: Share and Reconstruct:

• $\text{Share}_{t,n}(S)$. The procedure samples a uniformly random polynomial $f(\cdot)$ over $\mathbb{Z}_p$, of degree at most $t-1$, where the free coefficient is the secret S. That is, $f(x) = S + a_1 x + a_2 x^2 + \ldots + a_{t-1} x^{t-1}$, where a_j, $1 \leq j \leq t-1$, are selected independently and uniformly at random from $\mathbb{Z}_p$. The procedure outputs n values, $[[S]]_i = f(i)$, $i \in [n]$, where $[[S]]_i$ is the share given to P_i. The entire set of shares, denoted $[[S]] := \{[[S]]_i : i \in [n]\}$, is called a (t,n)-sharing of S.

• $\text{Reconstruct}_t([[S]])$. The procedure is given any selection of t shares out of the (t,n)-sharing of S. It then interpolates a polynomial $f(\cdot)$ of degree at most $t-1$ using the given points and outputs $S = f(0)$. Any selection of t shares will yield the secret S, as t points determine a unique polynomial of degree at most $t-1$. On the other hand, any selection of $t-1$ shares or less reveals nothing about the secret S.

Hereinafter, we set the threshold to be

$$t := \lfloor (n+1)/2 \rfloor . \tag{1}$$

Namely, to reconstruct the secret, at least half of the parties must collaborate. Hence, if the set of n parties has an honest majority, in the sense that a majority (more than half) of them act honestly, without trying to collude to reconstruct the secret illicitly, the shared secret will remain fully protected.

In what follows, we shall use the following terminology and notations. Let S be a secret known to some party P_i, $i \in [n]$. Then if P_i performs the procedure $\text{Share}_{t,n}(S)$, we will simply say that P_i distributes a (t,n)-sharing of S.

If the parties have a (t,n)-sharing $[[S]]$ of some secret S and they wish to let one of them, say P_i, reconstruct the secret S, then at least $t-1$ parties would send their shares to P_i who will proceed to apply the Reconstruct procedure on the t shares it has. We will describe this procedure shortly by writing $S \leftarrow \text{Reconstruct}([[S]]; P_i)$.

Protocols 1 and 2 summarize those two basic operations.

Protocol 1: Share: distributing a (t,n)-sharing of $S \in \mathbb{Z}_p$

Input: $S \in \mathbb{Z}_p$ – the secret; $i \in [n]$ – the index of the dealer P_i
Parameter: $t \in [n]$ – the desired threshold

forall $j \in [t-1]$ **do**
  P_i generates a random $a_i \in \mathbb{Z}_p$
P_i sets $f(x) \leftarrow S + \sum_{j=1}^{t-1} a_j x^j$
forall $j \in [n]$ **do**
  P_i sets $[[S]]_j \leftarrow f(j)$
  P_i sends $[[S]]_j$ to P_j

Output: A (t,n)-sharing $[[S]] = \{[[S]]_i : i \in [n]\}$ of S

2.2 MPC over Secret-Sharings—Overview

Let u and v be two secret values in the field $\mathbb{Z}_p$, and assume that $P_1, \ldots, P_n$ hold (t,n)-sharings of them, denoted $[[u]] = \{[[u]]_i :\in [n]\}$ and $[[v]] = \{[[v]]_i :\in [n]\}$. Let $G(\cdot,\cdot)$ be a publicly known function.

[1] For any integer k we let $[k]$ denote the set $\{1, \ldots, k\}$.

Protocol 2: Reconstruct: Reconstructing a secret from its shares

Input: $[[S]]$ – a secret sharing of $S \in \mathbb{Z}_p$; $i \in [n]$ – the index of the party P_i that reconstructs S
Parameter: t – the secret sharing threshold
1 P_{i_j}, $j \in [t]$, send their shares $[[S]]_{i_j}$ to P_i
2 P_i interpolates and recovers the polynomial f of degree at most $t-1$ such that $f(i_j) = [[S]]_{i_j}$, $j \in [t]$
3 P_i recovers $S = f(0)$
Output: The secret S

Then MPC over secret shares is the problem of computing a (t, n)-sharing of $G(u, v)$ from $[[u]]$ and $[[v]]$, without learning any information on u, v, or $G(u, v)$. We begin by presenting MPC protocols for two fundamental functionalities: polynomial evaluation (Section 3) and secure comparison (Section 4). These primitives serve as the foundation for a broader toolkit; Section 5 builds on them to describe protocols for arithmetic, numerical, and Boolean computations.

3 Secure Polynomial Evaluation

Here we discuss the case when $G(\cdot, \cdot)$ is a polynomial. We begin by discussing affine combinations and then explain how to perform secure multiplications. With those two ingredients, it is possible to compute a sharing of $G(u, v)$ from sharings of u and v for any polynomial.

3.1 Affine Combinations

Let $\alpha, \beta, \gamma \in \mathbb{Z}_p$ be three values publicly known to all. Then

$$[[\alpha]] + \beta[[u]] + \gamma[[v]] := \{\alpha + \beta[[u]]_i + \gamma[[v]]_i : i \in [n]\}$$

is a proper (t, n)-sharing of $w := \alpha + \beta u + \gamma v$, thanks to the affinity of secret sharing. By writing

$$[[w]] \leftarrow [[\alpha]] + \beta[[u]] + \gamma[[v]]$$

we mean that each party P_i, $i \in [n]$, sets $[[w]]_i \leftarrow \alpha + \beta[[u]]_i + \gamma[[v]]_i$, so that now the parties hold a (t, n)-sharing of $w = \alpha + \beta u + \gamma v$ without needing to interact or to perform any further polynomial computations.

3.2 Generating Shares in a Secret Random

Before moving on to discussing the secure implementation of multiplying two shared scerets, we describe here a simple subprotocol (Protocol 3) that will be used later on—the generation of shares in a secret random field element. First, each party P_i, $i \in [n]$, generates a uniformly random field element, $r_i \in \mathbb{Z}_p$, and (t, n)-shares it among $P_1, \ldots, P_n$ (Lines 1-3). Let $[[r_i]]_j$ denote the share of r_i that P_i had sent to P_j. Subsequently, each P_j, $j \in [n]$, adds the shares that it got from all n parties and gets $[[r]]_j := \sum_{i=1}^{n} [[r_i]]_j$ (Lines 4-5). It is easy to see that $\{[[r]]_j :\in [n]\}$ is a (t, n)-sharing of the uniformly random number $r = \sum_{i=1}^{n} r_i$, and that no party has any information on r.

Protocol 3: RNG: Random number sharing – creating a (t, n)-sharing of a random secret

Parameter: $t \in [n]$ – the desired threshold
1 **forall** $i \in [n]$ **do**
2 | P_i generates a random $r_i \in \mathbb{Z}_p$
3 | P_i runs Share$(r_i, i; t)$
4 **forall** $j \in [n]$ **do**
5 | P_j sets $[[r]]_j = \sum_{i=1}^{n} [[r_i]]_j$
Output: A (t, n)-sharing of a secret random $r \in \mathbb{Z}_p$

3.3 Secure Multiplication

A secure multiplication protocol,

$$[[w]] \leftarrow \text{SecureMult}([[u]], [[v]]) \,,$$

takes (t, n)-sharings of u and v and computes from them a (t, n)-sharing of $w = u \cdot v$ in a secure manner, namely, without revealing to the parties any information on u, v, or $w = uv$. Damgård and Nielsen [7] designed such a secure multiplication protocol, which was later improved by Chida et al. [6]. We proceed to describe it.

First, each P_i sets $[[y]]_i := [[u]]_i \cdot [[v]]_i$, $i \in [n]$. The set of values $\{[[y]]_i : i \in [n]\}$ constitutes a $(2t-1, n)$-sharing of uv. Indeed, if the random polynomial that was used to generate shares of u (resp. v) was f_u (resp. f_v) then $[[y]]_i = F(i)$ where $F() = f_u() \cdot f_v()$ is a polynomial of degree (at most) $2t-2$ and $F(0) = f_u(0) f_v(0) = uv$.

Next, the parties follow the protocol outlined in Section 3.2 and generate two sharings of the same uniformly distributed random (and unknown) field element $r = R$: a (t, n)-sharing, denoted $\{[[r]]_i : i \in [n]\}$, and a $(2t-1, n)$-sharing, denoted $\{[[R]]_i : i \in [n]\}$. Then, each party P_i, $i \in [n]$, computes $[[z]]_i = [[y]]_i + [[R]]_i$ and sends the result to P_1. Since $\{[[z]]_i : i \in [n]\}$ is a $(2t-1, n)$-sharing of $z := uv + R$, P_1 can use any $2t-1$ of those shares in order to reconstruct $z = uv + R$. P_1 broadcasts that value to all parties. Consequently, each P_i computes $[[w]]_i = z - [[r]]_i$, $i \in [n]$. Since z is a publicly known field element and $[[r]]_i$ is a (t, n)-share of $r = R$, then $[[w]]_i$ is a (t, n)-share of $z - r = uv + R - r = uv$. Hence, $\{[[w]]_i : \in [n]\}$ is a (t, n)-sharing of uv, as required.

The above procedure is summarized in Protocol 4. It reveals no information on u, v, or uv, since the only reconstructed value is $uv + R$, where R is a uniformly distributed secret random.

Later on, in Section 6.1, we identify the computations in Protocol 4 that can be carried out offline, and compare the online runtime of the protocol to its overall runtime.

Protocol 4: SecureMult: Secure multiplication of shared secrets

Input: $[[u]]$ and $[[v]]$ – (t, n)-sharings of two secrets, $u, v \in \mathbb{Z}_p$
Parameter: t – the threshold
forall $i \in [n]$ **do**
  P_i generates a random $r_i \in \mathbb{Z}_p$
  P_i sets $R_i \leftarrow r_i$
  P_i runs Share$(r_i, i; t)$
  P_i runs Share$(R_i, i; 2t-1)$
forall $j \in [n]$ **do**
  P_j sets $[[r]]_j = \sum_{i=1}^{n} [[r_i]]_j$
  P_j sets $[[R]]_j = \sum_{i=1}^{n} [[R_i]]_j$
forall $i \in [n]$ **do**
  P_i sets $[[z]]_i \leftarrow [[u]]_i \cdot [[v]]_i + [[R]]_i$
The parties run Reconstruct$([[z]], 1; 2t-1)$
P_1 broadcasts z
forall $i \in [n]$ **do**
  P_i sets $[[w]]_i \leftarrow z - [[r]]_i$
Output: A (t, n)-sharing $[[w]]$ of $w = u \cdot v$

4 Secure Comparison

A secure comparison protocol,

$$[[w]] \leftarrow \text{SecureCompare}([[u]], [[v]]) \,,$$

takes (t, n)-sharings of u and v and securely computes from them a (t, n)-sharing of $w = 1_{u<v}$, where hereinafter if $\mathcal{P}$ is a predicate then $1_{\mathcal{P}}$ is a bit that equals 1 if the predicate $\mathcal{P}$ holds and equals 0 otherwise. Nishide and Ohta [21] proposed such a secure comparison protocol. We describe their protocol in a top-down manner.

4.1 The Top Level of the Protocol

Here we describe the top level of the protocol, that invokes two lower level computations: secure multiplications (Section 3.3) and a restricted comparison protocol that will be described in the subsequent sections.

