

SPRUCE: Scalable Multiparty Private Set Union in Constant Rounds

Zhengwei Tong
Duke University

Saba Eskandarian
UNC Chapel Hill

Jonathan Katz
Google

Kartik Nayak
Duke University

Abstract

Multiparty private set union (MPSU) enables a group of mutually distrusting parties, each holding a private set, to learn the union of their sets without revealing additional information. MPSU is a fundamental primitive with applications to privacy-preserving tasks such as cyber-risk assessment and private record matching.

State-of-the-art MPSU protocols require parties to sequentially shuffle encrypted values in order to hide which party contributed which inputs to the final result. We introduce a new paradigm for semi-honest MPSU that eliminates the need for such shuffling. This allows the protocol to run in constant rounds regardless of how many parties are involved. The resulting protocol also benefits from improved parallelism, an information-theoretic online phase, and native support for unbalanced input sizes, a situation most prior works must handle by padding inputs to hide individual input sizes.

We implement and evaluate our protocol across a range of party counts, input sizes, and network conditions. In the unbalanced setting, where parties hold unequal-sized inputs, our protocol significantly outperforms the state-of-the-art schemes PULSE (CCS '25) and DZBC25 (USENIX Security '25). In the balanced setting, it achieves an order-of-magnitude improvement over PULSE in most settings and outperforms DZBC25 in WAN deployments with 100ms latency.

1 Introduction

Private Set Union is the problem where two or more parties compute the union of their private input sets while revealing no additional information beyond the union of the sets. Motivated by applications to cyber risk assessment [8, 23], private record matching [7], association rule mining [26], and joint network/graph analytics [6], private set union (PSU) is now a fundamental object of inquiry [16, 28, 35]. In the two-party setting, recent protocols have made PSU increasingly practical [20, 24, 25, 29, 33, 34, 37, 38]. While not as performant as two-party schemes, a number of recent works

also target the more general multiparty PSU (MPSU) setting [15, 17–19, 22, 31, 32, 36].

The extension from two-party PSU to MPSU is non-trivial. In a two-party protocol, it is often enough to privately test whether one party's elements already appear in the other party's set and then reveal only the missing elements. The fastest techniques for this rely on reverse private membership tests or inverted bloom lookup tables [33, 38]. In the multiparty setting, however, such a strategy can leak intermediate information about subset unions, intersections, or which party contributed a particular element. A secure MPSU protocol therefore requires more than private pairwise membership testing. After the union is revealed, parties should not learn which other participant contributed each union element, except where this is already implied by their own inputs and the final output.

Recent MPSU protocols [15, 17, 18, 29, 38] address this challenge by adapting the pairwise membership-testing structure of two-party PSU. At a high level, these protocols use shuffling protocols of various kinds to have parties add their inputs to an encrypted or secret-shared output set, using private membership tests to determine which of their inputs are new, and then mix and rerandomize the resulting set [11] to hide which elements are new. Once each party has done this in a round-robin fashion, the resulting set is revealed, and nobody can tell who contributed which element. Works following this paradigm have substantially improved the practicality of MPSU, and recent designs such as PULSE [17] further reduce the round complexity of several protocol components. Nevertheless, the overall design still relies on membership-testing and having each party sequentially perform a shuffle.

Our Contribution. This work explores a different design direction for semi-honest MPSU. Our approach, summarized in Section 2, eliminates the need for parties to pass sets around, reducing the round complexity of MPSU from $O(N)$ in the number of parties to $O(1)$. Our protocol has an online phase that is purely information-theoretic, and an offline phase that can be instantiated with any generic protocol for producing Beaver multiplication triples [1], both of which benefit sub-

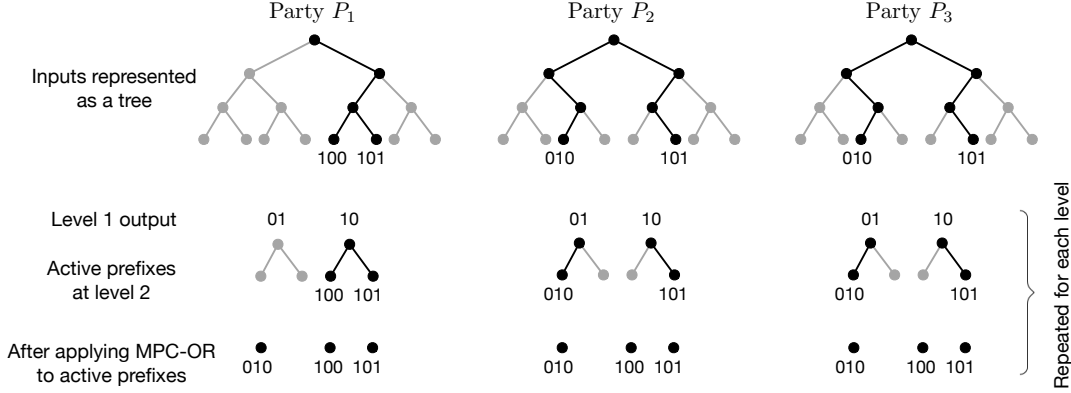


Figure 1: **Represents an example computation with 3 parties for 3-bit inputs.** Each party begins with 2 inputs. After the MPC computation at level 1, the outputs are {01, 10} – the final outputs are captured by this set of prefixes. The MPC computation at level 2 is thus performed on the set {010, 011, 100, 101}.

stantially from parallelization. Unlike prior works, we natively do not leak the size of individual parties’ inputs, only the final output size, which allows our protocol to achieve excellent performance in settings with unbalanced input sizes, where other schemes would need to pad inputs until they all match in order to hide this information. Along the way, we introduce a handful of optimizations that improve over the performance of using standard generic MPC protocols directly for computing bitwise ORs of inputs from several parties in constant rounds. This may be of independent interest.

Our approach relies on the parties collaboratively searching the space of all possible elements in the union by iteratively building up a union of prefixes, e.g., they first compute the union of all their prefixes of length 1, then length 2, and so on. At a high-level, this bears a similarity to techniques used for computation of private heavy hitters [5], albeit in a very different setting and with different security considerations. Crucially, we show how to convert this iterative process, where each step depends on the output of the previous one, into a constant-round protocol. The key innovation here is a new application of Bloom filters [4] that allows parties to non-interactively and speculatively compute the output set before conducting another constant round pruning-process that removes false positives from the output. Our use of Bloom filters is entirely different from prior two party PSU schemes that uses invertible bloom lookup tables (IBLTs) as part of a pairwise membership test, e.g., [33].

We call the resulting scheme **SPRUCE**: Scalable multi-party Private set Union with Constant-round Evaluation. We implement our scheme and compare its performance to PULSE [17] and DZBC25 [15], the most performant prior MPSU schemes, over a broad range of parameters. We find that SPRUCE is 2.67–28.17 $\times$ faster than PULSE and achieves speedups over DZBC25 under high latency settings; notably, under WAN setting of 100ms latency, it outperforms DZBC25 in every measured configuration by 1.40–7.86 $\times$.

Under unbalanced inputs, SPRUCE outperforms PULSE and DZBC25; the performance advantage may further increase with the degree of input imbalance. Details of our implementation and evaluation appear in Section 6.

2 Technical Overview

This section conveys the key ideas behind our protocol.

Prior work. Existing multiparty PSU protocols largely follow a common paradigm: parties iteratively determine which of their elements do not appear in another party’s set (often referred to as reconciliation), incorporate those elements into a growing representation of the union in a privacy-preserving manner, replace duplicate elements with dummy values, and finally apply a shuffle so that the output no longer reveals which party contributed which element [14, 15, 17, 18, 31, 32].

Considerable effort has been devoted to reducing the cost of the reconciliation and deduplication phases. Many of these operations can be implemented in a constant number of rounds [17]. However, the round complexity of the final shuffle remains inherently linear in the number of parties, as each party needs to permute the data to ensure that the permutation remains hidden even if all-but-one of the parties are corrupted, and therefore incurs $\Theta(N)$ work and communication.

High-level idea. Our protocol takes a fundamentally different viewpoint and presents a fresh way to approach the problem. Rather than asking which elements one party possesses that another party does not, we instead ask which values in the universe of all possible inputs should appear in the output. Suppose inputs are κ -bit values. Then there is a trivial protocol: each party encodes their set as a 2^κ -bit string (setting bit x to 1 if party holds x , and 0 otherwise) and the parties compute the union by computing the OR of their strings. Then there are 2^κ possible values that could appear in the union. The problem with this approach is that the universe contains 2^κ

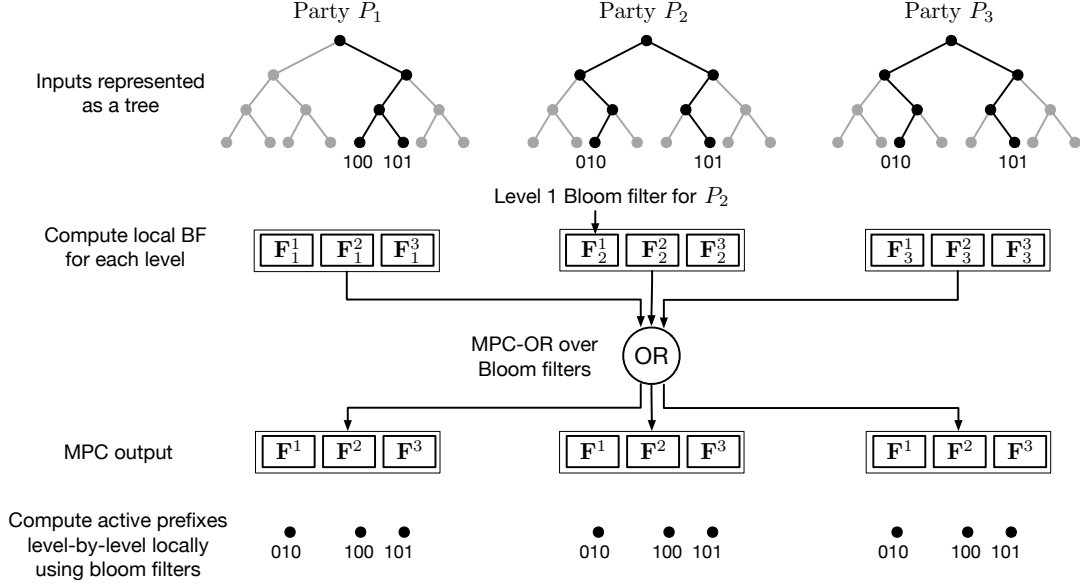


Figure 2: **Represents how Bloom filters are used to obtain $O(1)$ rounds.** The level-by-level active prefix computation is similar to that in Figure 1 but it is performed locally and it utilizes the computed Bloom filters to test membership of each of the active prefixes. The figure does not represent the existence of false positives, and their elimination using an MPC at the leaf level.

possible values, making an explicit representation infeasible.

