Peeter Laud*

Parallel Oblivious Array Access for Secure Multiparty Computation and Privacy-Preserving Minimum Spanning Trees

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, our technique has 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 the first-ever realization of a privacy-preserving MST algorithm with sublinear round complexity.

Keywords: Secure Multiparty Computation; Oblivious Arrays; Minimum Spanning Tree

DOI 10.1515/popets-2015-0011
Received 2015-02-15; revised 2015-05-09; accepted 2015-05-15.

1 Introduction

In Secure Multiparty Computation (SMC), p parties compute $(y_1, \ldots, y_p) = f(x_1, \ldots, x_p)$, 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 [27, 57]. A universally composable [12] abstraction for SMC is the arithmetic black box (ABB) [21]. 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 [4, 7, 11, 18, 31, 43], 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 [28] 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 $O(\log^3 m)$ overhead for accessing an element of an m-element array [34].

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 exploit this parallelism in designing oblivious data access methods, by bundling 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 are reasonable. The protocols themselves are surprisingly simple; the simplicity also being attenuated by the use of the ABB model. Our protocols can be interleaved with the rest of the SMC application in order to provide oblivious data access capability to it.

***Corresponding Author: Peeter Laud:** Cybernetica AS, E-mail: peeter.laud@cyber.ee

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 sparse 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 [15], the best-known algorithms for finding MST (without privacy considerations), are not well suited for a direct implementation on top of ABB, because of their inherent sequentiality. In this paper, we consider a parallel algorithm by Awerbuch and Shiloach [3] (itself an adoption of a MST algorithm by Borůvka [45]) that runs on a PRAM (Parallel Random Access Machine) 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 private data access 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, making the communication complexity of our protocol to be $O(|E| \log^2 |V|)$. 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, both for the ABB model of SMC and solving MST on PRAM 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 present the solution for the private MST problem, describing the adoption of the algorithm described in Sec. 3 to privacy-preserving execution in Sec. 6, again quantifying its performance. We conclude in Sec. 7.

2 Related work

Secure multiparty computation (SMC) protocol sets can be based on a variety of different techniques, including garbled circuits [57], secret sharing [8, 24, 48] or homomorphic encryption [16]. A highly suitable abstraction of SMC is the universally composable Arithmetic Black Box (ABB) [21], 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 [2, 14, 17, 41], also containing this paper.

Protocols for oblivious RAM [28] have received significant attention during recent years [37, 49, 51]. The overhead of these protocols is around $O(\log^2 m)$ when accessing an element of a vector of length m, where the elements themselves are short. 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 [25, 26, 29, 42], or over an SMC protocol set [20, 34]. 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 [54].

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 [9] 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)* [32, 36, 38, 44] (see Sec. 4.1 for details). Our OEP application protocol is more efficient (both in practice and asymptotically) than any other published construction built with SMC techniques. The building of an OEP in composable manner has only been considered in [38]; our construction is more efficient than theirs. Our writing protocol is similar to the associative map circuit [58], but optimized for SMC protocol sets based on secret sharing.

There exist privacy-preserving protocols for a number of graph algorithms, e.g. single-source shortest paths and for maximum flow [2, 6, 10, 34]. MST algorithms also belong to this list. Brickell and Shmatikov [10] present a two-party protocol that makes the resulting MST public. Blanton et al. [6] adapt Prim's algorithm for privacy-preserving execution in a way that is asymptotically optimal (in terms of communication complexity) for dense graphs. For both of them, the number of

communication rounds is proportional to the number of vertices in the graph.

3 Preliminaries

3.1 Secure Multiparty Computation

3.1.1 Universal Composability

Universal composability (UC) [12] 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 p users. The latter is *at least as secure as* the former (or: Π *securely implements* $\mathcal{F}$), 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 $\mathcal{Z}$ of either Π or $\mathcal{F}$. Here $\mathcal{X} \| \mathcal{Y}$ denotes the systems $\mathcal{X}$ and $\mathcal{Y}$ (consisting of one or more interactive Turing machines) running in parallel, possibly communicating with each other over the common interface.

