Privacy-Preserving Minimum Spanning Trees through Oblivious Parallel RAM for Secure Multiparty Computation

Peeter Laud
Cybernetica AS
peeter.laud@cyber.ee

November 25, 2014

Abstract

In this paper, we describe efficient protocols to perform in parallel many reads and writes in private arrays according to private indices. The protocol is implemented on top of the Arithmetic Black Box (ABB) and can be freely composed to build larger privacy-preserving applications. For a large class of secure multiparty computation (SMC) protocols, we believe our technique to have better practical and asymptotic performance than any previous ORAM technique that has been adapted for use in SMC.

Our ORAM technique opens up a large class of parallel algorithms for adoption to run on SMC platforms. In this paper, we demonstrate how the minimum spanning tree (MST) finding algorithm by Awerbuch and Shiloach can be executed without revealing any details about the underlying graph (beside its size). The data accesses of this algorithm heavily depend on the location and weight of edges (which are private) and our ORAM technique is instrumental in their execution. Our implementation is probably the first-ever realization of a privacy-preserving MST algorithm.

1 Introduction

In Secure Multiparty Computation (SMC), k parties compute $(y_1, \ldots, y_k) = f(x_1, \ldots, x_k)$, with the party P_i providing the input x_i and learning no more than the output y_i. For any functionality f, there exists a SMC protocol for it [Yao82, GMW87]. A universally composable [Can01] abstraction for SMC is the arithmetic black box (ABB) [DN03]. The ideal functionality $\mathcal{F}_{\mathsf{ABB}}$ allows the parties to store private data in it, perform computations with data inside the ABB, and reveal the results of computations. This means that the ABB does not leak anything about the results of the intermediate computations, but only those values whose declassification is explicitly requested by the parties. Hence, any secure implementation of ABB also protects the secrecy of inputs and intermediate computations. There exist a number of practical implementations of the ABB [BDNP08, DGKN09, BLW08, BSMD10, HKS$^+$10, MK10], differing in the underlying protocol sets they use and in the set of operations with private values that they make available for higher-level protocols.

These ABB implementations may be quite efficient for realizing applications working with private data, if the control flow and the data access patterns of the application do not depend on private values. For hiding data access patterns, oblivious RAM (ORAM) techniques [GO96] may be used. These techniques have a significant overhead, which is increased when they are combined with SMC. Existing combinations of ORAM with SMC report at least $O(\log^3 n)$ overhead for accessing an element of an n-element array [KS14].

In this work, we propose a different method for reading and writing data in SMC according to private addresses. We note that SMC applications are often highly parallelized, because

the protocols provided by ABB implementations often have significant latency. We can exploit this parallelism in designing oblivious data access methods, by trying to bundle several data accesses together. In the following, we assume that we have private vector $\vec{v}$ of m elements (the number m is public, as well as the sizes of other pieces of data). We provide two protocols on top of ABB: for reading its elements n times, and for writing its elements n times. These protocols receive as an input a vector of indices (of length n) and, in case of writing, a new vector of values, and return a vector of selected elements of $\vec{v}$, or the updated vector $\vec{v}$. The asymptotic complexity of both protocols is $O((m+n)\log(m+n))$, while the constants hidden in the O-notation should be reasonable. These protocols could be interleaved with the rest of the SMC application in order to provide oblivious data access capability to it.

To demonstrate the usefulness of our protocols in privately implementing algorithms with private data access, and to expand the set of problems for which there exist reasonably efficient privacy-preserving protocols, we provide a protocol for finding the minimum spanning tree (MST) of a weighted graph. The protocol is implemented on top of ABB, i.e. the are no assumptions on which computing party initially knows which parts of the description of the graph. Kruskal's and Prim's algorithms [CLRS01], the best-known algorithms for finding MST (without privacy considerations), are unsuitable for direct implementation on top of ABB, because of their inherent sequentiality. In this paper, we consider a parallel algorithm by Awerbuch and Shiloach [AS87] (itself an adoption of a MST algorithm by Borůvka [NMN01]) that runs on priority-CRCW PRAM in time logarithmic to the size of the graph, using as many processors as the graph has edges. Hence the workload of this algorithm is asymptotically the same as Kruskal's. We adapt it to run on top of our ABB implementation, using the ORAM protocols we've developed. The adoption involves the simplification of the algorithm's control flow, and choosing the most suitable variants of our protocols at each oblivious read or write. The efficiency of the resulting protocol is very reasonable; our adoption only carries the cost of an extra logarithmic factor. We believe that the adoption of other PRAM algorithms for SMC is a promising line of research.

This paper has the following structure. After reviewing the related work in Sec. 2 and providing the necessary preliminaries, in particular the ABB model of SMC in Sec. 3, we give the actual protocols for parallel reading and writing in Sec. 4 and discuss both their theoretical and practical performance in Sec. 5. We will then move on to solving the MST problem, reviewing a parallel MST algorithm in Sec. 6 and describing its adoption to privacy-preserving execution in Sec. 7, again quantifying its performance. We conclude in Sec. 8.

2 Related work

Secure multiparty computation (SMC) protocol sets can be based on a variety of different techniques, including garbled circuits [Yao82], secret sharing [Sha79, GRR98, BNTW12] or homomorphic encryption [CDN01]. A highly suitable abstraction of SMC is the universally composable Arithmetic Black Box (ABB) [DN03], the use of which allows very simple security proofs for higher-level SMC applications. Using the ABB to derive efficient privacy-preserving implementations for various computational tasks is an ongoing field of research [DFK$^+$06, CS10, ACM$^+$13, LT13], also containing this paper.

Protocols for oblivious RAM [GO96] have received significant attention during recent years [SCSL11, KLO12, SvDS$^+$13]. The overhead of these protocols is around $O(\log^2 m)$ when accessing an element of a vector of length m (with elements of size $O(\log m)$). These ORAM constructions assume a client-server model, with the client accessing the memory held by the server, which remains oblivious to the access patterns. This model is simpler than the SMC model, because the client's and server's computations are not shared among several parties.

In this paper, we use SMC techniques to achieve oblivious data access in SMC applications. This goal has been studied before, by implementing the client's computations in an ORAM protocol on top of a secure two-party computation protocol set [GKK+12, GGH+13, LHS+14, GHJR14], or over an SMC protocol set [DMN11, KS14]. For these protocol sets, the overhead of at least $O(\log^3 m)$ is reported. Recently, optimizing ORAM to perform well in the secure computation setting has become a goal of its own [WHC+14].

The ORAM constructions often allow only sequential access to data, as the updating of the data structures maintained by the server cannot be parallelized. Recently, Oblivious Parallel RAM [BCP14] has been proposed, which may be more suitable for SMC protocol sets where the computations have significant latency.

Our parallel reading protocol essentially builds and then applies an *oblivious extended permutation (OEP)* [KS08, HEK12, MS13, LW14] (see Sec. 4.1 for details). Our OEP application protocol is more efficient (both in practice and/or asymptotically) than any other published construction built with SMC techniques. The building of an OEP in composable manner has only been considered in [LW14]; our construction is more efficient than theirs.

We are aware of no previous attempts to compute the MST of a graph in a privacy-preserving manner. For graph algorithms, there exist privacy-preserving protocols for single-source shortest paths and for maximum flow [BS05, ACM+13, KS14].

3 Preliminaries

Universal composability (UC) [Can01] is a framework for stating security properties of systems. It considers an ideal functionality $\mathcal{F}$ and its implementation π with identical interfaces to the intended users. The latter is *at least as secure as* the former, if for any attacker $\mathcal{A}$ there exists an attacker $\mathcal{A}_S$, such that $\pi \| \mathcal{A}$ and $\mathcal{F} \| \mathcal{A}_S$ are indistinguishable to any potential user of π / $\mathcal{F}$. The value of the framework lies in the composability theorem: if π is at least secure as $\mathcal{F}$, then ξ^π is at least as secure as $\xi^\mathcal{F}$ for any system ξ that uses π / $\mathcal{F}$. We say that such ξ is implemented *in the $\mathcal{F}$-hybrid model*.

The *arithmetic black box* is an ideal functionality $\mathcal{F}_{\mathsf{ABB}}$. It allows its users (a fixed number p of parties) to securely store and retrieve values, and to perform computations with them. When a party sends the command $\mathsf{store}(v)$ to $\mathcal{F}_{\mathsf{ABB}}$, where v is some value, the functionality assigns a new *handle* h (sequentially taken integers) to it by storing the pair (h, v) and sending h to all parties. If a sufficient number (depending on implementation details) of parties send the command $\mathsf{retrieve}(h)$ to $\mathcal{F}_{\mathsf{ABB}}$, it looks up (h, v) among the stored pairs and responds with v to all parties. When a sufficient number of parties send the command $\mathsf{compute}(op; h_1, \ldots, h_k; params)$ to $\mathcal{F}_{\mathsf{ABB}}$, it looks up the values $v_1, \ldots, v_k$ corresponding to the handles $h_1, \ldots, h_k$, performs the operation op (parametrized with $params$) on them, stores the result v together with a new handle h, and sends h to all parties. In this way, the parties can perform computations without revealing anything about the intermediate values or results, unless a sufficiently large coalition wants a value to be revealed. In this paper our protocols are given in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model.