Tree-based representation of the universe. To avoid explicitly representing the entire universe, we conceptually organize the universe as a complete binary tree of depth κ .¹ Each root-to-leaf path corresponds to a unique κ -bit value. A party holding value x conceptually places a 1 at the leaf corresponding to x and at every ancestor of that leaf. All other nodes are assigned 0. Importantly, the protocol never materializes the full tree; the tree serves only as a conceptual representation.

This representation enables the protocol to reason about prefixes of values. If a node corresponding to prefix p evaluates to 1 after aggregating all parties' inputs, then at least one output value begins with prefix p . Conversely, if the node evaluates to 0, then no output value can appear in the subtree rooted at p .

The parties exploit this property by exploring the tree level by level. The root is processed first. At each subsequent level, only children of prefixes that evaluated to 1 at the previous level are explored. We refer to such prefixes as *active prefixes*. Since the MPC outputs are revealed to all parties, every party learns the same set of active prefixes and therefore explores the same portion of the conceptual tree. Figure 1 presents a visual representation with an example.

The key observation here is that, from a privacy standpoint, the union of prefixes can be computed from the union of the sets, and thus, revealing those intermediate prefixes does not leak additional information. From a performance standpoint,

the protocol explores only prefixes that may lead to output elements. The amount of work performed is therefore proportional to the number of active prefixes rather than to the size of the universe.

The round-complexity bottleneck. While the tree-based approach eliminates the dependence on the universe size, it introduces a new challenge. To determine which prefixes should be explored at level $\ell + 1$, the parties must first know the outputs computed at level ℓ . Consequently, the computation proceeds sequentially from the root to the leaves. Since the tree has depth κ , this approach requires $\Theta(\kappa)$ rounds of interaction. Thus, although the protocol avoids exploring an exponential space, it does so at the cost of introducing a linear dependence on the input bit-length.

Compressing levels using Bloom filters. Our key observation is that the information contained in a level of the tree can be represented compactly using a Bloom filter [4]. For a given level, each party inserts all prefixes marked with a 1 into a Bloom filter. The Bloom filter parameters are fixed and publicly known. Importantly, the size of the filter depends on the number of inserted prefixes rather than on the total number of possible prefixes at that level.

The protocol then proceeds as follows. For every level of the conceptual tree, each party locally constructs a Bloom filter containing the prefixes marked with a 1 at that level. The parties then execute MPC protocols that compute the bitwise OR of the Bloom filters corresponding to each level. Crucially, these MPC computations are independent across levels and can therefore be performed in parallel. At the end of this phase, every party obtains the OR'ed Bloom filter cor-

¹The intuition can be easily extended to any degree tree; in fact, in our implementation, we consider a prefix extension of 4, yielding a $2^4 = 16$ -ary tree.

responding to every level of the tree. A visual representation of this computation is presented in Figure 2.

Local computation of active prefixes. Once the Bloom filter phase completes, no further interaction is required to determine which portions of the tree should be explored. Starting from the root, parties can locally reconstruct the active-prefix structure. Suppose prefix p is active at level ℓ . To determine whether child $p||0$ should be explored, the party performs a Bloom filter membership query against the filter corresponding to level $\ell + 1$. The same procedure is applied to child $p||1$. Any child reported as present becomes active and is explored further. Repeating this process level by level constructs a set of candidate active prefixes all the way to the leaves. Observe that this computation is entirely local and requires no interaction between parties. The protocol has effectively removed the sequential MPC dependency between levels.

Eliminating Bloom filter false positives. The above procedure identifies a superset of the true output values. The source of inaccuracy is the possibility of Bloom filter false positives. A membership query may indicate that a prefix is present even though no party inserted that prefix. Such errors can propagate through layers of the tree and may cause additional candidate leaves to be explored.

To restore exact correctness, we perform one final MPC computation at the leaf level. After computing the candidate leaves for the last level (based on the Bloom filter outputs of the penultimate level), the parties conduct an MPC to filter the remaining active prefixes. For each potential active leaf, they input a 1 or 0 depending on whether they have the corresponding element in their set. The parties compute an MPC OR operation over these values. Since this computation uses the original inputs rather than Bloom filter approximations, any candidate value arising solely from a false positive evaluates to 0 and is removed. Consequently, the final output contains exactly the elements belonging to the union.

Efficiency and privacy intuition. The resulting protocol achieves constant round complexity. All Bloom filter MPC computations are performed in parallel, and only a single additional MPC computation is required to eliminate false positives. From a privacy perspective, the union of the prefixes of any length i can be computed from the union of the sets. Moreover, based on the set of all these elements known to a simulator, the Bloom filter of that set can be computed. Thus, revealing the OR’ed Bloom filters for each level does not reveal anything more to the parties that they could not compute based on the output.

Instantiating the MPC. It remains to show how we instantiate the MPC OR operation. One natural way to do this is to use a generic MPC that operates over secret shares of bits and implements an OR gate. There are generic MPC protocols such as BMR [2] that run in $O(1)$ rounds but they are inefficient. Other approaches such as the GMW protocol [21] have a round complexity that is linear in the depth of the circuit,

and thus will incur $O(\log N)$ rounds when computing the OR of all N parties’ inputs using a boolean circuit.

In order to compute a multi-input OR in a constant round, we instead use an MPC that operates over a prime-order field of appropriate size, and use the addition operation instead of logical OR, where any non-zero value is interpreted as a 1. Here, the field size only needs to be larger than the number of participating parties in the protocol. This is because each party’s contribution is always either 0 or 1, so for N parties, the sum is at most N . As long as the field size is greater than N , a nonzero sum will not wrap around to 0 in the field.

Unfortunately, this simple change would compromise the security of the protocol. A bitwise OR outputs 1 regardless of how many inputs are 1, but a sum will reveal how many parties have a particular prefix in their set or a particular bit set in their Bloom filter. We avoid this issue by randomizing any sum before it is revealed to the parties. Whenever shared sums are being revealed, the parties multiply by a secret-shared random value before revealing the sum.

MPC optimizations. The MPC approach described here can be instantiated generically, but we introduce three optimizations that significantly reduce communication and improve performance compared to a generic instantiation. See Section 4 for details on each.

- *Discount multiplications.* The process of multiplying shares by a random value can be implemented using standard multiplication with preprocessed Beaver triples [1]. However, we observe that since we are multiplying by a random value, not some specific secret shared value, we do not need the full power of Beaver multiplication. Instead, we use a simplified multiplication protocol that reduces online computation and communication for each multiplication by roughly a factor of two.
- *Perfect correctness.* Multiplying sums by a truly random value creates a risk of selecting zero as the random value and breaking the correctness of the protocol. Naively, this can be prevented by selecting a large enough field to make the probability of this bad event negligible. However, inflating the field size would significantly increase communication and computation costs. Instead, we ensure that all random values chosen for multiplication are non-zero. Since this can be completed in a preprocessing step, it requires no additional online computation and removes the correctness issue entirely.
- *Free sharing.* When using a generic MPC protocol, we would start by having each party secret share its inputs with all other parties. This adds a communication round and a great deal of communication cost. We observe that, in our protocol, parties immediately sum and randomize the shares they receive. As such, instead of having an initial sharing step, we have the parties behave as if their inputs are already secret shares of the sum. Since all shares are randomized

Ideal functionality $\mathcal{F}_{\text{MPSU}}$

Upon receiving a set $S_i \subseteq \{0, 1\}^k$ with $|S_i| \leq n$ from every party P_i , compute

$$U := \bigcup_{i=1}^N S_i$$

and send U to every party.

Figure 3: Ideal functionality for multi-party private set union.

before being revealed, this saves communication without affecting security.

3 Preliminaries

Notation. Let $p > N$ be prime and let $\mathbb{F}_p^* := \mathbb{F}_p \setminus \{0\}$. For a vector $\mathbf{x} \in (\mathbb{F}_p)^m$, an N -out-of- N additive sharing $[\mathbf{x}]$ consists of shares $[\mathbf{x}]_1, \dots, [\mathbf{x}]_N \in (\mathbb{F}_p)^m$ satisfying $\mathbf{x} = \sum_{i=1}^N [\mathbf{x}]_i$. We write $x \xleftarrow{\$} \mathbb{F}_p$ to denote that x is sampled uniformly at random from the field $\mathbb{F}_p$. For two vectors $\mathbf{x}, \mathbf{y}$ of equal dimension, we use $\mathbf{x} \odot \mathbf{y}$ to denote their entry-wise product.

Private set union. Private set union is a functionality where each party P_i to the protocol holds a set $S_i \subseteq \{0, 1\}^k$. The intended output $U := \bigcup_{i=1}^N S_i$ is delivered to every party. Figure 3 formalizes this functionality. We have N as the number of parties and n as the upper bound for input size. Our formalization of the MPSU functionality differs slightly from prior works. Whereas state of the art MPSU protocols [15, 17] require parties' inputs to be of exactly size n , our functionality allows parties to send inputs S_i of size $\leq n$. This subtle change highlights the fact that, unlike these prior works, our protocol does not reveal the size of users' inputs. The practical benefit of the change is that using prior works with unbalanced set sizes requires padding inputs to the same size, whereas our scheme does not.