The value of the framework lies in the composition theorem. The protocol Π may be implemented in the *$\mathcal{F}$-hybrid model*. In this case the Turing machines $M_1, \ldots, M_p$ that implement the steps of the protocol Π for the p users may additionally access an ideal functionality $\mathcal{F}$ for certain steps of the protocol. The protocol Π *securely implements* an ideal functionality $\mathcal{G}$ *in the $\mathcal{F}$-hybrid model* if for any $\mathcal{A}$ there exists $\mathcal{A}_S$, such that $(M_1 \| \cdots \| M_p \| \mathcal{F}) \| \mathcal{A}$ and $\mathcal{G} \| \mathcal{A}_S$ are indistinguishable.

Let Ξ, consisting of Turing machines $M'_1, \ldots, M'_p$ be a protocol that implements the ideal functionality $\mathcal{F}$. The machines $M'_1, \ldots, M'_p$ could be used to provide the functionality of $\mathcal{F}$ to the machines $M_1, \ldots, M_p$ of the protocol Π. Let M''_i be the "union" of M_i and M'_i, where M_i communicates with the user $\mathcal{Z}$, and all calls from M_i to $\mathcal{F}$ are replaced with calls to M'_i (hence M_i acts as the user for M'_i). Let Π^{Ξ} denote the protocol consisting of machines $M''_1, \ldots, M''_p$. The composition theorem states the following [12]:

Theorem 1. *If Π securely implements $\mathcal{G}$ in $\mathcal{F}$-hybrid model and Ξ securely implements $\mathcal{F}$ [in $\mathcal{H}$-hybrid model] then Π^{Ξ} securely implements $\mathcal{G}$ [in $\mathcal{H}$-hybrid model].*

3.1.2 Arithmetic Black Box

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 $\mathsf{declassify}(h)$ 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 $\mathsf{compute}(op; h_1, \ldots, h_k)$ 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 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, by issuing the command $\mathsf{declassify}(h)$ to $\mathcal{F}_{\mathsf{ABB}}$. Whenever $\mathcal{F}_{\mathsf{ABB}}$ executes a command, it also informs the adversary, forwarding the command to it (except for the value v in the store-command). This kind of informing is necessary, because the real adversary is able to perform traffic analysis, deducing the executed commands from the network traffic it can see between computing parties.

The existing implementations of ABB are protocol sets Π_{ABB} based on either secret sharing [7, 11, 18], threshold homomorphic encryption [21, 31] or garbled circuits [22, 42]. 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). An SMC application may be be built by having an ABB implementation as its component, and invoking the computation commands of the ABB according to the description of the computation realized by this application.

We have used the Sharemind SMC framework [7] for implementing the protocols proposed in this paper. Its implementation of the ABB consists of three parties, providing protection against one honest-but-curious party. The protocols of Sharemind are based on secret sharing over rings.

All ABB implementations provide protocols for computing linear combinations of private values (with public coefficients; formally, the coefficients are part of the operation name op) 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. There may be other operations provided by the ABB implementation. These operations typically have a cost at least as large as multiplying two private values, and usually much more [8].

The peformance profiles of ABB implementations based on different SMC constructions are very different. Implementations based on operations with values secret-shared over some ring require less communication among the parties than implementations based on garbled circuits, if the operations the SMC application performs map nicely into the arithmetic operations on that ring. On the other hand, protocols based on garbled circuits may run in constant rounds, while for secret-sharing based protocols the number of rounds is at least proportional to the circuit-depth of the computation realized by the SMC application. Hence the parallelizability of the application may be a criterion in choosing the ABB implementation.