First, the parties compute sharings of the following bits:

$$a = 1_{u<\frac{p}{2}} \quad , \quad b = 1_{v<\frac{p}{2}} \quad , \quad c = 1_{(u-v) \bmod p <\frac{p}{2}} \,. \tag{2}$$

Note that those three comparison bits a, b, c have a special form, since those are comparisons between a value in which the parties hold secret shares and the special value $\frac{p}{2}$. We postpone the discussion on how to compute such bits and assume for now that the parties had already computed such sharings, denoted $[[a]]$, $[[b]]$, and $[[c]]$. We observe that

$$w = 1_{u<v} = a\bar{b} \vee \bar{a}\bar{b}\bar{c} \vee ab\bar{c} \,. \tag{3}$$

The Boolean equality in Eq. (3) translates into an equivalent arithmetic expression:

$$\begin{aligned} w &= a(1-b) + (1-a)(1-b)(1-c) + ab(1-c) \\ &= 1 - b - c + bc + a(b + c - 2bc) \,. \end{aligned} \tag{4}$$

Eq. (4) may be rearranged as follows:

$$w = a \cdot (d - e) - d + 1 \,, \qquad \text{where} \qquad e = bc \,, \quad d = b + c - e \,.$$

Hence, the parties may proceed to compute a sharing of $w = 1_{u<v}$ as described in Protocol 5. The subprotocol LessThan_Half_P will be described in the next sections.

Before moving on, we note that in case the two secrets u and v are known to be smaller than $\frac{p}{2}$, we have $a = b = 1$. Hence, in that case Eq. (4) reduces to $w = 1 - c$, which means that the parties only need to compute shares of $c = 1_{(a-b) \bmod p <\frac{p}{2}}$ and then set $[[w]] \leftarrow 1 - [[c]]$. In Section 6.4 we demonstrate the reduction in runtime resulting from this simplified computation.

Protocol 5: SecureCompare: Secure comparison of shared secrets

Input: $[[u]]$ and $[[v]]$ – (t, n)-sharings of $u, v \in \mathbb{Z}_p$

1 $[[a]] \leftarrow$ LessThan_Half_P($[[u]]$)
2 $[[b]] \leftarrow$ LessThan_Half_P($[[v]]$)
3 $[[c]] \leftarrow$ LessThan_Half_P($[[u]] - [[v]]$)
4 $[[e]] \leftarrow$ SecureMult($[[b]], [[c]]$)
5 $[[d]] \leftarrow [[b]] + [[c]] - [[e]]$
6 $[[w]] \leftarrow$ SecureMult($[[a]], [[d]] - [[e]]$)
7 $[[w]] \leftarrow [[w]] - [[d]] + [[1]]$

Output: A (t, n)-sharing $[[w]]$ of $w = 1_{u<v}$

4.2 Comparing a Shared Value with $\frac{p}{2}$

Let $a \in \mathbb{Z}_p$ be any secret value that is (t, n)-secret-shared among the parties. Here we discuss how to compute shares of $w := 1_{a<\frac{p}{2}}$. The main observation is as follows: If $a < \frac{p}{2}$ then $2a < p$ and therefore $2a \bmod p$ equals $2a$ (no modular reduction is needed). Since that number is even, we infer that $(2a)_0 = 0$, where $(\cdot)_0$ denotes hereinafter the least significant bit (LSB). However, if $a \geq \frac{p}{2}$ then $2a \geq p$ and then $2a \bmod p$ equals $2a - p$. That number is odd (because p is odd) and then its LSB is 1. In view of the above, the parties can compute a sharing of $w = 1_{a<\frac{p}{2}}$ by running Protocol 6.

Our next task is to design the subprotocol LSB (Line 1 in Protocol 6) that takes a sharing of some secret and returns a sharing of its least significant bit.

4.3 LSB Computation

Assume that the parties hold a (t, n)-sharing $[[x]]$ of some secret $x \in \mathbb{Z}_p$, and they need to compute a (t, n)-sharing of $[[x_0]]$. Recall that $s := \lceil \log_2 p \rceil$, so each element in $\mathbb{Z}_p$ may be represented by s bits.

Protocol 6: LessThan_Half_P: Secure comparison of a shared secret a with $\frac{p}{2}$

Input: $[[a]]$ – a (t,n)-sharing of $a \in \mathbb{Z}_p$

1 $[[w]] \leftarrow 1 - \mathrm{LSB}(2[[a]])$

Output: A (t,n)-sharing $[[w]]$ of $w = 1_{a<\frac{p}{2}}$

The computation proceeds as follows. First, the parties compute (t,n)-sharings of s random bits r_i, $0 \le i \le s-1$. We defer the description of that computation to Subsection 4.3.1. Hence, at this stage each party P_j, $j \in [n]$, has a share $[[r_i]]_j$ for each $0 \le i \le s-1$, where $\{[[r_i]]_j : j \in [n]\}$ is a (t,n)-sharing of a random bit $r_i \in \{0,1\}$.

Then, they compute shares in the value

$$r = \sum_{i=0}^{s-1} 2^i r_i \bmod p\,. \tag{5}$$

To that end, each party P_j sets $[[r]]_j = \sum_{i=0}^{s-1} 2^i [[r_i]]_j$, $j \in [n]$, and then $\{[[r]]_j : j \in [n]\}$ is a (t,n)-sharing of the secret r that is given in Eq. (5). The value r distributes "almost" uniformly over $\mathbb{Z}_p$; it is possible to adjust the computation so that r distributes perfectly uniformly over $\mathbb{Z}_p$, as we disucss in Subsection 4.3.2.

In the next stage, the parties compute $[[c]] = [[x]] + [[r]]$ and proceed to publicly reconstruct $c = x + r$.

Lemma 1. *$x_0 = 1 - (c_0 \oplus r_0)$ if $c < r$ while $x_0 = c_0 \oplus r_0$ otherwise.*

Proof. The equality $c = x + r$ holds modulo p. We distinguish between two cases:

(a) If $x + r < p$ then $c = x + r$ without modular reduction and then $c \ge r$. In that case, since $c = x + r$ as integers in $\mathbb{Z}$, we have $c_0 = x_0 \oplus r_0$, and, consequently, $x_0 = c_0 \oplus r_0$.

(b) If $x + r \ge p$ then $c = x + r - p$. As p is odd then the latter equality implies that $c_0 = 1 - (x_0 \oplus r_0)$ which is equivalent to $x_0 = 1 - (c_0 \oplus r_0)$. Since $c = r - (p - x)$ and $p - x > 0$, we infer that in this case $c < r$. □

Let us denote $d_0 = c_0 \oplus r_0$. In view of Lemma 1, $x_0 = 1 - d_0$ if $c < r$ and $x_0 = d_0$ otherwise. Hence, if $e := 1_{c<r}$ then

$$x_0 = e \cdot (1 - d_0) + (1 - e) \cdot d_0 = e + d_0 - 2ed_0\,.$$

It follows that the parties can get a sharing of x_0 if they are able to compute sharings of e as well as of d_0, since then

$$[[x_0]] = [[e]] + [[d_0]] - 2[[ed_0]]\,. \tag{6}$$

Obtaining a sharing of d_0 is easy: indeed, since c is publicly known, so is c_0, and therefore the parties set $[[d_0]] = [[r_0]]$ if $c_0 = 0$ and $[[d_0]] = 1 - [[r_0]]$ otherwise. Regarding e, it is a comparison bit between a publicly known value, c, and another value, r, in which the parties hold sharings of each of its bits. In Section 4.4 we explain how to compute sharings of such comparison bits. Finally, a sharing of ed_0, as appears on the right hand side of Eq. (6), can be computed from $[[d_0]]$ and $[[e]]$ by invoking SecureMult (Section 3.3). Protocol 7 implements the above described computation.

At this point we are left with two "debts". How to compute a sharing in a random bit (the subprotocol GenRndBitSharing in Line 2 of Protocol 7) and how to perform the secure comparison between a public value and a bitwise-shared value (Bitwise_LessThan in Line 12 of Protocol 7). The former computation is described in Subsection 4.3.1 (see Protocol 8 there), while the latter one is discussed in Section 4.4 (see Protocol 9 there).

4.3.1 Generating shares in a secret random bit

Here we describe a protocol that computes a sharing of a secret random bit $b \in \{0,1\}$. First, the parties compute a sharing of a random $r \in \mathbb{Z}_p$, as described in Section 3.2. They then locally compute a $(2t-1,n)$-sharing of r^2 and proceed to recover r^2. If $r^2 = 0$ the parties select another random r. Otherwise, they compute, in polynomial time, a square root r' of r^2. As r^2 has two roots in the field we infer that r' can be either r or $-r$ with equal probabilities. Next, the parties set $[[b]] = 2^{-1} \cdot ((r')^{-1} \cdot [[r]] + 1)$. Since $(r')^{-1} r \in \{-1, 1\}$, with a uniform distribution, it follows that $b \in \{0,1\}$ is a uniformly distributed bit, as needed. Protocol 8 summarizes that computation.

Protocol 7: LSB: Secure computation of a sharing in the LSB of a shared secret

Input: $[[x]]$ – a (t, n)-sharing of $x \in \mathbb{Z}_p$
Parameter: $s = \lceil \log_2 p \rceil$
1 **forall** $0 \leq i \leq s-1$ **do**
2 | The parties run $[[r_i]] \leftarrow$ GenRndBitSharing()
3 **forall** $j \in [n]$ **do**
4 | P_j sets $[[r]]_j = \sum_{i=0}^{s-1} 2^i [[r_i]]_j$
5 $[[c]] \leftarrow [[x]] + [[r]]$
6 The parties run Reconstruct$([[c]], 1; t)$
7 P_1 broadcasts c
8 **if** $c_0 = 0$ **then**
9 | $[[d_0]] \leftarrow [[r_0]]$
10 **else**
11 | $[[d_0]] \leftarrow 1 - [[r_0]]$
12 $[[e]] \leftarrow$ Bitwise_LessThan$(c, \{[[r_i]]\}_{0 \leq i \leq s-1})$
13 $[[y]] \leftarrow$ SecureMult$([[e]], [[d_0]])$
14 $[[x_0]] \leftarrow [[e]] + [[d_0]] - 2[[y]]$
Output: A (t, n)-sharing of $[[x_0]]$

Protocol 8: GenRndBitSharing: Generating a (t, n)-sharing of a random bit

Parameter: $t \in [n]$ – the desired threshold
1 **forall** $i \in [n]$ **do**
2 | P_i generates a random $r_i \in \mathbb{Z}_p$
3 | P_i runs Share$(r_i, i; t)$
4 **forall** $j \in [n]$ **do**
5 | P_j sets $[[r]]_j = \sum_{i=1}^{n} [[r_i]]_j$
6 | P_j sets $[[r^2]]_j \leftarrow [[r]]_j \cdot [[r]]_j$
7 The parties run Reconstruct$([[r^2]], 1; 2t-1)$
8 P_1 broadcasts r^2
9 **if** $r^2 = 0$ **then**
10 | **go to** Line 1
11 Compute a square root r' of r^2
12 Compute $(r')^{-1}$
13 $[[b]] \leftarrow \frac{p+1}{2} \cdot ((r')^{-1} \cdot [[r]] + 1)$
Output: A (t, n)-sharing of a secret random bit $b \in \{0, 1\}$

4.3.2 The distribution of the bit-generated random value in $\mathbb{Z}_p$

The procedure described earlier for generating a random r by selecting randomly each of its bits does not produce a uniformly distributed random in $\mathbb{Z}_p$. Indeed, since r is defined through Eq. (5), the sum $\sum_{i=0}^{s-1} 2^i r_i$ may be greater than $p-1$. But then, after applying the modular reduction in order to produce a value in $\mathbb{Z}_p$, each of the values in the range $I_1 := [0, 2^s - p - 1]$ may be generated by two selections of random bits, while each value in the range $I_2 := [2^s - p, p-1]$ can be generated only by a single selection of bits. As a consequence, the probability of ending up with any $r \in I_1$ is 2^{-s+1}, while the probability of ending up with any $r \in I_1$ is 2^{-s}.