We assume throughout that at least one party is corrupted. This is not a crucial assumption, but dropping it would require separately modeling an eavesdropping adversary, to which the functionality would separately leak the size of the protocol output.

Semi-honest adversaries. We consider a static semi-honest (honest-but-curious) adversary that may corrupt and coordinate any strict subset of the parties. Corrupted parties faithfully follow the prescribed protocol but may attempt to infer additional information from their internal states and the messages they observe. We require that the adversary learn no information beyond what is implied by the inputs of the corrupted parties and the prescribed protocol output. We formalize this guarantee using the standard simulation-based definition of security [9, 30]: the view of the adversary in a real protocol execution must be simulatable given only the information available to it in the corresponding ideal execution.

Bloom filters. A Bloom filter [4] is a probabilistic data structure for set membership. A Bloom filter with parameters $\text{bf} = (m, k, H_1, \dots, H_k)$ consists of an m -bit array and k hash functions $H_j : X \rightarrow [m]$.

For a set $T \subseteq X$, we denote its Bloom-filter insertion

$$\text{BFInsert}(T)[t] := \mathbf{1}[\exists x \in T, \exists j \in [k] : H_j(x) = t].$$

For a Bloom filter $\mathbf{F} \in \{0, 1\}^m$ and $x \in X$, we define the membership test

$$\text{BFTest}(\mathbf{F}, x) := \bigwedge_{j=1}^k \mathbf{F}[H_j(x)].$$

Bloom filters have no false negatives, but may produce false positives. For n inserted elements, the standard false-positive estimate [4] is

$$\varepsilon = \left(1 - e^{-kn/m}\right)^k.$$

and the usual sizing rule to achieve a target false-positive rate ε is

$$m = \left\lceil \frac{n \ln \varepsilon^{-1}}{(\ln 2)^2} \right\rceil, \quad k = \left\lceil \frac{m}{n} \ln 2 \right\rceil.$$

4 Construction and Security

We now formalize the complete MPSU construction described in Section 2. There are $N \geq 2$ parties $P_1, \dots, P_N$. Without loss of generality we have P_1 serve as a *designated party* who will serve as the center of a star communication topology and perform additional computation in the protocol. Note that the designated party is not trusted and may be corrupted, just like any other party. We assume point-to-point channels between P_1 and each other party. In particular, values sent by P_1 to all parties are delivered over the corresponding point-to-point channels rather than through a public broadcast channel.

Our construction is modular. In Section 4.1 we show how to compute bitwise-OR. Section 4.2 then constructs an MPSU protocol that relies on any protocol securely computing bitwise-OR as a subroutine. Section 5.1 proves correctness and privacy, and Section 5.2 analyzes the resulting efficiency.

Preprocessing. Our protocol relies on Beaver triples [1] to multiply secret-shared values. Beaver multiplication allows for most of the work of multiplication to be moved to a preprocessing phase.

The preprocessing used in our bitwise OR protocol largely follows standard Beaver triple generation. A standard Beaver triple consists of N -out-of- N secret shares of vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$, where each entry of $\mathbf{a}$ and $\mathbf{b}$ is sampled uniformly from $\mathbb{F}_p$ and $\mathbf{c} = \mathbf{a} \odot \mathbf{b}$. We rely on a slightly non-standard variant in which the second random vector is required to be nonzero. To distinguish it from the standard Beaver triple notation, we denote this vector by $\mathbf{r}$ instead of $\mathbf{b}$. Each entry of $\mathbf{r}$ is sampled uniformly from $\mathbb{F}_p^*$ rather than $\mathbb{F}_p$.

Ideal functionality $\mathcal{F}_{\text{Beaver},p}$

Upon receiving (preprocess, m) from every party, sample

$$\mathbf{a} \xleftarrow{\$} (\mathbb{F}_p)^m, \quad \mathbf{r} \xleftarrow{\$} (\mathbb{F}_p^*)^m,$$

and compute $\mathbf{c} := \mathbf{a} \odot \mathbf{r}$. Independently sample an N -out-of- N additive sharing of each of $\mathbf{a}$, $\mathbf{r}$, and $\mathbf{c}$, and send $([\mathbf{a}]_i, [\mathbf{r}]_i, [\mathbf{c}]_i)$ to party P_i .

Figure 4: Beaver-style multiplication preprocessing, slightly modified for our protocol.

Such a Beaver triple can be obtained from a standard one by performing a secure zero test on $\mathbf{r}$ and discarding and regenerating any entry of $\mathbf{r}$ that is zero. Concretely, if the parties want m triples of the form required for our protocol, they sample m triples $\mathbf{a}, \mathbf{r}, \mathbf{c}$, as well as an additional vector $\mathbf{r}'$. They then compute and reveal the vector $\mathbf{r}'' \leftarrow \mathbf{r} \odot \mathbf{r}'$, using supplemental beaver triples $\mathbf{a}', \mathbf{b}', \mathbf{c}'$ to complete the multiplications. Thus the test requires generating 2 standard Beaver triples per modified one. For each zero entry of $\mathbf{r}''$, the parties discard that entry and repeat the sampling process. The optimizations described in Section 4.1 can also be applied to speed up this multiplication.

This simple process results in success with probability $(1 - 1/p)^2$ per entry since an entry r_i is discarded only if one of r_i or r'_i are zero. Otherwise the revealed value is the product of two random elements of $\mathbb{F}_p^*$, and the entry of r_i can be used so long as shares of r'_i are discarded and never revealed. To save rounds, parties can compute extra triples to replace any discarded ones up front in the first round.

Our formal analysis models preprocessing via the ideal functionality $\mathcal{F}_{\text{Beaver},p}$, shown in Figure 4.

4.1 Secure Computation of OR

Our MPSU construction requires the parties to compute the coordinate-wise OR of certain values. We give the corresponding ideal functionality in Figure 5.

Ideal functionality $\mathcal{F}_{\text{OR}}$ (parameterized by m)

Upon receiving a vector $\mathbf{w}_i \in \{0, 1\}^m$ from every party P_i , compute the coordinate-wise OR $\mathbf{y} := \bigvee_{i=1}^N \mathbf{w}_i$ and send $\mathbf{y}$ to every party.

Figure 5: Ideal functionality for coordinate-wise OR.

As mentioned in the technical overview, we can compute this functionality in constant rounds via preprocessed Beaver multiplication triples [1] used for multiplication over a final field $\mathbb{F}$. The key idea is to simply interpret users 0/1 inputs as elements of $\mathbb{F}$ and secret share them. The parties then sum their secret shares. After revealing the result, any nonzero

Algorithm 1 Π_{OR}

Input: Parties $P_1, \dots, P_N$ each have input $\mathbf{w}_i \in \{0, 1\}^m$.

Output: Every party learns $\mathbf{y} = \bigvee_{i=1}^N \mathbf{w}_i$

- 1: The parties invoke $\mathcal{F}_{\text{Beaver},p}$ with input m , where $p > N$. In return, each P_i receives $[\mathbf{a}]_i, [\mathbf{r}]_i, [\mathbf{c}]_i$
- 2: Each P_i computes $\mathbf{e}_i \leftarrow \mathbf{w}_i - [\mathbf{a}]_i$ and sends $\mathbf{e}_i$ to P_1
- 3: P_1 computes $\mathbf{e} \leftarrow \sum_{i=1}^N \mathbf{e}_i$ and sends $\mathbf{e}$ to all parties
- 4: Each P_i computes $\mathbf{z}_i \leftarrow \mathbf{e} \odot [\mathbf{r}]_i + [\mathbf{c}]_i$ and sends $\mathbf{z}_i$ to P_1
- 5: P_1 reconstructs $\mathbf{z} \leftarrow \sum_{i=1}^N \mathbf{z}_i$
- 6: **for** $j = 1$ to m **do**
- 7: P_1 sets $\mathbf{y}[j] = 1$ if $\mathbf{z}[j] \neq 0$, and $\mathbf{y}[j] = 0$ otherwise
- 8: P_1 sends $\mathbf{y}$ to each party, which outputs it

value in $\mathbb{F}$ is interpreted as the bit 1. Since the actual sum reveals the number of non-zero inputs in addition to simply the output bit, we multiply the sum by a random, non-zero value r before revealing it. Picking r to be non-zero ensures that the product does not turn a non-zero sum into a zero output, ensuring perfect correctness.

Optimizing Beaver multiplication. This first attempt protocol satisfies the required functionality, but it incurs unnecessary costs. Recall $x := \sum_{i=1}^N w_i \in \mathbb{F}_p^m$ is the coordinate-wise aggregate that must be randomized before it is revealed. A straightforward realization would first generate an independent secret-shared mask $[r]$, where $r \in (\mathbb{F}_p^*)^m$, and then multiply $[x]$ and $[r]$ using a standard Beaver triple $([a], [b], [c])$, where $c = a \odot b$. The parties would open $e := x - a$ and $d := r - b$ and compute $[x \odot r] = [c] + e \odot [b] + d \odot [a] + e \odot d$. Although e and d can be opened in the same communication round, both length- m vectors must be transmitted.

Our protocol exploits the fact that r is not input-dependent: it is completely random and introduced solely to hide the nonzero value of x . We therefore use r itself as the second factor of the preprocessed value. Specifically, we specialize conventional Beaver multiplication by setting $b = r$, so that the second masked difference $d = r - b$ is zero and need not be represented or opened. After opening only $e = x - a$, the parties compute $[x \odot r] = e \odot [r] + [c]$.

Thus, a single Beaver-style preprocessing tuple $([a], [r], [c])$ supplies both the multiplication correlation and the random multiplicative mask. Relative to the straightforward realization, this eliminates the independently shared mask in addition to the triple, the opening of d , and the terms $d \odot [a]$ and $e \odot d$ from the product computation. In our star topology, a generic realization would transmit $5(N - 1)m$ field elements: two masked-difference shares from each non-designated party, the two reconstructed differences in the reverse direction, and one final product share. Our specialized realization transmits only $3(N - 1)m$ field elements, excluding the final output bits. The two constructions can use the same number of communication rounds, but our construction substantially reduces communication.

