Private and Oblivious Set and Multiset Operations

Marina Blanton and Everaldo Aguiar
Department of Computer Science and Engineering
University of Notre Dame
{mblanton,eaguiar}@nd.edu

Abstract

Privacy-preserving set operations, and set intersection in particular, are a popular research topic. Despite a large body of literature, the great majority of the available solutions are two-party protocols and are not composable. In this work we design a comprehensive suite of secure multi-party protocols for set and multiset operations that are composable, do not assume any knowledge of the sets by the parties carrying out the secure computation, and can be used for secure outsourcing. All of our protocols have communication and computation complexity of $O(m \log m)$ for sets or multisets of size m, which compares favorably with prior work. Furthermore, we are not aware of any results that realize composable operations. Our protocols are secure in the information theoretic sense and are designed to minimize the round complexity. Practicality of our solutions is shown through experimental results.

1 Introduction

The ability to securely perform set operations on private inputs has numerous applications. As an example, we mention computing intersection of databases belonging to different agencies or organizations, which by law or other provisions are not permitted to share their records in the clear, but want to compute the set of records common to both of them. This can be used in contexts ranging from finding passengers of an airline who appear in the national watch list to determining customers common to two companies for more effective advertisement. The importance of the topic is also evidenced by a significant body of prior work (see, e.g., [34, 52, 35, 31, 49, 43, 5]).

Work on privacy-preserving set operations started with the seminal work of [34]. Consequently, many other publications appeared with the goal of extending the functionality or improving its performance. Secure protocols are known for set intersection (e.g., [34, 52, 42, 31, 49]), set union (e.g., [52, 35, 43]), set intersection cardinality or over-the-threshold cardinality (e.g., [68, 29]), multiset element reduction ([52]), and others. Most of publications assume the two-party setting, in which Alice and Bob each possess a private set A and B, respectively, apply a set operation to the sets, and learn the result (or only one party learns the result). In such protocols, the knowledge of the private input A or B is essential for correctly recovering the result.

While this problem formulation has a large number of applications, the existing solutions cannot be securely used as building blocks in larger protocols as they are not designed to be composable.[1] That is, a set operation has to comprise the entire computation as neither the output can remain private from both parties nor the existing solutions apply when A or B is the result of prior secure computation and is not known to either party. One example application that could benefit

[1]Here, by composability we mean the ability to use set operations as building blocks in larger computation using sequential composition. This is different from security under concurrent execution in the universal composability framework, which our protocols also achieve.

from composable set operations is privacy-preserving product network aggregation [62], in which set union and set intersection operations are executed one after another. Prior solutions to this problem had to reveal some information at an intermediate step because the existing set union and intersection protocols were not composable. The literature that provides solutions for the multi-party setting [52, 35] likewise assumes that each participant has access to her private set in the clear.

The recent emergence of cloud computing demands techniques for secure outsourcing that will allow the benefits of available cloud services to be utilized to the fullest extent, which otherwise might not be used due to the fear of information disclosure. In that setting the computational parties do not have access to the private inputs and it is essential that they do not learn any information about the data they process, while are able to carry out the computation correctly. In other words, the computation needs to be data-independent or oblivious. From that point of view, it is desirable to have protocols that are both composable and can be used in outsourced tasks, which we set as one of our goals. Note that in many existing solutions, the computation cannot be outsourced because performing the computation requires knowledge of some of the inputs.

We utilize secure multi-party computation (SMC) framework, where $n > 2$ computational parties carry out the computation using linear secret sharing and can be independent from input owners or output recipients. This means that the techniques are suitable for traditional secure multi-party computation as well as secure outsourcing by one or more parties who utilize multiple computational servers for secure computation. In the case of secure outsourcing, each client simply distributes its input to the servers and receives and reconstructs the output at the end of the computation, i.e., the computation is non-interactive for the client. As we employ secret sharing, only a fraction t of the computational parties (or servers to which the computation is being outsourced) can collude or misbehave to guarantee security, i.e., the majority of the computational parties must be honest. Related problem formulations from the literature include secure outsourcing to a single server (e.g., [70]), secure outsourcing to two non-colluding servers (e.g., [10]), and server-aided computation with a single server, where the client's work is not guaranteed to be linear in the input/output size (e.g., [51]).

Our contributions. In this work, we provide a suite of secure multi-party protocols for a number of set and multiset operations, which are union, intersection, difference, symmetric difference, equality, subset and superset relationships, and element reduction (for multisets only). Besides computing the main functionality, we provide variants of the protocols that produce cardinality of the resulting (multi)set or compute over-the-threshold cardinality and produce a bit. Furthermore, our protocols can be used to always hide the size of the input/output (multi)sets or the size can be revealed to make any computation that follows more efficient (since complexity of set operations is proportional to the size of their representation). We also provide a generic conversion from a multiset to a set that allows us to run our protocols for secure set operations on multisets, as well as a direct and more efficient implementation of multiset operations. Finally, we describe a number of optimizations that allow for faster performance of our protocols. In addition to minimizing the total number of operations, substantial part of this work is dedicated to reducing the number of rounds (i.e., sequential interactive operations) of set and multiset operations, which has a tremendous impact of the performance of our protocols in practice. We implement our solutions for selected set and multiset operations and show that their performance is practical and comparable to those of the latest (more restrictive) two-party solutions.

The advantages of our solutions over previously available results are as follows:

1. The requirement that each input set/multiset is known by a participant in the clear is removed. This implies that the elements of the input sets can be arbitrarily partitioned among the

participants. The input sets can also be a result of prior privacy-preserving computation and are not known in the clear to any participant.

2. Our protocols are composable. Because both the inputs and outputs are split among the participants, our protocols can be composed an arbitrary number of times or they can be used as building blocks in larger computations.
3. No intermediate results or other information are revealed to the participants, which makes the solution suitable for secure computation outsourcing. In other words, the parties who provide the inputs and/or learn the output can be different from the parties carrying out secure computation. This is in contrast to prior results, where the knowledge of a set in the clear was essential for correctness of the computation.
4. Our solution provides natural support for hiding the sizes of the sets. The input sets can be padded for additional security, and the size of the result is never revealed, unless the parties decide to do otherwise.
5. Unlike most prior literature, our techniques make no use of expensive operations based on public-key cryptography and achieve information-theoretic security (assuming the existence of secure channels between the participants).
6. All of our protocols are efficient and have $O(m \log m)$ communication and computation complexity where m is the sum of the input sets' sizes. This compares favorably with the existing solutions (which we detail below).

Security of our protocols is shown in both semi-honest (honest-but-curious or passive) and malicious (active) models. Part of this work appeared in [9].

2 Related Work

Privacy-preserving set operations. The first custom solutions for securely computing set operations were two-party set intersection and intersection cardinality protocols of [34] based on homomorphic encryption and polynomial representation of sets. The authors also proposed an optimization using balanced hash functions that reduced the computation overhead to $O(m \ln \ln m)$, while the overall communication complexity of the protocols was $O(m)$. Here m represents the set size and $n \geq 2$ will denote the number of parties. [52] extended that work by building a framework of multiset operations which included set union, intersection and element reduction. This work was also the first to establish protocols secure against malicious players when three or more parties were involved, which was done via zero-knowledge proofs. The protocols secure in the honest-but-curious model (and the set intersection protocol secure in the malicious model) presented in [52] had communication complexities of $O(n^2m)$, or $O(c^2m)$ when $c < n$ dishonest players collude, and computation complexity of $O(n^2m^2)$. [42] proposed the first two-party private set intersection protocol based on oblivious pseudo-random functions (OPRFs). If we denote m_1 and m_2 to be the number of elements in the sets, where the first set is considered to belong to the server and the second to the client, and t to be the size of the binary representation of input elements, the solution in [42] is constant round and have communication and computation (modular exponentiations) complexities of $O(m_1 \cdot t + m_2)$. An improvement to this work is presented in [48], and a similar protocol that replaces the OPRFs with unpredictable functions can be found in [49].

An efficient set union protocol for the malicious adversary, with communication complexity of $O(n^2m^2 + n^3m)$ and $O(n)$ rounds, was given by [35]. [46] suggest another set union protocol for the malicious adversary, which uses a modified ElGamal cryptosystem and achieves constant-round communication. Additionally, [21, 43, 30, 22] provide protocols for privacy preserving set intersection in the two-party setting secure against malicious adversaries. The approach in [30],

Reference	Operation	Computation	Commucation	Mult. party	Uncond. security	Size hiding	Composable
[30]	PSI	$O(m)$	$O(m)$				
[29]	PSI-CA	$O(m)$	$O(m)$				
[5]	PSI	$O(m \log m)$	$O(m)$			$\surd$	
[52]	PSI, PSI-CA	$O(n^2m^2)$	$O(n^2m)$	$\surd$			
[17]	PSI	$O(n^3m)$	$O(n^3m)$	$\surd$			
[22]	PSI	$O(nm^2)$	$O(nm + m\log^2 m)$	$\surd$			
[46]	PMU	$O(n^2m^2)$	$O(n^2m)$	$\surd$			
[59]	PSI	$O(n^3m^2 + n^4)$	$O(n^3m^2 + n^4)$	$\surd$	$\surd$		
This work	PSI, PSU, PSDiff, PER, PSR, PSI-CA, PSU-CA, PSDiff-CA, PER-CA, PMI, PMU, PMDiff, PSE, PSuR, PMI-CA, PMU-CA, PMDiff-CA	$O(nm\log m + n^2\log m + n^3)$	$O(nm\log m + n^2\log m + n^3)$	$\surd$	$\surd$	$\surd$	$\surd$

Table 1: Summary of secure set operations protocols with the best performance.

building on the efficient solution from [31], yields linear complexities (in the number of set elements) for both communication and computation achieving higher efficiency than prior linear-time work [48]. Also, publications [52, 68, 58, 29, 63] propose protocols that compute the set intersection cardinality (rather than the intersection itself), with [29] being the most efficient and having linear-time computation and communication complexities in the semi-honest model (in the random oracle model). One noteworthy recent work [5] adds to the two-party set intersection operation the feature of hiding the size of the input set (including the upper bound) held by the participant who learns the result. The computation performed locally by that party is then no longer linear, but instead is $O(m \log m)$, where m is size of that party's input set. Lastly, we highlight the work [63], which describes protocols for several set operations in the Universal Composability framework with malicious adversaries and $O(n^2m^2)$ complexities.

There are also publications that develop private set intersection protocols in the information-theoretic setting [56, 59, 58, 60]. [56] proposed the first unconditionally secure protocol using polynomial representation of sets and two-dimensional secret sharing (where values are shared among the players and each share is again shared among them) with communication complexity of $O(n^4m^2)$ in the malicious model and $t < n/3$ parties under control of an active adversary (though that complexity is contested in [59]). [59] revisit the problem and provide an information-theoretically secure set intersection protocol with communication complexity of $O(n^3m^2 + n^4)$. This solution is used in [60] to build a protocol that works when the adversary controls $t < n/2$ parties, in which case communication complexity becomes $O(n^4m^2 + n^5)$. [58] addresses private matching, set disjointness, and set intersection cardinality in the information-theoretic setting with semi-honest adversaries.

The only implementations of private set operations that we are aware of are for two-party set intersection in [47] that uses garbled circuits and in [32] that implements the protocol from [30].

Table 1 provides a brief comparison of the most relevant protocols listing their complexities and functionality. Notations PSI and PMI stand for "private set intersection" and "private multiset intersection," respectively. U stands for "union," ER stands for "element reduction," CA means "cardinality", SR denotes "subset relationship", and "set equality" and "superset relationship" are represented by SE and SuR. All complexities are listed for the malicious adversary and correspond to solutions with the best performance. In the table, a solution is marked as size hiding if the sizes of the input sets can be protected by means of padding. We note that [5] achieves a stronger notion of size hiding in which no information about one of the two input sets is revealed. We additionally achieve that no information about the size of the output set (beyond the bounds imposed by the sizes of the (padded) input sets) is revealed to the parties. The complexity of the results in Table 1 that rely on public-key cryptography is measured in public-key operations reported in them (i.e., not the total number of operations) and the security parameters for communication are implicit. The remaining solutions achieve information-theoretic security with communication measured in the number of field elements (of small size). All reported complexities reflect the combined work and communication of *all parties*.

Secure multi-party computation. The literature on SMC and function evaluation is very extensive and its review is beyond the scope of this work. In the multi-party setting, employed in this work, the available techniques are garbled circuit evaluation (e.g., [38, 8]), linear secret sharing techniques (e.g., [65, 19]), and threshold homomorphic encryption (e.g., [33, 27, 20]). We employ linear secret sharing and design efficient and information-theoretically secure protocols for set and multiset operations.

Parallel set operations. Set operations have also been examined in the realm of parallel computing. Early solutions [53, 66] utilized specially designed array structures to efficiently compute these operations directly in hardware. More recent parallel techniques [11] involve a careful arrangement of the data into random balanced binary trees. While these techniques allow set operations to be performed efficiently, they were not designed to be secure, are not data-oblivious, and do not naturally lend themselves to secure multi-party protocols.

3 Preliminaries

3.1 Framework

In this work we use the multi-party setting in which $n > 2$ parties $P_1, \ldots, P_n$ jointly execute prescribed functionality on private inputs. We utilize a linear secret sharing scheme (such as [65]) for representation of, and secure computation over private values. To ensure composability of our protocols, we assume P_1 through P_n receive their shares of the input prior to the computation and compute shares of the output. Then any party with private input will produce shares of it before the computation starts, and upon computation completion P_1 through P_n send their shares to the entities entitled to learn the result. This gives flexibility to the problem setting in that the input parties may be disjoint from the computational parties (as in the case with outsourcing). Similarly, the parties receiving the output do not have to coincide with the input parties or computational parties.

Throughout this work we assume that parties $P_1, \ldots, P_n$ are pair-wise connected by secure authenticated channels (the underlying communication model depends on the employed secret sharing; usually synchronous communication with broadcast channels is assumed). Each input and output party also establishes secure channels with P_1 through P_n. With (n, t)-secret sharing, any private value is secret-shared among n parties such that any $t + 1$ shares can be used to reconstruct it, while t or fewer shares information-theoretically reveal no information about the shared value. Therefore, the values of n and t should be chosen such that an adversary is unable to corrupt more than t computational parties.

In a linear secret sharing scheme, a linear combination of secret-shared values can be computed by each computational party locally, without any interaction, but multiplication of two secret-shared values requires communication between the computational parties. In other words, if we let $[x]$ denote that value x is secret-shared among $P_1, \ldots, P_n$, operations $[x] + [y]$, $[x] + c$, and $c[x]$ are performed by each P_i locally on its shares of x and y, while computation of $[x][y]$ is interactive. The most common way of implementing a multiplication operation is by sending the total of $O(n^2)$ messages (where each participant sends $n-1$ messages, one to each other participant) using, for instance, the techniques of [36], but recent results [45, 28, 7] lower the communication to $O(n)$ messages per multiplication.

All operations are assumed to be performed in a field $\mathbb{Z}_p$ for a small prime p greater than the maximum value used in the computation. We use ℓ to denote the bitlength of (multi)set elements, and therefore it must hold that $p \geq 2^\ell$. Without loss of generality, we assume that the domain of set elements consists of integers greater than 0, i.e., it is $[1, 2^\ell - 1]$ (i.e., if the domain is different, it can always be mapped to $[1, 2^\ell - 1]$ for some ℓ).

Performance of secure computation is of grand significance, as protecting secrecy of data throughout the computation often incurs substantial computational costs. For that reason, besides security, efficient performance of the developed techniques is one of our primary goals. Performance of a protocol in our setting is typically measured in terms of: (i) the number of interactive operations (multiplications, distributing shares of a private value or opening a secret-shared value) necessary to perform the computation, or *invocations*, and (ii) the number of sequential interactions, or *rounds*. We employ the same metrics here.

3.2 Building blocks

We now proceed with a brief description of building blocks which are used in our solutions, namely, oblivious sorting, comparisons, and prefix AND.

Oblivious sorting. When sorting is utilized in secure computation, the sequence of operations that the parties execute must be independent of the set they are sorting, or data-oblivious, to ensure that no information about the private data is revealed. While most sorting algorithms are not oblivious, using a sorting network results in an oblivious solution. Such techniques use compare-and-exchange operations (CEO), which are fixed and independent of the input. In our setting, a CEO can be implemented as follows:

$$\begin{array}{l} [s] \leftarrow \mathsf{GE}([a],[b],\ell) \\ [c] \leftarrow [s][b] + (1-[s])[a] \\ [d] \leftarrow [s][a] + (1-[s])[b] \end{array}$$

where GE denotes a "greater than or equal" operation for ℓ-bit operands (detailed below) that produces a bit. After comparing a and b, c corresponds to $\min(a,b)$ and d to $\max(a,b)$.