Example. Assume that $p = 19$. In that case $s = 5$ and $p = 10011_2$. Here, the two ranges in $\mathbb{Z}_{19}$ are $I_1 := [0, 2^5 - 19 - 1] = [0, 12]$ and $I_2 := [2^5 - 19, 19 - 1] = [13, 18]$. Each value $r \in I_1$ can be obtained by two selections of $s = 5$ bits—the one that corresponds to the binary representation of r and the one that corresponds to the binary representation of $r + 19$. Say, the value $r = 3$ will be generated either by the selection $(r_4 = 0, r_3 = 0, r_2 = 0, r_1 = 1, r_0 = 1)$ or by the selection $(r_4 = 1, r_3 = 0, r_2 = 1, r_1 = 1, r_0 = 0)$. On the other hand, any value $r \in I_2$ will be generated only by the single selection of bits that correspond to r's binary representation. Consequently, the probability of obtaining any given $r \in I_1$ is $\frac{1}{16}$ while that of obtaining any $r \in I_2$ is $\frac{1}{32}$. □

To mitigate that problem it is best advised to select a Mersenne prime, namely, a prime of the form $p = 2^s - 1$. In that case the range I_1 shrinks to $I_1 = \{0\}$ and then the resulting entropy of r's

distribution is $s - 2 \cdot 2^{-s}$ which is very close to the entropy of the desired uniform distribution, which is $\log_2 p = \log_2(2^s - 1) \approx s - \frac{1}{\ln 2} \cdot 2^{-s} = s - 1.44 \cdot 2^{-s}$. Furthermore, since with such primes there is only one selection of bits to rule out ($r_0 = \cdots = r_{s-1} = 1$), the parties can simply compute shares of $a := \sum_{i=0}^{s-1} r_i$ and then test securely whether $a = s$; such equality testing can be carried out by the IsZero protocol that we describe in Section 5.1. If equality holds, the parties can discard that selection and try a new one.

We note that a selection of a Mersenne prime is also advantageous in another manner, since it is possible to perform modular multiplication in such fields without an expensive modular reduction. Indeed, to compute $z = xy \bmod p$ one may compute $z = xy$, which is an integer of up to $2s$ bits, and then set

$$z_1 = z \wedge p\,, \qquad z_2 = z \gg s\,;$$

namely, z_1 is the number represented by the lower s bits of z, while z_2 is the number represented by the higher bits of z, shifted by s positions to the right. Finally, one sets $z = z_1 + z_2$, and if $z \geq p$ one updates its value to $z = z - p$. It can be easily verified that $z = xy \bmod p$.

4.4 Secure Bitwise Comparisons

Let a and b be two values in $\mathbb{Z}_p$ with corresponding binary representations $a = (a_{s-1}, \ldots, a_0)_2$ and $b = (b_{s-1}, \ldots, b_0)_2$. Assume that a is publicly known, while b is given through (t, n)-sharings of each of its s bits, $b_0, \ldots, b_{s-1}$. The goal is to compute a (t, n)-sharing of $f = 1_{a<b}$.

Let us first solve the problem in the clear, and only then translate it to a cryptographic solution. Define $c = a \oplus b$; namely, $c = (c_{s-1}, \ldots, c_0)_2$ where $c_i = a_i \oplus b_i$ for $0 \leq i \leq s-1$. The bits of c indicate the bit positions in which a and b differ. Next, define

$$d_i = \bigvee_{i \leq j \leq s-1} c_j\,, \qquad 0 \leq i \leq s-1\,.$$

Then $d = (d_{s-1}, \ldots, d_0)_2 = (0, \ldots, 0, 1, \ldots, 1)_2$ where the first 1 bit from the left indicates the most significant bit in which a and b differ, or $d = (0, \ldots, 0)_2$ in case $a = b$.

Next, set $e_{s-1} = d_{s-1}$ and then $e_i = d_i - d_{i+1}$ for all $0 \leq i \leq s-2$. It is easy to see that $e = (e_{s-1}, \ldots, e_0)_2 = (0, \ldots, 0, 1, 0, \ldots, 0)_2$ where the 1-bit indicates the most significant bit in which a and b differ, or $e = (0, \ldots, 0)_2$ if $a = b$.

Lemma 2. *Define* $\hat{e} := \sum_{i=0}^{s-1} e_i$ *and set* $f := \hat{e} \cdot (1 - \sum_{i=0}^{s-1} e_i a_i)$. *Then* $f = 1_{a<b}$.

Proof. First, we observe that $\hat{e} = 0$ if $a = b$ and $\hat{e} = 1$ if $a \neq b$. In the first case we get $f = 0$, as desired. In the second case we get $f := 1 - \sum_{i=0}^{s-1} e_i a_i$. Let j be the unique index where $e_j = 1$. If $a < b$ then $a_j = 0$ (and $b_j = 1$). Hence $f = 1$, which is indeed the value of $1_{a<b}$ in that case. If, however, $a > b$ then $a_j = 1$ (and $b_j = 0$). Hence $f = 0$, which is the value of $1_{a<b}$ in that case. □

Example. Assume that $p = 2^7 - 1 = 127$ (i.e., $s = 7$) and that the two inputs, given by their binary representations, are $a = 0101100_2$ and $b = 0101010_2$. Then $c = 0000110_2$ and, consequently, $d = 0000111_2$ and $e = 0000100_2$. Hence $\hat{e} = 1$ and then $f = 1 \cdot (1 - 1 \cdot 1\text{`}) = 0 = 1_{a<b}$. □

We proceed to discuss the secure implementation of the above comparison procedure. The computation starts with computing sharings of c_i, $0 \leq i \leq s-1$. Since $c_i = a_i \oplus b_i$ and a_i is publicly known, the parties set $[[c_i]] = [[b_i]]$ if $a_i = 0$ and $[[c_i]] = 1 - [[b_i]]$ otherwise.

Next, computing sharings of d_i is done from $i = s-1$ down to $i = 0$. First, $[[d_{s-1}]] = [[c_{s-1}]]$. Then, for $i = s-2, \ldots, 0$, we have $d_i = d_{i+1} \vee c_i = d_{i+1} + c_i - d_{i+1} \cdot c_i$. Hence, $[[d_i]] = [[d_{i+1}]] + [[c_i]] - \text{SecureMult}([[d_{i+1}]], [[c_i]])$. As for $[[e_i]]$, its computation is local and simple: first, $[[e_{s-1}]] = [[d_{s-1}]]$ and then, for $i = s-2, \ldots, 0$, $[[e_i]] = [[d_i]] - [[d_{i+1}]]$.

Afterwards, the parties compute $[[\hat{e}]] = \sum_{i=0}^{s-1} [[e_i]]$. Letting $g := 1 - \sum_{i=0}^{s-1} e_i a_i$, the parties compute shares of g by $[[g]] = 1 - \sum_{i=0}^{s-1} [[e_i]] a_i$; that computation is local since a_i are publicly known. Finally, the computation of shares of $f = \hat{e} \cdot g$ is carried out by $[[f]] = \text{SecureMult}([[\hat{e}]], [[g]])$.

Protocol 9 describes that computation.

Protocol 9: Bitwise_LessThan: Computing a sharing in $1_{a<b}$ where a is public and b is given through sharings of its bits

Input: $a \in \mathbb{Z}_p$ and $\{[[b_i]]\}_{0 \leq i \leq s-1}$, where $b_i \in \{0,1\}$ and $b = \sum_{i=0}^{s-1} 2^i b_i \in \mathbb{Z}_p$

Parameter: $s = \lceil \log_2 p \rceil$

1 **forall** $0 \leq i \leq s-1$ **do**
2 **if** $a_i = 0$ **then**
3 $[[c_i]] \leftarrow [[b_i]]$
4 **else**
5 $[[c_i]] \leftarrow 1 - [[b_i]]$
6 $[[d_{s-1}]] \leftarrow [[c_{s-1}]]$
7 $[[e_{s-1}]] \leftarrow [[c_{s-1}]]$
8 **forall** $i = s-2, \ldots, 0$ **do**
9 $[[d_i]] \leftarrow [[d_{i+1}]] + [[c_i]] - \text{SecureMult}([[d_{i+1}]], [[c_i]])$
10 $[[e_i]] \leftarrow [[d_i]] - [[d_{i+1}]]$
11 $[[\hat{e}]] \leftarrow \sum_{i=0}^{s-1} [[e_i]]$
12 $[[g]] \leftarrow 1 - \sum_{i=0}^{s-1} [[e_i]] a_i$
13 $[[f]] \leftarrow \text{SecureMult}([[\hat{e}]], [[g]])$

Output: A (t,n)-sharing $[[f]]$ of $f := 1_{a<b}$

4.4.1 Discussion

The secure bitwise comparison algorithm that we described herein differs from the one in [21] in two aspects.

The first one is in computing the bits d_i, $i = s-1, \ldots, 0$. While [21] implemented a technique due to [5] to compute all those bits in parallel in a constant number of rounds, we elected to perform that computation in the simpler manner as described herein, bearing the price of $O(s)$ computation rounds. In Section 6.6 we present an experimental comparison between our simplified computation and the approach proposed in [21].

The second one is in computing the final sharing of the comparison bit $f = 1_{a<b}$. By Lemma 2,

$$f = \hat{e} \cdot (1 - \sum_{i=0}^{s-1} e_i a_i)\,. \tag{7}$$

However, an alternative way to get f is through the equality

$$f = \sum_{i=0}^{s-1} e_i b_i\,. \tag{8}$$

While [21] compute the sharing of f using Eq. (8), it is preferable to base that final computation on Eq. (7). The reason is that a_i are publicly known while b_i are secret-shared. Hence, a computation that is based on Eq. (7) invokes only a single call to SecureMult to compute the sharing of the final bit $f = \hat{e} \cdot g$ (Line 13 in Protocol 9) , while a computation of such a sharing on the basis of Eq. (8) entails s invocations of SecureMult.

In Section 6.3 we illustrate the advantage, in terms of runtime, of using Eq. (7) instead of Eq. (8).

5 Other Computations

Here, we demonstrate how secure polynomial evaluation and secure comparison can serve as building blocks for other secure computations, including equality testing, conditional branching, field inversion, integer division, square root computation, bit extraction, Boolean evaluation, set-membership testing, and computing statistics over secret-shared datasets.

5.1 Equality Testing

Assume the parties hold a (t,n)-sharing of a secret u. The IsZero protocol takes this sharing as input and outputs a (t,n)-sharing of $w = 1_{u=0}$. Naturally, IsZero can also be used to test equality between

two secrets a and b by checking whether $u := a - b = 0$. Recall that by Fermat's little theorem, if $u \neq 0$ then $u^{p-1} = 1$. Hence, $w = 1 - u^{p-1}$. Consequently, IsZero can be implemented by applying the square and multiply algorithm and invoking the SecureMult protocol at most $2s$ times (recall that $s = \lceil \log_2 p \rceil$). Protocol 10 implements that computation.

Protocol 10: IsZero: Equality to zero

Input: $[[u]]$ – a (t, n)-sharing of $u \in \mathbb{Z}_p$

Parameter: t – the threshold; $p - 1 = (b_{s-1}, \ldots, b_0)_2$ – the binary representation of the order of $\mathbb{Z}_p^\times$

1 $[[z]] \leftarrow [[1]]$
2 **forall** $i = s - 1, \ldots, 0$ **do**
3 $\quad [[z]] \leftarrow \text{SecureMult}([[z]], [[z]])$
4 $\quad$ **if** $b_i = 1$ **then**
5 $\quad\quad [[z]] \leftarrow \text{SecureMult}([[z]], [[u]])$
6 $[[w]] \leftarrow [[1]] - [[z]]$

Output: A (t, n)-sharing of $w = 1_{u=0}$

5.2 Computations with Conditional Branching

Assume that the parties hold secret sharings of u_1, u_2, v_1, and v_2. Then a sharing of

$$w_1 = \begin{cases} v_1 & \text{if } \ u_1 = 0 \\ v_2 & \text{otherwise} \end{cases}$$

can be computed by the equality

$$w_1 = 1_{u_1=0} \cdot v_1 + (1 - 1_{u_1=0}) \cdot v_2 \,.$$

Such a computation, that invokes the IsZero and SecureMult protocols, does not leak to the parties any information on the inputs or the intermediate computed values, since all are secret-shared. Similarly, the parties can compute shares in

$$w_2 = \begin{cases} v_1 & \text{if } \ u_1 < u_2 \\ v_2 & \text{otherwise} \end{cases}$$

by

$$w_2 = 1_{u_1<u_2} \cdot v_1 + (1 - 1_{u_1<u_2}) \cdot v_2 \,.$$

The latter computation can be used to compute sharings of $w = \min\{u_1, u_2\}$ and of $w = \max\{u_1, u_2\}$.