Protocol description. The parties first generate m Beaver

triples $[\mathbf{a}], [\mathbf{r}], [\mathbf{c}]$. Each party P_i sends $\mathbf{e}_i = \mathbf{w}_i - [\mathbf{a}]_i$ to the designated party P_1 , who computes their sum $\mathbf{e} = \sum_i \mathbf{e}_i = (\sum_i \mathbf{w}_i) - \mathbf{a}$. Since the shares $\{[\mathbf{a}]_i\}$ are all independently uniform, neither the $\{\mathbf{e}_i\}$ nor their sum leak anything about the parties' inputs. Next, P_1 sends $\mathbf{e}$ to all parties, who each then compute $\mathbf{z}_i = \mathbf{e} \odot [\mathbf{r}]_i + [\mathbf{c}]_i$ and send it to P_1 . Finally, P_1 computes $\mathbf{z} = \sum_i \mathbf{z}_i$ and sets the j -th entry of $\mathbf{y}$ to 1 if the j -th entry of $\mathbf{z}$ is nonzero. The protocol is described formally in Algorithm 1.

Note that no input-sharing step is required for this protocol. The local input vectors already form additive shares of the intended sum, since $x = \sum_{i=1}^N w_i$. Party P_i can therefore use w_i directly as its “share” and send $e_i = w_i - [a]_i$. Every message e_i sent by an honest party is uniformly masked by the preprocessing share $[a]_i$.

To see that correctness holds, note first that because $p > N$ we have

$$\forall_i \mathbf{w}_i[j] = 1 \Leftrightarrow \sum_i \mathbf{w}_i[j] \neq 0$$

for any index j . Next, note that

$$\begin{aligned} \mathbf{z} = \sum_i \mathbf{z}_i &= \mathbf{r} \odot \mathbf{e} + \mathbf{c} \\ &= \mathbf{r} \odot \sum_i \mathbf{w}_i - \mathbf{r} \odot \mathbf{a} + \mathbf{c} \\ &= \mathbf{r} \odot \sum_i \mathbf{w}_i. \end{aligned} \quad (4.1)$$

Since $\mathbf{r}[j] \neq 0$, this means

$$\forall_i \mathbf{w}_i[j] = 1 \Leftrightarrow \sum_i \mathbf{w}_i[j] \neq 0 \Leftrightarrow \mathbf{z}[j] \neq 0,$$

as desired.

Next, we prove security.

Theorem 4.1 (Semi-honest security of Π_{OR}). *Π_{OR} securely realizes $\mathcal{F}_{\text{OR}}$ for any number of semi-honest corruptions in the $\mathcal{F}_{\text{Beaver},p}$ -hybrid model.*

Proof. Let I be the set of corrupted parties, and let $\mathcal{H} := [N] \setminus I$ be the set of honest parties. We assume P_1 is corrupted as when that is not the case the adversary sees strictly fewer protocol messages. We also assume $I \neq [N]$ since security is trivial if all parties are corrupted.

We define a simulator Sim as follows. Sim first forwards the inputs $\{\mathbf{w}_i\}_{i \in I}$ of the corrupted parties to $\mathcal{F}_{\text{OR}}$, and receives output $\mathbf{y} := \bigvee_{i=1}^N \mathbf{w}_i$. Then:

1. Sample uniform shares $\{[\mathbf{a}]_i, [\mathbf{r}]_i, [\mathbf{c}]_i\}_{i \in I}$ and give these to the corrupted parties on behalf of $\mathcal{F}_{\text{Beaver},p}$.
2. Choose uniform $\{\mathbf{e}_i\}_{i \in \mathcal{H}}$ and send those to P_1 on behalf of the honest parties.
3. Let $\mathbf{e} = \sum_{i \in \mathcal{H}} \mathbf{e}_i + \sum_{i \in I} (\mathbf{w}_i - [\mathbf{a}]_i)$. (Note the corrupted parties can already compute this from their view.)

4. For $j = 1, \dots, m$, set $\mathbf{z}[j] = 0$ if $\mathbf{y}[j] = 0$, and choose $\mathbf{z}[j] \leftarrow \mathbb{F}_p^*$ if $\mathbf{y}[j] = 1$. For $i \in I$, let $\mathbf{z}_i = \mathbf{e} \odot [\mathbf{r}]_i + [\mathbf{c}]_i$. Choose $\{\mathbf{z}_i\}_{i \in \mathcal{H}}$ uniformly subject to the constraint

$$\sum_{i \in \mathcal{H}} \mathbf{z}_i = \mathbf{z} - \sum_{i \in I} \mathbf{z}_i,$$

and send those to P_1 on behalf of the honest parties.

We argue that the view of the corrupted parties in the above simulation is distributed identically to their view in an execution of Π_{OR} on the same inputs. This is immediate for the simulation of the first step, above; for the second step, it follows since the $\{[\mathbf{a}]_i\}_{i \in \mathcal{H}}$ are uniform and independent of the $\{[\mathbf{a}]_i, [\mathbf{r}]_i, [\mathbf{c}]_i\}_{i \in I}$ in an execution of Π_{OR} . To see that the simulation of the fourth step is perfect, observe that when $\mathbf{y}[j] = 1$ (so $\bigvee_i \mathbf{w}_i[j] = 1$) then since $\mathbf{r}[j]$ is uniform over $\mathbb{F}_p^*$, Equation 4.1 in fact shows that $\mathbf{z}[j]$ is uniform over $\mathbb{F}_p^*$. (And, of course, when $\mathbf{y}[j] = 0$ then $\mathbf{z}[j] = 0$ as well.) Thus, the simulated distribution of $\mathbf{z}$ is identical to the distribution of $\mathbf{z}$ in an execution of Π_{OR} . Finally, the honest parties' simulated messages $\{\mathbf{z}_i\}_{i \in \mathcal{H}}$ are uniform subject to the constraint that the $\{\mathbf{z}_i\}_{i \in [N]}$ sum to $\mathbf{z}$, exactly as in a real execution of Π_{OR} . $\square$

4.2 Our MPSU Protocol

We next construct a protocol for MPSU assuming ideal access to $\mathcal{F}_{\text{OR}}$. Algorithm 2 shows the complete protocol in detail. Let f be the prefix extension parameter, i.e., number of bits by which a prefix is extended at each level, and assume that $f \mid \kappa$. Define the alphabet $\mathbb{A} := \{0, 1\}^f$, whose symbols correspond to all possible f -bit extensions, and define $D := \kappa/f$, the number of levels required to reconstruct a full κ -bit element. We identify every κ -bit element with a length- D word over $\mathbb{A}$, so that $S_i \subseteq \mathbb{A}^D$. For $x \in \mathbb{A}^D$ and $\ell \in \{0, \dots, D\}$, let $\text{pref}_\ell(x) \in \mathbb{A}^\ell$ denote the first ℓ symbols of x , with $\text{pref}_0(x) = \epsilon$. For a set $S \subseteq \mathbb{A}^D$, define

$$\text{Pref}_\ell(S) := \{\text{pref}_\ell(x) : x \in S\}.$$

Bloom-filter parameters. The protocol fixes a target false-positive rate ϵ_{BF} . At each layer ℓ , at most $\min\{\sum_i |S_i|, 2^{f\ell}\}$ distinct prefixes can occur. Accordingly, we choose (m_ℓ, k_ℓ) using the Bloom-filter sizing rule from Section 3.

The parties use the same public Bloom-filter parameters

$$\text{bf}_\ell := (m_\ell, k_\ell, H_{\ell,1}, \dots, H_{\ell,k_\ell})$$

for layer ℓ , where $H_{\ell,j} : \mathbb{A}^\ell \rightarrow [m_\ell]$. The set of hash functions should be agreed upon in advance.

Local sketch construction and private aggregation. For each depth $\ell \in [D-1]$, party P_i encodes all length- ℓ prefixes of its input as

$$\mathbf{F}_i^{(\ell)} := \text{BFInsert}_\ell(\text{Pref}_\ell(S_i)) \in \{0, 1\}^{m_\ell}.$$

Algorithm 2 Π_{MPSU} in the $\mathcal{F}_{\text{OR}}$ -hybrid model

Public parameters: $N, n, \kappa, f, D, \epsilon_{\text{BF}}$ and $\{\text{bf}_\ell\}_{\ell=1}^{D-1}$
Input: Each party P_i holds $S_i \subseteq \mathbb{A}^D$ with $|S_i| \leq n$
Output: Every party outputs $\hat{U} = \bigcup_{i=1}^N S_i$
Each party P_i does: ▷ Local sketch construction

- 1: **for** $\ell = 1$ to $D - 1$ **do**
- 2: $\mathbf{F}_i^{(\ell)} \leftarrow \text{BFInsert}_\ell(\text{Pref}_\ell(S_i))$
- 3: $\mathbf{F}_i \leftarrow \mathbf{F}_i^{(1)} \parallel \dots \parallel \mathbf{F}_i^{(D-1)}$
- 4: Parties invoke $\mathcal{F}_{\text{OR}}$ on $\mathbf{F}_1, \dots, \mathbf{F}_N$ and receive $\mathbf{F} = \mathbf{F}^{(1)} \parallel \dots \parallel \mathbf{F}^{(D-1)}$ ▷ Private aggregation
- Each party does:* ▷ Local candidate reconstruction
- 5: $\mathcal{B}_0 \leftarrow \{\epsilon\}$
- 6: **for** $\ell = 1$ to $D - 1$ **do**
- 7: $\mathcal{B}_\ell \leftarrow \emptyset$
- 8: **for all** $b \in \mathcal{B}_{\ell-1}$ and $\alpha \in \mathbb{A}$ **do**
- 9: $b' \leftarrow b \parallel \alpha$
- 10: **if** $\text{BFTest}_\ell(\mathbf{F}^{(\ell)}, b') = 1$ **then**
- 11: $\mathcal{B}_\ell \leftarrow \mathcal{B}_\ell \cup \{b'\}$
- 12: Let $c_1, \dots, c_M$ be the elements of $\{b \parallel \alpha : b \in \mathcal{B}_{D-1}, \alpha \in \mathbb{A}\}$ in lexicographic order
- Each party P_i does:* ▷ Exact final verification
- 13: **for** $j = 1$ to M **do**
- 14: $\mathbf{q}_i[j] \leftarrow \mathbf{1}[c_j \in S_i]$
- 15: Parties invoke $\mathcal{F}_{\text{OR}}$ on $\mathbf{q}_1, \dots, \mathbf{q}_N$ and receive $\mathbf{q}$
- 16: Each party outputs $\hat{U} \leftarrow \{c_j : \mathbf{q}[j] = 1\}$

It concatenates these vectors into

$$\mathbf{F}_i := \mathbf{F}_i^{(1)} \parallel \dots \parallel \mathbf{F}_i^{(D-1)}.$$

The parties invoke $\mathcal{F}_{\text{OR}}$ once on $\mathbf{F}_1, \dots, \mathbf{F}_N$ and parse the common output as

$$\mathbf{F} = \mathbf{F}^{(1)} \parallel \dots \parallel \mathbf{F}^{(D-1)}.$$

Local candidate reconstruction and exact verification.