Ajtai et al. [1] describe a sorting network with $O(m \log m)$ comparisons for a set of cardinality m, but it has a very high constant behind the big-O notation. More practically, Batcher's network [6] uses $O(m \log^2 m)$ comparisons and was the basis of secure multi-party sorting in [50]. More recent results [55, 39, 41] developed oblivious randomized sorting algorithms with $O(m \log m)$ comparisons and low constants which succeed with very high probability. Another recent solution is [72], in which oblivious sorting is achieved in constant round using $O(m^2)$ or $O(mR)$ communication and computation, where $[0, R]$ is the range of numbers to be sorted. Throughout the paper, we use notation $([y_1], \ldots, [y_m]) \leftarrow \mathsf{Sort}([x_1], \ldots, [x_m], \ell)$ to denote secure implementation of oblivious sorting on ℓ-bit values in this framework. In some cases, we also need to sort tuples, where the comparisons are performed using the first element of each tuple, but the entire tuples are swapped based on the outcome of a comparison. We denote this modification by SortT, e.g., $\langle [x_1], [y_1] \rangle, \ldots, \langle [x_m], [y_m] \rangle \leftarrow \mathsf{SortT}(\langle [a_1], [b_1] \rangle, \ldots, \langle [a_m], [b_m] \rangle, \ell)$ denotes sorting of 2-tuples.

Because performance is of particular importance to us and complexity of oblivious sorting dominates the complexity of all of our algorithms, we analyze the solutions of [39] and [6] in more detail. Goodrich's shellsort [39] uses asymptotically low $5m \log m - 7.5m + 9 \approx 5m \log m$ CEOs, but requires $5m - \log m + 1 \approx 5m$ of them to be executed consecutively. The number of rounds then corresponds to this value multiplied by the round complexity of a CEO. Batcher's network [6] that uses odd-even merge sort involves $\frac{1}{4}m(\log^2 m - \log n + 4) - 1 \approx \frac{1}{4}m \log^2 m$ CEOs, but they can be more effectively parallelized using $\frac{1}{2}\log m(\log m + 1) \approx \frac{1}{2}\log^2 m$ consecutive CEOs. Also, for $m \lesssim 10^6$, Batcher's network involves fewer comparisons than Goodrich's shellsort.

In some cases we also need to merge two sorted arrays, which can be accomplished faster than sorting all elements. For that reason, we define and use protocol $\mathsf{Merge}(([x_1], \ldots, [x_{m_1}]), ([y_1], \ldots, [y_{m_2}]), \ell)$, which is part of Batcher's oblivious merge sort. Oblivious bitonic merge from [6] uses $\frac{1}{2}m \log m$ CEOs and has depth (i.e., the number of consecutive CEOs) of $\log m$. Similar to sorting, use MergeT to denote the tuple version of Merge.

Other protocols. We also rely on the following secure protocols from prior literature:

- $[b] \leftarrow \mathsf{Eq}([x], [y], \ell)$ is an equality protocol that, on input two secret-shared values x and y of length at most ℓ bits, outputs (shares of) a bit b which is set to 1 iff $x = y$. The most efficient implementation of this operation that we are aware of is from [15] which uses $\ell + 4 \log \ell$ invocations in 4 rounds, where most of the cost is input independent and can be performed ahead of time.

- $[b] \leftarrow \mathsf{GE}([x], [y], \ell)$ is a comparison protocol that, on input two secret-shared ℓ-bit values x and y, outputs a bit b which is set to 1 iff $x \geq y$. Efficient implementations of this function also exist, e.g., we can use the protocol from [15] with 4 rounds and $4\ell - 2$ invocations, where precomputation can also reduce the cost.

- $([y_1], \ldots, [y_k]) \leftarrow \mathsf{PreAND}([x_1], \ldots, [x_k])$ computes prefix-AND, which on input a sequence of bits $x_1, \ldots, x_k$, outputs bits $y_1, \ldots, y_k$, where each $y_i = \bigwedge_{j=1}^{i} x_j$. Secure multi-party implementation of PreAND can be realized by utilizing prefix-OR, PreOR, by calling $\mathsf{PreOR}(1 - [x_1], \ldots, 1 - [x_k])$ and outputting the complements of the returned bits. The PreOR protocol from [15] uses 3 rounds and $5k - 1$ invocations, where, as before, input-independent precomputation can reduce the cost.

The complexities of Eq, GE, and PreAND functionalities cited above correspond to statistically secure protocols, but alternative implementations that achieve perfect secrecy are also available. All other parts of our solutions, with the exception of another building block described in Protocol 17 in optimizations section 7, are perfectly secure. Therefore, by using perfectly secure implementations of these building blocks the overall solutions will be perfectly secure as well. Because the separation between perfect and statistical security might be important with respect to what security properties we can obtain, we note that perfectly secure versions of Eq, GE, and PreAND that also run in a constant number of rounds and have linear complexities (in ℓ in case of Eq and GE and in k in case of PreAND) are available from [23]. They can be built from any linear secret sharing scheme with a multiplication protocol. The main difference between these protocols and the protocols from [15] is that the perfectly secure versions of comparison operations assume that the arguments are given in a bitwise form (i.e., x and y are represented as shares of ℓ bits each). This does not impose a limitation for the type of computations used in this work because the overwhelming number of operations are contributed by comparisons. We therefore can use bitwise representation of (multi)set elements throughout the protocols without increasing their asymptotic complexity. If at any point of the computation bit-decomposition is required (e.g., for computing over-the-threshold versions of set operations), it is also available from [23] and other sources.

Finally, another recent work [67] provides equality and comparison protocols of sublinear (in ℓ) complexity. In particular, the equality protocol in [67] uses $O(\delta)$ invocations in a constant number of rounds, where δ is a correctness parameter, and a comparison is performed using $O(\log \ell(\delta + \log\log \ell))$ invocations in $O(\log \ell)$ rounds or using $O(\sqrt{\ell}(\delta + \log \ell))$ interactive operations in a constant number of rounds for the same δ. These protocols are, however, more suitable for SMC based on homomorphic encryption and are applicable to our setting only when $t = 1$.

3.3 Security model

For each presented protocol, we define its secure functionality such that the computational parties do not provide any input and do not receive any output. Instead, it is assumed that prior to the beginning of each protocol the input parties secret-share their sets among the computational parties. Likewise, at the end of the computation, the computational parties send their shares to the entities entitled to learn the result who reconstruct the output.

We next formally define security using the standard definition in secure multi-party computation for semi-honest adversaries, who follow the protocol as prescribed, but might try to learn more than they entitled from the protocol execution. For this case we assume that the adversary is static, i.e., the set of corrupted parties is fixed prior to the protocol execution. When, however, treating the case of malicious adversaries who can follow any arbitrary strategy, we will assume the adaptive adversary who can adaptively corrupt the participants throughout the protocol execution.

Definition 1 *Let parties $P_1, \ldots, P_n$ with pair-wise secure channels engage in a protocol π that computes a (possibly probabilistic) n-ary function $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$, where P_i contributes input in_i and receives output out_i. Let $\mathrm{VIEW}_\pi(P_i)$ denote the view of participant P_i during the execution of protocol π. More precisely, P_i's view is formed by its input and internal random coin tosses r_i, as well as messages $m_1, \ldots, m_k$ passed between the parties during protocol execution: $\mathrm{VIEW}_\pi(P_i) = (\mathsf{in}_i, r_i, m_1, \ldots, m_k)$. Let $I = \{P_{i_1}, P_{i_2}, \ldots, P_{i_t}\}$ denote a subset of the participants, $\mathrm{VIEW}_\pi(I)$ denote the combined view of participants in I during the execution of protocol π (i.e., $\mathrm{VIEW}_\pi(I) = (\mathrm{VIEW}_\pi(P_{i_1}), \ldots, \mathrm{VIEW}_\pi(P_{i_t}))$), and $f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ denote the projection of $f(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ on the coordinates in I (i.e., $f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ consists of the i_1th, $\ldots$, i_tth elements that $f(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ outputs). We say that protocol π is t-private in presence of static semi-honest adversaries if for each coalition I of size at most t and all $\mathsf{in}_i \in \{0,1\}^*$ there exists a probabilistic polynomial time simulator S_I such that $\{(S_I(\mathsf{in}_I, f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)), f(\mathsf{in}_1, \ldots, \mathsf{in}_n))\} \equiv \{(\mathrm{VIEW}_\pi(I), (\mathsf{out}_1, \ldots, \mathsf{out}_n))\}$, where $\mathsf{in}_I = (\mathsf{in}_{i_1}, \ldots, \mathsf{in}_{i_t})$ and "$\equiv$" denotes perfect or statistical indistinguishability.*

By secure channels we mean private authenticated channels, in which case security is information-theoretic. In case of malicious adversaries, security is formalized by comparing a protocol execution to an ideal model where the participants simply send their inputs to a trusted third party and receive their outputs back.

Definition 2 *Let π be a protocol that computes function $f : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$, with party P_i contributing input in_i. Let $\mathcal{A}$ be an arbitrary algorithm with auxiliary input x and S be an adversary/simulator in the ideal model. Let $REAL_{\pi,\mathcal{A}(x),I}(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ denote the view of adversary $\mathcal{A}$ controling parties in I together with the honest parties' outputs after real protocol π execution. Similarly, let $IDEAL_{f,S(x),I}(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ denote the view of S and outputs of honest parties after ideal execution of function f. We say that π t-securely computes f if for each coalition I of size at most t, every probabilistic $\mathcal{A}$ in the real model, all $\mathsf{in} \in \{0,1\}^*$ and $x \in \{0,1\}^*$, there is probabilistic S in the ideal model that runs in time polynomial in $\mathcal{A}$'s runtime and $\{\mathrm{IDEAL}_{f,S(x),I}(\mathsf{in}_1, \ldots, \mathsf{in}_n)\} \equiv \{\mathrm{REAL}_{\pi,\mathcal{A}(x),I}(\mathsf{in}_1, \ldots, \mathsf{in}_n)\}$.*

Security in the semi-honest model holds for $t < n/2$ and in malicious for $t < n/3$.

4 Set Operations

This section presents our solutions for several set operations – set intersection, union, asymmetric and symmetric difference, subset and superset relationships, and set equality, as well as multiset element reduction. All other multiset operations are treated in consecutive sections. Our solutions assume that the set or multiset operations are performed on ℓ-bit values in integer representation, and the parameter ℓ is omitted from the notation.

Intuitively, correctly computing an operation on sets A and B of size m without any knowledge of what these sets contain appears to be hard if fewer than m^2 comparisons are used (one comparison for each $a_i \in A$ and $b_j \in B$). Indeed, if any given pair of elements a_i, b_j have not been (explicitly or implicitly) compared, then for arbitrary sets A and B the result is not guaranteed to be correct. If, however, the result is known to be correct with fewer comparisons, then some information about the input sets must be known which violates our security requirements. Fortunately, relationships between some pairs a_i, b_j can be determined implicitly, based on other explicit comparisons of elements of A and B, which eliminates the need for m^2 comparisons. We notice that once data-oblivious sorting is used as a building block, we can realize all of our set and multiset operations using $O(m \log m)$ interactive operations (comparisons) and their round complexity exceeds that of sorting by a small (additive) constant. We mark all interactive operations and rounds in our protocols.

4.1 Core protocols

The main idea behind our solutions consists of combining the input sets into one, sorting the resulting set, and comparing adjacent elements of the sorted set to determine what elements should be kept and what should be erased, depending on the desired set operation. For certain set operations such as set difference, we also maintain information about the origin of an element (e.g., coming from the first or the second input set) to implement the desired functionality. A more detailed description of each operation is given next.

Set union. The first protocol that we describe computes the set union $C = A \cup B$, where $A = \{a_1, \ldots, a_{m_1}\}$, $B = \{b_1, \ldots, b_{m_2}\}$, and $C = \{c_1, \ldots, c_{m_1+m_2}\}$. Initially the elements of A and B are combined into a new set and subsequently sorted. Next, we need to eliminate duplicates, as we wish to keep only a single instance of each item appearing in either of the sets. To accomplish this, our protocol looks at adjacent items in the sorted set, x_i and x_{i+1}. If the elements are the same, the first instance is erased by setting the corresponding item c_i in the resulting set to 0 (recall that 0 is not a valid element of A or B). The protocol makes no changes to those items that occur a single time.

Protocol 1. $[c_1], \ldots, [c_{m_1+m_2}] \leftarrow \mathsf{Union}([a_1], \ldots, [a_{m_1}], [b_1], \ldots, [b_{m_2}])$

```
1. $[x_1], \ldots, [x_{m_1+m_2}] \leftarrow \mathsf{Sort}([a_1], \ldots, [a_{m_1}], [b_1], \ldots, [b_{m_2}], \ell)$;        // section 3.2
2. for $i = 1$ to $m_1 + m_2 - 1$ do in parallel
3.     $[u_i] \leftarrow \mathsf{Eq}([x_i], [x_{i+1}], \ell)$;                                         // section 3.2
4.     $[c_i] \leftarrow [x_i](1 - [u_i])$;                                              // 1 round, $m_1 + m_2 - 1$ inv
5. $[c_{m_1+m_2}] \leftarrow [x_{m_1+m_2}]$;
6. return $[c_1], \ldots, [c_{m_1+m_2}]$;
```

For example, on input sets $\langle 2, 4, 1, 5\rangle$ and $\langle 4, 3, 2\rangle$, we obtain $\langle 1, 2, 2, 3, 4, 4, 5\rangle$ after step 1 and $\langle 1, 0, 2, 3, 0, 4, 5\rangle$ after step 5. Note that the computation in the protocol can be parallelized, and each element of the output is computed independently of others. While this protocol provides the most basic version, we subsequently describe how the size of the set C can be reduced to contain only non-zero elements (the actual members of the union) if desired.

Set intersection. Following the set union logic, we could implement our protocol for set intersection in a similar manner. This time, after sorting the combined set of size $m = m_1 + m_2$, we wish to erase (i.e., set to 0) each distinct element once (note that there will be either one or two instances of each distinct element). In its simplest form, in the protocol we could compare two consecutive elements x_i and x_{i+1} in the sorted set and keep x_i if they are equal. [47], however, notice that the size of the output set can be reduced in half if instead we compare each even element of the sorted set to its adjacent elements. Then if either comparison results in 1, we keep the current element and otherwise set it to 0. The output consists of only even elements, which gives us $\lfloor m/2 \rfloor$ elements in the output set. Implementing this logic in our framework results in similar (in fact, slightly more efficient) performance compared to the simpler logic, but the output size is reduced in half, which improves efficiency of the computations that follow. We also note that from the set operations that we implement in this work, set intersection is the only operation where the output size can be reduced to a fraction of the input set sizes without any knowledge of the inputs by computing values at certain fixed locations.

In our set intersection protocol we implement the logic described above, where we have to make an exception for the last element in case $m = m_1 + m_2$ is even (i.e., in that case the element at position m is compared only to its predecessor at position $m-1$). For any given element x_{2i} of the sorted set, let u_i denote the result of the comparison of x_{2i} with x_{2i-1} and v_i denote the result of x_{2i}'s comparison with x_{2i+1}. Then to compute the corresponding element of the output set c_i, we need to multiply x_{2i} with the OR of u_i and v_i. In general, Boolean OR $a \vee b$ can be implemented as $a + b - ab$, but we note that in our case u_i and v_i will never be simultaneously 1. This means that the sum $u_i + v_i$ will correspond to their OR, reducing the number of interactive operations. As before, computing all elements of the result $A \cap B$ proceeds in parallel, which is of grand importance because the size of A and B can be very large. For our example input sets, the protocol outputs $\langle 2, 0, 4 \rangle$.

Protocol 2. $[c_1], \ldots, [c_{\lfloor m/2 \rfloor}] \leftarrow \mathsf{Int}([a_1], \ldots, [a_{m_1}], [b_1], \ldots, [b_{m_2}])$

```
1. $[x_1], \ldots, [x_m] \leftarrow \mathsf{Sort}([a_1], \ldots, [a_{m_1}], [b_1], \ldots, [b_{m_2}], \ell)$;     // section 3.2
2. for $i = 1$ to $\lfloor (m-1)/2 \rfloor$ do in parallel
3.     $[u_i] \leftarrow \mathsf{Eq}([x_{2i}], [x_{2i-1}], \ell)$;     // section 3.2
4.     $[v_i] \leftarrow \mathsf{Eq}([x_{2i}], [x_{2i+1}], \ell)$;     // section 3.2
5.     $[c_i] \leftarrow ([u_i] + [v_i])[x_{2i}]$;     // 1 round, $\lfloor (m-1)/2 \rfloor$ inv
6. if ($m$ mod $2 = 0$)
7.     $[u_{m/2}] \leftarrow \mathsf{Eq}([x_m], [x_{m-1}], \ell)$;     // section 3.2
8.     $[c_{m/2}] \leftarrow [u_{m/2}][x_m]$;     // 1 inv
9. return $[c_1], \ldots, [c_{\lfloor m/2 \rfloor}]$;
```

Subset relationship. The subset protocol computes whether a given set A is contained in another set B, i.e., $A \overset{?}{\subseteq} B$. It returns a bit which is set to 1 if $A \subseteq B$ and 0 otherwise. The algorithm proceeds by comparing all pairs of adjacent elements in the aggregate sorted array and returns 1 iff the number of elements that were equal is exactly the size of the set A. Note that we run the protocol only when $m_1 \leq m_2$ (assuming no padding in the input sets); otherwise, the output bit is automatically set to 0. For example, for inputs $\langle 2, 4, 1, 5 \rangle$ and $\langle 4, 3, 2 \rangle$, the output is 0 because $m_1 > m_2$. For inputs $\langle 4, 3, 2 \rangle$ and $\langle 2, 4, 1, 5 \rangle$, on the other hand, the protocol is executed, but returns 0 because $t = 2 \neq m_1 = 3$.