Finally, if w is computed by a deeper conditional branching, it is possible to compute shares in it by combining and nesting computations as described above. For the sake of illustration, assume that the value of w is determined as follows:

$$\begin{array}{ll} \textbf{If } \ \mathcal{P}_1 \ \textbf{ then} & w = u_1 \\ \textbf{Else If } \ \mathcal{P}_2 \ \textbf{ then} & w = u_2 \\ \textbf{Else} & w = u_3 \,. \end{array}$$

Then

$$w = 1_{\mathcal{P}_1} \cdot u_1 + (1 - 1_{\mathcal{P}_1}) \cdot [1_{\mathcal{P}_2} \cdot u_2 + (1 - 1_{\mathcal{P}_2}) \cdot u_3] \ ,$$

and that expression can be used to compute shares of w using the basic toolkit that we described earlier.

5.3 Secure Inversion

Here we describe a secure inversion protocol that computes from a sharing of $u \in \mathbb{Z}_p^\times := \mathbb{Z}_p \setminus \{0\}$ a sharing of $v = u^{-1}$. The protocol starts by generating a sharing in a secret random field element, r, using Protocol 3 (Line 1). The parties then compute $w = u \cdot r$ (Lines 2-5). Since $u \neq 0$ and r is random, w reveals no information on u. As there is a negligible probability of $\frac{1}{p}$ that $r = 0$, the parties check whether $w = 0$ and if so they return to generate a sharing of a new random (Lines 6-7). Finally, they compute w^{-1} and each party P_i multiplies its share $[[r]]_i$ by w^{-1} (Line 8). The resulting shares constitute a sharing of $v = w^{-1} \cdot r = u^{-1}$.

Protocol 11: SecureInv: Secure inversion

Input: $[[u]]$ – a (t, n)-sharing of $u \in \mathbb{Z}_p^\times$

1 $[[r]] \leftarrow \text{RNG}(\cdot)$
2 **forall** $i \in [n]$ **do**
3 $\quad P_i$ sets $[[w]]_i \leftarrow [[u]]_i \cdot [[r]]_i$
4 The parties run Reconstruct$([[w]], 1; 2t-1)$
5 P_1 broadcasts w
6 **if** $w = 0$ **then**
7 $\quad$ Go to Line 1
8 $[[v]] \leftarrow w^{-1} \cdot [[r]]$

Output: A (t, n)-sharing of $v = u^{-1}$

5.4 Secure Division

A secure division protocol computes from the sharings of two secrets, u and v, sharings of the corresponding quotient $q = \lfloor \frac{u}{v} \rfloor$ and remainder $r = u \bmod v$. We start by recapping the binary long division algorithm. Assume that the binary representation of the numerator u is $(u_{s-1}, \ldots, u_0)_2$. Then Algorithm 12 performs a binary long division between u and v. Protocol 13 is a secure implementation of that algorithm. Note that, in view of our discussion in Section 5.2, Lines 5-6 in Protocol 13 implement in a secure manner Lines 5-9 in Algorithm 12. Because the algorithm and protocol are straightforward, we omit further explanation.

To the best of our knowledge, despite the fundamental nature of the secure division problem and its applicability to many privacy-preserving distributed machine learning computations, no prior work has presented such an algorithm. In particular, such a secure division protocol is essential to any privacy-preserving implementation of fundamental algorithms that involve divisions, e.g. k-means clustering [17], linear regression [9], collaborative filtering [10], or any algorithm that performs gradient descent. As the numerator and denominator in such computations are real-valued, while secret sharing is performed over a finite field, it is necessary to represent such real values by finite field elements. In order to perform such computations with accuracy of, say, d digits after the decimal point, each real-valued secret input x should be translated to an integer value that preserves its first d digits after the decimal point, $x \mapsto \hat{x} = \lfloor 10^d x + 0.5 \rfloor$, and then distribute a sharing of $\hat{x}$. (Of course, the selection of p, the size of the underlying field, should take into account both the estimated bound on all inputs as well as the rescaling factor 10^d.)

Algorithm 12: Binary long division

Input: $u, v \in \mathbb{Z}$, $u \in [0, p)$, $v \in (0, p)$; $u_{s-1}, \ldots, u_0$ such that $u = (u_{s-1}, \ldots, u_0)_2$

1 $q \leftarrow 0$
2 $r \leftarrow 0$
3 **forall** $j = s-1, \ldots, 0$ **do**
4 $\quad r \leftarrow 2r + u_j$
5 $\quad$ **if** $r < v$ **then**
6 $\quad\quad q_j \leftarrow 0$
7 $\quad$ **else**
8 $\quad\quad q_j \leftarrow 1$
9 $\quad\quad r \leftarrow r - v$
10 $\quad q \leftarrow 2q + q_j$

Output: The quotient $q = \lfloor \frac{u}{v} \rfloor$ and remainder $r = u \bmod v$

5.5 Secure Computation of Square Roots

Assume that the parties hold a (t, n)-sharing of a nonnegative integer u and they wish to compute a sharing of its real root, $\sqrt{u}$. Since $\sqrt{u}$ is typically irrational, we describe here a method to compute a sharing of its integer approximation, $\lfloor \sqrt{u} \rfloor$. In order to get an answer that is accurate to within d bits after the binary point, it is possible to apply the described method on $2^{2d}u$.

Algorithm 14 is a classical algorithm for extracting square roots [14]. The input bits are scanned in

Protocol 13: Secure binary long division

Input: $[[u]]$, $[[v]]$, $v \in (0, p)$, and $[[u_j]]$, $j \in \{0, 1, \ldots, s-1\}$, such that $u = (u_{s-1}, \ldots, u_0)_2$

1 $[[q]] \leftarrow [[0]]$
2 $[[r]] \leftarrow [[0]]$
3 **forall** $j = s-1, \ldots, 0$ **do**
4 $\quad [[r]] \leftarrow 2[[r]] + [[u_j]]$
5 $\quad [[q_j]] \leftarrow [[1]] - \text{SecureCompare}([[r]], [[v]])$
6 $\quad [[r]] \leftarrow [[r]] - \text{SecureMult}([[q_j]], [[v]])$
7 $\quad [[q]] \leftarrow 2[[q]] + [[q_j]]$

Output: A sharing of $q = \lfloor \frac{u}{v} \rfloor$ and of $r = u \bmod v$, as well as sharings of q's bits, $q_{s-1}, \ldots, q_0$

pairs, from the most significant pair of bits, u_{s-1}, u_{s-2}, to the least significant pair, u_1, u_0 (for the sake of simplicity, we assume that s is even). The value of the square root is built into w, bit by bit, from the MSB to the LSB, where in each iteration (namely, for each pair of bits in the input) a new bit is appended to w. The value of r stands for the remainder. Both w and r are initialized to zero (Lines 1-2). In each iteration, the next two input bits are appended from the right to the current value of r (Line 4). Then, we set $y = 4w + 1$ (Line 5) and check whether y can be deducted from r. If so, then the next bit in the root is 1. In that case, we deduct y from r and append the bit 1 to w (Lines 7-8). Otherwise, we leave r unchanged and append the bit 0 to w (Line 10). At the end, w will hold the sought-after rounded square root. The final value of r will be $r = u - w^2$.

The advantage of Algorithm 14 over other algorithms, such as Newton Raphson or binary search (that have the same time complexity, $O(s)$), is that it avoids divisions.

Algorithm 14: Bitwise extraction of roots

Input: $u \in \mathbb{Z}$, $u \in [0, p)$; $u_{s-1}, \ldots, u_0$ such that $u = (u_{s-1}, \ldots, u_0)_2$

1 $w \leftarrow 0$
2 $r \leftarrow 0$
3 **forall** $j = s-2, s-4, \ldots, 2, 0$ **do**
4 $\quad r \leftarrow 4r + 2u_{j+1} + u_j$
5 $\quad y \leftarrow 4w + 1$
6 $\quad$ **if** $y \leq r$ **then**
7 $\quad\quad r \leftarrow r - y$
8 $\quad\quad w \leftarrow 2w + 1$
9 $\quad$ **else**
10 $\quad\quad w \leftarrow 2w$

Output: The rounded square root, $w = \lfloor \sqrt{u} \rfloor$

Protocol 15 securely implements Algorithm 14. We believe that at this point it requires no further explanation. We note that the value of the bit b that is computed in Line 6 in the j-th iteration is the $\frac{j}{2}$-th bit in the output w. Hence, the protocol can issue not only a sharing of w but also a sharing of each of its bits. Another observation is that the final value of r is the remainder $u - w^2$. Hence, if at the end of the loop the parties proceed to compute a sharing of the bit $1_{r=0}$ it would indicate whether u is a perfect square, without revealing any other information on u.

5.6 Secure Extraction of Bits

Here we discuss the following problem: the parties hold a (t, n)-sharing of $u \in \mathbb{Z}_p$ and they wish to compute a (t, n)-sharing of u_j, for some $0 \leq j \leq s-1$, where $u = (u_{s-1}, \ldots, u_0)_2$ is the binary representation of u.

Nishide and Ohta [21] divised a protocol for concurrently computing (t, n)-sharings of all u's bits. That protocol (see [21, Fig. 2]) is computationally demanding. (Interested readers are referred to [21] for its description and the corresponding computational cost analysis.) Here, however, we focus on the case where it is needed to extract a sharing of only one specific bit, u_j, $0 \leq j \leq s-1$. To do so without performing a full bit decomposition, the parties may run Protocol 13 with the given $[[u]]$ and $[[v]] := [[2^j]]$; the output $[[q]]$ is a sharing of $q = \lfloor 2^{-j} u \rfloor = (0, \ldots, 0, u_{s-1}, \ldots, u_j)_2$. Running Protocol 7 on $[[q]]$ will issue a sharing of $q_0 = u_j$, as desired.

Protocol 15: Secure bitwise extraction of roots

Input: $[[u]]$, and $[[u_j]]$, $j \in \{0, 1, \ldots, s-1\}$, such that $u = (u_{s-1}, \ldots, u_0)_2$

1 $[[w]] \leftarrow [[0]]$
2 $[[r]] \leftarrow [[0]]$
3 **forall** $j = s-2, s-4, \ldots, 2, 0$ **do**
4 $\quad [[r]] \leftarrow 4[[r]] + 2[[u_{j+1}]] + [[u_j]]$
5 $\quad [[y]] \leftarrow 4[[w]] + [[1]]$
6 $\quad [[b]] \leftarrow [[1]] - \text{SecureCompare}([[r]], [[y]])$
7 $\quad [[r]] \leftarrow [[r]] - \text{SecureMult}([[b]], [[y]])$
8 $\quad [[w]] \leftarrow 2[[w]] + [[b]]$

Output: A sharing $[[w]]$ of $w = \lfloor \sqrt{u} \rfloor$

Note that since the denominator in the division is public in our case, $v = 2^j$, then Line 6 in Protocol 13 can be carried out locally, without invoking SecureMult, since the second multiplicand $v = 2^j$ is publicly known.

5.7 Computing Boolean Expressions

If the parties hold sharings of two secret bits u and v, they can securely compute sharings of the following Boolean expressions,

$$u \vee v = u + v - uv\,, \qquad u \wedge v = uv\,, \qquad \neg u = 1 - u\,, \tag{9}$$

using the previously described secure computation of affine combinations and multiplications. Hence, if the parties hold secret shares of the inputs of any Boolean circuit they can securely compute a sharing of the circuit's output.