The existing implementations of ABB are protocol sets π_{ABB} based on either secret sharing [DGKN09, BLW08, BSMD10] or threshold homomorphic encryption [DN03, HKS+10]. Depending on the implementation, the ABB offers protection against a honest-but-curious, or a malicious party, or a number of parties (up to a certain limit). E.g. the implementation of the ABB by Sharemind [BLW08] consists of three parties, providing protection against one honest-but-curious party.

All ABB implementations provide protocols for computing linear combinations of private values (with public coefficients) and for multiplying private values. The linear combination protocol is typically cheap, involving no communication and/or cheap operations with values.

When estimating the (asymptotic) complexity of protocols built on top of ABB, it is typical to disregard the costs of computing of linear combinations. Other operations, e.g. comparison, can be built on top of addition and multiplication [DFK+06, NO07], or the ABB implementation may have dedicated protocols for these operations [BNTW12]. In this paper, we require the ABB implementation to provide protocols also for equality and comparison operations. In the protocols we present, we let $[\![x]\!]$ denote that some value has been stored in the ABB and is accessible under handle x. The notation $[\![z]\!] \leftarrow [\![x]\!] \otimes [\![y]\!]$ means that the operation $\otimes$ is performed with values stored under handles x and y, and the result is stored under handle z. In ABB implementations this involves the invocation of the protocol for $\otimes$.

We also require the ABB to provide *oblivious shuffles* — private permutations of values. For certain ABB implementations, these can be added as described in [LWZ11]. A more general approach is to use Waksman networks [Wak68]. Given an oblivious shuffle $[\![\sigma]\!]$ for m elements, and a private vector $([\![v_1]\!], \ldots, [\![v_m]\!])$, it is possible to apply this shuffle to this vector, permuting its elements and resulting in the vector $([\![v_{\sigma(1)}]\!], \ldots, [\![v_{\sigma(m)}]\!])$. It is also possible to *unapply* the shuffle to this vector, performing the inverse permutation of its elements. The complexity of the protocols implementing these ABB operations is either $O(m)$ or $O(m \log m)$ (for constant number of parties).

With oblivious shuffles and comparison operations, vectors of private values (of length m) can be sorted in $O(m \log m)$ time, where the size of the constants hidden in the O-notation is reasonable [HKI+12]. In our protocols, we let $[\![\sigma]\!] \leftarrow \mathsf{sort}([\![\vec{v}]\!])$ denote the sorting operation applied to the private vector $[\![\vec{v}]\!]$. This operation does not actually reorder $\vec{v}$, but produces an oblivious shuffle $[\![\sigma]\!]$, the application of which to $\vec{v}$ would bring it to sorted order. We require the sorting to be stable and the sorting protocol to be universally composable. In effect, this makes sort a yet another operation provided by the ABB.

4 Oblivious data access

4.1 Protocol for reading

In Alg. 1, we present our protocol for obliviously reading several elements of an array. Given a vector $\vec{v}$ of length m, we let $\mathsf{prefixsum}(\vec{v})$ denote a vector $\vec{w}$, also of length m, where $w_i = \sum_{j=1}^{i} v_j$ for all $j \in \{1, \ldots, m\}$. Computing $\mathsf{prefixsum}([\![\vec{v}]\!])$ is a free operation in existing ABB implementations, because addition of elements, not requiring any communication between the parties, is counted as having negligible complexity. We can also define the inverse operation $\mathsf{prefixsum}^{-1}$: if $\vec{w} = \mathsf{prefixsum}(\vec{v})$ then $\vec{v} = \mathsf{prefixsum}^{-1}(\vec{w})$. The inverse operation is even easier to compute: $v_1 = w_1$ and $v_i = w_i - w_{i-1}$ for all $i \in \{2, \ldots, m\}$.

We see that in Alg. 1, the permutation σ orders the indices which we want to read, as well as the indices $1, \ldots, n$ of the "original array" $\vec{v}$. Due to the stability of the sort, each index of the "original array" ends up before the reading indices equal to it. In $\mathsf{apply}(\sigma, \vec{u})$, each element v'_i of $\vec{v}'$, located in the same position as the index i of the "original array" in sorted $\vec{t}$, is followed by zero or more 0-s. The prefix summing restores the elements of $\vec{v}$, with the 0-s also replaced with the element that precedes them. Unapplying σ restores the original order of $\vec{u}$ and we can read out the elements of $\vec{v}$ from the latter half of $\vec{u}'$. A small example is presented in Fig. 1.

The protocol presented in Alg. 1 clearly preserves the security guarantees of the implementation of the underlying ABB, as it applies only ABB operations, classifies only public constants and declassifies nothing. Its complexity is dominated by the complexity of the sorting operation, which is $O((m+n)\log(m+n))$. We also note that the round complexity of Alg. 1 is $O(\log(m+n))$.

Instead of reading elements from an array, the elements of which are indexed with $1, \ldots, m$,

Algorithm 1: Reading n values from the private array

Data: A private vector $[\![\vec{v}]\!]$ of length m
Data: A private vector $[\![\vec{z}]\!]$ of length n, with $1 \leq z_i \leq m$ for all i
Result: A private vector $[\![\vec{w}]\!]$ of length n, with $w_i = v_{z_i}$ for all i

1 **foreach** $i \in \{1, \ldots, m\}$ **do** $[\![t_i]\!] \leftarrow i$;
2 **foreach** $i \in \{1, \ldots, n\}$ **do** $[\![t_{m+i}]\!] \leftarrow [\![z_i]\!]$;
3 $[\![\sigma]\!] \leftarrow \mathsf{sort}([\![\vec{t}]\!])$
. . . . . .
4 $[\![\vec{v}']\!] \leftarrow \mathsf{prefixsum}^{-1}([\![\vec{v}]\!])$
5 **foreach** $i \in \{1, \ldots, m\}$ **do** $[\![u_i]\!] \leftarrow [\![v'_i]\!]$;
6 **foreach** $i \in \{1, \ldots, n\}$ **do** $[\![u_{m+i}]\!] \leftarrow 0$;
7 $[\![\vec{u}']\!] \leftarrow \mathsf{unapply}([\![\sigma]\!]; \mathsf{prefixsum}(\mathsf{apply}([\![\sigma]\!]; [\![\vec{u}]\!])))$
8 **foreach** $i \in \{1, \ldots, n\}$ **do** $[\![w_i]\!] \leftarrow [\![u'_{m+i}]\!]$;
9 **return** $[\![\vec{w}]\!]$

Let $\vec{v} = (1, 4, 9, 16, 25)$. Let $\vec{z} = (3, 2, 4, 3)$. The intermediate values are the following.

- $\vec{t} = (1, 2, 3, 4, 5, 3, 2, 4, 3)$
- σ is the permutation

1	2	3	4	5	6	7	8	9
1	2	7	3	6	9	4	8	5

  meaning that e.g. the 3rd element in the sorted vector is the 7th element in the original vector.
- $\vec{v}' = (1, 3, 5, 7, 9)$ and $\vec{u} = (1, 3, 5, 7, 9, 0, 0, 0, 0)$.
- After applying σ to $\vec{u}$, we obtain the vector $(1, 3, 0, 5, 0, 0, 7, 0, 9)$.
- After prefixsumming, we get the vector $(1, 4, 4, 9, 9, 9, 16, 16, 25)$. Denote it with $\vec{y}$.
- After applying the inverse of σ, we get $\vec{u}' = (1, 4, 9, 16, 25, 9, 4, 16, 9)$. Indeed, to find e.g. u'_7, we look for "7" in the lower row of the description of σ. We find "3" in the upper row, meaning that $u'_7 = y_3$.
- Finally, we return the last n elements of $\vec{u}'$, which are $\vec{w} = (9, 4, 16, 9)$.

All values are private, i.e. stored in the ABB.

Figure 1: Example of private reading according to Alg. 1

the presented protocol could also be used to read the private values from a dictionary, the elements of which are indexed with (private) $[\![j_1]\!], \ldots, [\![j_m]\!]$. In this case, in line 1, $[\![t_i]\!]$ is not initialized with i, but with $[\![j_i]\!]$. Note that in this case, the algorithm cannot detect if all indices that we attempt to read are present in the dictionary.