Starting from $\mathcal{B}_0 := \{\epsilon\}$, every party reconstructs the same candidate frontier. At depth ℓ , it expands every $b \in \mathcal{B}_{\ell-1}$ by every symbol $\alpha \in \mathbb{A}$ and keeps $b \parallel \alpha$ exactly when it passes the public membership test against $\mathbf{F}^{(\ell)}$. Bloom filters have no false negatives, so every true union prefix survives; false positives may introduce additional candidates.

After depth $D - 1$, the parties construct the set of full-length candidate elements $\mathcal{C} := \{b \parallel \alpha : b \in \mathcal{B}_{D-1}, \alpha \in \mathbb{A}\}$, and let $c_1, \dots, c_M$ be the elements of $\mathcal{C}$ in the canonical lexicographic order.

Each party P_i then forms an exact membership indicator vector $\mathbf{q}_i \in \{0, 1\}^M$ with $\mathbf{q}_i[j] := \mathbf{1}[c_j \in S_i]$. A second invocation of $\mathcal{F}_{\text{OR}}$ returns $\mathbf{q} = \bigvee_i \mathbf{q}_i$, and the parties retain exactly those c_j for which $\mathbf{q}[j] = 1$. This final invocation uses the parties' original sets rather than Bloom filter answers and therefore removes every false-positive candidate.

5 Security and Complexity Analysis

This section proves the security of our MPSU scheme and discusses the complexity of the protocol.

5.1 Correctness and Privacy

We first record the aggregation property that connects the first $\mathcal{F}_{\text{OR}}$ invocation to a Bloom filter of the union prefixes. In analyzing this protocol, we use the following aggregation property of Bloom filters [13].

Lemma 5.1 (Bloom-filter aggregation). *Fix a depth ℓ and a common public Bloom-filter description bf_ℓ . For any sets $T_1, \dots, T_N \subseteq \mathbb{A}^\ell$,*

$$\bigvee_{i=1}^N \text{BFInsert}_\ell(T_i) = \text{BFInsert}_\ell\left(\bigcup_{i=1}^N T_i\right).$$

Theorem 5.2 (Correctness of Π_{MPSU}). *For every choice of public Bloom-filter parameters and hash functions, Protocol Π_{MPSU} correctly computes $U := \bigcup_{i=1}^N S_i$.*

Proof. First note that $U \subseteq \bigcup_i S_i$ due to the final exact verification step. It remains to show that $\bigcup_i S_i \subseteq U$. Fix $u \in \bigcup_i S_i$. We prove by induction that $\text{pref}_\ell(u) \in \mathcal{B}_\ell$ for all $\ell \in [D - 1]$. This trivially holds for $\ell = 0$. Assuming $\text{pref}_{\ell-1}(u) \in \mathcal{B}_{\ell-1}$, the parties compute $\text{BFTest}_\ell(\mathbf{F}^{(\ell)}, \text{pref}_\ell(u))$ (because they test all extensions of $\text{pref}_{\ell-1}(u)$) which will return 1 by Lemma 5.1 (since $\mathbf{F}^{(\ell)} = \bigvee_i \mathbf{F}_i^{(\ell)}$ and Bloom filters have no false negatives). This proves the inductive claim.

Since $\text{pref}_{D-1}(u) \in \mathcal{B}_{D-1}$, some party P_i with $u \in S_i$ will set the corresponding entry of $\mathbf{q}_i$ equal to 1, and hence u will be included in the final output U . $\square$

Theorem 5.3 (Semi-honest security of Π_{MPSU}). *Protocol Π_{MPSU} securely realizes $\mathcal{F}_{\text{MPSU}}$ for any number of semi-honest corruptions in the $\mathcal{F}_{\text{OR}}$ -hybrid model.*

Proof. Let I be the set of corrupted parties, and let $\mathcal{H} := [N] \setminus I$ be the set of honest parties. We assume $I \neq [N]$ since security is trivial if all parties are corrupted.

We define a simulator Sim as follows. Sim first forwards the inputs $\{S_i\}_{i \in I}$ of the corrupted parties to $\mathcal{F}_{\text{MPSU}}$, and receives output U . Then:

1. For $\ell = 1, \dots, D - 1$, let $\mathbf{F}^{(\ell)} \leftarrow \text{BFInsert}_\ell(\text{Pref}_\ell(U))$. Give the corrupted parties $\mathbf{F}^{(1)} \parallel \dots \parallel \mathbf{F}^{(D-1)}$ on behalf of $\mathcal{F}_{\text{OR}}$.
2. Compute $\mathcal{B}_{D-1}$ from $\mathbf{F}^{(1)} \parallel \dots \parallel \mathbf{F}^{(D-1)}$ exactly as in lines 5–11 of Algorithm 2. Let $c_1, \dots, c_M$ be the elements of $\{b \parallel \alpha\}_{b \in \mathcal{B}_{D-1}, \alpha \in \mathbb{A}}$ in lexicographic order.
3. For $j \in [M]$, set $\mathbf{q}[j] = 1$ if $c_j \in U$ and $\mathbf{q}[j] = 0$ otherwise. Give the corrupted parties $\mathbf{q}$ on behalf of $\mathcal{F}_{\text{OR}}$.

We argue that the view of the corrupted parties in the above simulation is distributed identically to their view in an execution of Π_{MPSU} on the same inputs. Let $\{S_i\}_{i \in \mathcal{H}}$ be the inputs of the honest parties. In the ideal world, these are not known to simulator. In an execution of Π_{MPSU} , we have

$$\mathbf{F}^{(\ell)} = \bigvee_i \text{BFInsert}_\ell(\text{Pref}_\ell(S_i))$$

for $\ell = 1, \dots, D-1$. In the ideal world, we have

$$\begin{aligned} \mathbf{F}^{(\ell)} &= \text{BFInsert}_\ell(\text{Pref}_\ell(\bigcup_i S_i)) \\ &= \text{BFInsert}_\ell(\bigcup_i \text{Pref}_\ell(S_i)) \\ &= \bigvee_i \text{BFInsert}_\ell(\text{Pref}_\ell(S_i)) \end{aligned}$$

(using Lemma 5.1), and so the distributions of the $\mathbf{F}^{(\ell)}$ in the real and ideal worlds are identical. In words, observe that the set of prefixes of the union is equal to the union of prefixes of the sets, allowing the simulator to simulate intermediate state based on the outputs. The values $c_1, \dots, c_M$ computed by the simulator are identical to the values that would be computed by all parties in an execution of the protocol. Finally, in both the real and ideal worlds $\mathbf{q}$ is a boolean vector of length M that is set to 1 exactly at indices corresponding to elements in $U = \bigcup_i S_i$. $\square$

By the standard modular composition theorem [10], replacing the $\mathcal{F}_{\text{OR}}$ invocations in Π_{MPSU} with Π_{OR} yields a protocol that securely realizes $\mathcal{F}_{\text{MPSU}}$ in the $\mathcal{F}_{\text{Beaver},p}$ -hybrid model.

5.2 Complexity Analysis

In our design, each party's vote is represented by a field element in $\mathbb{F}_p$. The only requirement on p is that $p > N$, which ensures that the sum of N input bits is zero in $\mathbb{F}_p$ if and only if all input bits are zero. The field size affects communication through the $\log p$ -bit representation of each field element. Since p can be chosen close to N , we treat $\log p$ as $O(\log N)$ in the following analysis.

Effect of false positives. At each depth, every surviving prefix is expanded into 2^f children, and each false child survives the Bloom-filter test with probability ϵ_{BF} . Thus, a false-positive prefix produces $2^f \epsilon_{\text{BF}}$ surviving false-positive children in expectation. False positives can be introduced from true prefixes at every depth and can themselves propagate to subsequent depths. Since there are at most Nn true prefixes at each depth, the expected number of surviving prefixes at depth ℓ is bounded by $Nn \left(1 + \sum_{j=1}^{\ell} (2^f \epsilon_{\text{BF}})^j\right)$. Therefore, when $2^f \epsilon_{\text{BF}} < 1$, the expected prefix size remains bounded by

$$\frac{Nn}{1 - 2^f \epsilon_{\text{BF}}}.$$

In particular, if $2^f \epsilon_{\text{BF}} \leq 1/2$, the expected last layer contains at most $2Nn$ prefix candidates.

Online rounds. Excluding preprocessing, an execution of Π_{OR} requires 4 rounds. The MPSU protocol invokes two sequential executions of Π_{OR} , with only local computation between them. It therefore uses 8 rounds, independent of N , n , and κ .

Online communication.