Assume that the parties need to compute (a secret sharing of) an m-ary OR, $\bigvee_{i=1}^{m} u_i$, or an m-ary AND, $\bigwedge_{i=1}^{m} u_i$. They can do that sequentially over $m-1$ rounds of computation. However, Nishide and Ohta [21] devised a protocol that computes $\bigvee_{i=1}^{m} u_i$ in a constant number of rounds. That protocol can also be used to efficiently compute $\bigwedge_{i=1}^{m} u_i = 1 - \bigvee_{i=1}^{m}(1 - u_i)$.

Let $f(x) = \sum_{i=0}^{m} \alpha_i x^i$ be the unique m-degree polynomial that satisfies $f(1) = 0$ and $f(i) = 1$ for $i = 2, \ldots, m+1$. Hence, if

$$w := 1 + \sum_{i \in [m]} u_i \tag{10}$$

then $f(w) = \bigvee_{i=1}^{m} u_i$. Therefore, the parties need to compute $f(w)$ where f is a public m-degree polynomial and w is given in Eq. (10).

Protocol 16 does that. It starts by generating sharings in m randoms b_i and their inverses $c_i = b_i^{-1}$, $i \in [m]$ (Lines 1-3). The parties then compute, locally, a sharing of w, Eq. (10) (Line 4). Next, they jointly compute, in the clear, the values $e_i := w b_{i-1} b_i^{-1}$, $i \in [m]$, where $b_0 := 1$ (Lines 5-10). Since the definition of w, Eq. (10), implies that it is always nonzero, and b_i, $i \in [m]$, are nonzero randoms, the values of e_i reveal no information on w. Subsequently, they compute shares in w^i for all $i = 2, \ldots, m$ (Lines 11-12) and then in the desired output $v = f(w)$ (Line 13).

Lines 1-3 in Protocol 16 can be executed offline as they do not depend on the inputs. Lines 4-5 and 11-13 describe local computations. The parties interact only in the loop in Lines 6-10. The computations in that loop can be parallelized and, hence, it can be executed over a constant number of rounds.

5.8 Set Membership Testing

Assume that D is an interval in $\mathbb{Z}_p$, namely $D = [a, b]$ where $0 \le a \le b < p$. Then $u \in D$ iff $u \ge a$ and $u \le b$. Hence,

$$1_{u \in D} = 1_{u \ge a} \cdot 1_{u \le b} = 1_{a < u+1} \cdot 1_{u < b+1}\,. \tag{11}$$

The latter two bits can be computed using SecureCompare (Section 4). The interval endpoints a and b can be publicly known or secret-shared. In case they are publicly known then a corresponding sharing (to be used in the SecureCompare protocol) would be $[[a]] = \{[[a]]_i = a : i \in [n]\}$, and similarly for b.

If D is a discrete set of values, say $D = \{a_1, \ldots, a_k\}$, then

$$1_{u \in D} = 1_{\prod_{i=1}^{k}(u - a_i) = 0}\,. \tag{12}$$

Protocol 16: Computing an m-ary OR

Input: $[[u_i]]$, $i \in [m]$

Parameter: $f(x) = \sum_{i=0}^{m} \alpha_i x^i$

forall $i \in [m]$ **do**
  $[[b_i]] \leftarrow \text{RNG}(\cdot)$
  $[[c_i]] \leftarrow \text{SecureInv}([[b_i]])$
$[[w]] \leftarrow 1 + \sum_{i \in [m]} [[u_i]]$
$[[b_0]] \leftarrow [[1]]$
forall $i \in [m]$ **do**
  $[[d_i]] \leftarrow \text{SecureMult}([[w]], [[b_{i-1}]])$
  $[[e_i]] \leftarrow \text{SecureMult}([[d_i]], [[c_i]])$
  The parties run $\text{Reconstruct}([[e_i]], 1; t)$
  P_1 broadcasts e_i
forall $i = 2, \ldots, m$ **do**
  $[[w^i]] \leftarrow \left(\prod_{j=1}^{i} e_j\right) \cdot [[b_i]]$
$[[v]] \leftarrow \sum_{i=0}^{m} \alpha_i [[w^i]]$

Output: A sharing $[[v]]$ of $v = \bigvee_{i \in [m]} u_i$

Evaluating a sharing of $1_{u \in D}$ by Eq. (12) entails invoking SecureMult $k-1$ times followed by one invocation of IsZero.

Note that if D is an interval of short length then it is more efficient to compute sharings of $1_{u \in D}$ by viewing D as a discrete set of $b - a + 1$ elements and then relying on Eq. (12), rather than using Eq. (11).

Finally, if D is a union of intervals then a sharing of $1_{u \in D}$ can be computed by the above described techniques and a computation of logical OR using Eq. (9) or Protocol 16.

5.9 Statistical Computations

Assume that the parties hold secret shares of the values in some dataset $U = \{u_1, \ldots, u_m\}$, and they wish to compute statistics of U. The most basic statistics are the mean and standard deviation. Their computation (or computing secret shares of them) is easy as it involves only arithmetic operations. Hence, we focus in this section on computing other important statistics—quantiles. If $q > 1$ is an integer, the q-quantiles are $q-1$ values that partition U into q "bins" of sizes $b := \lfloor \frac{m}{q} \rfloor$ and (if $b < \frac{m}{q}$) also $b+1$. The determination of those values reduces to finding the k-th ranked element in U for $k \in K := \{k_1 < \cdots < k_{q-1}\}$, where k_i, $i \in [q-1]$, are defined as follows. If $m = q \cdot b + r$, where $r \in [0, q)$ (i.e., b and r are the quotient and remainder, respectively, when dividing m by q), then the sequence of indices k_i, $i \in [q-1]$, is:

$$k_0 = 0\,; \quad k_i = k_{i-1} + \Delta\,, \quad \text{where} \quad \Delta = \begin{cases} b+1 & \text{if} \quad i \leq r \\ b & \text{if} \quad i > r \end{cases}, \quad i \in [q-1]\,.$$

The size of the first r bins is $b+1$, while all other bins are of size b. Finally, the value of the k-th ranked element in U is the smallest integer M_k for which

$$|\{j \in [m] : u_j \leq M_k - 1\}| < k \tag{13}$$

and

$$|\{j \in [m] : u_j \leq M_k\}| \geq k\,. \tag{14}$$

(The median is M_1 when $q = 2$.)

The value of M_k, for any given k, can be found easily once the parties sort the array U. Using Merge Sort, it is possible to sort U, without disclosing the values in U, using $O(m \log_2 m)$ invocations of SecureCompare. Let $V = \{v_1, \ldots, v_m\}$ be the sorted array. Namely, $v_j = u_{i_j}$, for all $j \in [m]$, where $(i_1, \ldots, i_m)$ is the permutation of $(1, \ldots, m)$ that was computed by the sorting procedure and represents the sorted order of U's elements. At the completion of the secure implementation of Merge Sort the parties will hold sharings of v_j for each $j \in [m]$. Finally, the k-th ranked element in U is simply v_k, so $[[M_k]] = [[v_k]]$.

While the above strategy does not reveal the values in U, it does reveal their ranking. In some application scenarios it might be necessary to prevent such information leakage. For example, if U represents the salaries of employees in some organization, we would like to compute quantile statistics over U without disclosing any information on those salaries. Therefore, if an employee might be associated with her index j in U, revealing the relative position of her salary in the organization must be avoided.

Hence, we proceed to describe a protocol for computing M_k that does not reveal any information on U's entries. Given k, the corresponding k-th ranked element M_k is found by a binary search over the interval $[0, M)$, where M is a known upper bound on all elements in U. Such an upper bound must be known a priori in order to properly select the size p of the underlying field. For convenience, we will take M to be a power of 2, say $M = 2^\ell$.

Algorithm 17 implements a binary search to find the k-th ranked element in U, when U is public (as opposed to the setting which interests us, where U is secret-shared and has to remain secret). The variable α will store the current guess for the value of M_k. It is initialized to the middle of the range $[0, M)$ (Line 1). Then a binary search begins (Lines 2-7). First, the algorithm counts the number κ_1 of dataset values that are at most $\alpha - 1$ (Line 3). If $\alpha \leq M_k$, where M_k is the sought-after k-th ranked element, then that count would be smaller than k, as implied by Eq. (13). If, however, that count is already greater than or equal to k, then $\alpha > M_k$. Hence, in the first case we increment α while in the second case we decrement it (Lines 4-7). The value by which α is updated in iteration i is 2^{i-1} (Lines 5 and 7). At the end of the loop, we check the compliance of α with condition (14) and update its value if necessary (Line 8-10). The final value of α is the k-th ranked element in U (Line 11).

Algorithm 17: Computing the k-th ranked element

Input: m – the size of the dataset; $\{u_j\}_{j\in[m]}$ – the dataset; $M = 2^\ell$ – an upper bound on $\{u_j\}_{j\in[m]}$; $k \in [m]$

1 $\alpha \leftarrow 2^{\ell-1}$
2 **forall** $i = \ell - 1, \ldots, 1$ **do**
3 $\quad \kappa_1 \leftarrow \sum_{j\in[m]} 1_{u_j \leq \alpha - 1}$
4 $\quad$ **if** $\kappa_1 < k$ **then**
5 $\quad\quad \alpha \leftarrow \alpha + 2^{i-1}$
6 $\quad$ **else**
7 $\quad\quad \alpha \leftarrow \alpha - 2^{i-1}$
8 $\kappa_2 \leftarrow \sum_{j\in[m]} 1_{u_j \leq \alpha}$
9 **if** $\kappa_2 > k$ **then**
10 $\quad \alpha \leftarrow \alpha - 1$
11 $M_k \leftarrow \alpha$

Output: The k-th ranked element M_k

Lemma 3. *Algorithm 17 is correct and it performs $(m+1)\ell$ comparisons.*

Proof. Line 3 is executed $\ell - 1$ times and it involves m comparisons. Line 8 involves m additional comparisons. Lines 4 and 9 involve ℓ additional comparisons. Hence, the overall number of comparisons in the algorithm is as stated.

For each $i = \ell - 1, \ldots, 1$ let α_i denote the value of α at the beginning of the i-th iteration. Assume that in iteration i Line 5 was executed. Then $M_k \geq \alpha_i$, as implied by inequality (13) and the fact that $\kappa_1 < k$ in that case. If, however, Line 7 was executed then $M_k < \alpha_i$.

Let us now denote by β_i the value of α_j where j is the minimal index $j \in \{\ell - 1, \ldots, i + 1\}$ such that in iteration j Line 7 was executed, while if there is no such j then $\beta_i := 2^\ell$. Similarly, we denote by γ_i the value of α_j where j is the minimal index $j \in \{\ell - 1, \ldots, i + 1\}$ such that in iteration j Line 5 was executed, while if there is no such j then $\gamma_i := 0$.

We claim that

$$\gamma_i \leq M_k < \beta_i\,, \quad i = \ell - 1, \ldots, 1\,. \tag{15}$$

To prove the upper bound in Eq. (15), we observe that initially $\beta_i = 2^\ell$ which is greater than all values in U. On the other hand, if β_i equals α_j in an iteration where Line 7 was executed then, as argued above, $M_k < \alpha_j = \beta_i$. The proof of the lower bound in Eq. (15) is similar: it clearly holds for $\gamma_i = 0$ while if γ_i equals α_j in an iteration where Line 5 was executed then $\gamma_i = \alpha_j \leq M_k$, as argued before.

(For the sake of clarity, we provide below this proof an example that illustrates the values of α_i, β_i and γ_i.)