Note that in Alg. 1, the argument $[\![\vec{v}]\!]$ is only used after the dotted line. At the same time, the step that dominates the complexity of the protocol — sorting of $[\![\vec{t}]\!]$ in line 3 — takes place before the dotted line. Hence, if we read the same positions of several vectors, we could execute the upper part of Alg. 1 only once and the lower part as many times as necessary. In Sec. 7, we will denote the upper part of Alg. 1 with `prepareRead` (with inputs $[\![\vec{z}]\!]$ and m, and output $[\![\sigma]\!]$), and the lower part with `performRead` (with inputs $[\![\vec{v}]\!]$ and $[\![\sigma]\!]$).

An *extended permutation* [MS13] from m elements to n elements is a mapping from $\{1,\dots,n\}$ to $\{1,\dots,m\}$ (note the contravariance). The *application* of an extended permutation ϕ to a vector $(x_1,\dots,x_m)$ produces a vector $(y_1,\dots,y_n)$, where $y_i = x_{\phi(i)}$. An oblivious extended permutation (OEP) protocol preserves the privacy of $\vec{x}$, $\vec{y}$ and ϕ. The `prepareRead` protocol essentially constructs the representation $[\![\phi]\!]$ of an OEP and the `performRead` protocol applies it, with better performance than previous constructions.

4.2 Protocol for writing

For specifying the parallel writing protocol, we have to fix how multiple attempts to write to the same field are resolved. We thus require that each writing request comes with a numeric *priority*; the request with highest priority goes through (if it is not unique, then one is selected arbitrarily). We can also give priorities to the existing elements of the array. Normally they should have the lowest priority (if any attempt to write them actually means that they must be overwritten). However, in case where the array element collects the maximum value during some process (e.g. finding the best path from one vertex of some graph to another), with the writes to this element representing candidate values, the priority of the existing element could be equal to this element. This is useful in e.g. the Bellman-Ford algorithm for computing shortest distances.

Thus assume that there exists an algorithm compute_priority which, when applied to an element $[\![w_i]\!]$ of the vector $[\![\vec{w}]\!]$, as well as to its index i, returns the priority of keeping the current value of w_i. The parallel writing protocol is given in Alg. 2. The writing algorithm receives a vector of values $[\![\vec{v}]\!]$ to be written, together with the indices $[\![\vec{j}]\!]$ showing where they have to be written, and the writing priorities $[\![\vec{p}]\!]$. Alg. 2 transforms the current vector $[\![\vec{w}]\!]$ (its indices and priorities) to the same form and concatenates it with the indices and priorities of the write requests. The data are then sorted according to indices and priorities (with higher-priority elements coming first). The operation *zip* on two vectors of equal length transforms them to a vector of pairs; the ordering on these pairs is determined lexicographically. The vector $[\![\vec{b}]\!]$ is used to indicate the highest-priority position for each index: $b_i = 0$ iff the i-th element in the vector $\vec{j}'$ is the first (hence the highest-priority) value equal to j'_i. Note that all equality checks in line 10 can be done in parallel. Here and elsewhere, **foreach**-statements denote parallel execution. Performing the sort in line 11 moves the highest-priority values to the first m positions. The sorting is stable, hence the values correspond to the indices $1,\dots,m$ in this order. We thus have to apply the shuffles induced by both sorts to the vector of values $\vec{v}' = \vec{v}\|\vec{w}$, and take the first m elements of the result.

We find Alg. 2 to be conceptually simpler than Alg. 1 and thus do not provide an example. The writing protocol is secure for the same reasons as the reading protocol. Its complexity is dominated by the two sorting operations, it is $O((m+n)\log(m+n))$, with the round complexity being $O(\log(m+n))$. Similarly to the reading protocol, the writing protocol can be adapted to write into a dictionary instead. Another similarity is the dotted line — the complex sorting operations above the line only use the indices and priorities, while the actual values are used only in cheap operations below the line. For the purposes of Sec. 7, we thus introduce the protocols `prepareWrite` which executes the operations above the dotted line, and `performWrite`, executing the operations below the line. The protocol `prepareWrite` receives as inputs $\vec{j}$, $\vec{p}$, and the length m of $\vec{w}$. It lets the existing elements of $\vec{w}$ to have the least possible priority, i.e. they will be definitely written over if there is at least one request to do it (prioritizing existing elements of $\vec{w}$ is not needed in Sec. 7). The output of `prepareWrite` is the pair of oblivious shuffles $([\![\sigma]\!],[\![\tau]\!])$. These are input to `performWrite` together with $[\![\vec{v}]\!]$ and $[\![\vec{w}]\!]$.

Algorithm 2: Obliviously writing n values to a private array

Data: Private vectors $[\![\vec{j}]\!]$, $[\![\vec{v}]\!]$, $[\![\vec{p}]\!]$ of length n, where $1 \le j_i \le m$ for all i
Data: Private array $[\![\vec{w}]\!]$ of length m
Result: Updated $\vec{w}$: values in $\vec{v}$ written to indices in $\vec{j}$, if priorities in $\vec{p}$ are high enough

1 **foreach** $i \in \{1, \dots, n\}$ **do**
2 $\quad [\![j'_i]\!] \leftarrow [\![j_i]\!]$
3 $\quad [\![p'_i]\!] \leftarrow -[\![p_i]\!]$
4 **foreach** $i \in \{1, \dots, m\}$ **do**
5 $\quad [\![j'_{n+i}]\!] \leftarrow i$
6 $\quad [\![p'_{n+i}]\!] \leftarrow -\mathsf{compute\_priority}(i, [\![w_i]\!])$
7 $[\![\sigma]\!] \leftarrow \mathsf{sort}(zip([\![\vec{j'}]\!], [\![\vec{p'}]\!]))$
8 $[\![\vec{j''}]\!] \leftarrow \mathsf{apply}([\![\sigma]\!]; [\![\vec{j'}]\!])$
9 $[\![b_1]\!] \leftarrow 0$
10 **foreach** $i \in \{2, \dots, N\}$ **do** $[\![b_i]\!] \leftarrow [\![j''_i]\!] \stackrel{?}{=} [\![j''_{i-1}]\!]$;
11 $[\![\tau]\!] \leftarrow \mathsf{sort}([\![\vec{b}]\!])$
. . . . . .
12 **foreach** $i \in \{1, \dots, n\}$ **do** $[\![v'_i]\!] \leftarrow [\![v_i]\!]$;
13 **foreach** $i \in \{1, \dots, m\}$ **do** $[\![v'_{n+i}]\!] \leftarrow [\![w_i]\!]$;
14 $[\![\vec{w'}]\!] \leftarrow \mathsf{apply}([\![\tau]\!]; \mathsf{apply}([\![\sigma]\!]; [\![\vec{v'}]\!]))$
15 **foreach** $i \in \{1, \dots, m\}$ **do** $[\![w_i]\!] \leftarrow [\![w'_i]\!]$;
16 **return** $[\![\vec{w}]\!]$

4.3 Sorting bits

Alg. 2 makes two calls to the sorting protocol. While the first one of them is a rather general sort, the second one in line 11 only performs a stable sort on bits, ordering the "0" bits before the "1" bits (and the sort does not actually have to be stable on the "1"-bits). In the following we show that the second sort can be performed with the complexity similar to that of a random shuffle, instead of a full sort. Our method leaks the number of 0-s among the bits, but this information was already public in Alg. 2 (being equal to the length of $\vec{w}$). The sorting protocol is given in Alg. 3. Here $\mathsf{random\_shuffle}(n)$ generates an oblivious random shuffle for vectors of length n. The protocol ends with a composition of an oblivious and a public shuffle; this operation, as well as the generation of a random shuffle, is supported by existing implementations of shuffles.

We see that the most complex operations of Alg. 3 are the applications of the oblivious shuffle $[\![\tau]\!]$. If the communication complexity of these is $O(m)$ and the round complexity of these is $O(1)$, then this is also the complexity of the entire protocol. The protocol declassifies a number of things, hence it is important to verify that the declassified values can be simulated. The vector $\vec{b'}$ is a random permutation of 0-s and 1-s, where the number of 0-bits and 1-bits is the same as in $[\![\vec{b}]\!]$. Hence the number of 0-bits is leaked. But beside that, nothing is leaked: if the simulator knows the number n of 0-bits, then $\vec{b'}$ is a uniformly randomly chosen bit-vector with n bits "0" and $(m-n)$ bits "1".