Following Section 3, a Bloom filter for at most Nn elements and false-positive rate ϵ_{BF} has size at most $Nn \cdot \frac{\log(1/\epsilon_{\text{BF}})}{(\log 2)^2}$. Across the κ/f Bloom-filter layers, the private aggregation phase therefore processes $O\left(Nn \frac{\kappa}{f} \log(1/\epsilon_{\text{BF}})\right)$ field elements.

For final exact verification, each surviving prefix is expanded into 2^f candidates. Our false-positive bound gives at most $2Nn$ surviving prefixes in expectation, and hence at most $2^{f+1}Nn$ final candidates.

Accounting for the $O(\log N)$ -bit representation of each field element and communication among N parties, the total communication complexity is $O\left(N^2 n \log N \left(\frac{\kappa}{f} \log \frac{1}{\epsilon_{\text{BF}}} + 2^f\right)\right)$ bits.

Local computation. Each party inserts at most n prefixes at each of the κ/f depths. Since a Bloom filter with false-positive rate ϵ_{BF} uses $O(\log(1/\epsilon_{\text{BF}}))$ hash functions, local Bloom-filter construction requires $O\left(n \frac{\kappa}{f} \log \frac{1}{\epsilon_{\text{BF}}}\right)$ hashes per party.

For candidate reconstruction, each surviving prefix is expanded into 2^f children, and each child requires $O(\log(1/\epsilon_{\text{BF}}))$ Bloom-filter tests. Under our false-positive bound, the expected number of surviving prefixes at each depth is at most $2Nn$. Therefore, reconstruction requires at most $2^{f+1}Nn$ Bloom-filter tests per depth in expectation, for a total of $O\left(2^{f+1}Nn \frac{\kappa}{f} \log \frac{1}{\epsilon_{\text{BF}}}\right)$ Bloom-filter tests per party.

Finally, the exact verification phase considers at most $2^{f+1}Nn$ candidates in expectation, where the factor 2^f accounts for the children generated from each surviving prefix and the additional factor 2 accounts for candidates introduced by Bloom-filter false positives. Constructing the final verification vector requires $O(2^{f+1}Nn)$ local set-membership checks per party.

Thus, the overall local computation is dominated by candidate reconstruction and is $O\left(2^f Nn \frac{\kappa}{f} \log \frac{1}{\epsilon_{\text{BF}}}\right)$ per party in expectation.

Table 1: Summary of complexity of SPRUCE protocol.

Metric	Complexity
Online Rounds	$O(1)$
Online Communication	$O\left(N^2 n \log N \left(\frac{\kappa}{f} \log \frac{1}{\epsilon_{\text{BF}}} + 2^f\right)\right)$
Local Computation	$O\left(2^f Nn \frac{\kappa}{f} \log \frac{1}{\epsilon_{\text{BF}}}\right)$

6 Implementation and Evaluation

In this section, we present the implementation and evaluation details of SPRUCE.

6.1 Implementation Details

SPRUCE codebase. We implemented our protocol SPRUCE in the Go language. Our codebase is available at <https://github.com/timetz/SPRUCE>. In our implementation, each prefix layer is represented using a Bloom filter from the *bits-and-blooms* library [3]. Bloom-filter positions are derived using the unkeyed, non-cryptographic *MurmurHash3* function provided in the library.

All arithmetic is performed over $\mathbb{F}_p$ with $p = 31$, using a field implementation generated with *gnark-crypto* [12]. We require the number of parties to satisfy $N < p$, ensuring that the aggregate of a nonempty Bloom-filter position cannot wrap around to zero in the field. Since field elements take values below 31, they are serialized using one byte.

Parties communicate over TLS-wrapped TCP connections in a star topology, with P_1 coordinating MPC openings and reconstruction. Given the way our Π_{OR} functionality is constructed in Algorithm 1, our implementation is specialized to our PSU protocol (i.e., we do not use general-purpose MPC). The implementation parallelizes computation and communication using configurable Go goroutines and using individual TLS connections between pairs of parties.

Experimental setup. All experiments were conducted on six virtual machines connected through a local-area network. Four machines report Intel Xeon 6530P processors, while the remaining two report Intel Xeon Gold 6226R processors. Each machine exposes 16 logical CPUs and has 60 GiB of memory. The `eth0` interface on every machine reports a link speed of 50 Gb/s.

We configured each machine’s outgoing `eth0` traffic using Linux `tc/netem`. Traffic was rate-limited to 10 Gbps to simulate LAN connections, and to 1 Gbps with one-way network delays of 40ms and 100ms to simulate WAN connections. We used BBR congestion control and an `fq` queue beneath the `netem` discipline. We disabled TCP slow start after idle and raised the TCP send and receive buffer maxima to $2^{31} - 1$ bytes to solve host-side TCP bottlenecks in experiments and maximize performance.

The implementations use the same party placement on our six-machine testbed. For $N \leq 6$, each party runs on a distinct machine. For $N > 6$, parties are assigned cyclically across the six machines, and hence some machines host multiple parties.

Parameters. Our protocol parameters include the number of parties N , the per-party input size n , and the Bloom filter false positive rate ϵ_{BF} . The implementation parameters include the input bit length κ , prefix extension f (or a fanout of 2^f), MPC field size p , and the number of threads. The size of the Bloom

filter m and the number of hash functions k are computed based on the formula from Section 3 as a function of the total number of inputs and the fanout parameter.

Unless specified otherwise, the default setting is $N = 6$ parties, $n = 2^{14}$ inputs per party, $\kappa = 128$ bits, prefix extension $f = 4$, $p = 31$, $\epsilon_{\text{BF}} = 0.02$, and each party uses 8 threads.

We evaluated our implementation with varying numbers of parties, $\{3, 4, 6, 8, 12, 18, 24\}$, and representative per-party input sizes, $\{2^{12}, 2^{14}, 2^{16}\}$, under both LAN and WAN settings. The results can be found in Table 2 and 3.

Benchmark methodology. Our evaluation reports the online phase of the protocol, where our main performance improvements apply. Accordingly, the current implementation uses benchmark-only simulated preprocessing: the required preprocessing material is generated locally without executing a secure distributed preprocessing protocol. As a reference point, computing Beaver-triple generation using the semi-honest *semi* protocol provided by the MP-SPDZ [27] library incurs ~ 21 s for the setting of 6 parties with 2^8 inputs across six machines. Since pre-processing is input-independent and parallelizable, it can be computed in advance, and its performance can be improved through better implementations and additional computational resources.

Comparison with prior work. We compare SPRUCE against state-of-the-art semi-honest MPSU implementations: the better performed symmetric-key implementation in [15], which we refer to as DZBC25, and PULSE [17]. To enable a fair comparison, we evaluate the codebases provided in these prior works on the same infrastructure that we use to run our implementation.

Both baseline artifacts represent protocol values using 128-bit blocks provided by their `cryptoTools` dependency. In [15], each block contains both the set element and auxiliary protocol data, limiting the usable element bit-length to 64 bits. PULSE’s membership-test layer operates on 128-bit elements. We therefore configure SPRUCE with $\kappa = 64$ when comparing against DZBC25 and with $\kappa = 128$ when comparing against PULSE.

Both implementations have all parties run on a single host. We made modifications to replace hard-coded loopback addresses with configurable network addresses and ports and to allow each party to communicate through the network. These changes do not modify the cryptographic protocols or their online message flows.

6.2 Evaluation

In our evaluation, we address three primary questions:

1. **Scalability.** How does the performance of SPRUCE change as a function of the number of parties, input sizes, and input bit length?
2. **Microbenchmarks.** What parts of the protocol contribute

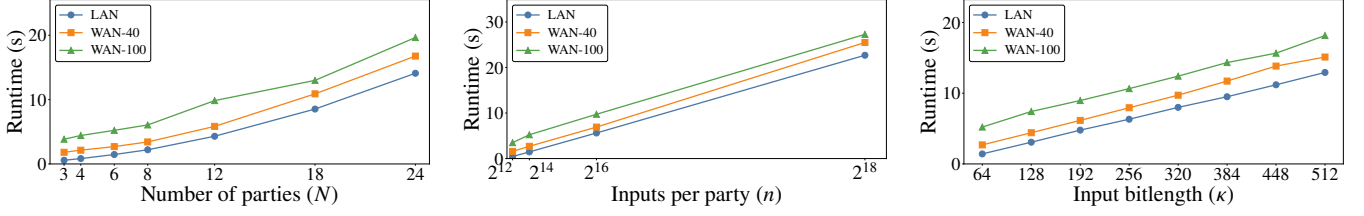


Figure 6: Runtime scalability across (a) number of parties, (b) input size, and (c) bit length.

to execution overhead? How do parameter choices affect performance?

3. Comparison with prior work. How does SPRUCE compare with prior work DZBC25 and PULSE under balanced and unbalanced settings?

6.2.1 Scalability

To understand our protocol performance as a function of parameters independently, we run our experiments for varying numbers of parties, $\{3, 4, 6, 8, 12, 18, 24\}$, and per-party input sizes, $\{2^{12}, 2^{14}, 2^{16}, 2^{18}\}$, under both LAN and WAN settings. The results are shown in Figure 6. We analyze the effect of one parameter when keeping others fixed.

First, as the number of parties increases, the execution time increases quadratically. This is because (i) the aggregated input size, and hence the Bloom-filter size, grows linearly with N , and (ii) the number of communication connections also grows linearly with N . Second, for each of the settings, the execution time varies linearly with input size. This is expected because Bloom filter size is linear in the total number of inserted prefixes. Third, the execution time increases linearly with input bit length κ . A linear increase in κ results in a linear increase in the depth of the tree. This causes a linear increase in the number of Bloom filters, which linearly affects the MPC computation size. Also, a linear increase in the depth of the tree causes a linear increase in the amount of local computation. Finally, since we have a constant round protocol, in all three scenarios, observe that adding more latency, i.e., comparing WAN vs. LAN setting only has a constant effect on the runtime under all settings. Together, this explains the trend observed in Figure 6.