Protocol 3. $[s] \leftarrow \mathsf{Sub}([a_1], \ldots, [a_{m_1}], [b_1], \ldots, [b_{m_2}])$

```
1. $[x_1], \ldots, [x_{m_1+m_2}] \leftarrow \mathsf{Sort}([a_1], \ldots, [a_{m_1}], [b_1], \ldots, [b_{m_2}], \ell)$;        // section 3.2
2. for $i = 2$ to $m_1 + m_2$ do in parallel $[u_i] \leftarrow \mathsf{Eq}([x_i], [x_{i-1}], \ell)$;   // section 3.2
3. $[t] \leftarrow \sum_{i=2}^{m_1+m_2} [u_i]$;
4. $[s] \leftarrow \mathsf{Eq}([t], m_1, \lceil \log m_1 \rceil)$;                                       // section 3.2
5. return $[s]$;
```

Utilizing the logic above, we can also derive a similar protocol to compute **set equality**. In that scenario, our first step would be to check if $m_1 = m_2$, as otherwise we can automatically report that the sets are not equal. The rest of the protocol will be exactly the same as the steps of Sub. Similarly, we can also produce a protocol for verifying a **superset relationship** between sets A and B from the logic provided in Protocol 3. In fact, the algorithm need not be changed in this case either, as a subset relationship directly implies an inverse superset relation between the same sets. That is, if the return bit $[s]$ indicates that A is a subset of B, we can conversely say that B is a superset of A. Hence, the two operations can be done interchangeably by simply switching the order in which the sets are passed to Sub.

Set difference. An intuitive solution to computing the set difference $A \setminus B$ is to combine sets A and $A \cap B$, sort the combined set, and eliminate all values that appear twice in the resulting multiset (by erasing both instances). This approach, however, results in running sorting twice (where the second time it is executed on a set of size $2|A| + |B|$) and thus more than doubling the overhead compared to other protocols. Our solution instead is to label the elements of the two sets with opposite bits which will allow us to perform this operation using a single sort. In detail, we associate a 0 bit with all elements of set A and a bit with value 1 with the elements of B and sort the concatenation of these tuples. After sorting, we compare (in parallel) each element of the sorted set to its successor and store the results into a bit vector u. Based on these results, the protocol will then erase (set to 0) each pair of elements that have the same value, while keeping those that have unique values unchanged. To erase both instances of duplicate elements, we can compute values c_i's as

$$[c_i] \leftarrow [x_i](1 - [u_i]); \quad [c_{i+1}] \leftarrow [x_{i+1}](1 - [u_i]);$$

for each i, where x_i's represent the previously sorted concatenation of the elements of A and B. Although this logic can be safely realized when the computation is executed sequentially, it needs to be modified if we want it to be parallelized. To achieve this, we make sure that the value of each c_i in the resulting set depends on the result of the comparison of x_i with x_{i-1} and x_{i+1}, and each c_i is set only once. In particular, we set c_i to 0 if either u_{i-1} or u_i is true and it is set to x_i otherwise. Similar to the OR computation in the set intersection, because at most one of u_{i-1} and u_i can be set for each value of i, the OR computation is performed as $u_{i-1} + u_i$ instead of full $u_{i-1} + u_i - u_{i-1}u_i$.

Finally, as the last step of the protocol we compute the elements c_i's of the set difference $A \setminus B$ by erasing all elements of B that still remain. This is accomplished using the second element of each tuple of the sorted set, which stores information about the input set from which the value originated. For the example inputs $\langle 2, 4, 1, 5 \rangle$ and $\langle 4, 3, 2 \rangle$, the protocol produces $c = \langle 1, 0, 0, 3, 0, 0, 5 \rangle$ after step 5 and $c = \langle 1, 0, 0, 0, 0, 0, 5 \rangle$ after step 6.

Protocol 4. $[c_1], \ldots, [c_{m_1+m_2}] \leftarrow \mathsf{Diff}([a_1], \ldots, [a_{m_1}], [b_1], \ldots, [b_{m_2}])$

```
1. $\langle [x_1], [y_1] \rangle, \ldots, \langle [x_{m_1+m_2}], [y_{m_1+m_2}] \rangle \leftarrow \mathsf{SortT}(\langle [a_1], [0] \rangle, \ldots, \langle [a_{m_1}], [0] \rangle, \langle [b_1], [1] \rangle, \ldots, \langle [b_{m_2}], [1] \rangle,
   $\ell$);                                                                                  // section 3.2
```

```
2. for $i = 1$ to $m_1 + m_2 - 1$ do in parallel $[u_i] \leftarrow \mathsf{Eq}([x_i], [x_{i+1}], \ell)$;    // section 3.2
3. $[c_1] \leftarrow [x_1](1 - [u_1])$;    // 1 round, 1 inv
4. $[c_{m_1+m_2}] \leftarrow [x_m](1 - [u_{m_1+m_2-1}])$;    // 1 inv
5. for $i = 2$ to $m_1 + m_2 - 1$ do in parallel $[c_i] \leftarrow [x_i](1 - [u_i] - [u_{i-1}])$;    // $m_1 + m_2 - 2$ inv
6. for $i = 1$ to $m_1 + m_2$ do in parallel $[c_i] \leftarrow [c_i](1 - [y_i])$;    // 1 round, $m_1 + m_2$ inv
7. return $[c_1], \ldots, [c_{m_1+m_2}]$;
```

Symmetric difference. Given two sets A and B, symmetric difference $A \Delta B$ computes the elements that belong to either of the sets while not being common to both. A naive approach to implementing the operation would be to compose a new protocol that computes $(A \cup B) \setminus (A \cap B)$. To improve efficiency, however, this operation can be done directly by modifying the above set difference protocol. Note that the last step of Protocol 4 (line 6) removes from the resulting set the elements of B that are not part of the intersection. Hence, by not executing that operation, we automatically obtain the symmetric difference protocol SDiff. This also implies that the SortT routine on line 1 can be replaced by regular sorting.

Element reduction. Element reduction is applied to a single multiset A, during which one instance of each distinct element is erased. The logic for its implementation is very similar to that of the intuitive implementation of set intersection (which we mention but do not use), but now each distinct element can appear any number of times in the sorted combined set instead of only once or twice. We therefore erase the first instance of each distinct element. This is implemented by comparing two adjacent elements x_i and x_{i+1} in the sorted multiset and setting the element at position $i + 1$ in the result, c_{i+1}, to 0 iff x_i and x_{i+1} differ (i.e., x_{i+1} is a new distinct element). For correctness, the first element c_1 is always set to 0. For example, if the sorted input is $\langle 1, 2, 2, 3, 4, 5, 5, 5 \rangle$, the protocol outputs $\langle 0, 0, 2, 0, 0, 0, 5, 5 \rangle$. As before, computation of each element of the resulting multiset can proceed in parallel.

Protocol 5. $[c_1], \ldots, [c_m] \leftarrow \mathsf{Red}([a_1], \ldots, [a_m])$

```
1. $[x_1], \ldots, [x_m] \leftarrow \mathsf{Sort}([a_1], \ldots, [a_m], \ell)$;    // section 3.2
2. $[c_1] \leftarrow 0$;
3. for $i = 1$ to $m - 1$ do in parallel
4.     $[u_i] \leftarrow \mathsf{Eq}([x_i], [x_{i+1}], \ell)$;    // section 3.2
5.     $[c_{i+1}] \leftarrow [u_i][x_{i+1}]$;    // 1 round, $m - 1$ inv
6. return $[c_1], \ldots, [c_m]$;
```

4.2 Protocol variants

The above protocols implement the basic functionality of multi-party set operations. In this section we show how they can be modified or extended to enable a number of new features.

Opening the result of a (multi)set operation. The output of the protocols presented in section 4.1 cannot be safely opened without leaking information about their inputs because the locations of erased items will be revealed. If the result is to be opened (e.g., when one of the above operations is the last operation in the computation), the parties will need to additionally sort the result, or randomly permute it, prior to the opening to hide all patterns. To do so, the last line of each protocol in section 4.1 should be changed from

return $[c_1], \ldots, [c_k]$;

to

return $\mathsf{Sort}([c_1], \ldots, [c_k], \ell)$;

for the appropriate value of k. In section 7 we also show how this step can be performed more efficiently using set compaction.

Reducing the size of the result of a (multi)set operation. The way our protocols are specified does not reveal the size of the resulting set or multiset. In certain cases, however, for efficiency reasons it is desirable to reveal the size of the output and eliminate all extra elements. We distinguish between these two modes of computation by referring to them as length-hiding and length-preserving, respectively. To perform a length-preserving operation, the parties follow each protocol as defined in section 4.1, after which they sort the outcome and discard zero elements by comparing each of them to 0 and opening the result of the comparison. More precisely, each "return $[c_1], \ldots, [c_k]$" operation (for the appropriate value of k) in the original protocol now needs to be replaced with the following:

```
1. $[d_1], \ldots, [d_k] \leftarrow \mathsf{Sort}([c_1], \ldots, [c_k], \ell)$;          // section 3.2
2. $S \leftarrow \emptyset$;
3. for $i = 1$ to $k$ in parallel
4.     $[b] \leftarrow \mathsf{Eq}([d_i], 0, \ell)$;                                 // section 3.2
5.     $b \leftarrow \mathsf{Open}([b])$;                                          // 1 round, 1 inv
6.     if $(b = 0)$ $S \leftarrow S \cup \{[d_i]\}$;
7. return $S$;
```

The Open operation corresponds to broadcasting the shares of its argument, so that all parties can reconstruct its value. As before, faster compaction can be used instead of sorting.

Computing (multi)set cardinality or over-the-threshold cardinality. Our basic protocols for set operations compute the resulting set, while in certain applications different information such as set cardinality needs to be computed. It is, however, rather straightforward to modify our protocols to instead compute the cardinality (e.g., $|A \cap B|$ for set intersection) or over-the-threshold cardinality (e.g., $|A \cap B| \overset{?}{\geq} T$ for set intersection and threshold T) of the resulting set. For completeness, we next describe such modifications, which give us even simpler protocols than the original versions.

To compute set union cardinality, it is no longer necessary to compute the c_i's in the Union protocol. Instead, it suffices to compute only the number of elements that differ from the next adjacent element in the combined sorted set $x_1, \ldots, x_{m_1+m_2}$. In particular, we replace lines 2–6 in Union with the following computation:

```
2. for $i = 1$ to $m_1 + m_2 - 1$ do in parallel $[u_i] \leftarrow \mathsf{Eq}([x_i], [x_{i+1}], \ell)$;     // section 3.2
3. return $m_1 + m_2 - \sum_{i=1}^{m_1+m_2-1} [u_i]$;
```

The set union over-the-threshold cardinality can likewise compute and return $\mathsf{GE}(m_1 + m_2 - \sum_{i=1}^{m_1+m_2-1} [u_i], T, \ell)$.

The set intersection cardinality and the cardinality of a multiset after element reduction follow a similar logic, where now the parties compute and return $\sum_{i=1}^{\lfloor m/2 \rfloor} [u_i] + \sum_{i=1}^{\lfloor m/2 \rfloor} [v_i]$ and $\sum_{i=1}^{m-1} [u_i]$, respectively. The over-the-threshold versions are formed analogously.

To compute the set difference cardinality, the parties need to produce the count of the number of elements that do not get erased from the resulting set. This can be achieved by replacing lines 3–7 of the Diff protocol with the following:

```
3. return $m_1 - \sum_{i=1}^{m_1+m_2-1} [u_i]$;
```

Finally, the symmetric difference cardinality can be obtained by replacing lines 3–7 of the Diff protocol with the following:

3. return $m_1 + m_2 - 2\sum_{i=1}^{m_1+m_2-1}[u_i]$;

As before, the over-the-threshold cardinality version is produced analogously.

Performing set operations on multiple input sets. Our protocols have been defined to work on two input sets, while existing literature on multi-party set operations considers the problem of computing set intersection or union of n input sets with n participating parties. Here we show that it is not difficult to modify our set union, intersection, and equality protocols to work on k inputs for any $k \geq 2$ (i.e., k may or may not depend on n). We consider only these three set operations as we are not aware of a standard way of defining other operations on multiple input sets.

First, observe that a protocol for multiple-input set union $[c_1], \ldots, [c_m] \leftarrow \mathsf{Union}([a_1^{(1)}], \ldots, [a_{m_1}^{(1)}], \ldots, [a_1^{(k)}], \ldots, [a_{m_k}^{(k)}])$, where $C = \bigcup_{i=1}^{k} A^{(i)}$, $A^{(i)} = \{a_1^{(i)}, \ldots, a_{m_i}^{(i)}\}$ for $i = 1, \ldots, k$, and $m = \sum_{i=1}^{k} m_i$, can be obtained from the original Protocol 1 with virtually no changes. The only obvious difference is that the step 1 now consists of sorting the concatenation of all of the $A^{(i)}$'s, i.e., $[x_1], \ldots, [x_m] \leftarrow \mathsf{Sort}([a_1^{(1)}], \ldots, [a_{m_1}^{(1)}], \ldots, [a_1^{(k)}], \ldots, [a_{m_k}^{(k)}], \ell)$. As before, the algorithm keeps a single instance of each present distinct value and eliminates the rest.

In order to implement multiple-input set intersection $[c_1], \ldots, [c_{\lceil (m-1)/k \rceil}] \leftarrow \mathsf{Int}([a_1^{(1)}], \ldots, [a_{m_1}^{(1)}], \ldots, [a_1^{(k)}], \ldots, [a_{m_k}^{(k)}])$, where now $C = \bigcap_{i=1}^{k} A^{(i)}$, the algorithm in Protocol 2 needs to be modified. This time we would like to keep only the elements that appear exactly k times in the sorted array. To do that, instead of checking two consecutive elements, we need to compare two elements $k-1$ positions apart. Similar to Protocol 2, instead of producing a set of size m, this time we output a set of size $\lceil (m-1)/k \rceil$ and the OR of multiple bits from which at most one is set is computed as their sum. More precisely, we obtain:

Protocol 6. $[c_1], \ldots, [c_{\lceil (m-1)/k \rceil}] \leftarrow \mathsf{Int}([a_1^{(1)}], \ldots, [a_{m_1}^{(1)}], \ldots, [a_1^{(k)}], \ldots, [a_{m_k}^{(k)}])$

```
1. $[x_1], \ldots, [x_m] \leftarrow \mathsf{Sort}([a_1^{(1)}], \ldots, [a_{m_1}^{(1)}], \ldots, [a_1^{(k)}], \ldots, [a_{m_k}^{(k)}], \ell)$;      // section 3.2
2. for $i = 1$ to $m - k + 1$ do in parallel $[u_i] \leftarrow \mathsf{Eq}([x_i], [x_{i+k-1}], \ell)$;      // section 3.2
3. $d \leftarrow \lfloor (m-1)/k \rfloor$;
4. for $i = 1$ to $d$ do in parallel $[c_i] \leftarrow \sum_{j=1}^{k}([u_{(i-1)k+j}] \cdot [x_{(i-1)k+j}])$;      // 1 round, $d \cdot k$ inv
5. if $((m-1) \bmod k \neq 0)$ $c_{\lceil (m-1)/k \rceil} \leftarrow \sum_{j=1}^{(m-1) \bmod k}[u_{d \cdot k+j}][x_{d \cdot k+j}]$;      // $(m-1) \bmod k$ inv
6. return $[c_1], \ldots, [c_{\lceil (m-1)/k \rceil}]$;
```

Lastly, to obtain a set equality protocol that works on multiple input sets, we only need to sort the concatenation of all k sets and compare the elements of the sorted set $k-1$ positions apart instead of the original 1 position apart in the Sub protocol. It is obvious that $m_i = m_j$ must also hold for every i and j.

4.3 Length-hiding set operations

Recall that our original protocols do not reveal any information about the size of the resulting set. To enable their use in the full length-hiding mode, we need to make sure that our protocols work correctly when the length of the input sets is also protected. To hide the actual length of a set, one adds to that set a number of additional elements that have value 0. In this framework, for instance, all sets can be padded to be of the same size (or one of few fixed sizes). It remains to show that correctness of our protocols is preserved when the input sets contain dummy zero elements. We consider each protocol in turn.