The vector $\vec{y}$ (computed in constant number of rounds, as all declassifications can be done in parallel) is a random vector of numbers, such that $(m-n)$ of its entries equal $(m+1)$, and the rest are a uniformly random permutation of $\{1, \dots, n\}$. The numbers $\{1, \dots, n\}$ in $\vec{y}$ are located at the same places as the 0-bits in $\vec{b'}$. Hence the simulator can generate $\vec{y}$ after generating $\vec{b'}$. Beside $\vec{b'}$ and $\vec{y}$, the sorting protocol does not declassify anything else. The rest

Algorithm 3: Stable sorting of 0-bits in a bit-vector

Data: Vector of private values $[\![\vec{b}]\!]$ of length m, where each $b_i \in \{0, 1\}$
Result: Oblivious shuffle $[\![\sigma]\!]$, such that $\mathsf{apply}([\![\sigma]\!]; [\![\vec{b}]\!])$ is sorted and the order of 0-bits is not changed
Leaks: The number of 0-bits in $[\![\vec{b}]\!]$

1 **foreach** $i \in \{1, \ldots, m\}$ **do** $[\![c_i]\!] \leftarrow 1 - [\![b_i]\!]$;
2 $[\![\vec{x}]\!] \leftarrow \mathsf{prefixsum}([\![\vec{c}]\!])$
3 $[\![\tau]\!] \leftarrow \mathsf{random\_shuffle}(m)$
4 $\vec{b'} \leftarrow \mathsf{retrieve}(\mathsf{apply}([\![\tau]\!]; [\![\vec{b}]\!]))$
5 $[\![\vec{x'}]\!] \leftarrow \mathsf{apply}([\![\tau]\!]; [\![\vec{x}]\!])$
6 **foreach** $i \in \{1, \ldots, m\}$ **do**
7 $\quad y_i \leftarrow \mathsf{if}\ b'_i = 0\ \mathsf{then}\ \mathsf{retrieve}([\![x'_i]\!])\ \mathsf{else}\ m + 1$
8 Let ξ be a public shuffle that sorts $\vec{y}$
9 $[\![\sigma]\!] \leftarrow [\![\tau]\!] \circ \xi$
10 **return** $[\![\sigma]\!]$

of Alg. 3 consists of invoking the functionality of the ABB or manipulating public data.

5 Performance and applicability

Using our algorithms, the cost of n parallel data accesses is $O((m+n)\log(m+n))$, where m is the size of the vector from which we're reading values. Dividing by n, we get that the cost of one access is $O((1+\frac{m}{n})\log(m+n))$. In practice, the cost will depend a lot on our ability to perform many data accesses in parallel. Fortunately, this goal to parallelize coincides with one of the design goals for privacy-preserving applications in general, at least for those where the used ABB implementation is based on secret sharing and requires ongoing communication between the parties. Parallelization allows to reduce the number of communication rounds necessary for the application, reducing the performance penalty caused by network latency.

Suppose that our application is such that on average, we can access in parallel a fraction of $1/f(m)$ of the memory it uses (where $1 \leq f(m) \leq m$). Hence, we are performing $m/f(m)$ data accesses in parallel, requiring $O(m \log m)$ work in total, or $O(f(m) \log m)$ for one access. Recall that for ORAM implementations over SMC, the reported overheads are at least $O(\log^3 m)$. Hence our approach has better asymptotic complexity for applications where we can keep $f(m)$ small.

Parallel random access machines (PRAM) are a theoretical model for parallel computations, for which a sizable body of efficient algorithms exists. Using our parallel reading and writing protocols, any algorithm for priority-CRCW PRAM (PRAM, where many processors can read or write the same memory cell in parallel, with priorities determining which write goes through) can be implemented on an ABB, as long as the control flow of the algorithm does not depend on private data. A goal in designing PRAM algorithms is to make their running time polylogarithmic in the size of the input, while using a polynomial number of processors. There is even a large class of tasks, for which there exist PRAM algorithms with logarithmic running time.

An algorithm with running time t must on each step access on average at least $1/t$ fraction of the memory it uses. A PRAM algorithm that runs in $O(\log m)$ time must access on average at least $\Omega(1/\log m)$ fraction of its memory at each step, i.e. $f(m)$ is $O(\log m)$. When implementing such algorithm on top of SMC using the reading and writing protocols presented in this note, we can say that the overhead of these protocols is $O(\log^2 m)$. For algorithms that access a larger

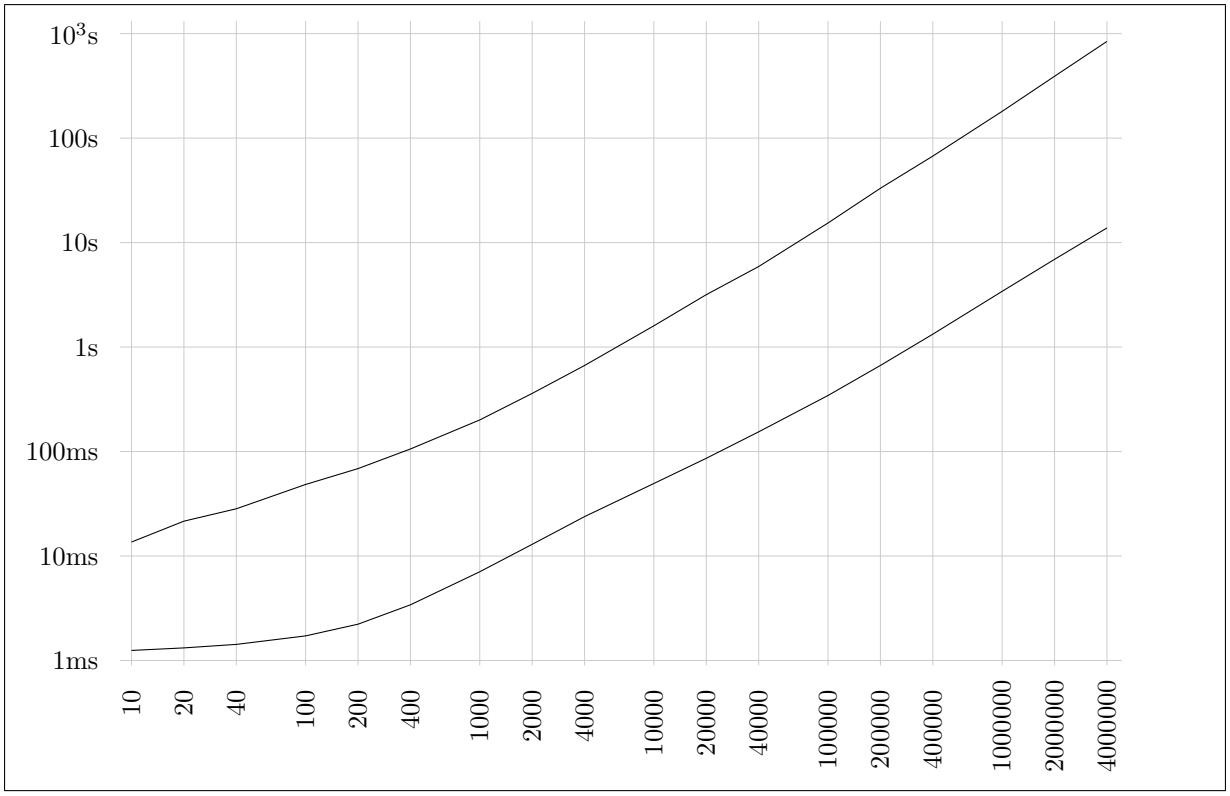

Figure 2: Times for preparing (upper) and performing (lower) a parallel read, depending on $m + n$

fraction of their memory at each step (e.g. the Bellman-Ford algorithm for finding shortest paths in graphs; for which also the optimization described above applies), the overhead is even smaller.

Experimental results

We have implemented protocols in Sec. 4 on the Sharemind secure multiparty computation platform (providing security against passive attacks by one party out of three in total) [BNTW12] and tested their performance. We measured the time it took to read n values from a vector of length m, or to write n values to a vector of length m. Due to the structure of the algorithms, the timings almost completely depend only on $m + n$ and this has been the quantity we have varied (we have always picked $m = n$).

Our performance tests are performed on a cluster of three computers with 48 GB of RAM and a 12-core 3 GHz CPU with Hyper Threading running Linux (kernel v.3.2.0-3-amd64), connected by an Ethernet local area network with link speed of 1 Gbps. The execution time of the reading protocol on this cluster for various values of $m + n$ is depicted in Fig. 2. We have split the running time into two parts, for preparing the read and for performing the read. We see that for larger values of $m + n$ the preparation is slower by almost two orders of magnitude, hence in the design of privacy-preserving algorithms one should aim for reuse of the results of preparation.

Similarly, the performance measuring results of the parallel writing are depicted in Fig. 3. Again, we distinguish the running times for preparing and performing the write, with similar differences in running times.