6.2.2 Microbenchmarks

In this subsection, we discuss how different aspects of protocol design contribute to the performance of our protocol. We also discuss the choice of lower level parameters such as parallelism, Bloom filter false positive rate ϵ_{BF} , and prefix extension f .

Figure 7 represents the runtime for different components of the protocol under the default parameterization when run on a LAN setting. We represent three logical components of

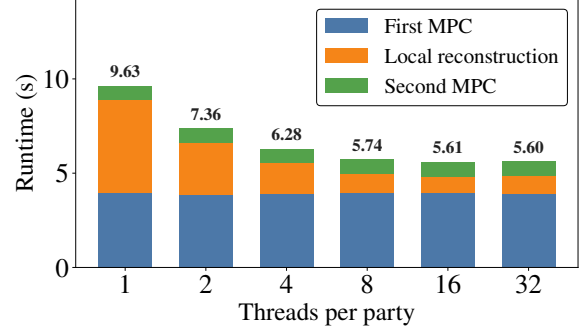


Figure 7: Impact of different phases of the protocol and parallelism.

our protocol: (i) First MPC involving OR-computation for Bloom filters, (ii) Local reconstruction to compute the active prefixes of the last layer, and (iii) Second MPC to compute final outputs from last layer prefixes. In the default setting with 8 threads, runtime is dominated by the First MPC.

Parallelism. Increasing the number of threads per party from one to 32 reduces local reconstruction time from 4.96 to 0.92 seconds, corresponding to a speedup of approximately $5.4\times$ for this phase. The complete online runtime decreases from 9.63 to 5.60 seconds, an overall speedup of approximately $1.7\times$.

Since our machine has 16 vCPUs (or 8 physical cores), most of the available speedup is obtained by eight threads: increasing the thread count further reduces total runtime only from 5.74 seconds at eight threads to 5.60 seconds at 32 threads. In principle, all of the phases of our protocol are parallelizable; however, our implementation currently only parallelizes local reconstruction; due to this, the two interactive MPC phases eventually impose a lower bound on the runtime.

Impact of Bloom filter false positive rate ϵ_{BF} and prefix extension f . The Bloom-filter false-positive rate ϵ_{BF} and prefix extension parameter f jointly determine the costs of private aggregation and local reconstruction. We discuss the impact of tuning these parameters under the default parameter setting. If $f \nmid \kappa$, we used a shorter extension of $\kappa \bmod f$ bits for the final layer. The results are shown in Fig. 8.

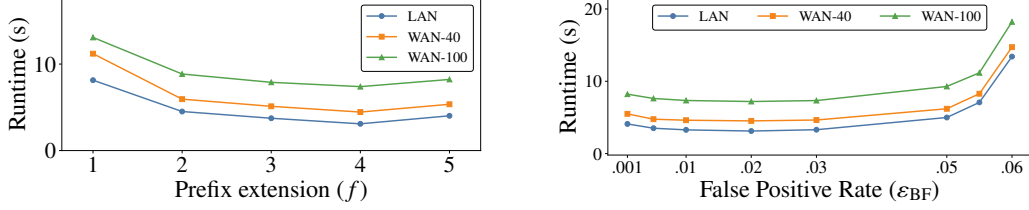


Figure 8: Runtime sensitivity to (a) prefix extension and (b) FPR.

A smaller f produces more layers. This incurs more sequential local work for local reconstruction and more workload for Bloom filter insertion. A larger f reduces the tree depth but increases the branching factor to 2^f , causing every surviving prefix (including false-positive prefixes) to have more children candidates to test against. Empirically, we observe that $f = 4$ achieves the lowest runtime across both evaluated false-positive rates and network settings.

A lower ϵ_{BF} requires larger Bloom filters, thereby increasing the computation and communication for private aggregation. A higher ϵ_{BF} produces smaller Bloom filters but admits more false-positive prefixes, increasing local reconstruction and exact-verification costs. Figure 8(b) therefore exhibits a U-shaped trend. Setting $f = 4$, $\epsilon_{BF} = 0.02$ minimizes runtime, so we use these values as the default.

6.2.3 Comparison with Prior Work

In this subsection, we compare SPRUCE with prior works PULSE and DZBC25 under balanced (i.e., every party contributes an equal number of inputs) and unbalanced (i.e., parties contribute unequal number of inputs).

Comparison under balanced settings. Table 2 compares SPRUCE with PULSE using 128-bit inputs. SPRUCE is faster in every configuration, achieving speedups between $2.67\times$ and $28.17\times$. PULSE additionally exceeds the runtime or memory limit in several of the largest configurations that SPRUCE completes. Across completed configurations, SPRUCE’s total communication ranges from approximately $1.12\times$ to $1.15\times$ that of PULSE. Thus, the runtime improvement is achieved without a significant increase in communication.

Table 3 compares SPRUCE with DZBC25 using 64-bit inputs. We observe that neither implementation uniformly dominates in the LAN setting. SPRUCE performs particularly well for smaller input sets, whereas DZBC25 is faster in several of the larger LAN configurations. This behavior reflects a tradeoff between communication volume and interaction: SPRUCE sends more data but uses a constant number of online rounds, while DZBC25 communicates less but incurs more rounds. The advantage of SPRUCE consequently increases as latency grows. Under WAN-100, SPRUCE is faster in every measured configuration, with speedups ranging from

$1.40\times$ to $7.86\times$. These results demonstrate the benefit of constant-round interaction in high-latency environments.

Comparison under unbalanced settings. As described in Section 3, SPRUCE does not reveal parties’ input cardinalities. In effect, the computation and thus the performance are completely dependent on the number of outputs (although Bloom filter sizes do have a coarse dependence on this value too, so an estimate of this value is required). On the other hand, prior protocols are parameterized by a common per-party input capacity that is a function of $n_{\max}$, the maximum number of values held by any party. The effective workload is thus a function of $Nn_{\max}$ even if other parties hold substantially fewer elements.

To evaluate such an unbalanced setting, we consider six parties where P_1 holds 2^{14} elements and every other party holds only 2^8 elements. As shown in Figure 9, SPRUCE completes the unbalanced workload in 0.63, 1.94, and 4.22 seconds under LAN, WAN-40, and WAN-100, respectively. On the other hand, DZBC25 and PULSE pay the same cost as that of running with 2^{14} inputs. Thus, compared with ours, PULSE is $98.32\times$ and $17.14\times$ slower under LAN and WAN-100 respectively, while DZBC25 is $3.78\times$ and $5.01\times$ slower. These results demonstrate that output-size aware Bloom filter sizing allows SPRUCE to exploit highly skewed input distributions rather than paying the cost of padding every party to the largest input size.

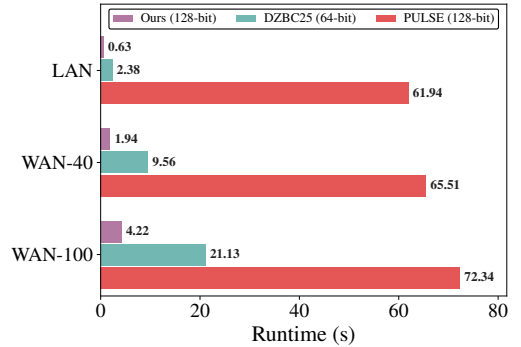


Figure 9: Runtime under unbalanced input distributions. P_1 holds 2^{14} elements, other parties hold 2^8 elements.

Table 2: Runtime for 128-bit inputs. Communication is total traffic on the wire, counting each transmitted byte once. Lower is better; blue marks the better value (if applicable). O/M denotes running out of memory.

	n	Protocol	$N = 3$	$N = 4$	$N = 6$	$N = 8$	$N = 12$	$N = 18$	$N = 24$
LAN (s)	2^{12}	PULSE	4.387	7.425	15.837	26.790	57.888	123.508	218.509
		Ours	0.331	0.496	0.809	1.297	2.448	4.788	7.757
	2^{14}	PULSE	17.261	30.281	61.832	104.090	223.589	486.645	851.430
		Ours	1.228	1.795	3.146	4.928	9.056	18.019	30.708
	2^{16}	PULSE	69.129	117.956	243.658	415.137	> 900	O/M	O/M
		Ours	4.912	7.009	12.159	18.881	36.292	72.929	121.714
WAN-40 (s)	2^{12}	PULSE	6.508	9.680	19.279	30.987	64.431	134.403	231.030
		Ours	1.638	1.759	2.083	2.484	3.629	6.563	10.406
	2^{14}	PULSE	19.683	32.942	65.859	108.752	232.715	502.380	864.826
		Ours	2.625	3.200	4.485	6.078	11.860	20.592	32.926
	2^{16}	PULSE	72.308	120.361	246.906	417.631	> 900	O/M	O/M
		Ours	6.107	8.301	14.579	22.416	39.538	85.765	130.857
WAN-100 (s)	2^{12}	PULSE	9.849	13.827	24.258	38.690	76.428	153.575	255.341
		Ours	3.690	3.960	4.359	4.937	6.023	8.969	12.872
	2^{14}	PULSE	24.183	37.233	71.352	115.800	237.993	510.261	> 900
		Ours	5.156	5.875	7.165	8.731	12.902	24.155	38.852
	2^{16}	PULSE	76.909	126.312	253.921	432.057	> 900	O/M	O/M
		Ours	9.209	11.170	16.321	24.373	45.630	82.051	131.928
Comm. (MiB)	2^{12}	PULSE	17.729	35.457	88.644	165.468	390.031	904.163	1631.039
		Ours	19.923	39.822	99.551	185.968	437.940	1015.597	1832.166
	2^{14}	PULSE	70.622	141.243	353.108	659.134	1553.673	3601.697	6497.179
		Ours	79.625	159.375	398.297	743.640	1752.515	4061.785	7329.920
	2^{16}	PULSE	276.226	552.451	1381.128	2578.105	–	O/M	O/M
		Ours	318.639	637.225	1593.461	2973.893	7010.114	16250.393	29314.281

Table 3: Runtime for 64-bit inputs. Communication is total traffic counting once. Lower is better; blue marks the better value.