3.1.3 Extending an ABB

Let $\mathcal{F}_{\mathsf{ABB}}$ support a certain set of operations. Let $\mathcal{F}'_{\mathsf{ABB}}$ support a larger set of operations. If we already have a secure implementation of $\mathcal{F}_{\mathsf{ABB}}$ and want to securely implement $\mathcal{F}'_{\mathsf{ABB}}$, it makes a lot of sense in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model, especially if the implementation Π'_{ABB} consisting of the machines $M'_1, \ldots, M'_k$ works as follows [52]:

1. Any store- or declassify-command received by the machines $M'_1, \ldots, M'_k$ from the user $\mathcal{Z}$ is forwarded by them to $\mathcal{F}_{\mathsf{ABB}}$, and the resulting handle or value returned to the user. The machines M'_i contain a translation table for handles known by $\mathcal{Z}$ and handles known by $\mathcal{F}_{\mathsf{ABB}}$ — each command from $\mathcal{Z}$ may be translated to several commands to $\mathcal{F}_{\mathsf{ABB}}$, as we explain below. The translation table is used by M'_i while forwarding the handles in both directions. The translation table is public, as it can be deduced from the information $\mathcal{F}_{\mathsf{ABB}}$ has sent to the adversary.
2. Any compute-command with an operation *op* known to $\mathcal{F}_{\mathsf{ABB}}$ is similarly forwarded to $\mathcal{F}_{\mathsf{ABB}}$ and the resulting handle returned to $\mathcal{Z}$.
3. Any compute-command with a novel operation *op* is converted to several commands to $\mathcal{F}_{\mathsf{ABB}}$ according to a public program P_{op} (the same programmed is used by all machines M'_i). There is a proof that the execution of P_{op} makes the result of *op* to be stored inside $\mathcal{F}_{\mathsf{ABB}}$. The handle to this result is returned to $\mathcal{Z}$.

We see that the machines M'_i do not directly communicate with each other. Neither do they directly communicate with the adversary $\mathcal{A}$, except when they have been corrupted. To prove that Π' securely implements $\mathcal{F}'_{\mathsf{ABB}}$, we have to show how the attacker $\mathcal{A}_S$ working against $\mathcal{F}'_{\mathsf{ABB}}$ is constructed from any attacker $\mathcal{A}$ working against Π'. We construct $\mathcal{A}_S = \mathit{Sim} \| \mathcal{A}$, where the task of the *simulator Sim* is to translate between the adversarial interface of $\mathcal{F}'_{\mathsf{ABB}}$ and the interface $\mathcal{A}$ has for communicating with $\mathcal{F}_{\mathsf{ABB}}$.

The simulator *Sim* receives from $\mathcal{F}'_{\mathsf{ABB}}$ the commands it is executing, and translates them to commands $\mathcal{F}_{\mathsf{ABB}}$ would have executed. For the first two kinds of commands listed above, the translation is equal to the command (except for the translation of handles; *Sim* maintains the same translation table as M'_i). For the third kind of command, *Sim* internally executes P_{op} and sends to $\mathcal{A}$ the commands it contains. This translation is trivial if there are no declassifications in P_{op}. If there are, we have to explain how *Sim* generates the declassified values in a way that is indistinguishable from Π. Also, if some computing parties store new values in $\mathcal{F}_{\mathsf{ABB}}$, such that these values are not *a priori* known to other computing parties, then we must argue that P_{op} allows the validity of these values to be verified. In this paper, there are no stores of such values. Having constructed such simulator, we have shown that Π' is a secure implementation of $\mathcal{F}'_{\mathsf{ABB}}$, tolerating any corruptions. Hence the corruptions tolerated by an actual implementation of Π' in the "plain" model are the same as those tolerated by the used implementation of $\mathcal{F}_{\mathsf{ABB}}$.

The extension of a basic ABB has been demonstrated in e.g. [1, 5, 13, 17, 46], where operations like bit-decomposition, equality and less-than comparisons, fixed- and floating-point, set and multiset operations have been added to the ABB.

We present our oblivious data access, as well as private MST operations as extensions to the ABB described in Sec. 3.1.4. In the programs P_{op} 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 $\mathcal{F}_{\mathsf{ABB}}$ is asked to perform the operation $\otimes$ 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 $\otimes$.