We see that 2 million data accesses against an array of length 2 million require about 1000

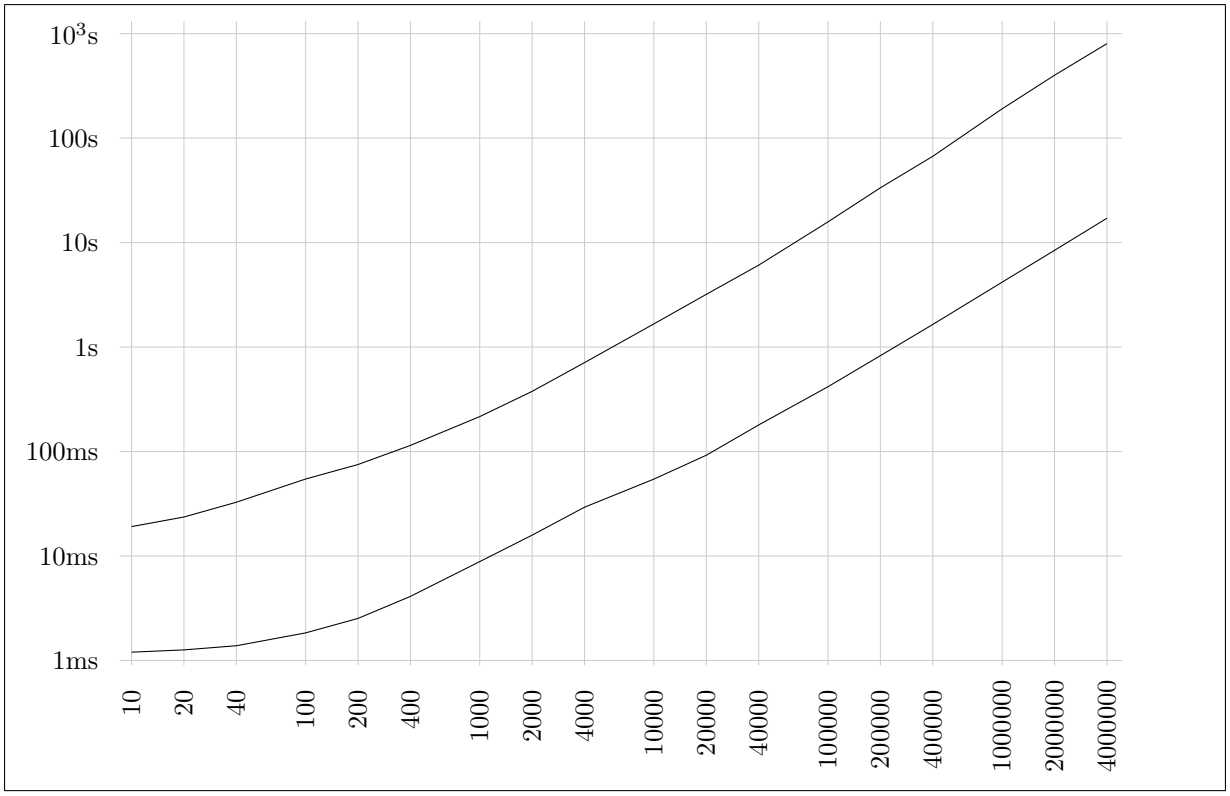

Figure 3: Times for preparing (upper) and performing (lower) a parallel write, depending on $m + n$

seconds. This makes 0.5 ms per access, which is almost two orders of magnitude faster than the times reported in [KS14]. Of course, such efficiency is possible only if the overlying application supports parallelism to this level. Also, active security is achieved in [KS14], while SHAREMIND only provides passive security. On the other hand, the protocols of [KS14] also require an expensive preprocessing phase, while nothing of that sort is required for our constructions.

6 Parallel algorithms for MST

Let $G = (V, E)$ be an undirected graph, where the set of vertices V is identified with the set $\{1, \ldots, |V|\}$ and the set of edges E with a subset of $V \times V$ (the edge between vertices u and v occurs in E both as (u, v) and as (v, u)). We assume that the graph is connected. Let $\omega : E \to \mathbb{N}$ give the weights of the edges (ω must be symmetric). A *minimum spanning tree* of G is a graph $T = (V, E')$ that is connected and for which the sum $\sum_{e \in E'} \omega(e)$ takes the smallest possible value.

Kruskal's and Prim's algorithms are the two most well-known algorithms for finding the MST of a graph. These algorithms work in time $O(|E| \log |V|)$ or $O(|E| + |V| \log |V|)$ [CLRS01]. They are inherently sequential and therefore unsuitable for an SMC implementation.

Other algorithms for MST have been proposed. Borůvka's algorithm [NMN01] works in iterations. At the beginning of each iteration, the set of vertices V has been partitioned into $V_1 \dot\cup \cdots \dot\cup V_k$ and for each V_i, the minimum spanning tree has already been found (at the start of the algorithm, each vertex is a separate part). For each i, let e_i be a minimum-weight edge connecting a vertex in V_i with a vertex in $V \backslash V_i$. We add all edges e_i to the MST we are constructing and join the parts V_i that are now connected. We iterate until all vertices are in the same part. Clearly, the number of iterations is at most $\log_2 |V|$ because the number of parts

drops to at most half during each iteration.

Borůvka's algorithm seems amenable for parallelization, as the edges e_i can all be found in parallel. Parallelizing the joining of parts is more involved. Awerbuch and Shiloach [AS87] have proposed a parallel variant of Borůvka's algorithm that introduces data structures to keep track of the parts of V, and delays the joining of some parts. Due to the delays, the number of iterations may increase, but it is shown to be at most $\log_{3/2}|V|$. Each iteration requires constant time, when executed by $|E|$ processors on priority-CRCW PRAM. The algorithm assumes that all edges have different weights (this does not lessen the generality). The rest of this section describes their algorithm.

Algorithm 4: MST algorithm by Awerbuch and Shiloach

```
   Data: Connected graph G = (V, E), edge weights ω
   Result: E' ⊆ E, such that (V, E') is the MST of G
 1 foreach (u, v) ∈ E do
 2     T[{u, v}] ← false
 3     A[(u, v)] ← true
 4 foreach v ∈ V do
 5     F[v] ← v
 6     W[v] ← NIL
 7 while ∃(u, v) ∈ E : A[(u, v)] do
 8     foreach (u, v) ∈ E where A[(u, v)] do
 9         if in_star(u) ∧ F[u] ≠ F[v] then
10             F[F[u]] ← F[v] with priority ω(u, v)
11             W[F[u]] ← {u, v} with priority ω(u, v)
12         Synchronize
13         if W[F[u]] = {u, v} then T[{u, v}] ← true;
14         if u < F[u] ∧ u = F[F[u]] then F[u] ← u;
15         Synchronize
16         if in_star(u) then
17             A[(u, v)] ← false
18         else
19             F[F[u]] ← F[u]
20 return {(u, v) ∈ E | T[{u, v}]}
```

Algorithm 5: Checking for stars in Alg. 4

```
   Data: A set V, a mapping F : V → V
   Result: Predicate St on V, indicating which elements of V belong to stars
 1 foreach v ∈ V do St[v] ← true;
 2 foreach v ∈ V do
 3     if F[v] ≠ F[F[v]] then
 4         St[v] ← false
 5         St[F[F[v]]] ← false
 6 foreach v ∈ V do St[v] ← St[v] ∧ St[F[v]];
 7 return St
```

The Awerbuch-Shiloach algorithm [AS87] is presented in Alg. 4. It uses the array $\mathcal{T}$ to record which edges have been included in the MST. For keeping track of the partitioning of V, Alg. 4 uses a union-find data structure — the array F that for each vertex records its current "parent". At each moment, F defines a forest of rooted trees, with the vertices v satisfying $v = F[v]$ being the roots. The trees of this forest correspond to the parts in the current partitioning. A rooted tree is called a *star* if its height is at most 1.

The array $\mathcal{A}$ records which edges are "active". The algorithm iterates as long as any active edges remain. The body of the **while**-loop is multi-threaded, creating one thread for each active edge. The changes a thread makes in common data are not visible to other threads until the next **Synchronize**-statement. In particular, the reads and writes of F in line 10 by different threads do not interfere with each other.

The Awerbuch-Shiloach algorithm joins two parts in the current partitioning of V or, two rooted trees in the forest defined by F, only if at least one of them is a star. Computing, which vertices belong in stars, can be done in constant time with $|V|$ processors. At each iteration of Alg. 4, before executing the lines 9 and 16, the algorithm Alg. 5 is invoked and its output is used to check whether the vertex u belongs to a star.

An iteration of Alg. 4 can be seen as a sequence of three *steps*, separated by the **Synchronize**-statements. In first step, the edges to be added to the tree are selected. For each star with root $r \in V$, the lightest outgoing edge is selected and stored in $\mathcal{W}[r]$. This selection crucially depends on the prioritized writing; the writing with smallest priority will go through. Also, the star is made a part of another tree, by changing the F-ancestor of r. In the second step, we break the F-cycles of length 2 that may have resulted from joining two stars. Independently, we also record the edges added to the MST. In the third step, we decrease the height of F-trees, as well as deactivate the edges that attach to a component that is still a star at this step. These edges definitely cannot end up in the MST.