	n	Protocol	$N = 3$	$N = 4$	$N = 6$	$N = 8$	$N = 12$	$N = 18$	$N = 24$
LAN (s)	2^{12}	DZBC25	2.060	2.085	2.140	3.215	4.502	4.325	6.680
		Ours	0.192	0.267	0.450	0.656	1.149	2.267	3.961
	2^{14}	DZBC25	2.134	2.203	2.403	3.685	4.953	7.372	13.224
		Ours	0.574	0.834	1.462	2.201	4.304	8.534	14.095
	2^{16}	DZBC25	2.435	2.708	3.484	5.618	9.582	19.675	38.249
		Ours	2.108	3.200	5.616	8.673	16.415	32.988	52.195
WAN-40 (s)	2^{12}	DZBC25	6.976	7.468	8.714	10.932	13.575	18.895	22.952
		Ours	1.276	1.427	1.578	1.842	2.211	3.459	5.165
	2^{14}	DZBC25	7.427	8.219	9.571	11.940	14.938	21.054	36.771
		Ours	1.818	2.139	2.696	3.421	5.840	10.893	16.768
	2^{16}	DZBC25	8.170	8.948	10.499	13.425	19.533	48.477	102.858
		Ours	3.501	4.597	6.912	10.342	19.428	37.730	60.241
WAN-100 (s)	2^{12}	DZBC25	14.863	16.280	19.203	23.440	29.971	43.265	56.545
		Ours	2.961	3.287	3.491	3.975	4.497	5.771	7.190
	2^{14}	DZBC25	16.220	18.131	21.056	25.909	34.460	44.986	60.560
		Ours	3.845	4.422	5.224	6.063	9.846	12.993	19.651
	2^{16}	DZBC25	17.784	19.619	23.145	28.156	37.903	57.670	115.394
		Ours	6.248	7.392	9.725	12.826	22.110	41.266	65.605
Comm. (MiB)	2^{12}	DZBC25	1.690	3.145	7.351	13.121	29.483	66.093	117.193
		Ours	10.495	20.993	52.392	97.862	230.650	534.826	964.761
	2^{14}	DZBC25	6.788	12.814	29.544	52.752	118.583	265.909	471.570
		Ours	41.936	83.881	209.731	391.416	922.271	2138.457	3859.066
	2^{16}	DZBC25	27.871	51.694	119.115	212.616	477.781	1071.095	1904.586
		Ours	167.686	335.606	838.927	1565.804	3690.130	8554.838	15432.210

7 Conclusion

We have presented a new semi-honest multi-party PSU design that departs dramatically from prior approaches. Our scheme combines a prefix-tree data structure maintained and updated via a small, optimized MPC with a novel applica-

tion of Bloom filters to save communication rounds without adding false positives to the protocol output. The resulting protocol outperforms prior work in many settings, sometimes by significant margins, validating the effectiveness of our techniques as a new paradigm for MPSU protocols.

Acknowledgment

This material is based upon work supported in part by the National Science Foundation under Grant Nos. 2234408 and 2237814. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Implementation Access

Our implementation of SPRUCE is available at: <https://github.com/timetzv/SPRUCE>. We have also updated code base of PULSE [17] and DZBC25 [15]. We have the compared updated codebase available in the repo as well.

References

- [1] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology—CRYPTO ’91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1992.
- [2] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC ’90)*, pages 503–513. ACM, 1990.
- [3] bits-and-blooms contributors. Bloom: A Go package implementing Bloom filters. GitHub repository. Version 3.7.1. Accessed August 19, 2026. URL: <https://github.com/bits-and-blooms/bloom>.
- [4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [5] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 762–776. IEEE, 2021.
- [6] Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In *Advances in Cryptology—ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 2005.
- [7] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Private matching for compute. Cryptology ePrint Archive, Paper 2020/599, 2020. URL: <https://eprint.iacr.org/2020/599>.
- [8] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *19th USENIX Security Symposium (USENIX Security 10)*, pages 223–240, Washington, DC, August 2010. USENIX Association.
- [9] Ran Canetti. Security and composition of multi-party cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
- [10] Ran Canetti. Security and composition of multi-party cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
- [11] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In *Advances in Cryptology—ASIACRYPT 2020, Part III*, volume 12493 of *Lecture Notes in Computer Science*, pages 342–372. Springer, 2020.
- [12] ConsenSys. gnark-crypto: Efficient cryptographic primitives in Go. GitHub repository. Version 0.20.1. Accessed August 19, 2026. URL: <https://github.com/ConsenSys/gnark-crypto>.
- [13] Adina Crainiceanu and Daniel Lemire. Bloofi: Multidimensional Bloom filters. *Information Systems*, 54:311–324, December 2015. arXiv:1501.01941 [cs.DB]. URL: <https://arxiv.org/abs/1501.01941>.
- [14] Minglang Dong, Yu Chen, Cong Zhang, Yujie Bai, and Yang Cao. Multi-party private set operations from predicative zero-sharing. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS ’25)*, pages 4079–4093. ACM, 2025.
- [15] Minglang Dong, Cong Zhang, Yujie Bai, and Yu Chen. Efficient Multi-Party private set union without Non-Collusion assumptions. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 2005–2024, Seattle, WA, August 2025. USENIX Association.
- [16] Keith B. Frikken. Privacy-preserving set union. In *Applied Cryptography and Network Security (ACNS 2007)*, volume 4521 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2007.
- [17] Jiahui Gao, Son Nguyen, Marina Blanton, and Ni Trieu. PULSE: Parallel private set union for large-scale entities. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS ’25)*, pages 1784–1798. ACM, 2025.
- [18] Jiahui Gao, Son Nguyen, and Ni Trieu. Toward a practical multi-party private set union. *Proceedings on Privacy Enhancing Technologies*, 2024(4):622–635, 2024.

- [19] Yixiao Gao, Xuexin Zheng, and Changhui Hu. A multi-party private set union protocol against malicious adversary. In *Proceedings of Innovative Computing 2024, Vol. 4*, volume 1217 of *Lecture Notes in Electrical Engineering*, pages 159–167. Springer Singapore, 2024.
- [20] Gayathri Garimella, Payman Mohassel, Mike Rosulek, Saeed Sadeghian, and Jaspal Singh. Private set operations from oblivious switching. In *Public-Key Cryptography—PKC 2021, Part II*, volume 12711 of *Lecture Notes in Computer Science*, pages 591–617. Springer, 2021.
- [21] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play ANY mental game. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC '87)*, pages 218–229. ACM, 1987.
- [22] Xuhui Gong, Qiang-Sheng Hua, and Hai Jin. Nearly optimal protocols for computing multi-party private set union. In *2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2022.
- [23] Kyle Hogan, Noah Luther, Nabil Schear, Emily Shen, David Stott, Sophia Yakoubov, and Arkady Yerukhimovich. Secure multiparty computation for cooperative cyber risk assessment. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 75–76. IEEE, 2016.
- [24] Yanxue Jia, Shi-Feng Sun, Hong-Sheng Zhou, Jiajun Du, and Dawu Gu. Shuffle-based private set union: Faster and more secure. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2947–2964, Boston, MA, August 2022. USENIX Association.
- [25] Yanxue Jia, Shi-Feng Sun, Hong-Sheng Zhou, and Dawu Gu. Scalable private set union, with stronger security. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6471–6488, Philadelphia, PA, August 2024. USENIX Association.
- [26] Murat Kantarcioglu and Chris Clifton. Privacy-preserving distributed mining of association rules on horizontally partitioned data. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1026–1037, 2004.
- [27] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*, pages 1575–1590. ACM, 2020.
- [28] Lea Kissner and Dawn Song. Privacy-preserving set operations. In *Advances in Cryptology—CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 241–257. Springer, 2005.
- [29] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. Scalable private set union from symmetric-key techniques. In *Advances in Cryptology—ASIACRYPT 2019, Part II*, volume 11922 of *Lecture Notes in Computer Science*, pages 636–666. Springer, 2019.
- [30] Yehuda Lindell. How to simulate it—a tutorial on the simulation proof technique. In Yehuda Lindell, editor, *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer International Publishing, 2017.
- [31] Qiang Liu and Joon-Woo Lee. Efficient multi-party private set union resistant to maximum collusion attacks. Cryptology ePrint Archive, Paper 2024/2096, 2024. URL: <https://eprint.iacr.org/2024/2096>.
- [32] Xiang Liu and Ying Gao. Scalable multi-party private set union from multi-query secret-shared private membership test. In *Advances in Cryptology—ASIACRYPT 2023, Part I*, volume 14438 of *Lecture Notes in Computer Science*, pages 237–271. Springer, 2023.
- [33] Lucas Piske and Ni Trieu. Is PSI really faster than PSU? achieving efficient PSU with invertible Bloom filters. In *Advances in Cryptology—EUROCRYPT 2026, Part II*, volume 16542 of *Lecture Notes in Computer Science*, pages 438–467. Springer, 2026.
- [34] Sihang Pu, Jiahui Gao, and Ni Trieu. Malicious private set union with two-sided output. In *Advances in Cryptology—EUROCRYPT 2026, Part II*, volume 16542 of *Lecture Notes in Computer Science*, pages 408–437. Springer, 2026.
- [35] Jae Hong Seo, Jung Hee Cheon, and Jonathan Katz. Constant-round multi-party private set union using reversed laurent series. In *Public Key Cryptography—PKC 2012*, volume 7293 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2012.
- [36] Katsunari Shishido and Atsuko Miyaji. Efficient and quasi-accurate multiparty private set union. In *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 309–314. IEEE, 2018.
- [37] Binbin Tu, Yujie Bai, Cong Zhang, Yang Cao, and Yu Chen. Fast enhanced private set union in the balanced and unbalanced scenarios. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 3437–3456, Seattle, WA, August 2025. USENIX Association.
- [38] Cong Zhang, Yu Chen, Weiran Liu, Min Zhang, and Dongdai Lin. Linear private set union from multi-query reverse private membership test. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 337–354, Anaheim, CA, August 2023. USENIX Association.