3.1.4 Our Initial ABB

In this paper, we require the values stored in the ABB to come from a sufficiently large ring, such that the array indices could be represented by them. Regarding the provided functionality, we require the ABB implementation to provide protocols for equality and comparison operations [8, 17]. We also require the ABB to have *oblivious shuffles* — private permutations of values — together with operations to *apply* and *unapply* them to vectors of values. Given an oblivious shuffle $[\![\sigma]\!]$ for m elements, and a private vector $([\![v_1]\!], \ldots, [\![v_m]\!])$, the apply operation returns handles to elements of a new private vector $([\![v'_1]\!], \ldots, [\![v'_m]\!])$ with $v'_i = v_{\sigma(i)}$. The unapply operation similarly returns handles to $([\![v'_1]\!], \ldots, [\![v'_m]\!])$, but this time the equalities $v'_{\sigma(i)} = v_i$ hold. Additionally, there are operations that return handles to oblivious shuffles. The operation random_shuffle(m) returns $[\![\sigma]\!]$, where σ is a uniformly randomly chosen element of the symmetric group S_m. The operation $[\![\sigma]\!] \circ \xi$, where $\sigma, \xi \in S_m$, returns $[\![\tau]\!]$, where $\tau = \sigma \circ \xi$.

It is up to the implementation of an ABB, how oblivious shuffles are represented. We note that Waksman networks [53] can be used to add oblivious shuffles to any ABB, in the sense of Sec. 3.1.3. For certain ABB implementations, including Sharemind, a more efficient implementation is described in [40]. The complexity of the protocols implementing the ABB operations apply, unapply and random_shuffle is either $O(m)$ or $O(m \log m)$ (for constant number of parties). The complexity of the composition of a private and a public shuffle is constant.

With oblivious shuffles and comparison operations, the ABB can be extended to sort vectors of private values (of length m) in $O(m \log m)$ time, where the size of the constants hidden in the O-notation is reasonable [30]. In our protocols, we let $[\![\sigma]\!] \leftarrow \mathsf{sortperm}([\![\vec{v_1}]\!], \ldots, [\![\vec{v_k}]\!])$, where $\vec{v_1}, \ldots, \vec{v_k}$ are vectors of length m, denote the operation that produces an oblivious shuffle $[\![\sigma]\!]$, such that the application of σ to each of $\vec{v_1}, \ldots, \vec{v_k}$ would bring them to an order that is componentwise lexicographically sorted (with the $\vec{v_1}$-component being the most significant). We require the sorting obtained through sortperm and shuffle application to be stable.

3.2 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 $O(|E| + |V| \log |V|)$ [15]. They are inherently sequential and therefore unsuitable as a basis for our privacy-preserving implementation on top of an ABB implementation based on secret sharing.

Other algorithms for MST have been proposed. Borůvka's algorithm [45] 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 [3] 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 $\log_{3/2} |V|$.

Awerbuch-Shiloach algorithm executes on *priority-CRCW PRAM*. Parallel random access machines (PRAM) are a theoretical model for parallel computations. In this model, an arbitrary number of processors are available, executing synchronously and sharing common memory. There exist several subclasses of PRAM, depending on how the read/write conflicts to the same memory location are resolved. A CRCW PRAM allows many processors to read and write the same location at the same time. In priority-CRCW PRAM, concurrent writes to the same location are resolved by (numeric) priorities assigned to the write: only the writing operation with the highest priority gets through [33].

One iteration of the Awerbuch-Shiloach algorithm requires constant time, when executed by $|E|$ processors. The algorithm assumes that all edges have different

weights (this does not lessen the generality). We give a thorough description of their algorithm, as we are going to adapt it for privacy-preserving execution in Sec. 6.

The Awerbuch-Shiloach algorithm [3] is presented in Alg. 1. 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. 1 uses a union-find data structure — the array F that for each vertex records its current "parent". When interpreting the array F as a set of directed edges $(v, F[v])$ for each $v \in V$, then the directed graph (V, F) will be a forest of rooted trees throughout the execution of Alg. 1. Initially (as set in line 5), this graph consists of trees of size 1, i.e. each vertex is a root of a tree consisting only of this vertex. In general, a vertex v is currently a root of a tree if $F[v] = v$. The trees in the forest (V, F) define the parts in the current partitioning. A rooted tree is called a *star* if its height is at most 1, i.e. it is either an isolated vertex or a tree consisting of a root and a number of leaves. The correctness of the Awerbuch-Shiloach algorithm depends on the following property: if v is a vertex in a star, then $F[v]$ is the root of that star.