Now we proceed to derive an upper (resp. lower) bound on M_k in iterations where Line 5 (resp. Line 7) was executed. In case Line 5 was executed, we claim that $M_k \leq \alpha_i + 2^i - 1$ since $\alpha_i + 2^i = \beta_i$ and, as argued earlier, $M_k < \beta_i$. On the other hand, if Line 7 was executed then $M_k \geq \alpha_i - 2^{i-1}$ since $\alpha_i - 2^{i-1} = \gamma_i$ and, as argued earlier, $M_k \geq \gamma_i$.

In summary, for each $i = \ell - 1, \ldots, 1$, if Line 5 was executed in iteration i then

$$\alpha_i \leq M_k \leq \alpha_i + 2^i - 1\,,$$

while if Line 7 was executed then

$$\alpha_i - 2^{i-1} \leq M_k \leq \alpha_i - 1\,.$$

Hence, at the completion of the loop, we have

$$\alpha_1 \leq M_k \leq \alpha_1 + 1\,, \tag{16}$$

if Line 5 was executed in the last iteration, or

$$\alpha_1 - 1 \leq M_k \leq \alpha_1 - 1\,, \tag{17}$$

if Line 7 was executed in that iteration. Note that if Line 5 was executed in the last iteration $i = 1$ then the final value of α is $\alpha_1 + 1$, while if Line 7 was executed then the final value of α is $\alpha_1 - 1$. Therefore, in the first case, by Eq. (16), $M_k \in \{\alpha - 1, \alpha\}$, while in the second case, by Eq. (17), $M_k = \alpha$. Finally, the computation in Lines 9-10, which takes place only when in the last iteration Line 5 was executed, updates the value of α to the correct value of M_k. □

Example. Assume that $\ell = 6$ and that Line 5 was executed in iterations $i = 5, 3, 1$, while Line 7 was executed in iterations $i = 4, 2$. Then the values of α_i and β_i will be as shown in Table 1.

i	α_i	β_i	γ_i	Executed
5	$2^5 = 32$	$2^6 = 64$	0	Line 5
4	$32 + 2^4 = 48$	64	32	Line 7
3	$48 - 2^3 = 40$	48	32	Line 5
2	$40 + 2^2 = 44$	48	40	Line 7
1	$44 - 2^1 = 42$	44	40	Line 5

Table 1: Illustrating the values of α_i, β_i, γ_i in Algorithm 17.

Protocol 18 implements Algorithm 17 in a secure manner, namely, when the dataset is secret and is given to the parties by secret sharings of its entries. It computes a sharing of the k-ranked element while keeping all inputs and intermediate computed values secret.

The sharing of α is initialized in Line 1; specifically, each party P_i sets $[[\alpha]]_i \leftarrow 2^{\ell-1}$ (recall that if a is any publicly known value then $[[a]]$ is the sharing in which $[[a]]_i = a$ for all $i \in [n]$).

The main loop (Lines 2-7) emulates the main loop in Algorithm 17. For all $j \in [m]$ we compute a sharing of $x_j := 1_{u_j \leq \alpha - 1} = 1_{u_j < \alpha}$ (Lines 3-4). These bits' sharings are then added into a sharing of κ_1 (Line 5) which is then compared to k (Line 6). Finally, the computation in Line 7 emulates the if-then-else command in Lines 4-7 in Algorithm 17. The computation in Lines 8-10 at the end of the main loop issues a sharing of κ_2, as in Line 8 in Algorithm 17. Consequently, the computation in Lines 11-12 emulates the final computation in Lines 9-11 in Algorithm 17.

Privacy. Since all values remain secret-shared and the only exchange of messages takes place in the SecureMult and SecureCompare protocols that are perfectly secure, Protocol 18 is also perfectly secure under our working assumption of honest majority.

Complexity. Protocol 18 involves $(m + 1)\ell$ comparisons (Lemma 3), where $\ell = \log_2 M$, M being the upper bound on all values in U. In comparison, the cost of an algorithm that finds the k-th ranked element by first sorting U is $O(m \log_2 m)$ comparisons. However, an algorithm that sorts U issues the k-th ranked element for every $k \in [m]$, in contrast to Protocol 18 that must be executed for every query for a k-th ranked element. (Say, if it is needed to compute all quartiles, the protocol would have to be repeated 3 times.) That is the price of privacy in this context. Sorting is a possible solution in application scenarios where the ranking of U's elements may be disclosed (but not their actual values); if, however, such an information leakage is prohibited, Protocol 18 offers a suitable solution

Protocol 18: Computing the k-th ranked element in a shared dataset

Input: m – the size of the dataset; $\{[[u_j]]\}_{j\in[m]}$ – a (t,n)-sharing of the dataset; $M = 2^\ell$ – an upper bound on $\{u_j\}_{j\in[m]}$; $k \in [m]$

Parameter: t – the threshold

1 $[[\alpha]] \leftarrow [[2^{\ell-1}]]$
2 **forall** $i = \ell - 1, \ldots, 1$ **do**
3 **forall** $j \in [m]$ **do**
4 $[[x_j]] \leftarrow \text{SecureCompare}([[u_j]], [[\alpha]])$
5 $[[\kappa_1]] \leftarrow \sum_{j\in[m]}[[x_j]]$
6 $[[\lambda]] \leftarrow \text{SecureCompare}([[\kappa_1]], [[k]])$
7 $[[\alpha]] \leftarrow \text{SecureMult}([[\lambda]], [[\alpha]] + [[2^{i-1}]]) + \text{SecureMult}([[1]] - [[\lambda]], [[\alpha]] - [[2^{i-1}]])$
8 **forall** $j \in [m]$ **do**
9 $[[x_j]] \leftarrow \text{SecureCompare}([[u_j]], [[\alpha]] + [[1]])$
10 $[[\kappa_2]] \leftarrow \sum_{j\in[m]}[[x_j]]$
11 $[[\lambda]] \leftarrow \text{SecureCompare}([[k]], [[\kappa_2]])$
12 $[[M_k]] \leftarrow [[\alpha]] - [[\lambda]]$

Output: A (t,n)-sharing $[[M_k]]$ of the k-th ranked element

6 Improving the Runtime of the MPC Protocols

In this section we discuss several manners in which the MPC protocols over secret-shared values can be made more efficient, and we demonstrate the advantage of those improvements through experimentation.[2] All of our experiments were carried out on three prime-order fields, with $p_1 = 2^{31} - 1$, $p_2 = 2^{61} - 1$, and $p_3 = 2^{127} - 1$.

6.1 Offline Computations in the Secure Multiplication Protocol

Protocol 4 for secure multiplication includes computations that can be carried out offline, before the inputs are given. Specifically, the generation of two random sharings of the same random value may be executed offline. To estimate the offline runtime versus the online runtime we performed the following experiments.

First, we ran the full multiplication protocol (Protocol 4) 50 times and computed the average runtime; we then repeated the same experiment when the two random value sharings are already given, namely, under the assumption that Lines 1-8 in Protocol 4 were executed offline. We conducted the evaluations described above for $n \in \{5, 10, 15, 20, 25\}$, where n is the number of parties, and for $p \in \{p_1, p_2, p_3\}$. After running the above described experiment and measuring the average runtime for a single multiplication (the full computation versus the online part only) we repeated the same experiment also for batch computations of multiplications. Namely, instead of performing a single multiplication we computed a batch of 64 parallel multiplications and computed the average runtimes, for a single multiplication. Figure 1 reports all results.

By preparing the random sharings before the online phase of the protocol, we achieve an improvement factor close to n, both for the single multiplication and for the batch multiplication. To explain that improvement factor, we observe that the runtime of Protocol 4 is affected mainly by the synchronization points in the protocol, i.e., the places in the protocol where a party has to wait for inputs from all other parties. The synchronization points in Protocol 4 are between Line 4 and Line 7 (namely, all messages that are sent in the sharing procedure in Line 4 must be completed before Line 7 can be executed), between Line 5 and Line 8, and then in Line 11, where P_1 can reconstruct z only after receiving the shares of z from all other parties. Note that the first two synchronization points are n-fold, as each of the n parties must wait for inputs from all parties, while the latter one requires only P_1 to wait for inputs from all parties. As the former two synchronization points occur in the offline phase while only the latter one occurs in the online phase, that explains the improvement factor, that is close to n, when comparing the online runtime to the full runtime (both in the single multiplication mode and in the batch mode).

When comparing the batch runtimes to the single multiplication runtime, the batch computation is more efficient, as it allows parallelization. The improvement factor here is also $O(n)$.

[2] The authors plan to release the source code upon acceptance of the paper.

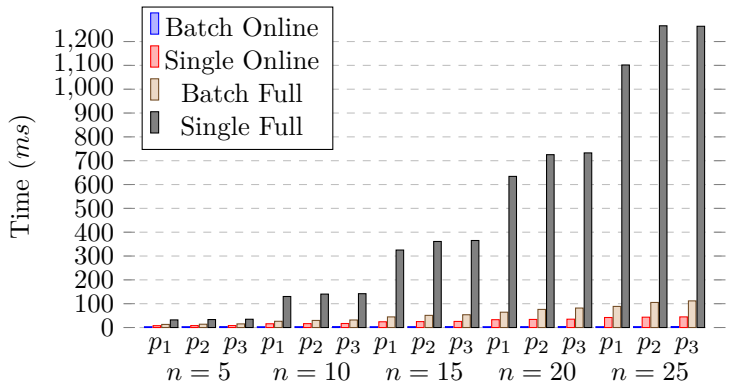

Figure 1: Average runtimes (milliseconds) of a single multiplication in four modes: full execution vs running the online part only (i.e., when all random sharings are already given), and a single multiplication vs a batch computation.

As for the runtime's dependence on the field size p, we see that it is minor, and noticeable mainly for the larger values of n, as Python treats all of the p values that we experimented with as BigInt.

6.2 Offline Computations in the Secure Comparison Protocol

The secure comparison protocol (Section 4) also performs computations that can be executed offline. Those include all random sharings generation that are carried out within SecureMult (Lines 1-8 in Protocol 4), as discussed in Section 6.1 above, and also the generation of random bits (Protocol 8), as discussed in Section 4.3. To compare the online runtime of the secure comparison to the full runtime we performed similar experimentation as that reported in Section 6.1 above. Figure 2 shows the runtime of the full SecureCompare (averaged over 50 repetitions) alongside the online runtime only, for $n \in \{5, 10, 15, 20, 25\}$, and $p \in \{p_1, p_2, p_3\}$. (Hereinafter, we focus on single computations, and do not experiment with batch computations.)

The online runtime is faster than the full runtime, with the improvement factor increasing with n. The average improvement factor (over all p's) increases from 3.1 for $n = 5$ to 4.2 for $n = 25$. As for the dependence on p, in this experiment, as opposed to the previous experiment with SecureMult, we see a noticeable impact of p, since the computation in SecureCompare is more involved.

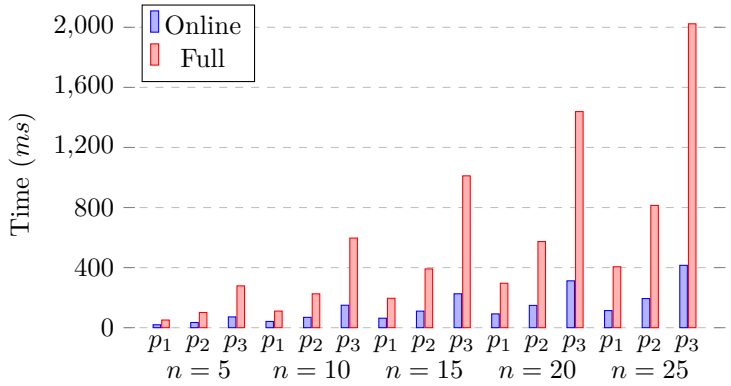

Figure 2: Average runtimes (milliseconds) of SecureCompare in two modes: full execution vs running the online part only.