Alg. 5 for checking which vertices belong in stars is simple. If the parent and the grandparent of a vertex differ, then this vertex, as well as its grandparent are not in a star. Also, if a parent of some vertex is not in a star, then the same holds for this vertex.

7 Privacy-preserving MST

Let the ABB store the information about the structure of a graph and the weights of its edges. This means that there are public numbers n and m, denoting the number of vertices and edges of the graph. The vertices are identified with numbers $1, 2, \ldots, n$. The structure of the graph is private — the ABB stores m pairs of values, each one between 1 and n, giving the endpoints of the edges. For each edge, the ABB also stores its weight. The preamble of Alg. 7 specifies the actual data structures (arrays) and the meaning of their elements.

Thanks to working in the ABB model, it is unnecessary to specify which parties originally hold which parts of the description of the graph. No matter how they are held, they are first input to the ABB, after which the privacy-preserving MST algorithm is executed. Even more generally, some data about the graph might initially be held by no party at all, but be computed by some previously executed protocol. The benefit of working in the ABB model are the strong composability results it provides.

The algorithms in Sec. 4 can be used to implement Alg. 4 in privacy-preserving manner, if the dependencies of the control flow from the private data (there is a significant amount of such dependencies, mostly through the array $\mathcal{A}$) could be eliminated without penalizing the performance too much. Also, when implementing the algorithm, we would like to minimize the number of calls to `prepareRead` and `prepareWrite` algorithms, due to their overheads.

Let us first describe the checking for stars in privacy-preserving manner, as Alg. 5 has a

relatively simple structure. Its privacy-preserving version is depicted in Alg. 6. It receives the same mapping F as an input, now represented as a private vector $[\![\vec{F}]\!]$. As the first step, the protocol finds $F \circ F$ in privacy-preserving manner, and stores in in $[\![\vec{G}]\!]$. To find $[\![\vec{G}]\!]$, we have to read from the array $[\![\vec{F}]\!]$ according to the indices also stored in $[\![\vec{F}]\!]$. This takes place in lines 1–2 of Alg. 6. We can now privately compare whether the parent and grandparent of a vertex are equal (in parallel for all i, as denoted by the use of **foreach**). The result is stored in $[\![\vec{b}]\!]$ which serves as an intermediate value for the final result $[\![\overrightarrow{St}]\!]$. After line 3 of Alg. 6, the value of $[\![\vec{b}]\!]$ is the same as the value of $\overrightarrow{St}$ after the assignments in lines 1 and 4 of Alg. 5.

Algorithm 6: Privacy-preserving checking for stars

Data: Private vector $[\![\vec{F}]\!]$ of length n, where $1 \leq F_i \leq n$

Result: Private predicate $[\![\overrightarrow{St}]\!]$, indicating which elements of $\{1, \ldots, n\}$ belong to stars according to $\vec{F}$

1 $[\![\sigma]\!] \leftarrow$ `prepareRead`$([\![\vec{F}]\!], n)$

2 $[\![\vec{G}]\!] \leftarrow$ `performRead`$([\![\vec{F}]\!], [\![\sigma]\!])$

3 **foreach** $i \in \{1, \ldots, n\}$ **do** $[\![b_i]\!] \leftarrow [\![F_i]\!] \stackrel{?}{=} [\![G_i]\!]$;

4 $[\![b_{n+1}]\!] \leftarrow$ false

5 **foreach** $i \in \{1, \ldots, n\}$ **do** $[\![a_i]\!] \leftarrow [\![b_i]\!] \; ? \; (n+1) : [\![G_i]\!]$;

6 $[\![\vec{b'}]\!] \leftarrow$ `obliviousWrite`$([\![\vec{a}]\!], \overrightarrow{\text{false}}, \vec{1}, [\![\vec{b}]\!])$

7 $[\![\vec{p}]\!] \leftarrow$ `performRead`$([\![\vec{b'}]\!], [\![\sigma]\!])$ `// Ignore` b'_{n+1}

8 **foreach** $i \in \{1, \ldots, n\}$ **do** $[\![St_i]\!] \leftarrow [\![b'_i]\!] \wedge [\![p_i]\!]$;

9 **return** $[\![\overrightarrow{St}]\!]$

As next, we prepare to privately perform the assignment in line 4 of Alg. 5. We only want to perform the assignment if $[\![F_i]\!] \neq [\![G_i]\!]$, hence the number of assignments we want to perform depends on private data. Algorithm 2 presumes that the number of writes is public. We overcome this dependency by assigning to a dummy position each time Alg. 5 would have avoided the assignment in its line 5. We let the vector $[\![\vec{b}]\!]$ to have an extra element at the end and assign to this element for each dummy assignment. In line 5 we compute the indices of vector $[\![\vec{b}]\!]$ where false has to be assigned. Here the operation ? : has the same semantics as in C/C++/Java — it returns its second argument if its first argument is true (1), and its third argument if the first argument is false (0). It can be easily implemented in the ABB: $[\![b]\!] ? [\![x]\!] : [\![y]\!]$ is computed as $[\![b]\!] \cdot ([\![x]\!] - [\![y]\!]) + [\![y]\!]$.

In line 6 of Alg. 6, the oblivious write is performed. The arguments of `obliviousWrite` are in the same order as in the preamble of Alg. 2: the vector of addresses, the vector of values to be written, the vector of writing priorities, and the original array. All arguments can be private values. All public values are assumed to be automatically classified. In line 6, all values to be written are equal to false, as in Alg. 5. Hence the priorities do not really matter; we make them all equal to 1 (with the assumption that the priorities for existing elements of $[\![\vec{b}]\!]$, output by compute_priority in line 6 of Alg. 2, are equal to 0). The result of the writing is a private vector $[\![\vec{b'}]\!]$ of length $n+1$ that is equal to $[\![\vec{b}]\!]$ in positions that were not overwritten.

Lines 7 and 8 of Alg. 6 correspond to the assignment in line 6 of Alg. 5. First we compute $St[F[v]]$ for all v (in terms of Alg. 5) by reading from $[\![\vec{b}]\!]$ according to the indices in $[\![\vec{F}]\!]$. In line 1 we prepared the reading according to these indices. As $[\![\vec{F}]\!]$ has not changed in the meantime, this preparation is still valid and can be reused. Hence we apply `performRead` to first n elements of $[\![\vec{b'}]\!]$. The conjunction is computed in line 8.

The privacy-preserving MST protocol is given in Alg. 7. We explain it below.

Algorithm 7: Privacy-preserving minimum spanning tree

Data: Number of vertices n, number of edges m
Data: Private vector $[\![\vec{E}]\!]$ of length $2m$ (endpoints of edges, i-th edge is (E_i, E_{i+m}))
Data: Private vector $[\![\vec{\omega}]\!]$ of length m (edge weights)
Result: Private boolean vector $[\![\vec{\mathcal{T}}]\!]$ of length m, indicating which edge belongs to the MST