In the Union protocol, after the first step, all dummy elements will occupy the lowest indices in the sorted set which we denote 1 through s. During the loop execution, the zero elements will be set to 0 again, which has no effect on the result of the operation. The only place where a care needs to be exercised is during comparison of zero element x_s and the next non-zero element x_{s+1}. Notice that in the Union protocol, the result of computing $\mathsf{Eq}([x_s], [x_{s+1}], \ell)$ has no effect on x_{s+1}. We therefore obtain that the output of the protocol will be correct regardless of the number of regular elements contained in the sets A and B (including the case when A and B are entirely composed of dummy elements). By applying similar reasoning to other protocols, we obtain that regardless of whether zero elements are reset to zero or their values are preserved, the result of the computation is not affected. In the intersection protocol Int, we have that computation "at the border" of dummy and regular elements, namely, x_s and x_{s+1}, can possibly affect x_{s+1} only when $s+1$ is even, but we see that in that case $c_{(s+1)/2}$ will be set correctly to the result of the comparison of x_{s+1} and x_{s+2}. Thus, the protocol works as expected on padded inputs. In the element reduction protocol Red, we can also see that the result of $\mathsf{Eq}([x_s], [x_{s+1}], \ell)$ will be 0 and x_{s+1} will be set to 0 as required. Finally, in the set difference protocols Diff and SDiff protocols, the value of u_s will be 0 as well and therefore will not affect the correctness of the value of c_{s+1}.

The only protocol that cannot be executed as previously described on padded inputs is subset relationship Sub. In contrast to other protocols that erase elements from the input sets, the subset protocol counts the number of matched elements (which the padding can increase) and requires the knowledge of the input set size. We therefore next describe a more elaborate version of Sub protocol that works on padded input sets.

In the protocol below, we preserve information about the origin of each element during sorting (note that elements from set A are marked with bit 1). After comparing the adjacent elements of the sorted set, we prepend the array of computed bits u_i with 1 if the first element of the sorted set is 0, and with 0 otherwise. Now notice that if the sorted set contains k zero elements (which will precede all other elements), $u_1 = \ldots = u_k = 1$, while $u_{k+1} = 0$. Thus, if we perform prefix-AND on bits $u_1, \ldots, u_{m_1+m_2}$, the output will consist of k 1's followed by $m_1 + m_2 - k$ 0's. This gives us a mechanism to identify all zero elements within the sorted set. We then count the number of non-zero elements in A and store the value in t_1, and count the number of matches between non-zero elements in A and B and store the value in t_2. If the values are the same, the protocol outputs 1, and otherwise it outputs 0.

Protocol 7. $[s] \leftarrow \mathsf{Sub}([a_1], \ldots, [a_{m_1}], [b_1], \ldots, [b_{m_2}])$

```
1. $\langle [x_1], [y_1] \rangle, \ldots, \langle [x_{m_1+m_2}], [y_{m_1+m_2}] \rangle \leftarrow \mathsf{SortT}(\langle [a_1], [1] \rangle, \ldots, \langle [a_{m_1}], [1] \rangle, \langle [b_1], [0] \rangle, \ldots, \langle [b_{m_2}], [0] \rangle,$
   $\ell)$;                                                          // section 3.2
2. for $i = 2$ to $m_1 + m_2$ do in parallel $[u_i] \leftarrow \mathsf{Eq}([x_i], [x_{i-1}], \ell)$;   // section 3.2
3. $[u_1] \leftarrow \mathsf{Eq}([x_1], 0, \ell)$;                                        // section 3.2
4. $([v_1], \ldots, [v_{m_1+m_2}]) \leftarrow \mathsf{PreAND}([u_1], \ldots, [u_{m_1+m_2}])$;          // section 3.2
5. $[t_1] \leftarrow \sum_{i=1}^{m_1+m_2} ([y_i](1 - [v_i]))$;                       // 1 round, $m_1 + m_2$ inv
6. $[t_2] \leftarrow \sum_{i=1}^{m_1+m_2} ([u_i] - [v_i])$;
7. $[s] \leftarrow \mathsf{Eq}([t_1], [t_2], \lceil \log m_1 \rceil)$;                          // section 3.2
8. return $[s]$;
```

This protocol also computes set equality when $m_1 = m_2$.

We conclude that all our protocols except Sub can be used unmodified on inputs padded with zero elements so that the size of both the input and output sets is protected. For the subset operation, Protocol 7 should be used instead of Protocol 3.

4.4 Security

Correctness of the computation has been discussed with each respective protocol. We only comment on the performance of randomized sorting algorithms, and randomized shellsort [39] in particular, that can fail to sort the input with a small probability. In our context, failure to sort the input can potentially become a security leak that reveals some information about the input sets. Toward this end, we note that the algorithm of [39] can fail with probability at most $1/m^b$ for some $b \geq 1$ and has not failed on any tested input. Furthermore, increasing the number c of region compare-exchange operations can be used to reduce the probability of failure to the desired $1/2^\kappa$ for a security parameter κ, which will result in statistical security. Lastly, we note that our protocols can run in $O(m \log m)$ time even without using randomized sorting algorithms by employing optimizations described in section 7. Security of our protocols can be shown as follows:

Theorem 4.1 *The above set operations protocols are t-private in presence of semi-honest participants with private channels with $t < n/2$.*

Proof First, note that the (n, t)-threshold linear secret sharing scheme achieves perfect secrecy in presence of collusions of size at most $t \leq n$ (i.e., zero information can be learned about secret-shared values by t or fewer parties) in the case of passive adversaries. Also, the multiplication operation does not reveal any information when $t < n/2$ (see, e.g., [3] for a formal security proof). Furthermore, because most other building blocks used in this work (i.e., Eq, GE, and PreAND) have been previously shown to be secure, information is not revealed during their execution as well. Their most efficient implementations are statistically secure (as opposed to perfectly secure) for any desired security parameter κ. Then if our protocols call only secure building blocks, the security of the overall protocols will follow. In particular, by Canetti's composition theorem [12], (sequential) composition of secure sub-protocols results in security of the overall solution.

More formally, to comply with the security definition 1, we need to build a simulator S_I for each protocol that can simulate the views of the corrupted parties I using their inputs and outputs in a way which is indistinguishable from real protocol execution. We can easily build this simulator by invoking simulators for the corresponding building blocks to simulate views for the entire protocol. The resulting views are guaranteed to be indistinguishable from the real protocol execution by the participants.

The only missing piece is security of Sort protocol. First, note that any candidate sorting algorithm suitable for use in secure computation consists of a sequence of compare-and-exchange operations. Each compare-and-exchange operation consists of a comparison GE, multiplications, and additions/subtractions as shown in section 3.2. We thus can easily build a simulator for it by invoking the corresponding simulators for the underlying operations. Second, we employ only oblivious sort, in which the sequence of compare-and-exchange operations is input-independent and therefore cannot leak information about the input. Thus, security of the overall Sort follows from the security of compare-and-exchange operations where we invoke the corresponding simulator the necessary number of times.

Lastly, we mention that the extension to set operation protocols that allows the parties to learn information about the actual size of the resulting set is also secure, because in this case both the function f and our protocol π reveal this information. □

Before we proceed with security in presence of malicious participants, we note that when the building blocks Eq, GE, and PreAND are perfectly secure (i.e., implemented as arithmetic circuits), all of our protocols are perfectly secure as well. It then follows from [14, 3] that security in presence of adaptive adversaries comes "for free," and the protocols are secure in presence of both static and adaptive adversaries (this applies to the malicious setting as well). When, however, the building blocks are statistically secure, according to [14] static and adaptive models are equivalent when the number of computational parties is small (as a function of the security parameter), e.g., fixed, which means that we also automatically obtain security against adaptive adversaries.

To show security in presence of malicious adversaries, we need to ensure that (i) all participants prove that their input is well-formed, (ii) all participants comply with the prescribed computation by proving that each step was performed correctly, and (iii) if some dishonest participants quit, others will be able to reconstruct their shares and proceed with the rest of the computation. When the computation corresponds to an arithmetic circuit, (ii) and (iii) are normally achieved using a verifiable secret sharing scheme (VSS), and a large number of results have been developed over the years (e.g., [36, 18, 44, 45, 7, 26, 24, 25] and many others). When, however, the participants are expected to additionally perform other operations, we need to employ the corresponding zero-knowledge proofs of knowledge. Similarly, if the input has a specified form, zero-knowledge proofs will need to be employed.

When each input is a set (as opposed to a multiset), each element needs to be unique. Therefore, to ensure correctness of a set operation, the participants need to verify this property prior to execution of the operation. To minimize the overhead associated with such verification, we suggest the following approach: on input two or more sets, the participants sort each set separately, then verify that the difference between two consecutive elements in each sorted set is non-zero, merge the sorted sets, and proceed with the rest of the operation as before. Then if any observed value is zero, the participants abort the protocol. For example, if the input consists of two sets $a_1, \ldots, a_{m_1}$ and $b_1, \ldots, b_{m_2}$, we replace sorting $\mathsf{Sort}([a_1], \ldots, [a_{m_1}], [b_1], \ldots, [b_{m_2}], \ell)$ in any protocol with the following steps:

```
 1. $[x_1], \ldots, [x_{m_1}] \leftarrow \mathsf{Sort}([a_1], \ldots, [a_{m_1}], \ell)$;
 2. $[y_1], \ldots, [y_{m_2}] \leftarrow \mathsf{Sort}([b_1], \ldots, [b_{m_2}], \ell)$;
 3. for $i = 1$ to $m_1 - 1$ do in parallel
 4.     $[c_i] \leftarrow \mathsf{Eq}([x_{i+1}] - [x_i], 0, \ell)$;
 5.     $c_i \leftarrow \mathsf{Open}([c_i])$;
 6.     if $c_i = 1$, output $\perp$;
 7. for $i = 1$ to $m_2 - 1$ do in parallel
 8.     $[c'_i] \leftarrow \mathsf{Eq}([y_{i+1}] - [y_i], 0, \ell)$;
 9.     $c'_i \leftarrow \mathsf{Open}([c'_i])$;
10.     if $c'_i = 1$, output $\perp$;
11. $[z_1], \ldots, [z_{m_1+m_2}] \leftarrow \mathsf{Merge}(([x_1], \ldots, [x_{m_1}]), ([y_1], \ldots, [y_{m_2}]), \ell)$;
```

Clearly, opening the values on lines (5) and (9) does not reveal any information about the private values and is the exactly the condition that the participants want to verify.

When padding is used, each input set is allowed to have multiple instances of zero elements.[2] In such a case, the difference between two consecutive elements in a sorted set is allowed to be zero as long as the elements are zero. We then modify the above verification to work with padded sets as follows: now the participants privately compare each element of the sorted set to 0, privately compare each difference between two consecutive elements of the sorted set to 0, and open the value of the form $(d_i \neq 0) \vee ((d_i = 0) \wedge (x_i = 0))$ for each position i, where x_i denotes the ith element of the sorted set and d_i the difference between x_i and x_{i+1}. Let u_i denote the result of comparison of x_i to 0 and v_i the result of comparison $x_{i+1} - x_i$ to 0. The participants then compute and open

[2]The verification algorithm described above does not enforce absence of padding, which is normally not needed. If, however, the participants want to ensure that no zero elements are present, they can simply compare the first element of the sorted set to 0 and open the result of the comparison.

value $v_i u_i + 1 - v_i$ for each i and abort if any of the opened values is 0. It is straightforward to modify the verification steps given above for sets with no padding to incorporate the computation of the u_i's and opening a modified expression on lines (5) and (9). It is interesting to note that input verification is not needed when inputs are multisets since the inputs can be arbitrary.

Theorem 4.2 *Given a $(n, n/3)$-VSS scheme with support for multiplication, generating a random field element, and opening a secret-shared value, the above set operations protocols are t-secure in presence of malicious participants with private channels with $t < n/3$.*

Proof When the overall computation corresponds to an arithmetic circuit, all we need to obtain security in presence of malicious participants is to employ a VSS scheme which ensures that (i) each multiplication protocol is performed correctly, (ii) each input is secret-shared correctly in case the dealer is corrupt, and (iii) a secret can be properly reconstructed from it shares (when not already implied by the above). There are many such results for a variety of settings and assumptions, normally for $t < n/3$, and we in particular mention the result of [4] which provides perfect security with $t < n/3$. Then if at any point of the computation the participants are required to input values of a specific form, they would have to prove that the values they supplied are well formed. For our constructions such proofs are necessary only if statistically secure building blocks (Eq, LE, and PreAND) are used, where the computational parties need to supply private random values of a specific length. While enforcement of this constraint can be performed by using a range proof from prior literature, e.g., [61], we propose an alternative solution that avoids computational assumptions. In particular, when using Eq, LE, and PreAND from [15], collectively choosing a random bit by the computational parties (using protocol RandBit) involves only generating a random field element that VSS techniques already cover. Then to generate a random value of bitlength k, the parties can call RandBit k times in parallel obtaining bits $b_0, \ldots, b_{k-1}$, after which each of them locally computes $r = \sum_{i=0}^{k-1} 2^i b_i$. We thus obtain that the security of our protocols in the malicious model follows from VSS techniques (e.g., [4, 36, 19]) when either perfectly secure or fast statistically secure implementations of the building blocks from [15] are used. $\square$

Note that in the malicious model the complexity of RandInt algorithm increases, which now uses $O(k)$ interactive operations to generate a k-bit random value instead of being local using PRSS in the semi-honest setting. This slightly increases the overall number of interactive operations, but has no effect on the asymptotic complexity of set operations.

As mentioned before, security in presence of adaptive and static participants in the malicious model are equivalent for perfectly secure protocols [14] and in that setting we automatically gain security in presence of adaptive adversaries. Then security in presence of adaptive adversaries can only be obtained if the (statistically secure) building blocks are proven secure in the adaptive adversarial model. Lastly, security under concurrent general composition [57] (or, equivalently, universal composability [13]) is also free in the information-theoretic setting according to [54]. That is, every perfectly secure protocol in the stand-alone setting is also secure under concurrent general composition, and every statistically secure protocol in the stand-alone setting can be easily modified to be secure under concurrent general composition (by adding the so-called start synchronization to ensure that all inputs are ready before the computation starts).

5 General Conversion from a Multiset to a Set

Our previous protocols do not work correctly when they are run on multisets. To enable computation on multisets, we describe a general conversion from a multiset to a set, which will allow all previous protocols to be run on multisets with only notational changes.

Our solution converts a multiset $a_1, \ldots, a_m$ to a representation $\langle x_1, y_1 \rangle, \ldots, \langle x_m, y_m \rangle$, where x_i's correspond to the a_i's, and indices y_i's count the number of instances of each distinct value in the multiset. That is, if a value v appears k times in the multiset, the indices of the corresponding elements in the multiset will be numbered 1 through k. This makes each pair $\langle x_i, y_i \rangle$ unique and our protocols for set operations apply. The multiset-to-set protocol below illustrates how this multiset representation can be computed.

Protocol 8. $\langle [x_1], [y_1] \rangle, \ldots, \langle [x_m], [y_m] \rangle \leftarrow \mathsf{M2S}([a_1], \ldots, [a_m])$

```
1. $[x_1], \ldots, [x_m] \leftarrow \mathsf{Sort}([a_1], \ldots, [a_m], \ell)$;        // section 3.2
2. $[y_1] \leftarrow 1$;
3. for $i = 1$ to $m - 1$ do
4.     $[u_i] \leftarrow \mathsf{Eq}([x_i], [x_{i+1}], \ell)$;                   // section 3.2
5.     $[y_{i+1}] \leftarrow [u_i][y_i] + 1$;                                  // $m-1$ rounds, $m-1$ inv
6. return $\langle [x_1], [y_1] \rangle, \ldots, \langle [x_m], [y_m] \rangle$;
```

In this protocol, the indices y_i have to be computed sequentially. In the attempt to design an algorithm that does not require the number of rounds to be linear in the size of the multiset, we resort to the techniques that were used in [23] to design constant-round protocols for other integer arithmetic operations. In particular, suppose we are given an associative binary operator $\circ$. Also suppose that we can securely compute this operation on m inputs $\circ_{i=1}^{m}[a_i]$ in R rounds and $C(m)$ operations. Given this, [16] describe a method for computing prefix-$\circ$, $\mathsf{Pre}_\circ$, that uses $2R$ rounds and $\sum_{i=1}^{\log_2 m} 2^i C(m \cdot 2^{-i}) + mC(\log_2 m) \le \log_2 mC(m) + mC(\log_2 m)$ operations. Secure prefix-$\circ$ functionality is defined as $([b_1], \ldots, [b_m]) \leftarrow \mathsf{Pre}_\circ([a_1], \ldots, [a_m])$, where $b_i = \circ_{j=1}^{i} a_j$. In the context of Protocol 8, this means that if we define a procedure for computing $\langle [x_m], [y_m] \rangle = \circ_{i=1}^{m}[a_i]$ in the multiset-to-set conversion using R rounds, we will be able to use their method to compute all $\langle [x_i], [y_i] \rangle$ as $(\langle [x_1], [y_1] \rangle, \ldots, \langle [x_m], [y_m] \rangle) \leftarrow \mathsf{Pre}_\circ([a_1], \ldots, [a_m])$ in $2R$ rounds.