6.3 Efficient Secure Bitwise Comparisons

As noted in Section 4.4, the secure computation of the bit $f = 1_{a<b}$ when a is publicly known while b is known through secret shares of each of its s bits, can be done either by Eq. (8), as done in [21], or

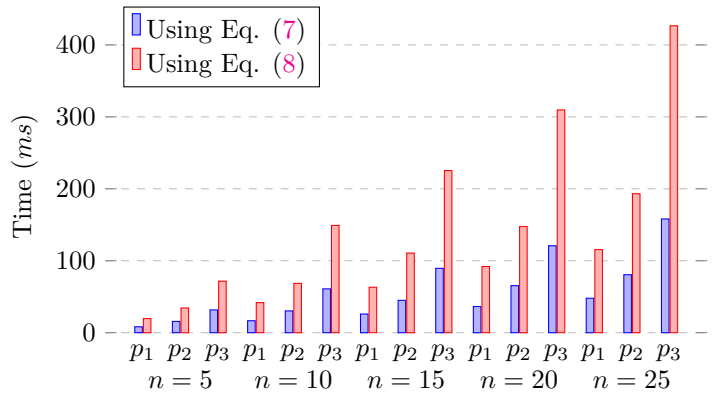

Figure 3: Average online runtimes (milliseconds) of SecureCompare using Eq. (7) vs Eq. (8).

by Eq. (7), as we suggest here. In our next experiment we evaluate the improvement offered by Eq. (7) over the original computation in Eq. (8) (for general inputs).

Figure 3 shows the average *online* runtime over 50 executions of two versions of Protocol 5. One that invokes Protocol 9 that uses the efficient computation in Eq. (7), and another that computes shares in f using Eq. (8). As can be seen, the improvement factor in all settings was over 2.

6.4 Secure Comparison of Small Values

As noted in Section 4.1, the secure comparison protocol for computing shares of $1_{u<v}$ becomes much simpler if the two compared values, u and v, are known to be smaller than $\frac{p}{2}$, since then the bits a and b are known to be 1 and the parties need only to compute shares in the bit c (see Eq. (2)). In our next experiment we measured the runtime of computing shares in $1_{u<v}$ when it is given that $u, v < \frac{p}{2}$ to the runtime of computing shares in that bit when that information is not given. Namely, we compare the runtime of Protocol 5 which executes only Line 3 for computing $[[c]]$ and then returns $[[w]] = 1 - [[c]]$ to the runtime of the full protocol. Figure 4 shows those runtimes for various values of n and p, averaged over 50 repetitions. It can be seen that the improvement factor is roughly 3 since, as discussed in Section 4.1, when the two inputs are known to be smaller than $\frac{p}{2}$ it is needed to compute secret shares only in the comparison bit c and not in all three bits a, b, c as needed in the general case.

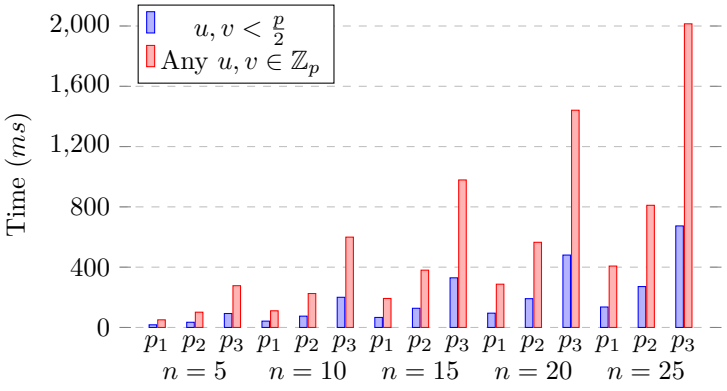

Figure 4: Average runtimes (milliseconds) for performing the full SecureCompare in two cases: when the two compared values are known to be smaller than $\frac{p}{2}$ and when such information is not given.

6.5 Overall Improvement

Here we combine all the improvements considered in the previous sections. Namely, we compare the runtime of the secure comparison protocol as presented in [21]—including all offline computations, for

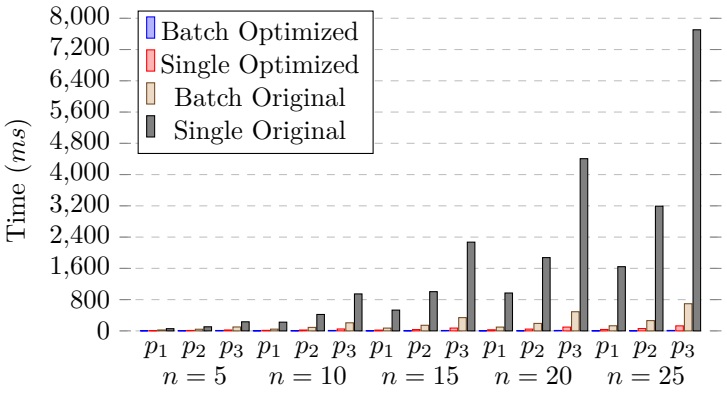

Figure 5: Runtimes (milliseconds) of SecureCompare, as single computation or as parallel batch, and for online and full modes.

general inputs, and using Eq. (8)—to a secure comparison protocol that relies on given offline random computations (Sections 6.1+6.2), for restricted inputs (Section 6.4), and with the improved computation in Eq. (7) (Section 6.3). The average runtimes over 50 repetitions are shown in Figure 5 for a single SecureCompare, when executed alone and when executed in the framework of a batch of 64 comparisons. The improvement factor grows linearly with n. Specifically, for $n = 5$ the average improvement factor (over all values of p) is approximately 9, both for the single mode and the batch mode, while for $n = 25$ it is approximately 54.

6.6 Parallel versus Sequential Computation of Prefix-Or Bits

Here we evaluate the cost in terms of runtime when performing the computation of the prefix OR bits $d_{s-1}, \ldots, d_0$ in the simplified sequential manner that we described in Section 4.4, versus the original suggestion of [21] to compute all those s bits in parallel. The goal of this experiment is to see for which input bit lengths s it is preferable to switch from the simplified computation to the more involved computation. In Figure 6 we see the average runtimes (over 50 repetitions) to compute all s prefix-or bits in the two methods, for various values of s that correspond to a Mersenne prime $p_s = 2^s - 1$, for $n = 10$.

For the online computation (that assumes that all required random sharings are already given) the simplified sequential manner is faster for smaller prime numbers, with diminishing returns for larger primes numbers. While for $s = 5$ we have an improvement factor of 2.5, it reduces to 1.2 for $s = 17$, and then, for $s = 31$ and $s = 61$ the parallel method becomes faster. Those findings are supported by the theoretical analysis of the two methods. The simplified sequential method has a depth complexity of $s - 1$ rounds with one SecureMult operation per round, while the parallel method [21] has a depth complexity of $3 \cdot \lceil \sqrt{s} \rceil + 2$ rounds with $O(\sqrt{s})$ SecureMult operations per round. By comparing the two round complexities we see that the parallel round complexity becomes smaller for $s \geq 15$, in agreement with our findings.

As for the full runtimes, the advantage of the parallel method begins to show only for $s = 61$, because of the additional runtimes to generate the required random sharings.

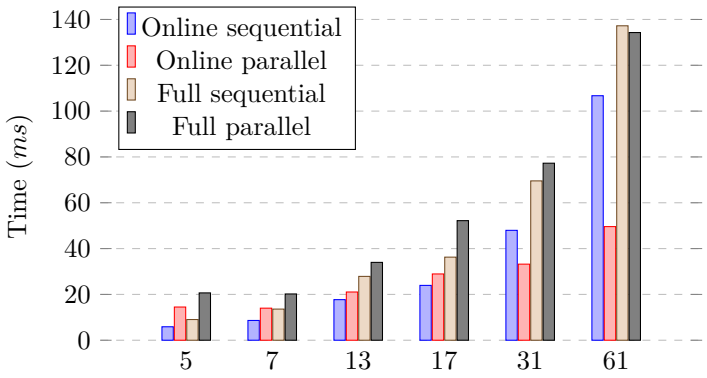

Figure 6: Average runtimes (milliseconds) for computing the s prefix-OR bits in the parallel and sequential methods, for different values of s, with $n = 10$.

7 Applications

In this section, we describe applications of MPC over secret shares in three domains: collaborative filtering (the main technique used in recommender systems), distributed optimization platforms, and e-voting infrastructures.

7.1 Collaborative filtering

Collaborative filtering (CF) is a central technique in recommender systems, where user preferences are inferred by identifying similarities in rating behavior across a population. CF techniques typically rely on collecting large amounts of user data—such as movie ratings or product purchases—and leveraging matrix factorization or neighborhood-based methods to predict unknown preferences. However, despite their effectiveness, CF algorithms pose serious privacy risks as users must share detailed, sensitive preference data with a central service or among multiple parties.

To address these concerns, secure multiparty computation (MPC) frameworks have been proposed as a way to implement CF algorithms without compromising user privacy. In such approaches, each user secret-shares their private ratings among a set of computation parties. These parties then emulate the CF algorithm—whether it involves matrix factorization, similarity computation, or other forms of optimization—entirely on secret shares, revealing only the final recommendations or model outputs. This strategy maintains the benefits of collective learning while ensuring that no single party gains access to any individual's raw data.

In their paper, Bickson et al. [3] tackle secure CF in a decentralized peer-to-peer environment. They focus on item-based neighborhood models, in which a user's unknown preference for some item is inferred by aggregating ratings from similar items rated by the same user. Preserving privacy in such models is challenging—sharing raw ratings leaks sensitive information, while centralized protocols may not scale or may violate trust assumptions. The authors remedy this by using secret-sharing–based MPC protocols, where each user splits their ratings into secret shares (using the Shamir threshold scheme) that are distributed to neighbors. Mathematical operations—such as dot products between rating vectors and similarity computations—are then performed collaboratively without revealing the raw data.

The implementation stands out for both its scale and realism. Instead of confining experiments to small-scale testbeds, the authors simulate peer-to-peer networks with millions of nodes and hundreds of millions of edges, applying their secure computation framework to perform recommendation tasks across this vast topology. They show that their model achieves accuracy on par with a non-private baseline while maintaining acceptable performance—demonstrating not only feasibility but also scalability, a critical requirement for real-world recommender systems. Furthermore, they compare the runtime of their secret-sharing-based MPC method to a homomorphic encryption-based alternative (specifically, the Paillier cryptosystem [22]), showing that the former is several orders of magnitude faster.

Tassa and Ben Horin [28] present a scalable, privacy-preserving framework for item-based collaborative filtering based on a distributed mediation model. Their protocol replaces the single semi-trusted mediator used in [24] with multiple non-colluding mediators, improving both privacy guarantees and computational efficiency. Using Shamir's secret sharing, vendors distribute user-item rating data in a

way that conceals the sensitive rating data. The mediators operate directly on these shares to compute item similarities and generate personalized recommendations through secure multiparty protocols that protect the privacy of both users and vendors. Compared to the single-mediator approach based on costly homomorphic encryption that was proposed in [24], their secret-sharing-based method achieves a runtime improvement of three to four orders of magnitude. It also offers stronger robustness: it tolerates collusion among any group of vendors and any minority subset of mediators—unlike the scheme in [24], which relies on the assumption that the single mediator remains honest. The framework also accommodates realistic, overlapping vendor-user-item distributions and includes mechanisms for secure data updates and efficient top-h recommendation queries, making it a practical and scalable solution for privacy-preserving collaborative filtering.

Collectively, these works demonstrate that MPC protocols based on secret sharing have evolved into a robust and adaptable framework for addressing the privacy and scalability challenges inherent in collaborative filtering.