foreach $i \in \{1, \ldots, 2m\}$ **do**
  $[\![\omega'_i]\!] \leftarrow -[\![\omega_{i \bmod m}]\!]$
  $[\![E'_i]\!] \leftarrow [\![E_{(i+m) \bmod 2m}]\!]$
foreach $i \in \{1, \ldots, n+1\}$ **do**
  $[\![F_i]\!] \leftarrow i$
  $[\![\mathcal{W}_i]\!] \leftarrow (m+1)$
foreach $i \in \{1, \ldots, m+1\}$ **do** $[\![\mathcal{T}_i]\!] \leftarrow \mathsf{false}$;
$[\![\sigma^{\mathrm{e}}]\!] \leftarrow \texttt{prepareRead}([\![\vec{E}]\!], n)$
for *iteration_number* $:= 1$ **to** $\lfloor \log_{3/2} n \rfloor$ **do**
  $[\![\overrightarrow{St}]\!] \leftarrow \texttt{StarCheck}([\![\vec{F}]\!])$
  $[\![\vec{F}^{\mathrm{e}}]\!] \leftarrow \texttt{performRead}([\![\vec{F}]\!], [\![\sigma^{\mathrm{e}}]\!])$ `// Ignore` F_{n+1}
  $[\![\overrightarrow{St}^{\mathrm{e}}]\!] \leftarrow \texttt{performRead}([\![\overrightarrow{St}]\!], [\![\sigma^{\mathrm{e}}]\!])$
  foreach $i \in \{1, \ldots, m\}$ **do** $[\![d_i]\!] \leftarrow [\![F^{\mathrm{e}}_i]\!] \stackrel{?}{=} [\![F^{\mathrm{e}}_{i+m}]\!]$;
  foreach $i \in \{1, \ldots, 2m\}$ **do**
    $[\![a_i]\!] \leftarrow [\![St^{\mathrm{e}}_i]\!] \wedge \neg [\![d_{i \bmod m}]\!] \; ? \; [\![F^{\mathrm{e}}_i]\!] : (n+1)$
  $([\![\sigma^{\mathrm{v}}]\!], [\![\tau^{\mathrm{v}}]\!]) \leftarrow \texttt{prepareWrite}([\![\vec{a}]\!], [\![\vec{\omega}']\!], n+1)$
  $[\![\vec{F}]\!] := \texttt{performWrite}([\![\sigma^{\mathrm{v}}]\!], [\![\tau^{\mathrm{v}}]\!], [\![\vec{E}']\!], [\![\vec{F}]\!])$
  $[\![\vec{\mathcal{W}}]\!] := \texttt{performWrite}([\![\sigma^{\mathrm{v}}]\!], [\![\tau^{\mathrm{v}}]\!], (i \bmod m)_{i=1}^{2m}, [\![\vec{\mathcal{W}}]\!])$
  $[\![\vec{\mathcal{T}}]\!] := \texttt{obliviousWrite}([\![\vec{\mathcal{W}}]\!], \overrightarrow{\mathsf{true}}, \vec{1}, [\![\vec{\mathcal{T}}]\!])$
  $[\![\sigma^{\mathrm{f}}]\!] \leftarrow \texttt{prepareRead}([\![\vec{F}]\!], n+1)$
  $[\![\vec{G}]\!] \leftarrow \texttt{performRead}([\![\vec{F}]\!], [\![\sigma^{\mathrm{f}}]\!])$
  $[\![\vec{H}]\!] \leftarrow \texttt{performRead}([\![\vec{G}]\!], [\![\sigma^{\mathrm{f}}]\!])$
  foreach $i \in \{1, \ldots, n\}$ **do**
    $[\![c_i^{(1)}]\!] \leftarrow i \stackrel{?}{=} [\![G_i]\!]$
    $[\![c_i^{(2)}]\!] \leftarrow i \stackrel{?}{<} [\![F_i]\!]$
    $[\![c_i^{(3)}]\!] \leftarrow [\![F_i]\!] \stackrel{?}{=} [\![H_i]\!] \wedge [\![F_i]\!] \stackrel{?}{<} [\![G_i]\!]$
    $$[\![F_i]\!] := \begin{cases} i, & \text{if } [\![c_i^{(1)}]\!] \wedge [\![c_i^{(2)}]\!] \\ [\![F_i]\!], & \text{if } [\![c_i^{(1)}]\!] \wedge \neg [\![c_i^{(2)}]\!] \\ [\![F_i]\!], & \text{if } \neg [\![c_i^{(1)}]\!] \wedge [\![c_i^{(3)}]\!] \\ [\![G_i]\!] & \text{if } \neg [\![c_i^{(1)}]\!] \wedge \neg [\![c_i^{(3)}]\!] \end{cases}$$
return $([\![\mathcal{T}_1]\!], \ldots, [\![\mathcal{T}_m]\!])$

To adapt Alg. 4 for execution on ABB, we first we have to simplify its control flow. Fortunately, it turns out that it is not necessary to keep track which edges are still "active". The outcome of Alg. 4 does not change if all edges are assumed to be active all the time. In this case, only the stopping criterion of Alg. 4 (that there are no more active edges) has to be changed

to something more suitable. One could keep track of the number of edges already added to the MST, or to execute the main loop of the algorithm sufficiently many times. We opt for the second solution, as otherwise we may leak something about the graph through the running time of the algorithm.

Alg. 7 first copies around some input data, effectively making the set of edges E symmetric. Throughout this algorithm we assume that x mod m returns a value between 1 and m. In line 2 we negate the weights, as lower weight of some edge means that it has higher priority of being included in the MST. In lines 4 to 7 we initialize $[\![\vec{F}]\!]$, $[\![\vec{\mathcal{W}}]\!]$ and $[\![\vec{\mathcal{T}}]\!]$ similarly to Alg. 4. All these vectors have an extra element in order to accommodate dummy assignments. In $[\![\vec{\mathcal{W}}]\!]$, the value $(m+1)$ corresponds to the value NIL in Alg. 4 — the elements of $[\![\vec{\mathcal{W}}]\!]$ are used below as addresses to write into $[\![\vec{\mathcal{T}}]\!]$ and $(m+1)$ indicates a dummy assignment.

Before starting the iterative part of the algorithm, in line 8 we prepare for reading according to the endpoints of edges. The actual reads are performed in each iteration.

As mentioned before, the number of iterations of Alg. 7 (line 9) will be sufficient for all edges of the MST to be found. As discussed before, $\lfloor \log_{3/2} n \rfloor$ is a suitable number. All iterations are identical; the computations do not depend on the sequence number of the current iteration.

An iterations starts very similarly to Alg. 4, running the star checking algorithm and finding for each endpoint u of each edge e the values $F[u]$ and $St[u]$ (in terms of Alg. 4). In line 9 of Alg. 4, a decision is made whether to update an element of $\vec{F}$ and an element of $\vec{\mathcal{W}}$. The same decision is made in lines 13–15 of Alg. 7: we choose the address of the element to update. If the update should be made then this address is $[\![F_i^{\mathrm{e}}]\!]$. Otherwise, it is the dummy address $n+1$. In lines 16–18 the actual update is made. As the writes to both $[\![\vec{F}]\!]$ and $[\![\vec{\mathcal{W}}]\!]$ are according to the same indices $[\![\vec{a}]\!]$ and priorities $[\![\vec{\omega}']\!]$, their preparation phase has to be executed only once. If the write has to be performed, we write the other endpoint of the edge to $\vec{F}$ and the index of the edge to $\vec{\mathcal{W}}$. In line 19 we update $[\![\vec{\mathcal{T}}]\!]$ similarly to line 13 of Alg. 4.

Compared to Alg. 4, we have redesigned the breaking of F-cycles and decreasing the height of F-trees, in order to reduce the number of calls to algorithms in Sec. 4. In Alg. 4, the cycles are broken (which requires data to be read according to the indices in $\vec{F}$, and thus the invocation of `prepareRead`, as $\vec{F}$ has just been updated), and then the F-grandparent of each vertex is taken to be its F-parent (which again requires a read according to $\vec{F}$ and another invocation of `prepareRead`). Instead, we will directly compute what will be the F-grandparent of each vertex after breaking the F-cycles, and take this to be its new F-parent.

For this computation, we need the F-parents of each vertex, which we already have in the vector $\vec{F}$. We also need their F-grandparents which we store in $\vec{G}$, and F-great-grandparents, which we store in $\vec{H}$. We only need a single call to `prepareRead` to find both $[\![\vec{G}]\!]$ and $[\![\vec{H}]\!]$. After breaking the cycles, the F-grandparent of the vertex i can be either i, F_i or G_i. It is not hard to convince oneself that the computation in lines 24–27 finds the F-grandparent of i and assigns it to $[\![F_i]\!]$. As before, the computations for different vertices are made in parallel. The *case*-construction in line 27 is implemented as a composition of ? : operations.

Finally, we return the private boolean vector $[\![\vec{\mathcal{T}}]\!]$ indicating which edges belong to the MST, except for its final dummy element $[\![\mathcal{T}_{m+1}]\!]$.

Alg. 7 is UC-secure for the same reasons as Alg. 1 and Alg. 2 — it only applies the operations of ABB, classifies only public constants and declassifies nothing. The amount of work it performs (or: the amount of communication it requires for typical ABB implementations) is $O(|E| \log^2 |V|)$. Indeed, it performs $O(\log |V|)$ iterations, the complexity of which is dominated by reading and writing preparations requiring $O(\log |E|) = O(\log |V|)$ work. For typical ABB implementations, the round complexity of Alg. 7 is $O(\log^2 |V|)$ — each private reading or writing preparation requires $O(\log |V|)$ rounds.

Awerbuch-Shiloach algorithm (Alg. 4) accesses all of its memory during each of its iterations.