Before we proceed with further description, we need to specify the operator $\circ$ used to perform the conversion. The $\mathsf{M2S}$ protocol can be viewed as starting with individual elements, each with count 1, and aggregating the first i of them to compute the count at position i. Because the operator must work on "individual" and "aggregate" values, we define it as:

$\langle [c_1], [c_2] \rangle \leftarrow \langle [a_1], [a_2] \rangle \circ \langle [b_1], [b_2] \rangle$

```
1. $[u] \leftarrow \mathsf{Eq}([a_1], [b_1], \ell)$;        // section 3.2
2. $[c_1] \leftarrow [b_1]$;
3. $[c_2] \leftarrow [u][a_2] + [b_2]$;                    // 1 round, 1 inv
4. return $\langle [c_1], [c_2] \rangle$;
```

The above assumes that the operands are well-formed, i.e., $b_1 \ge a_1$. We refer to this operation as addition with reset, i.e., the count is reset if the value of the current multiset element has changed, and the count is incremented otherwise. The operator can be shown to be associative.

To be able to use the method from [16] for computing $\mathsf{Pre}_\circ([a_1], \ldots, [a_m])$ using a solution to $\circ_{i=1}^{m}[a_i]$, we need a constant round procedure for computing $\circ_{i=1}^{m}[a_i]$, where $a_i = \langle x_i, y_i \rangle$. We realize it as shown below. Note that in this protocol each y_i can be an arbitrary count (i.e., if $y_i > 1$, the pair $\langle x_i, y_i \rangle$ corresponds to an "aggregate" of several multiset elements with the same value), but the x_i's must form a non-decreasing sequence.

Protocol 9. $\langle [x], [y] \rangle \leftarrow \circ_{i=1}^{m} \langle [x_i], [y_i] \rangle$

```
1. for $i = 1$ to $m-1$ do in parallel $[u_i] \leftarrow \mathsf{Eq}([x_i],[x_{i+1}],\ell)$;      // section 3.2
2. $([v_{m-1}],\ldots,[v_1]) \leftarrow \mathsf{PreAND}([u_{m-1}],\ldots,[u_1])$;      // section 3.2
3. for $i = 1$ to $m-1$ do in parallel $[w_i] \leftarrow [v_i][y_i]$;      // 1 round, $m-1$ inv
4. $[y] \leftarrow [y_m] + \sum_{i=1}^{m-1}[w_i]$;
5. $[x] \leftarrow [x_m]$;
6. return $\langle [x],[y]\rangle$;
```

In the protocol above, as a result of prefix-AND in step 2, we obtain an array of bits $v_{m-1}, \ldots, v_1$, where v_i is set to 1 iff all elements x_i through x_m are equal. This allows us to count the number of elements in the input which have the same value as x_m. Their corresponding counts are added together in step 4 and are returned as the count for the entire set. This computation in particular implies that if $x_m > x_{m-1}$, then the pair $\langle x_m, y_m\rangle$ will be returned as required. This protocol allows us to obtain a new solution for multiset-to-set conversion where the round complexity is the round complexity of sorting plus a small constant.

Protocol 10. $\langle [x_1],[y_1]\rangle,\ldots,\langle [x_m],[y_m]\rangle \leftarrow \mathsf{M2S}([a_1],\ldots,[a_m])$

```
1. $[x'_1],\ldots,[x'_m] \leftarrow \mathsf{Sort}([a_1],\ldots,[a_m],\ell)$;      // section 3.2
2. for $i = 1$ to $m$ do in parallel $[y'_i] \leftarrow 1$;
3. $\langle [x_1],[y_1]\rangle,\ldots,\langle [x_m],[y_m]\rangle \leftarrow \mathsf{Pre}_\circ(\langle [x'_1],[y'_1]\rangle,\ldots,\langle [x'_m],[y'_m]\rangle)$;      // Protocol 9
4. return $\langle [x_1],[y_1]\rangle,\ldots,\langle [x_m],[y_m]\rangle$;
```

This concludes our description of the conversion. To illustrate how it can be used to perform multiset operations, we sketch a solution for multiset union $A \cup B$. It assumes that the input multisets are already available in the proper format with numbered instances of each distinct value. This can be achieved by executing the conversion protocol twice as $\langle [x'_1],[y'_1]\rangle,\ldots,\langle [x'_{m_1}],[y'_{m_1}]\rangle \leftarrow \mathsf{M2S}([a_1],\ldots,[a_{m_1}])$ and $\langle [x''_1],[y''_1]\rangle,\ldots,\langle [x''_{m_2}],[y''_{m_2}]\rangle \leftarrow \mathsf{M2S}([b_1],\ldots,[b_{m_2}])$. Alternatively, the input multisets might already be available in the proper format as a result of prior processing. For instance, the output of the multiset union protocol presented next produces an (unsorted) multiset with properly numbered elements. The only exception are zero elements that have been erased as the result of union computation. In particular, their counts are also set to 0 to ensure that such elements do not affect correctness of our protocols during their composition.

Protocol 11. $\langle [x_1],[y_1]\rangle\ldots,\langle [x_{m_1+m_2}],[y_{m_1+m_2}]\rangle \leftarrow \mathsf{MUnion}(\langle [x'_1],[y'_1]\rangle,\ldots,\langle [x'_{m_1}],[y'_{m_1}]\rangle, \langle [x''_1],[y''_1]\rangle, \ldots,\langle [x''_{m_2}],[y''_{m_2}]\rangle)$

```
1. $k \leftarrow \max(m_1,m_2)+1$;
2. $\langle [\alpha_1],[\beta_1],[\gamma_1]\rangle,\ldots,\langle [\alpha_{m_1+m_2}],[\beta_{m_1+m_2}],[\gamma_{m_1+m_2}]\rangle \leftarrow \mathsf{SortT}(\langle k[x'_1]+[y'_1],[x'_1],[y'_1]\rangle,\ldots,\langle k[x'_{m_1}]+$
   $[y'_{m_1}],[x'_{m_1}],[y'_{m_1}]]\rangle, \langle k[x''_1]+[y''_1],[x''_1],[y''_1]\rangle,\ldots,\langle k[x''_{m_2}]+[y''_{m_2}],[x''_{m_2}],[y''_{m_2}]\rangle, \ell+\lceil\log k\rceil)$;
                                                                  // section 3.2
3. for $i = 1$ to $m_1+m_2-1$ do in parallel
4.     $[u_i] \leftarrow \mathsf{Eq}([\alpha_i],[\alpha_{i+1}],\ell+\lceil\log k\rceil)$;      // section 3.2
5.     $[x_i] \leftarrow [\beta_i](1-[u_i])$;      // 1 round, $m_1+m_2-1$ inv
6.     $[y_i] \leftarrow [\gamma_i](1-[u_i])$;      // optional
7. $[x_{m_1+m_2}] \leftarrow [\beta_{m_1+m_2}]$;
8. $[y_{m_1+m_2}] \leftarrow [\gamma_{m_1+m_2}]$;      // optional
9. return $\langle [x_1],[y_1]\rangle,\ldots,\langle [x_{m_1+m_2}],[y_{m_1+m_2}]\rangle$;
```

In the protocol k should be set to a value larger than any y'_i and y''_i (which are bounded by the size of the multisets). In that way, the values will be sorted by the first elements x_i''s and x_i'''s, but in case of their equality, the ties will be resolved – and the tuples will be sorted – by the second elements y_i''s and y_i'''s. The safest way to set k is therefore to use $k = \max(m_1, m_2) + 1$.

As we indicate above, lines 6 and 8 are optional. That is, if the counts for each value do need to be maintained, the protocol returns only $[x_i]$'s. Otherwise, the counts can be computed at low cost (i.e., significantly lower than executing the M2S protocol).

The remaining operations (such as intersection, difference, etc.) can be constructed similarly, and we sketch such protocols in Appendix A. Security of these protocols can be shown analogously to the security of set operations.

6 Direct Operations on Multisets

The previous section described efficient algorithms for private multiset operations using a general multiset-to-set conversion. It is, however, often the case that direct implementations are more efficient than utilizing general procedures. This is true for secure multiset operations as well. In particular, by directly computing a multiset operation, both communication and round complexity is reduced approximately by a factor of two because sorting is used only once instead of calling it once for the conversion procedure for each input multiset and once on the combined multiset for the set operation itself. Therefore in this section we describe our solutions that provide direct implementation of multiset operations.

6.1 Overview of the technique

To be able to perform a multiset operation, we first sort the concatenation of two input multisets in such a way that all elements from the first input set A appear before the elements of the same value from the second input set B. It is accomplished by setting indices associated with the elements of A to 0 and indices associated with the elements of B to 1. We then use values $2a_i + 0$ and $2b_j + 1$ to compare two elements during sorting, where a_i's and b_j's are elements of A and B, respectively. This will ensure that all elements with the same original value will be grouped together in the sorted multiset, but the elements from A appear before the elements with the same value from B. After the sorting, we assign to all elements with the same value counts. The elements from A have counts that start from 1 and increment, while the counts of the elements from B decrement from the highest count of the elements with same value from A. That is, if the first occurrence of a distinct value comes from A, its count is set to 1 (and otherwise it is set to -1). When another element with the same value from A is observed, its count is incremented, but once elements from B with the same value are observed, the count will be decremented after each occurrence. For instance, a sorted combined multiset $\langle 1, 0\rangle, \langle 2, 0\rangle, \langle 2, 0\rangle, \langle 2, 1\rangle, \langle 2, 1\rangle, \langle 2, 1\rangle, \langle 3, 1\rangle$ will be converted to the multiset with counts $\langle 1, 0, 1\rangle, \langle 2, 0, 1\rangle$,
$\langle 2, 0, 2\rangle, \langle 2, 1, 1\rangle, \langle 2, 1, 0\rangle, \langle 2, 1, -1\rangle, \langle 3, 1, -1\rangle$. The first element with a negative count corresponds to an element from B for which there is no matching element from A. Then depending on what operation needs to be performed, either elements with non-negative or elements with negative counts might need to be erased. For instance, to compute multiset union, we erase all elements with non-negative counts from B (i.e., erase the duplicates); to compute multiset intersection, we erase all element of A and all elements of B with negative counts (i.e., those that do not have matching elements from A). Verifying if a subset relation $B \subseteq A$ exists is also very simple and it requires only that we check for negative counts, since that would indicate an unmatched element in B, denoting that a subset relation does not hold.

In order to efficiently calculate the multiset difference, we slightly modify the logic. This time, we associate index 1 with the elements of A and index 0 with the elements of B. We then use $2a_i + 1$ and $2b_j + 0$ for comparisons, which will force the elements of B to precede those from A in the sorted multiset when the values of the elements are equal. As before, following the sorting procedure we assign counts to all elements (note that this time the elements of B will have positive counts and the first distinct occurrences will be given a -1 count if the element belongs to A). After this preprocessing step, we can easily compute $A \setminus B$ by erasing all elements from B along with all elements of A with non-negative counts.

The (non-private) algorithm below for direct multiset union illustrates the logic for this operation, where the sorting procedure *Sort* sorts tuples using their first elements.

Algorithm 1. $c_1, \ldots, c_{m_1+m_2} \leftarrow \mathsf{DMUnion}(a_1, \ldots, a_{m_1}, b_1, \ldots, b_{m_2})$

```
1.  $\langle x_1, y_1, z_1\rangle, \ldots, \langle x_{m_1+m_2}, y_{m_1+m_2}, z_{m_1+m_2}\rangle \leftarrow Sort(\langle 2a_1, a_1, 0\rangle, \ldots, \langle 2a_{m_1}, a_{m_1}, 0\rangle, \langle 2b_1{+}1, b_1, 1\rangle,$
    $\ldots, \langle 2b_{m_2}+1, b_{m_2}, 1\rangle)$;
2.  $count_1 \leftarrow 1 - 2z_1$;
3.  for $i = 2$ to $m_1 + m_2$ do
4.     if $(y_{i-1} = y_i)$
5.        if $(z_i)$ $count_i \leftarrow count_{i-1} - 1$;
6.        else $count_i \leftarrow count_{i-1} + 1$;
7.     else $count_i \leftarrow 1 - 2z_i$;
8.  $c_1 \leftarrow y_1$;
9.  for $i = 2$ to $m_1 + m_2$ do in parallel
10.    if $(y_i \wedge (count_i \geq 0))$ $c_i \leftarrow 0$;
11.    else $c_i \leftarrow y_i$;
12. return $c_1, \ldots, c_{m_1+m_2}$;
```

The two for loops can be easily combined into one (i.e., the y_i's can be reset to 0 inside the first loop). We separate them for clarity of presentation: the computation in the first loop will be common to all of our multiset operations, and the computation in the second loop is specific to multiset union. We also note that, for the purposes of the above sequential algorithm, it is not necessary to maintain negative counts. Instead, all elements from the second multiset that have no matching elements from the first multiset can have the same count (e.g., -1). If the same count is used, the comparison on line 10 can be replaced with an equality check, which would result in a slightly more efficient implementation. We, however, need to maintain the exact counts for the elements coming from both multisets for the purposes of a constant-round implementation of this functionality.

To compute multiset intersection, it is sufficient to replace lines 8–11 above with:

```
8.  $c_1 \leftarrow 0$;
9.  for $i = 2$ to $m_1 + m_2$ do in parallel
10.    if $(\neg y_i)$ $c_i \leftarrow 0$;
11.    else if $(count_i < 0)$ $c_i \leftarrow 0$;
12.       else $c_i \leftarrow x_i$;
```

Computing the subset relation ($B \overset{?}{\subseteq} A$) can be achieved by replacing lines 8–12 with:

```
8.  $s \leftarrow 1$;
9.  for $i = 1$ to $m_1 + m_2$ do in parallel
10.    if $(count_i < 0)$ $s \leftarrow 0$;
```

```
11. return $s$;
```

To compute multiset difference, we change line 1 of the union algorithm to:

```
1. $\langle x_1, y_1, z_1\rangle, \ldots, \langle x_{m_1+m_2}, y_{m_1+m_2}, z_{m_1+m_2}\rangle \leftarrow Sort(\langle 2a_1{+}1, a_1, 1\rangle, \ldots, \langle 2a_{m_1}{+}1, a_{m_1}, 1\rangle, \langle 2b_1, b_1, 0\rangle, \ldots, \langle 2b_{m_2}, b_{m_2}, 0\rangle)$;
```

and replace lines 8–11 with the appropriate logic:

```
8.  for $i = 1$ to $m_1 + m_2$ do in parallel
9.    if $(\neg y_i)$ $c_i \leftarrow 0$;
10.   else if $(count_i \geq 0)$ $c_i \leftarrow 0$;
11.     else $c_i \leftarrow x_i$;
```

What is important to notice is that the proposed approach for representing sorted multisets is asymmetric with respect to the inputs A and B, which makes it a natural choice for asymmetric (i.e., not commutative) set operations such as (asymmetric) difference and subset relation. As shown above, it also works for symmetric operations such as union and intersection. If, however, we would like to implement an improved logic for the set intersection that produces a multiset of size $(m_1 + m_2)/2$ instead of $m_1 + m_2$ or symmetric difference, we are not aware of a convenient way to modify Algorithm 1 for that purpose.

To use our multiset-to-set conversion approach for symmetric set operations, we observe that the procedure for computing the counts can be applied to the input multisets independently, after which the two multisets can be merged. This gives us a mechanism for realizing symmetric functionalities using an asymmetric function. We provide additional information on how this functionality can be implemented below.

6.2 Efficient secure implementation

All of the algorithms for performing multiset operations directly that we described so far are sequential and involve a linear number of rounds. To be able to compute these multiset operations in a constant number of rounds, all that is necessary is to design a mechanism for computing all counts $count_i$ in a constant number of rounds. Using the intuition developed in the previous section, we define a new operator for the purposes of computing counts, which can be securely implemented on two operands as follows:

```
$\langle [x], [y], [count]\rangle \leftarrow \langle [x_1], [y_1], [count_1]\rangle \diamond \langle [x_2], [y_2], [count_2]\rangle$

1. $[u] \leftarrow \mathsf{Eq}([x_1], [x_2], \ell)$;                      // section 3.2
2. $[x] \leftarrow [x_2]$;
3. $[y] \leftarrow [y_2]$;
4. $[count] \leftarrow [u][count_1] + [count_2]$;               // 1 round, 1 inv
5. return $\langle [x], [y], [count]\rangle$;
```

In the above, each x_1 and x_2 is a multiset element and y_1 and y_2 are bits. It is expected that the inputs are well-formed, which means that $x_1 \leq x_2$ and if $x_1 = x_2$, then $y_1 \leq y_2$. Then we obtain that if $x_1 = x_2$, the counts are simply added. Otherwise, $count_1$ is ignored and $count_2$ is used in the result. This operator can also be shown to be associative.