7.2 Distributed Constraint Optimization

Distributed Constraint Optimization Problems (DCOPs) provide a foundational framework for modeling cooperative multi-agent systems where each agent controls a variable and aims to collectively maximize global utility (or minimize cost) subject to local constraints. Applications span diverse areas such as distributed sensor placement, smart grid control, supply chain coordination, and multi-robot planning. However, despite their distributed nature, classical DCOP algorithms often assume full trust between agents or require them to reveal sensitive information such as utility functions, constraint structures, or variable assignments during execution. This assumption limits their applicability in real-world environments where agents are autonomous, competitive, or privacy-constrained.

To address this, recent works have begun leveraging secret-sharing-based MPC to implement DCOP algorithms in a privacy-preserving manner. This approach allows the agents to jointly emulate standard DCOP solvers—such as branch-and-bound, inference-based message passing, or local search—without ever revealing their private inputs. Each agent secret-shares its data among a set of untrusted computation parties—either peer agents or external computing mediators—and the DCOP algorithm is reformulated as a sequence of arithmetic or logical operations collaboratively executed over these shares. The result is a secure emulation of the original algorithm, where intermediate states and agent-specific data remain confidential, but the collective optimization outcome is still correctly computed.

The studies summarized below exemplify this paradigm across three major classes of DCOP solvers: PC-SyncBB [27], which securely emulates the complete branch-and-bound algorithm [13]; MD-Max-Sum [15], which applies the technique to an inference-based, message-passing solver [11]; and P-DSA [12], which adapts a stochastic local-search method [31]. Each demonstrates how MPC over secret shares can be used to preserve algorithmic fidelity and strong privacy guarantees in a collaborative optimization setting.

The PC-SyncBB algorithm is a privacy-preserving, collusion-resistant extension of the classic SyncBB algorithm [13]. SyncBB is a complete branch-and-bound solver where statically ordered agents explore assignments while pruning the search using an upper bound on the optimal cost, inferred and updated during execution. PC-SyncBB preserves this structure but replaces all sensitive exchanges with secure MPC subprotocols that combine additive secret sharing and homomorphic encryption. When an agent assigns a value, a secure protocol updates the cost of the current partial assignment with neighbors. Finally, when a full assignment is reached, another MPC subprotocol verifies whether it improves the global optimum, by applying threshold secret sharing and then performing secure comparison, as described in Section 4 herein. Through this design, PC-SyncBB perfectly emulates SyncBB, and it provides topology, constraint, and decision privacy. PC-SyncBB is the first DCOP solver resilient to collusions of agents; this resilience holds under the standard assumption that a strict majority of agents are honest.

The MD-Max-Sum protocol adapts the well-known Max-Sum inference-based DCOP solver [11] into a privacy-preserving version using MPC over secret shares. Max-Sum relies on iterative message passing between agents to propagate cost functions and compute optimal assignments. In MD-Max-Sum, all utility messages are secret-shared across a set of computation parties (called mediators), who simulate the summation and maximization logic in a privacy-preserving way, using MPC primitives described herein. The original algorithm is emulated faithfully, while preserving topology, constraint, and assignment/decision privacy. The emulation allows the Max-Sum solver to operate in untrusted environments, enabling agents to collaboratively solve global optimization problems without leaking any private preferences or structure.

In contrast, P-DSA targets a different class of DCOP algorithms—incomplete solvers based on local search—specifically, the Distributed Stochastic Algorithm (DSA) [31]. DSA allows agents to iteratively propose and accept new assignments based on local utility improvements. P-DSA securely emulates this process using secret-sharing and MPC to evaluate whether a proposed change yields an improvement in the total global utility. As with the other protocols, individual utilities and constraint costs remain private at all times. P-DSA enables agents to perform privacy-preserving optimization in dynamic or scalable settings where complete exploration is computationally infeasible.

Together, these three frameworks illustrate how diverse paradigms of DCOP solving can be re-engineered to operate securely through secret-sharing and MPC techniques, where each framework preserves the logic of its original algorithm. By spanning this spectrum of solutions, the approach enhances the practical applicability of DCOPs in sensitive domains where distributed agents must safeguard proprietary information while still cooperating on global tasks. This highlights the flexibility and generality of the secure emulation paradigm.

7.3 Voting Systems

Other representative examples of secure computation via secret sharing are the systems of [8] and [26], which apply multiparty computation to securely implement score-based and order-based voting rules. In these frameworks, voters distribute secret shares of their ballots—represented as a vector (for score-based rules) or a matrix (for order-based rules)—across a set of talliers. The talliers first execute validation subprotocols to verify the legality of each ballot—ensuring compliance with the underlying voting rule—without disclosing any information on its contents. They then apply aggregation subprotocols across all valid ballots to compute the election outcome (e.g., determining the top-K winners, with or without ranking).

A central technique is to represent the tallying logic using secure arithmetic and comparison operations over secret shares. By leveraging efficient MPC primitives in this setting, the talliers jointly validate the legality of submitted ballots and compute election outcomes, while remaining oblivious to the ballots themselves. This design achieves perfect privacy: under the assumption of an honest majority, talliers learn nothing about individual ballots beyond their validity. This perfect privacy guarantee is strictly stronger than anonymity, where ballots remain visible but cannot be linked to voters. By ensuring stronger protection, perfect privacy enhances trust in the system and promotes both voter participation and truthful preference reporting.

The secure implementation of the Copeland voting rule in [26] was adopted by the Gentoo Linux project[3]—a community-driven project with a decentralized governance model—for the election of their council (interested readers are referred to [26, Section 7.2]).

These works demonstrate how well-established social choice mechanisms can be re-engineered as secure distributed protocols that retain correctness, scalability, and fault tolerance. By preserving the semantics of the original voting rules and simultaneously providing provable privacy and verifiability, they illustrate the power and generality of secret-sharing–based MPC as a tool for privacy-sensitive domains.

8 Conclusion

This work systematizes key techniques for secure multiparty computation (MPC) over secret-shared values, with a focus on efficient and composable protocols for secure arithmetic and comparison. By providing a unified and self-contained presentation of these protocols, along with a range of fundamental computations that build upon them, we aim to make the core techniques of secret-sharing-based MPC more accessible and practically applicable. Additionally, we introduce optimizations that reduce communication and online computation costs, contributing to the effort of narrowing the gap between theoretical frameworks and real-world deployment.

Looking ahead, several directions warrant further investigation. First, efficient support for real-valued computations, including floating-point operations and fixed-point arithmetic with precision guarantees, remains a critical challenge for privacy-preserving data science. Second, automated generation of optimized MPC circuits tailored to specific applications could improve both performance and usability. Third, integrating secret-sharing-based MPC with hybrid models (e.g., combining secret sharing with

[3] https://www.gentoo.org/

homomorphic encryption or garbled circuits) may offer better tradeoffs in mixed trust or adversarial settings. Finally, putting these protocols to use in real-world, large-scale systems—particularly in healthcare, finance, and federated learning—continues to present valuable opportunities that have yet to be fully explored.

We hope this work serves as a valuable reference and catalyst for advancing secure computation over shared secrets in both research and practice.

References

[1] Emmanuel A. Abbe, Amir E. Khandani, and Andrew W. Lo. Privacy-Preserving Methods for Sharing Financial Risk Exposures. *American Economic Review*, 102:65–70, 2012.

[2] Sunpreet Arora, Andrew Beams, Panagiotis Chatzigiannis, Sebastian Meiser, Karan Patel, Srinivasan Raghuraman, Peter Rindal, Harshal Shah, Yizhen Wang, Yuhang Wu, Hao Yang, and Mahdi Zamani. Privacy-Preserving Financial Anomaly Detection via Federated Learning & Multi-Party Computation. In *Annual Computer Security Applications Conference Workshops (ACSAC workshops)*, pages 270–279, December 2024.

[3] Danny Bickson, Danny Dolev, Genia Bezman, and Benny Pinkas. Peer-to-peer secure multi-party numerical computation. In *Peer-to-Peer Computing*, pages 257–266, 2008.

[4] Kallista A. Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *CCS*, pages 1175–1191, 2017.

[5] Ashok K. Chandra, Steven Fortune, and Richard J. Lipton. Lower bounds for constant depth circuits for prefix problems. In *ICALP*, pages 109–117, 1983.

[6] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *CRYPTO*, pages 34–64, 2018.

[7] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, pages 572–590, 2007.

[8] Lihi Dery, Tamir Tassa, and Avishay Yanai. Fear not, vote truthfully: Secure multiparty computation of score based rules. *Expert Syst. Appl.*, 168:114434, 2021.

[9] Norman R. Draper and Harry Smith. *Applied Regression Analysis*. Wiley, 3rd edition, 1998.

[10] Michael D. Ekstrand, John Riedl, and Joseph A. Konstan. Collaborative filtering recommender systems. *Foundations and Trends in Human-Computer Interaction*, 4:175–243, 2011.

[11] Alessandro Farinelli, Alex Rogers, Adrian Petcu, and Nicholas R. Jennings. Decentralised coordination of low-power embedded devices using the max-sum algorithm. In *AAMAS*, pages 639–646, 2008.

[12] Shmuel Goldklang, Tal Grinshpoun, and Tamir Tassa. Privacy preserving solution of dcops by local search. In *IJCAI*, 2025.

[13] Katsutoshi Hirayama and Makoto Yokoo. Distributed partial constraint satisfaction problem. In *Principles and Practice of Constraint Programming - CP*, pages 222–236, 1997.

[14] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, 1st edition, 1969.

[15] Pablo Kogan, Tamir Tassa, and Tal Grinshpoun. Privacy preserving solution of dcops by mediation. *Artif. Intell.*, 319:103916, 2023.

[16] Zhaohao Lin, Weike Pan, and Zhong Ming. FR-FMSS: federated recommendation via fake marks and secret sharing. In *RecSys*, pages 668–673, 2021.

[17] James B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, 1967.

[18] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *CCS*, pages 19–38, 2017.

[19] Yoshiki Nakagawa, Satsuya Ohata, and Kana Shimizu. Efficient privacy-preserving variable-length substring match for genome sequence. *Algorithms Mol. Biol.*, 17:9, 2022.

[20] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving ridge regression on hundreds of millions of records. In *IEEE Symposium on Security and Privacy (S&P)*, pages 334–348, 2013.

[21] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *PKC*, pages 343–360, 2007.

[22] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Eurocrypt*, pages 223–238, 1999.

[23] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[24] Erez Shmueli and Tamir Tassa. Mediated secure multi-party protocols for collaborative filtering. *ACM Trans. Intell. Syst. Technol.*, 11:15:1–15:25, 2020.

[25] Haris Smajlović, Ariya Shajii, Bonnie Berger, Hyunghoon Cho, and Ibrahim Numanagić. Sequre: a high-performance framework for secure multiparty computation enables biomedical data sharing. *Genome Biology*, 24:5, 2023.

[26] Tamir Tassa, Lihi Dery, and Arthur Zamarin. Secure order based voting using distributed tallying. *J. Inf. Secur. Appl.*, 93:104141, 2025.

[27] Tamir Tassa, Tal Grinshpoun, and Avishay Yanai. Pc-syncbb: A privacy preserving collusion secure DCOP algorithm. *Artif. Intell.*, 297:103501, 2021.

[28] Tamir Tassa and Alon Ben Horin. Privacy-preserving collaborative filtering by distributed mediation. *ACM Trans. Intell. Syst. Technol.*, 13:102:1–102:26, 2022.

[29] Marino Tejedor-Romero, David Orden, Ivan Marsa-Maestre, Javier Junquera-Sanchez, and Jose Manuel Gimenez-Guzman. Distributed Remote E-Voting System Based on Shamir's Secret Sharing Scheme. *Electronics*, 10:3075, 2021.

[30] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *Proc. Priv. Enhancing Technol.*, 2019:26–49, 2019.

[31] Weixiong Zhang, Guandong Wang, Zhao Xing, and Lars Wittenburg. Distributed stochastic search and distributed breakout: properties, comparison and applications to constraint optimization problems in sensor networks. *Artif. Intell.*, 161:55–87, 2005.