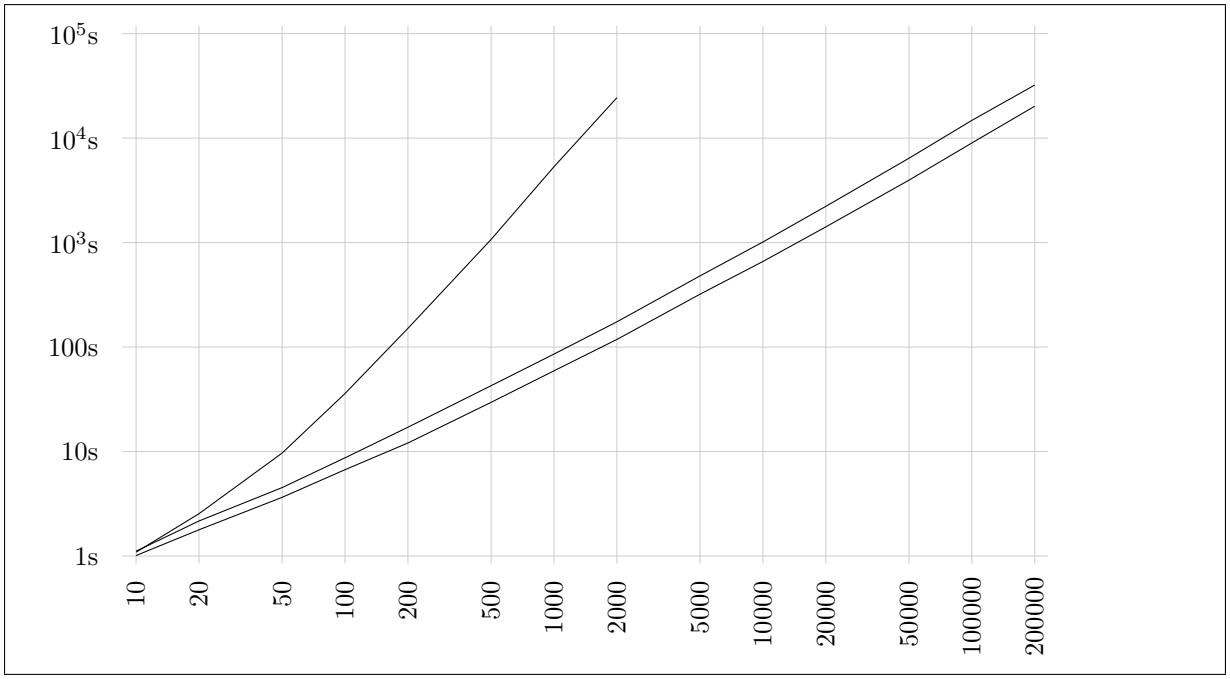

Figure 4: Running times for the private MST algorithm, depending on the number of vertices n. Number of edges is $m = 3n$ (lower line), $m = 6n$ (middle line), $m = n(n-1)/2$ (upper line)

Hence we can say that for this algorithm the overhead of our private data access techniques is only $O(\log m)$.

Experimental results

We have also implemented Alg. 7 (and Alg. 6) on the Sharemind SMC platform and tested its performance on the same setup as described in Sec. 5. We have varied the number of vertices n and selected the number of edges m based on it and on the most likely applications of our private MST protocol.

We believe the most relevant cases for our protocol to be planar graphs and complete graphs. Planar graphs have $m \approx 3n$, if most of its faces are triangles. Complete graphs have $m = n(n-1)/2$. We have also considered the case $m = 6n$ as a "generic" example of sparse graphs.

The results of our performance tests are depicted in Fig. 4. These will serve as the baseline for any further investigations in this direction.

8 Conclusions

We have presented efficient privacy-preserving protocols for performing in parallel many reads or writes from private vectors according to private indices. We have used these protocols to provide a privacy-preserving protocol for finding the minimum spanning tree in a graph; no protocols for this task have been investigated before. To achieve these results, this paper makes use of several novel ideas.

First, we noted that multiparty computations by necessity have to process their data in a parallel fashion, otherwise the costs of network latency are prohibitive. Hence one does not need a protocol for reading or writing one value from a private vector according to private

index (with the intent to run many copies of this protocol in parallel). It is sufficient to look for protocols that are efficient only when performing many reads or writes in parallel.

Second, we have noticed that the operations available relatively cheaply in existing ABB implementations allow us to construct such protocols. Our protocols in Sec. 4 are somewhat inspired by the techniques first appearing in [LW14], but are significantly more efficient.

Third, we have noticed that many PRAM algorithms become amenable to privacy-preserving implementations without too much overhead, if the protocols in Sec. 4 are available. In this sense, the private MST protocol of Sec. 7 serves just as an example. We have chosen this example because it is very difficult to imagine it to be implemented without the ideas in this paper.

Performance-wise, we have certainly obtained impressive results: a graph with 200,000 vertices and 1,200,000 edges can be processed in nine hours. Also, the set-up of our performance tests is realistic — LAN speeds between servers under control of different parties can easily be achieved through a co-located hosting service that provides physical barriers to the access of individual servers.

References

[ACM+13] Abdelrahaman Aly, Edouard Cuvelier, Sophie Mawet, Olivier Pereira, and Mathieu Van Vyve. Securely solving simple combinatorial graph problems. In Ahmad-Reza Sadeghi, editor, *Financial Cryptography*, volume 7859 of *Lecture Notes in Computer Science*, pages 239–257. Springer, 2013.

[AS87] Baruch Awerbuch and Yossi Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Computers*, 36(10):1258–1263, 1987.

[BCP14] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram. Cryptology ePrint Archive, Report 2014/594, 2014. `http://eprint.iacr.org/`.

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266, New York, NY, USA, 2008. ACM.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[BNTW12] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.

[BS05] Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 2005.

[BSMD10] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, pages 223–239, Washington, DC, USA, 2010.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.

[CDN01] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2001.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 23.2 The algorithms of Kruskal and Prim, pages 567–574. MIT Press and McGraw-Hill, 2nd edition, 2001.

[CS10] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *Financial Cryptography and Data Security*, volume 6052 of *LNCS*, pages 35–50. Springer, 2010.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious ram without random oracles. In Yuval Ishai, editor, *TCC*, volume 6597 of *Lecture Notes in Computer Science*, pages 144–163. Springer, 2011.

[DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

[GGH+13] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing oram and using it efficiently for secure computation. In Emiliano De Cristofaro and Matthew Wright, editors, *Privacy Enhancing Technologies*, volume 7981 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[GHJR14] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private database access with he-over-oram architecture. Cryptology ePrint Archive, Report 2014/345, 2014. `http://eprint.iacr.org/`.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 513–524. ACM, 2012.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, pages 218–229. ACM, 1987.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified vss and fact-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111, 1998.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*. The Internet Society, 2012.

[HKI+12] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon, editors, *ICISC*, volume 7839 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2012.

[HKS+10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In *CCS '10: Proceedings of the 17th ACM conference on Computer and communications security*, pages 451–462, New York, NY, USA, 2010. ACM.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 143–156. SIAM, 2012.

[KS08] Vladimir Kolesnikov and Thomas Schneider. A practical universal circuit construction and secure evaluation of private functions. In Gene Tsudik, editor, *Financial Cryptography*, volume 5143 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2008.

[KS14] Marcel Keller and Peter Scholl. Efficient, Oblivious Data Structures for MPC. Cryptology ePrint Archive, Report 2014/137, 2014. `http://eprint.iacr.org/`.

[LHS+14] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating Efficient RAM-Model Secure Computation. In *Proceedings of 2014 IEEE Symposium on Security and Privacy*. IEEE, 2014.

[LT13] Helger Lipmaa and Tomas Toft. Secure equality and greater-than tests with sublinear online complexity. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *ICALP (2)*, volume 7966 of *Lecture Notes in Computer Science*, pages 645–656. Springer, 2013.

[LW14] Peeter Laud and Jan Willemson. Composable Oblivious Extended Permutations. Cryptology ePrint Archive, Report 2014/400, 2014. `http://eprint.iacr.org/`.

[LWZ11] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-Efficient Oblivious Database Manipulation. In *Proceedings of the 14th International Conference on Information Security. ISC'11*, pages 262–277, 2011.

[MK10] Lior Malka and Jonathan Katz. Vmcrypt - modular software architecture for scalable secure computation. Cryptology ePrint Archive, Report 2010/584, 2010. `http://eprint.iacr.org/`.

[MS13] Payman Mohassel and Seyed Saeed Sadeghian. How to Hide Circuits in MPC: an Efficient Framework for Private Function Evaluation. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 557–574. Springer, 2013.

[NMN01] Jaroslav Nešetřil, Eva Milkovà, and Helena Nešetřilovà. Otakar Borůvka on minimum spanning tree problem; Translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1-3):3–36, 2001.

[NO07] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *Public Key Cryptography*, volume 4450 of *Lecture Notes in Computer Science*, pages 343–360. Springer, 2007.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with o((logn)3) worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 197–214. Springer, 2011.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[SvDS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM Conference on Computer and Communications Security*, pages 299–310. ACM, 2013.

[Wak68] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, 1968.

[WHC+14] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, abhi shelat, and Elaine Shi. Scoram: Oblivious ram for secure computation. Cryptology ePrint Archive, Report 2014/671, 2014. `http://eprint.iacr.org/`, to appear at ACM CCS '14.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.