The last piece that remains before we are ready to present our direct implementations of private multiset operations is to show how to compute unbounded fan-in $\diamond$ operator $\diamond_{i=1}^{m}\langle x_i, y_i, count_i\rangle$ in a constant number of rounds. This can be accomplished in a similar way to computing $\circ_{i=1}^{m}\langle x_i, y_i\rangle$ in section 5. In more detail, we have:

Protocol 12. $\langle [x], [y], [count] \rangle \leftarrow \diamond_{i=1}^{m} \langle [x_i], [y_i], [count_i] \rangle$

```
1. for $i = 1$ to $m-1$ do in parallel $[u_i] \leftarrow \mathsf{Eq}([x_i], [x_{i+1}], \ell)$;            // section 3.2
2. $([v_{m-1}], \ldots, [v_1]) \leftarrow \mathsf{PreAND}([u_{m-1}], \ldots, [u_1])$;                    // section 3.2
3. for $i = 1$ to $m-1$ do in parallel $[w_i] \leftarrow [v_i][count_i]$;                   // 1 round, $m-1$ inv
4. $[x] \leftarrow [x_m]$;
5. $[y] \leftarrow [y_m]$;
6. $[count] \leftarrow [count_m] + \sum_{i=1}^{m-1} [w_i]$;
7. return $\langle [x], [y], [count] \rangle$;
```

Now, for example, the multiset union protocol becomes:

Protocol 13. $[c_1], \ldots, [c_{m_1+m_2}] \leftarrow \mathsf{DMUnion}([a_1], \ldots, [a_{m_1}], [b_1], \ldots, [b_{m_2}])$

```
1. $\langle [x'_1], [y'_1], [z'_1] \rangle, \ldots, \langle [x'_{m_1+m_2}], [y'_{m_1+m_2}], [z'_{m_1+m_2}] \rangle \leftarrow \mathsf{SortT}(\langle 2[a_1], [a_1], [0] \rangle, \ldots, \langle 2[a_{m_1}],$
   $[a_{m_1}], [0] \rangle, \langle 2[b_1]+1, [b_1], [1] \rangle, \ldots, \langle 2[b_{m_2}]+1, [b_{m_2}], [1] \rangle, \ell+1)$;   // section 3.2
2. for $i = 1$ to $m_1+m_2$ do in parallel $[count'_i] \leftarrow 1 - 2[y'_i]$;
3. $\langle [x_1], [y_1], [count_1] \rangle, \ldots, \langle [x_{m_1+m_2}], [y_{m_1+m_2}], [count_{m_1+m_2}] \rangle \leftarrow \mathsf{Pre}_\diamond(\langle [y'_1], [z'_1], [count'_1] \rangle \ldots,$
   $\langle [y'_{m_1+m_2}], [z'_{m_1+m_2}], [count'_{m_1+m_2}] \rangle)$;                          // Protocol 11
4. for $i = 2$ to $m_1+m_2$ do in parallel $[u_i] \leftarrow \mathsf{GE}([count_i], 0, \lceil \log \max(m_1, m_2) \rceil)$; // section 3.2
5. for $i = 2$ to $m_1+m_2$ do in parallel $[v_i] \leftarrow [x_i][y_i]$;                  // $m_1+m_2-1$ inv
6. $[c_1] \leftarrow [x_1]$;
7. for $i = 2$ to $m_1+m_2$ do in parallel $[c_i] \leftarrow [x_i] - [u_i][v_i]$;          // 1 round, $m_1+m_2-1$ inv
8. return $[c_1], \ldots, [c_{m_1+m_2}]$;
```

The subset relation protocol DMSub can be obtained from $\mathsf{DMUnion}$ by replacing lines 4–8 with:

```
4. for $i = 1$ to $m_1+m_2$ do in parallel $[u_i] \leftarrow \mathsf{GE}([count_i], 0, \lceil \log m_1 \rceil)$;       // section 3.2
5. $[t] \leftarrow \sum_{i=1}^{m_1+m_2} [u_i]$;
6. $[s] \leftarrow \mathsf{Eq}([t], m_1+m_2, \lceil \log(m_1+m_2) \rceil)$;                              // section 3.2
7. return $[s]$;
```

It is assumed above that $m_1 \geq m_2$; otherwise, the result of the operation is 0 based on the multiset sizes (when no padding is used). To form a private multiset difference protocol DMDiff, one needs to change the loops on lines 4 and 5 of the $\mathsf{DMUnion}$ protocol to start from $i = 1$, as well as replace line 1 with:

```
1. $\langle [x'_1], [y'_1], [z'_1] \rangle, \ldots, \langle [x'_{m_1+m_2}], [y'_{m_1+m_2}], [z'_{m_1+m_2}] \rangle \leftarrow \mathsf{SortT}(\langle 2[a_1]+1, [a_1], [1] \rangle, \ldots, \langle 2[a_{m_1}] +$
   $1, [a_{m_1}], [1] \rangle, \langle 2[b_1], [b_1], [0] \rangle, \ldots, \langle 2[b_{m_2}], [b_{m_2}], [0] \rangle, \ell+1)$;              // section 3.2
```

and lines 6–7 with:

```
6. for $i = 1$ to $m_1+m_2$ do in parallel $[c_i] \leftarrow (1 - [u_i])[v_i]$;                // 1 round, $m_1+m_2$ inv
```

As was mentioned earlier, we use a different logic for set intersection and symmetric difference protocols, in which $\mathsf{Pre}_\diamond$ is executed on each input multiset separately, and the results are merged to produce a single sorted set. In what follows, we provide a set intersection protocol that implements the same computation for multisets as Protocol 2 for sets, but uses different variable naming for ease of consecutive description. Below, $m = m_1 + m_2$.

Protocol 14. $[c_1], \ldots, [c_{\lfloor m/2 \rfloor}] \leftarrow \mathsf{DMInt}([a_1], \ldots, [a_{m_1}], [b_1], \ldots, [b_{m_2}])$

```
1. $[a'_1], \ldots, [a'_{m_1}] \leftarrow \mathsf{Sort}([a_1], \ldots, [a_{m_1}], \ell)$;                          // section 3.2
```

```
 2. $[b'_1], \ldots, [b'_{m_2}] \leftarrow \mathsf{Sort}([b_1], \ldots, [b_{m_2}], \ell)$;                          // section 3.2
 3. $\langle [x'_1], [y'_1], [count'_1] \rangle, \ldots, \langle [x'_{m_1}], [y'_{m_1}], [count'_{m_1}] \rangle \leftarrow \mathsf{Pre}_\diamond(\langle [a_1], [0], [1] \rangle, \ldots, \langle [a_{m_1}], [0], [1] \rangle)$;
                                                                                     // Protocol 11
 4. $\langle [x''_1], [y''_1], [count''_1] \rangle, \ldots, \langle [x''_{m_2}], [y''_{m_2}], [count''_{m_2}] \rangle \leftarrow \mathsf{Pre}_\diamond(\langle [b_1], [0], [1] \rangle, \ldots, \langle [b_{m_2}], [0], [1] \rangle)$;
                                                                                     // Protocol 11
 5. $k = \max(m_1, m_2) + 1$;
 6. $\langle [z_1], [x_1], [count_1] \rangle, \ldots, \langle [z_m], [x_m], [count_m] \rangle \leftarrow \mathsf{MergeT}((\langle k[x'_1] + [count'_1], [x'_1], [count'_1] \rangle, \ldots,$
    $\langle k[x'_{m_1}] + [count'_{m_1}], [x'_{m_1}], [count'_{m_1}] \rangle), (\langle k[x''_1] + [count''_1], [x''_1], [count''_1] \rangle, \ldots, \langle k[x''_{m_2}] + [count''_{m_2}],$
    $[x''_{m_2}], [count''_{m_2}] \rangle), \ell + \lceil \log k \rceil)$;                                // section 3.2
 7. for $i = 1$ to $m - 1$ do in parallel $[u_i] \leftarrow \mathsf{GE}(k[x_i] + [count_i], k[x_{i+1}] + [count_{i+1}], \ell + \lceil \log k \rceil)$;
                                                                                     // section 3.2
 8. for $i = 1$ to $\lfloor (m-1)/2 \rfloor$ do in parallel $[c_i] \leftarrow ([u_{2i-1}] + [u_{2i}])[x_{2i}]$; // 1 round, $\lfloor (m-1)/2 \rfloor$ inv
 9. if ($m$ mod $2 = 0$) $[c_{m/2}] = [u_{m-1}][x_m]$;                                   // 1 inv
10. return $[c_1], \ldots, [c_{\lfloor m/2 \rfloor}]$;
```

To implement multiset symmetric difference DMSDiff, all we need is to replace lines 8–10 in DMInt with the following:

```
 8. $[c_1] \leftarrow [x_1](1 - [u_1])$;                                           // 1 round, 1 inv
 9. $[c_m] \leftarrow [x_m](1 - [u_{m-1}])$;                                       // 1 inv
10. for $i = 2$ to $m - 1$ do in parallel
        $[c_i] \leftarrow [x_i](1 - [u_i] - [u_{i-1}])$;                           // $m - 2$ inv
11. return $[c_1], \ldots, [c_m]$;
```

As before, security of these protocols can be shown in both passive and active models using the same argument as in section 4.4.

While in the most general case our direct implementation of multiset operations yields more efficient results, there are circumstances when the general approach described in section 5 achieves a performance not significantly different from the direct implementation of the respective multiset operation. In particular, if we can guarantee that the conversion procedure M2S will be executed over the elements of all multisets as the initial step and that, as a result, each input multiset is properly sorted, it then becomes possible to replace the SortT procedure present on all set operations protocols by a more efficient MergeT. In that case, the cost of using the general conversion and running the protocols for regular set operations will be very close to that of executing the protocols presented in this section. The main difference in the performance of the two solutions then comes from the need to operate on longer values in the general conversion than in the direct solutions for most multiset operations while comparing the multiset elements. For example, sorting (or merging) in MUnion executes compare-and-exchange operations on $(\ell + \lceil \log k \rceil)$-bit values, while DMUnion performs this operation on values of length $\ell + 1$. In both cases the modulus p of the secret sharing scheme must be chosen appropriately to allow for correct representation of integers of the specified length.

We next show how the multiset protocols described in this section can be made suitable for length-hiding operations. Similar to set operations, all direct operations on multisets with the exception of subset relationship work correctly when input multisets are padded with zero elements to hide the actual number of elements in a multiset. It therefore remains to show how DMSub needs to be modified to be suitable for length-hiding computation.

To ensure correct operation of DMSub on padded multisets, what is needed is to guarantee that zero elements will not affect the outcome. This means that the excess of zero elements in the second multiset B which have negative counts needs to be ignored in determining the result of the operation. The simplest way to achieve this is to compare each element of the merged sorted set to 0 and disregard zero elements with negative counts. This is what the protocol below computes, where the total number of non-zero elements with negative counts should be 0 to result in the output bit being set.

Protocol 15. $[s] \leftarrow \mathsf{DMSub}([a_1], \ldots, [a_{m_1}], [b_1], \ldots, [b_{m_2}])$

```
1. $\langle [x'_1], [y'_1], [z'_1] \rangle, \ldots, \langle [x'_{m_1+m_2}], [y'_{m_1+m_2}], [z'_{m_1+m_2}] \rangle \leftarrow \mathsf{SortT}(\langle 2[a_1], [a_1], [0] \rangle, \ldots, \langle 2[a_{m_1}], [a_{m_1}], [0] \rangle,$
   $\langle 2[b_1]+1, [b_1], [1] \rangle, \ldots, \langle 2[b_{m_2}]+1, [b_{m_2}], [1] \rangle, \ell+1)$;        // section 3.2
2. for $i = 1$ to $m_1 + m_2$ do in parallel $[count'_i] \leftarrow 1 - 2[y'_i]$;
3. $\langle [x_1], [y_1], [count_1] \rangle, \ldots, \langle [x_{m_1+m_2}], [y_{m_1+m_2}], [count_{m_1+m_2}] \rangle \leftarrow \mathsf{Pre}_\diamond(\langle [y'_1], [z'_1], [count'_1] \rangle \ldots,$
   $\langle [y'_{m_1+m_2}], [z'_{m_1+m_2}], [count'_{m_1+m_2}] \rangle)$;        // Protocol 11
4. for $i = 1$ to $m_1 + m_2$ do in parallel $[u_i] \leftarrow \mathsf{GE}([count_i], 0, \lceil \log \max(m_1, m_2) \rceil)$; // section 3.2
5. for $i = 1$ to $m_1 + m_2$ do in parallel $[v_i] \leftarrow \mathsf{Eq}([x_i], 0, \ell)$;        // section 3.2
6. $[t] \leftarrow \sum_{i=1}^{m_1+m_2} (1 - [u_i])[v_i]$;        // 1 round, $m_1 + m_2$ inv
7. $[s] \leftarrow \mathsf{Eq}([t], 0, \lceil \log \max(m_1, m_2) \rceil)$;        // section 3.2
8. return $[s]$;
```

This change to the original DMSub, however, involves $m_1 + m_2$ additional equality tests, which generally can be avoided. In particular, as we represent multiset elements using positive numbers, we can replace lines 5–7 above with

```
5. $[t] \leftarrow \sum_{i=1}^{m_1+m_2} (1 - [u_i])[x_i]$;        // 1 round, $m_1 + m_2$ inv
6. $[s] \leftarrow \mathsf{Eq}([t], 0, \ell + \lceil \log \max(m_1, m_2) \rceil)$;        // section 3.2
```

which completely avoids the extra equality tests. Note that, instead of being the sum of bits, the value of t is now larger and contains the sum of the elements themselves with negative counts. The correctness of the result, however, is still guaranteed if we appropriately increase the number of bits considered in the final equality test when comparing the value of t to 0. Unlike adding $m_1 + m_2$ equality checks, this change has a negligible effect on the performance of the operation.

7 Optimizations

In this section we describe techniques for improving efficiency of the protocols by optimizing sorting or replacing it with more efficient alternatives.

7.1 Operating on sorted inputs

As the first optimization, we notice that if the input sets in our set operations are always given in a sorted form, the sorting step of our algorithms (which introduces their main complexity) can be replaced by a merge operation. Because the merging step has lower complexity than sorting, the efficiency of the overall protocol improves. In particular, as mentioned earlier, oblivious bitonic merge [6] uses $\frac{1}{2} m \log m$ compare-and-exchange operations and, perhaps more importantly, has depth of $\log m$ as opposed to $\frac{1}{4} m \log^2 m$ and $\frac{1}{2} \log^2 m$, respectively, for merge sort.

In order to be able to use merging instead of sorting in our protocols, we need to ensure that inputs are given in a sorted form and the outputs also correspond to sorted (multi)sets. When each set is originally coming from a single input party, it can be locally sorted prior to distributing its shares to the computational parties. Alternatively, if the entire set is not known to any individual party, every portion of it known to a single party can still be sorted and multiple portions are merged by the computational parties prior to a protocol execution. Then the complexity of the first set operation which handles that set will be higher than that of merging, but all other uses of the same set save the cost of sorting.

To ensure that the output produced by a protocol is a sorted set, notice that non-zero elements of all output sets are already sorted. Thus, instead of performing full sorting to produce a sorted set, all that is necessary is to use set compaction which will place all zero elements before non-zero elements. Producing a sorted set as the output will also eliminate the need to sort the set at the end of the overall computation when the set is to be revealed to the output parties. Efficient oblivious set compaction is therefore what we address next.

7.2 Utilizing (multi)set compaction

Our starting point for realizing set compaction obliviously was tight order-preserving compaction for the external memory [40] that places all zero elements before non-zero elements while preserving the order of the non-zero elements. We adopt the solution of [40] to our setting and optimize to minimize the number of interactive operation as well as the number of rounds. The algorithm uses butterfly-like network that consists of $\log m$ levels for sets of size m $x_1, \ldots, x_m$. Initially, at level L_0, the cells store the original set to be compacted (cells with non-zero values x_j are considered occupied). Cell j at level L_i is connected to cells j and $j - 2^i$ at level L_{i+1}, which means that it can be routed to either cell at the next level. Initially, each non-zero element is labeled with the number of cells that it needs to be moved to the left to create a tight compaction. In other words, the label corresponds to the number of 0s in front of a non-zero element. For instance, if the input set is 1, 0, 2, 0, 0, 3, the labels of 1, 2, and 3 will be 0, 1, and 3, respectively (and zero elements can be assumed to be labeled with 0). These labels can be produced by a single scan of the array, which we parallelize to run only in one round in our solution. Then for each level L_i for $0 \le i \le \log m - 1$, the content of each occupied cell j with label y_j is routed to cell $j - (y_j \bmod 2^{i+1})$ (which will be either j or $j - 2^i$) at level L_{i+1}, after which the label is updated to $y_j = y_j - (y_j \bmod 2^{i+1})$.

Note that in the above description, non-zero elements of the input set are collected on the left, at low indices, while for our set protocols we would like zero elements to be moved to the left. This can be easily corrected by calling set compaction on the set $x_m, \ldots, x_1$ with the order of the element reversed instead of the original $x_1, \ldots, x_m$ and consequently reversing the order of the elements in the returned set. Because the compaction algorithm is order preserving, it will work in either situation.

In our compaction protocol Comp below, we first determine all non-zero elements and produce their labels. The labels are incremented from element j to $j + 1$ only if element $j + 1$ is non-zero. Then the labels of zero elements are erased (reset to 0). Because all additions are performed locally, producing the labels (lines 2–4) involves only a single round. After computing the labels, we process one level of the routing network at a time, during which for each cell j at level i we compute the bit $v_j = (y_j \bmod 2^{i+1} \overset{?}{=} 0)$. The value of the cell j at level $i + 1$ is then determined based on the routing decisions for cells j and $j + 2^i$ at level i. That is, if both cells j and $j + 2^i$ at level i are occupied, the content of either of them can be copied to cell j at level $i + 1$. Otherwise, it may or may not be occupied. Due to the algorithm's correctness at most one occupied cell from level i will be routed to any given cell j at level $i + 1$. This logic is encoded on lines 9–14 of the protocol, which updates the cell contents as well as their labels for level $i + 1$.

Protocol 16. $[x_1], \ldots, [x_m] \leftarrow \mathsf{Comp}([a_1], \ldots, [a_m])$

```
1. for $i = 1$ to $m$ do in parallel $[z_i] \leftarrow \mathsf{Eq}([a_i], 0, \ell)$;        // section 3.2
2. $[count_1] \leftarrow [z_1]$;
3. for $i = 2$ to $m$ do $[count_i] \leftarrow [count_{i-1}] + [z_i]$;
```

```
4.  for $i = 1$ to $m$ do in parallel $[y_i] \leftarrow (1 - [z_i])[count_i]$;          // 1 round, $m$ inv
5.  for $i = 1$ to $m$ do in parallel $[x_i] \leftarrow [a_i]$;
6.  for $i = 0$ to $\log m - 1$ do
7.      for $j = 1$ to $m$ do in parallel $[u_j] \leftarrow \mathsf{Mod2k}([y_j], \ell, i+1)$;     // see below
8.      for $j = 1$ to $m$ do in parallel $[v_j] \leftarrow \mathsf{Eq}([u_j], 0, i+1)$;          // section 3.2
9.      for $j = 1$ to $m - 2^i$ do in parallel
10.         $[x_j] \leftarrow [v_j][x_j] + (1 - [v_{j+2^i}])[x_{j+2^i}]$;                   // 1 round, $2(m - 2^i)$ inv
11.         $[y_j] \leftarrow [v_j][y_j] + (1 - [v_{j+2^i}])([y_{j+2^i}] - (1 - [u_{j+2^i}])2^i)$;   // $2(m - 2^i)$ inv
12.     for $j = m - 2^i$ to $m$ do in parallel
13.         $[x_j] \leftarrow [v_j][x_j]$;                                              // $2 \cdot 2^i$ inv
14.         $[y_j] \leftarrow [v_j][y_j]$;                                              // $2 \cdot 2^i$ inv
15. return $[x_1], \ldots, [x_m]$;
```

This protocol uses a new function which computes $a \bmod 2^k$ for a secret-shared integer a, the description of which we present next. Our protocol Mod2k takes a secret-shared value a, its length ℓ in bits and an integer k and produces value $a \bmod 2^k - 2^k u$ (modulo p), where u is a bit. Our protocol is a much reduced version of similar functionality in [15], which computes the operation precisely as $a \bmod 2^k$ by removing the error factor $2^k u$. We next describe our protocol and then explain why the error factor does not affect correctness of compaction.

Protocol 17. $[b] \leftarrow \mathsf{Mod2k}([a], \ell, k)$

```
1. $[r''] \leftarrow \mathsf{RandInt}(\kappa + \ell - k)$;
2. $[r'] \leftarrow \mathsf{RandInt}(k)$;
3. $c \leftarrow \mathsf{Open}([a] + 2^k[r''] + [r'])$;          // 1 round, 1 inv
4. $c' \leftarrow c \bmod 2^k$;
5. $[b] \leftarrow c' - [r']$;
6. return $[b]$;
```

In the above, RandInt produces a random value of the bitlength given as its argument and requires no interaction (see, e.g., [15] for more detail). Also recall that Open allows the parties to reconstruct the value given as its argument. In the protocol, κ corresponds to the statistical security parameter, and after the first three steps of the protocol the parties learn $a + r$, where the length of random $r = 2^k r'' + r'$ is at least κ bits more than the length of a. Then note that the output b is equal to $(a \bmod 2^k) - 2^k u$, where bit $u = 1$ iff $(a \bmod 2^k) + r' > 2^k$. We obtain that the result is $a \bmod 2^k$ when $(a \bmod 2^k) + r' < 2^k$, otherwise, when the sum overflows k-bit integers, the result is $(a \bmod 2^k) - 2^k$.

Returning back to compaction, we note that the above computation with a possible error 2^k does not pose a problem for our compaction algorithm. That is, the only values that $a \bmod 2^k$ can take during compaction are 0 and 2^{k-1}. This means that 0 will always be computed correctly (no overflow is possible), while 2^{k-1} can be computed as either 2^{k-1} or $2^{k-1} - 2^k$. Because the only information that we need based on this computation is whether the result was equal to zero or not (i.e., equality test on line 7 of Comp), the result of the comparison will always be correct, i.e., neither 2^{k-1} nor $2^{k-1} - 2^k$ can be 0 in our representation to produce an error. This is true in our setting (i.e., for any odd modulus p) even if we consider only $k + 1$ significant bits when comparing the result of Mod2k to 0.

Remarkably, we obtain the cost of (reduced) modulo reduction of only one interactive operation. We obtain that the overall cost of compaction is dominated by $m \log m$ equality tests, where each Eq protocol is executed on short values and the operation itself is substantially faster than GE used

Adv	Security	Communication	Reference
passive	perfect/statistical	$O(n\ell m \log m + n^2)$	[28]
active	perfect	$O(n\ell m \log m + n^2 \log m + n^3)$	[7]
active	statistical	$O(n(\ell+\kappa)m \log m + n^2 \log m + n^3)$	[7]

Table 2: Communication complexity of set and multiset protocols measured in the number of field elements.

in compare-and-exchange operations. This means that compaction runs in a small fraction of time of either sorting or merging protocols. The round complexity of Comp is $(\mathsf{round}(\mathsf{Eq})+1)(\log m+1)$, i.e., similar to that of merging.

Finally, we would like to mention that compaction is not the only mechanism of a cost lower than sorting for protecting private information about the output of a (multi)set operation before revealing it to the output parties. [47], for example, use Waksman switching network [69] that computes a random permutation of a set, which allows the parties to randomly shuffle the elements of the output set and thus hide any patterns in it. Waksman network is implemented in [47] for the two-party setting based on garbled circuits using a number of computation optimizations which allow for an efficient implementation of the switching network. In particular, in [47] one party supplies a random permutation and "hard-wires" it in the circuit, and the representation of the wires associated with the comparison operations in the switching network is optimized as well. In our setting, however, implementing such a network becomes substantially more costly. That is, in addition to having the computational parties obliviously choose a random permutation not known to any of them, implementing the network itself will include $\approx m \log m$ GE operations as well as other computation. The compaction algorithm that we instead choose in this work to accomplish this (and other) goal allows for a significantly faster implementation: while requiring a similar number of operations, it uses only equality tests Eq which are noticeably faster in our framework than GE comparisons and does not involve a significant amount of other work.

The security of the protocols presented in this section follows the same argument as before. In particular, it relies on the same elementary building blocks as other sub-protocols from prior literature used throughout our solutions (such as comparisons).

8 Complexity Analysis

After presenting optimizations to the protocols, we are ready to evaluate their complexities under different security settings. The complexities of all of our protocols are dominated by $O(m \log m)$ compare and exchange operations needed for sorting, where m is the combined size of the input (multi)sets, or $O(\ell m \log m)$ invocations, where ℓ is the bitlength of set elements. When the computation proceeds on sorted sets, the depth or round complexity of all protocols is $O(\log m)$ (with and without compaction). The communication complexity of our protocols measured in the number of field elements is shown in Table 2 using the results from prior literature. The computation is the same as communication. The results assume $t < n/2$ for passive adversaries and $t < n/3$ for active adversaries (although results for $t < n/2$ are available as well). The results with perfect security use perfectly secure building blocks. When statistically secure building blocks are used in the malicious setting, as discussed in section 4.4, complexity of LT, Eq, and PreAND becomes $O(\ell+\kappa)$ invocations, where κ is a (statistical) security parameter. This is due to the fact that protocol $\mathsf{RandInt}$ is called to generate random integers κ bits longer than integers used in the computation. This change is reflected in Table 2. The constants, however, are small enough that we expect the solution that uses statistically secure building blocks is faster in the malicious model as well. When statistically

Protocol	Set size							
	16	32	64	128	256	512	1024	2048
Set union	0.127	0.247	0.515	1.104	2.411	5.384	11.886	24.880
Set intersection	0.125	0.245	0.510	1.097	2.359	5.327	11.734	24.875
Set intersection with compaction	0.164	0.316	0.640	1.337	2.880	6.323	13.849	30.716
Multiset intersection	0.163	0.310	0.634	1.297	2.855	6.287	14.242	29.598

Table 3: Runtime of set and multiset operations protocols in seconds.

secure building blocks are used in any more, the field size is increased by a security parameter κ.

9 Performance Evaluation

In order to fully evaluate performance of our techniques, we implemented several protocols and measured their runtime for a number of set sizes. We used 32-bit integers to represent set elements (i.e., $\ell = 32$), and following the implementation of related primitives in [64] set the statistical security parameter κ to 48. This requires that the field $\mathbb{Z}_p$ used for the secret sharing scheme has modulus p of size greater than $\ell + \kappa$, and we use $|p| = 81$ in our implementation of set operations. For the multiset operations, the modulus size is increased by $\log(\max(m_1, m_2) + 1)$ bits, where as before m_1 and m_2 are the number of elements in the input multisets. For the experiments we used (3, 1)-secret sharing scheme, where each of the three computational parties was run on a 2.4GHz AMD Opteron computer. The computational parties were connected by 1Gb Ethernet. The code was written in C++ using the GMP library [37] for large number arithmetic. All integer operations were implemented as described in [2].

We implemented optimized set union and intersection protocols, as well as multiset-to-set conversion which correspondingly allows us to run multiset union and intersection. In more detail, we used bitonic merge [6] instead of full sort together with the building blocks' instantiations listed in section 3.2. As described in section 7, this setup assumes that the input sets are already individually sorted.

Our implementation used a limited degree of parallelism. In particular, when a number of operations of the same type could be carried out in parallel, they were executed in a single batch. For instance, in bitonic merge $m/2$ independent compare-and-exchange operations can be carried out simultaneously, and in our implementation each computational party first batched computation and communication of all of them together using the same number of rounds as that of a single compare-and-exchange operation. While this type of processing allows us to greatly reduce the communication time compared to the sequential execution of each operation by the computational parties, it by no means is optimal in terms of its runtime and the performance can be improved. In particular, with a full support for parallelism, the computation could be split among the multiple cores of the computational parties. In addition, the number of communication rounds could be lowered as data-independent rounds of the comparison protocol of a bitonic merge iteration could be carried out in parallel with the comparison computation of its previous iteration.

We measured performance of implemented protocols on sets of size from 16 to 2048, where the set size was increased by a factor of 2 for each consecutive experiment. Table 3 lists the running times of our set union (Protocol 1), set intersection (Protocol 2), set intersection followed by compaction (Protocols 2 and 16), and multiset intersection (Protocol 18 in Appendix A) in seconds. Each reported runtime corresponds to the average running time over five identical runs of the corresponding operation and the set size. As can be seen from the table, we obtain practical results which can scale to sets and multisets of rather large sizes. As expected, the runtime grows

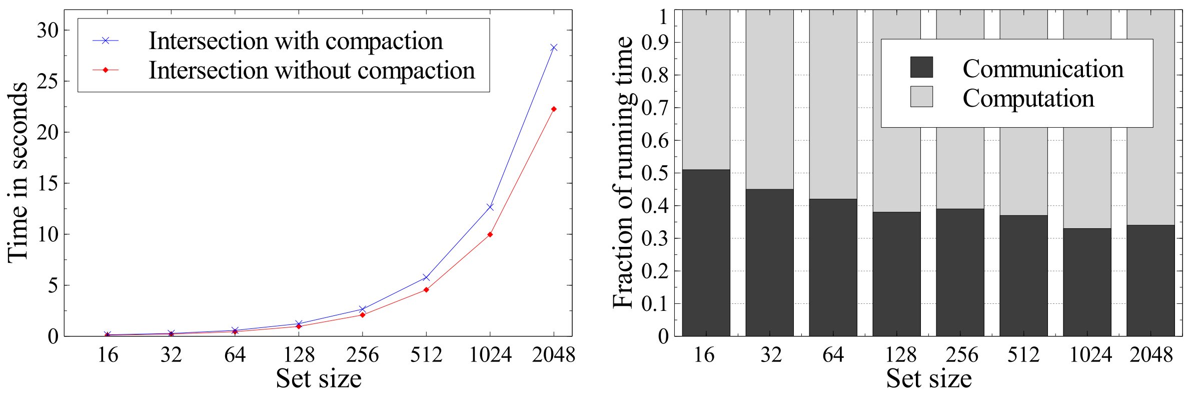

Figure 1: Performance of set intersection protocols.

slightly faster than a linear function in the size of the (multi)set.

One can notice that the performance of the set union protocol is very close to that of set intersection. This is due to the fact that almost all of the time is being spent in the merging step common to both operations, and the remaining computations are also very similar. Both Table 3 and Figure 1 report times for the set intersection protocol with and without compaction. As can be seen, despite a larger constant in the complexity of compaction compared to that of merging, performance of compaction is noticeably faster than that of the set intersection itself.

Table 3 also shows that performance of multiset operations is only slightly slower than that of the corresponding set operations.

Lastly, we measure the amount of time used for communication compared to that for computation. As previously discussed, our protocols were designed to minimize the round complexity and consequently reduce the communication time. We therefore were interested in determining the portion of the overall runtime due to communication, and the results for the set intersection protocol (without compaction) are given in Figure 1. Following our expectation, for small set sizes most of the overall runtime is due to communication, and the fraction of time spent on communication gradually decreases as we increase the set size.

Recently, implementations of secure set intersection protocols in the two-party setting have appeared in the literature [47, 32]. Because of the drastic differences in our setting and the setting adopted in those publications, a direct performance comparison of our solutions and those in [47, 32] is not possible. We can therefore only provide a discussion of the relative performance and capabilities of the solutions. In particular, [47] propose protocols for set intersection in the two-party setting based on Yao's generic garbled-circuit evaluation [71]. For sets with 1024 elements and the security parameter set to guarantee short-term security (112 bits), their most efficient implementation yields a runtime of 11.8 seconds for elements represented using 32 bits (exact runtimes are not available for sets of other sizes). In another recent implementation of two-party set intersection [32], the authors measured the performance of a custom linear-time RSA-based protocol from [31]. The implementation was optimized and fully parallelized, in that the computation was partitioned among the cores of a 4-core server and dual core client. The authors achieve a notable runtime of 1.8 seconds for sets of size 1000 and the same 112-bit short-term security parameter. Although our implementation results in a slower performance, this gap is largely expected for a variety of reasons. The most prominent reason is the fact that our multi-party framework incurs numerous interactive rounds during computation while these two-party solutions require a single interaction. Second, our solution is oblivious with respect to the inputs, while in the solutions implemented in both [47] and [32] the fact that the parties have knowledge of the sets (and in some cases other information) results in faster performance. Third, not taking advantage of the available multiple CPU cores in our implementation contributes to the amount of time spent on computation, although this can be substantially reduced in an implementation that parallelizes the computation. Lastly, the flexibility of our framework, composability of the protocols, and support for a large number of set and multiset operations offer advantages not available in other settings, and our solution can be preferred for those reasons despite its longer runtime.

10 Conclusions

This work is the first to provide a comprehensive suite of protocols for multi-party set and multiset operations that are data-oblivious and composable. The list of covered operations consists of set and multiset union, intersection, equality, symmetric and asymmetric difference, subset and superset relationships, and element reduction (for multisets). The flexibility of the framework allows these operations to be employed in a variety of settings ranging from the traditional secure multi-party computation to secure outsourcing by one or more parties. The solutions have a natural support for hiding the output size and can be easily extended to compute cardinality or over-the-threshold cardinality of the result. All solutions are information-theoretically secure against malicious adversaries, achieve low communication and computation cost of $O(m \log m)$ for data sets of size m, and were designed to minimize round complexity. Experimental results show practicality of our solution.

Acknowledgments

Portions of this work were sponsored by grant AFOSR-FA9550-09-1-0223 from the Air Force Office of Scientific Research. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the AFOSR.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *STOC*, pages 1–9, 1983.

[2] M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele. Secure computation on floating point numbers. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[3] G. Asharov and Y. Lindell. A full proof of the BGW protocol for perfectly-secure multiparty computation. Electronic Colloqium on Computational Complexity (ECCC), Report No. 36, 2011.

[4] G. Asharov, Y. Lindell, and T. Rabin. Perfectly-secure multiplication for any $t < n/3$. In *CRYPTO*, 2011.

[5] G. Ateniese, E. De Cristofaro, and G. Tsudik. (If) size matters: Size-hiding private set intersection. In *Public Key Cryptography (PKC)*, volume 6571 of *LNCS*, pages 156–173, 2011.

[6] K. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, 1968.

[7] Z. Beerliova-Trubiniova and M. Hirt. Perfectly-secure MPC with linear communication complexity. In *Theory of Cryptography Conference (TCC)*, pages 213–230, 2008.

[8] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A system for secure multi-party computation. In *ACM Conference on Computer and Communications Security (CCS)*, pages 257–266, 2008.

[9] M. Blanton and E. Aguiar. Private and oblivious set and multiset operations. In *ASIACCS*, 2012.

[10] M. Blanton, M. Atallah, K. Frikken, and Q. Malluhi. Secure and efficient outsourcing of sequence comparisons. In *ESORICS*, pages 505–522, 2012.

[11] G. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *SPAA*, pages 16–26, 1998.

[12] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[13] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[14] R. Canetti, I. Damgård, S. Dziembowski, Y. Ishai, and T. Malkin. Adaptive versus non-adaptive security of multi-party protocols. *Journal of Cryptology*, 17(3):153–207, 2004.

[15] O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In *Security and Cryptography for Networks (SCN)*, pages 182–199, 2010.

[16] A. Chandra, S. Fortune, and R. Lipton. Unbounded fan-in circuits and associative functions. In *ACM Symposium on Theory of Computing (STOC)*, pages 52–60, 1983.

[17] J. H. Cheon, S. Jarecki, and J. H. Seo. Multi-party privacy-preserving set intersection with quasi-linear complexity. *IEICE Trans. on Fund. of Electr., Comm. and Comp. Sci.*, E95-A(8):1366–1378, 2012.

[18] R. Cramer, I. Damgård, S. Dziembowski, M. Hirt, and T. Rabin. Efficient multiparty computations secure against an adaptive adversary. In *Advances in Cryptology – EUROCRYPT*, pages 311–326, 1999.

[19] R. Cramer, I. Damgård, and U. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In *Advances in Cryptology – EUROCRYPT*, pages 316–334, 2000.

[20] R. Cramer, I. Damgård, and J. Nielsen. Multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology – EUROCRYPT*, pages 280–300, 2001.

[21] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. Efficient robust private set intersection. In *Applied Cryptography and Network Security (ACNS)*, pages 125–142, 2009.

[22] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. Secure efficient multiparty computing of multivariate polynomials and applications. In *ACNS*, pages 130–146, 2011.

[23] I. Damgård, M. Fitzi, E. Kiltz, J. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, pages 285–304, 2006.

[24] I. Damgård, M. Geisler, M. Krøigaard, and J. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Public Key Cryptography (PKC)*, pages 160–179, 2009.

[25] I. Damgård, Y. Ishai, and M. Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Advances in Cryptology – EUROCRYPT*, pages 445–465, 2010.

[26] I. Damgård, Y. Ishai, M. Krøigaard, J. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *Advances in Cryptology – CRYPTO*, pages 241–261, 2008.

[27] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *Public Key Cryptography (PKC)*, pages 119–136, 2001.

[28] I. Damgård and J. Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, pages 572–590, 2007.

[29] E. De Cristofaro, P. Gasti, and G. Tsudik. Fast and private computation of cardinality of set intersection and union. In *International Conference on Cryptology and Network Security (CANS)*, 2012.

[30] E. De Cristofaro, J. Kim, and G. Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In *Advances in Cryptology – ASIACRYPT*, volume 6477 of *LNCS*, pages 213–231, 2010.

[31] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography and Data Security (FC)*, volume 6052 of *LNCS*, pages 143–159, 2010.

[32] E. De Cristofaro and G. Tsudik. Experimenting with fast private set intersection. In *International Conference on Trust and Trustworthy Computing (TRUST)*, pages 55–73, 2012.

[33] P.-A. Fouque, G. Poupard, and J. Stern. Sharing decryption in the context of voting or lotteries. In *International Conference on Financial Cryptography (FC)*, volume 1962 of *LNCS*, pages 90–104, 2000.

[34] M. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology – EUROCRYPT*, volume 3027 of *LNCS*, pages 1–19, 2004.

[35] Keith Frikken. Privacy-preserving set union. In *ACNS*, volume 4521 of *LNCS*, pages 237–252, 2007.

[36] R. Gennaro, M. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *ACM PODC*, pages 101–111, 1998.

[37] The GNU multiple precision arithmetic library release 5.0.5. `http://gmplib.org/`, 2012.

[38] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC*, pages 218–229, 1987.

[39] M. Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. In *SODA*, pages 1262–1277, 2010.

[40] M. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *ACM Symposium on Parallelism in Algorithms and Architectures*, pages 379–388, 2011.

[41] M. Goodrich. Spin-the-bottle sort and annealing sort: Oblivious sorting via round-robin random comparisons. In *Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, 2011.

[42] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *Theory of Cryptography Conference (TCC)*, pages 155–175, 2008.

[43] C. Hazay and K. Nissim. Efficient set operations in the presence of malicious adversaries. In *PKC*, 2010.

[44] M. Hirt and U. Maurer. Robustness for free in unconditional multi-party computation. In *Advances in Cryptology – CRYPTO*, pages 101–118, 2001.

[45] M. Hirt and J. Nielsen. Robust multiparty computation with linear communication complexity. In *CRYPTO*, pages 463–482, 2006.

[46] J. Hong, J. W. Kim, J. Kim, K. Park, and J. H. Cheon. Constant-round privacy preserving multiset union. Cryptology ePrint Achive Report 2011/138, 2011. `http://eprint.iacr.org/2011/138`.

[47] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *Network & Distributed System Security Symposium (NDSS)*, 2012.

[48] S. Jarecki and X. Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In *Theory of Cryptography Conference (TCC)*, pages 577–594, 2009.

[49] S. Jarecki and X. Liu. Fast secure computation of set intersection. In *SCN*, pages 418–435, 2010.

[50] K. Jónsson, G. Kreitz, and M. Uddin. Secure multi-party sorting and applications. Cryptology ePrint Archive Report 2011/122, 2011.

[51] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing multi-party computation. Cryptology ePrint Archive report 2011/272, 2011.

[52] L. Kissner and D. Song. Privacy-preserving set operations. In *CRYPTO*, pages 241–257, 2005.

[53] H. T. Kung and P. Lehman. Systolic (VLSI) arrays for relational database operations. In *ACM SIGMOD International Conference on Management of Data*, pages 105–116, 1980.

[54] E. Kushilevitz, Y. Lindell, and T. Rabin. Information-theoretically secure protocols and security under composition. *SIAM Journal of Computing*, 39(5):2090–2112, 2010.

[55] T. Leighton and C. Plaxton. Hypercubic sorting networks. *SIAM Journal of Computing*, 27:1–47, 1998.

[56] R. Li and C Wu. An unconditionally secure protocol for multi-party set intersection. In *ACNS*, 2007.

[57] Y. Lindell. General composition and universal composability in secure multi-party computation. In *FOCS*, pages 394–403, 2003.

[58] G. Narayanan, T. Aishwarya, A. Agrawal, A. Patra, A. Choudhary, and C. Rangan. Multi party distributed private matching, set disjointness and cardinality of set intersection with information theoretic security. In *Cryptology and Network Security (CANS)*, pages 21–40, 2009.

[59] A. Patra, A. Choudhary, and C. Rangan. Information theoretically secure multi party set intersection re-visited. In *Selected Areas in Cryptography*, pages 71–91, 2009.

[60] A. Patra, A. Choudhary, and C. Rangan. Round efficient unconditionally secure MPC and multiparty set intersection with optimal resilience. In *INDOCRYPT*, pages 398–417, 2009.

[61] K. Peng and F. Bao. An efficient range proof scheme. In *IEEE PASSAT*, pages 826–833, 2010.

[62] T. Raeder, M. Blanton, N. Chawla, and K. Frikken. Privacy-preserving network aggregation. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 198–207, 2010.

[63] Y. Sang and H. Shen. Efficient and secure protocols for privacy-preserving set operations. *ACM Transactions on Information and System Security*, 13(1):9:1–9:35, 2009.

[64] Information security in supply chain management (SecureSCM) project deliverable D9.2. University of Mannheim, July 2009.

[65] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[66] A. K. Sood and M. Abdelguerfi. Parallel and pipelined processing of some relational algebra operations. *International Journal of Electronics*, 59(4):477–482, 1985.

[67] T. Toft. Sub-linear, secure comparison with two non-colluding parties. In *PKC*, pages 174–191, 2011.

[68] J. Vaidya and C. Clifton. Secure set intersection cardinality with applications to association rule mining. *Journal of Computer Security*, 13(4):593–622, 2005.

[69] A. Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, 1968.

[70] C. Wang, K. Ren, and J. Wang. Secure and practical outsourcing of linear programming in cloud computing. In *INFOCOM*, pages 820–828, 2011.

[71] A. Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167, 1986.

[72] B. Zhang. Generic constant-round oblivious sorting algorithm for MPC. In *ProvSec*, pages 240–256, 2011.

A Multiset Protocols using General Multiset-to-Set Conversion

The multiset intersection protocol, MInt, is somewhat similar to MUnion. To obtain MInt with the optimized performance of Protocol 2, we replace lines 3–9 in MUnion (Protocol 11) with the appropriate logic, resulting in the following protocol (as before, m is compact for $m_1 + m_2$):

Protocol 18. $\langle[x_1],[y_1]\rangle\ldots,\langle[x_{m_1+m_2}],[y_{m_1+m_2}]\rangle \leftarrow \mathsf{MInt}(\langle[x'_1],[y'_1]\rangle,\ldots,\langle[x'_{m_1}],[y'_{m_1}]\rangle,\langle[x''_1],[y''_1]\rangle,\ldots,\langle[x''_{m_2}],[y''_{m_2}]\rangle)$

```
 1. $k \leftarrow \max(m_1, m_2) + 1$;
 2. $\langle[\alpha_1],[\beta_1],[\gamma_1]\rangle,\ldots,\langle[\alpha_{m_1+m_2}],[\beta_{m_1+m_2}],[\gamma_{m_1+m_2}]\rangle \leftarrow \mathsf{SortT}(\langle k[x'_1]+[y'_1],[x'_1],[y'_1]\rangle,\ldots,\langle k[x'_{m_1}]+[y'_{m_1}],[x'_{m_1}],[y'_{m_1}]\rangle,\langle k[x''_1]+[y''_1],[x''_1],[y''_1]\rangle,\ldots,\langle k[x''_{m_2}]+[y''_{m_2}],[x''_{m_2}],[y''_{m_2}]\rangle,\ell+\lceil\log k\rceil)$;
                                                                   // section 3.2
 3. for $i = 1$ to $\lfloor(m-1)/2\rfloor$ do in parallel
 4.     $[u_i] \leftarrow \mathsf{Eq}([\alpha_{2i}],[\alpha_{2i-1}],\ell+\lceil\log k\rceil)$;          // section 3.2
 5.     $[v_i] \leftarrow \mathsf{Eq}([\alpha_{2i}],[\alpha_{2i+1}],\ell+\lceil\log k\rceil)$;          // section 3.2
 6.     $[x_i] \leftarrow ([u_i]+[v_i])[\beta_i]$;                          // 1 round, $\lfloor(m-1)/2\rfloor$ inv
 7.     $[y_i] \leftarrow ([u_i]+[v_i])[\gamma_i]$;                          // optional
 8. if ($m \bmod 2 = 0$)
 9.     $[u_{m/2}] \leftarrow \mathsf{Eq}([\alpha_m],[\alpha_{m-1}],\ell+\lceil\log k\rceil)$;            // section 3.2
10.     $[x_{m/2}] \leftarrow [u_{m/2}][\beta_m]$;                             // 1 inv
11.     $[y_{m/2}] \leftarrow [u_{m/2}][\gamma_m]$;                             // optional
12. return $[x_1],\ldots,[x_{\lfloor m/2\rfloor}]$;
```

The multiset version of our subset relation protocol MSub returns only a single bit and can be constructed from the multiset union by simply replacing lines 3–9 with:

```
3. for $i = 2$ to $m_1 + m_2$ do in parallel $[u_i] \leftarrow \mathsf{Eq}([\gamma_i],[\gamma_{i-1}],\ell+\lceil\log k\rceil)$;    // section 3.2
4. $[t] \leftarrow \sum_{i=2}^{m_1+m_2}[u_i]$;
5. $[s] \leftarrow \mathsf{Eq}([t], m_1, \lceil\log m_1\rceil)$;                                  // section 3.2
6. return $[s]$;
```

It is also not very difficult to derive the multiset difference protocol MDiff from its set version Diff, which we provide next.

Protocol 19. $\langle[x_1],[y_1]\rangle\ldots,\langle[x_{m_1+m_2}],[y_{m_1+m_2}]\rangle \leftarrow \mathsf{MDiff}(\langle[x'_1],[y'_1]\rangle,\ldots,\langle[x'_{m_1}],[y'_{m_1}]\rangle,\langle[x''_1],[y''_1]\rangle,\ldots,\langle[x''_{m_2}],[y''_{m_2}]\rangle)$

```
 1. $k \leftarrow \max(m_1, m_2) + 1$;
 2. $\langle[\alpha_1],[\beta_1],[\gamma_1],[\delta_1]\rangle,\ldots,\langle[\alpha_{m_1+m_2}],[\beta_{m_1+m_2}],[\gamma_{m_1+m_2}],[\delta_{m_1+m_2}]\rangle \leftarrow \mathsf{SortT}(\langle k[x'_1]+[y'_1],[x'_1],[y'_1],[0]\rangle,\ldots,\langle k[x'_{m_1}]+[y'_{m_1}],[x'_{m_1}],[y'_{m_1}],[0]\rangle,\ldots,\langle k[x''_1]+[y''_1],[x''_1],[y''_1],[1]\rangle,\ldots,\langle k[x''_{m_2}]+[y''_{m_2}],[x''_{m_2}],[y''_{m_2}],[1]\rangle,$
    $\ell+\lceil\log k\rceil)$;                                            // section 3.2
 3. for $i = 1$ to $m_1 + m_2 - 1$ do in parallel $[u_i] \leftarrow \mathsf{Eq}([\alpha_i],[\alpha_{i+1}],\ell+\lceil\log k\rceil)$; // section 3.2
 4. $[x_1] \leftarrow [\beta_1](1-[u_1])$;                                    // 1 round, 1 inv
 5. $[y_1] \leftarrow [\gamma_1](1-[u_1])$;                                    // optional
 6. $[x_{m_1+m_2}] \leftarrow [\beta_{m_1+m_2}](1-[u_{m_1+m_2-1}])$;                    // 1 inv
 7. $[y_{m_1+m_2}] \leftarrow [\gamma_{m_1+m_2}](1-[u_{m_1+m_2-1}])$;                    // optional
 8. for $i = 2$ to $m_1 + m_2$ do in parallel
 9.     $[x_i] \leftarrow [\beta_i](1-[u_i]-[u_{i-1}])$;                        // $m_1 + m_2 - 2$ inv
10.     $[y_i] \leftarrow [\gamma_i](1-[u_i]-[u_{i-1}])$;                        // optional
```

```
11. for $i = 1$ to $m_1 + m_2$ do in parallel
12.     $[x_i] \leftarrow [\beta_i](1 - [\delta_i])$;        // 1 round, $m_1 + m_2$ inv
13.     $[y_i] \leftarrow [\gamma_i](1 - [\delta_i])$;        // optional
14. return $\langle [x_1], [y_1] \rangle, \ldots, \langle [x_{m_1+m_2}], [y_{m_1+m_2}] \rangle$;
```

In this protocol sorting is done with respect to the first element of each (4-)tuple. Symmetric difference can be obtained by skipping lines 11–13. As before, we will execute the lines marked as optional only if the counts need to be preserved.