Algorithm 1: MST algorithm by Awerbuch and Shiloach

Data: Connected graph $G = (V, E)$, edge weights ω
Result: $E' \subseteq E$, such that (V, E') is the MST of G

1 **foreach** $(u, v) \in E$ **do**
2 $\mathcal{T}[\{u, v\}] \leftarrow$ false
3 $\mathcal{A}[(u, v)] \leftarrow$ true
4 **foreach** $v \in V$ **do**
5 $F[v] \leftarrow v$
6 $\mathcal{W}[v] \leftarrow$ NIL
7 **while** $\exists (u, v) \in E : \mathcal{A}[(u, v)]$ **do**
8 **foreach** $(u, v) \in E$ *where* $\mathcal{A}[(u, v)]$ **do**
9 **if** in_star$(u) \wedge F[u] \neq F[v]$ **then**
10 $F[F[u]] \leftarrow F[v]$ with priority $\omega(u, v)$
11 $\mathcal{W}[F[u]] \leftarrow \{u, v\}$ with priority $\omega(u, v)$
12 **Synchronize**
13 **if** $\mathcal{W}[F[u]] = \{u, v\}$ **then** $\mathcal{T}[\{u, v\}] \leftarrow$ true;
14 **if** $u < F[u] \wedge u = F[F[u]]$ **then** $F[u] \leftarrow u$;
15 **Synchronize**
16 **if** in_star(u) **then**
17 $\mathcal{A}[(u, v)] \leftarrow$ false
18 **else**
19 $F[F[u]] \leftarrow F[u]$
20 **return** $\{(u, v) \in E \mid \mathcal{T}[\{u, v\}]\}$

Algorithm 2: Checking for stars in Alg. 1

Data: A set V, a mapping $F : V \to V$
Result: Predicate St on V, indicating which elements of V belong to stars

1 **foreach** $v \in V$ **do** $St[v] \leftarrow$ true;
2 **foreach** $v \in V$ **do**
3 **if** $F[v] \neq F[F[v]]$ **then**
4 $St[v] \leftarrow$ false
5 $St[F[F[v]]] \leftarrow$ false
6 **foreach** $v \in V$ **do** $St[v] \leftarrow St[v] \wedge St[F[v]]$;
7 **return** St

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. 1, before executing the lines 9 and 16, the algorithm Alg. 2 is invoked and its output is used to check whether the vertex u belongs to a star.

An iteration of Alg. 1 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. 2 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.

4 Oblivious data access

We present the protocols for obliviously reading and writing elements of an array as extensions to the ABB specified in Sec. 3.1.4.

4.1 Protocol for reading

In Alg. 3, 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 $i \in \{1, \ldots, m\}$. Computing $\mathsf{prefixsum}(\llbracket\vec{v}\rrbracket)$ 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 $i \in \{2, \ldots, m\}$.

Algorithm 3: Reading n values from the private array

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

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

We see that in Alg. 3, 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

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.

Fig. 1. Example of private reading according to Alg. 3

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.

By the arguments in Sec. 3.1.3, the protocol presented in Alg. 3 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. 3 is $O(\log(m+n))$.

Instead of reading elements from an array, the elements of which are indexed with $1, \ldots, m$, the presented protocol could also be used to read the private values from a dictionary. In this case, the elements of the vector $\vec{v}$ would not be indexed with $1, \ldots, m$, but with (private) $\llbracket j_1\rrbracket, \ldots, \llbracket j_m\rrbracket$. For reading from a dictionary, $\llbracket t_i\rrbracket$ is not initialized with i, but with $\llbracket j_i\rrbracket$ in line 1. The algorithm has to be slightly modified to detect if all indices that we attempt to read are present in the dictionary.

Note that in Alg. 3, the argument $\llbracket\vec{v}\rrbracket$ is only used after the dotted line. At the same time, the step that dominates the complexity of the protocol — sorting of $\llbracket\vec{t}\rrbracket$ 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. 3 only once and the lower part as many times as necessary. In Sec. 6, we will denote the upper part of Alg. 3 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* [44] 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 $[\![\sigma]\!]$ 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. As the priorities of existing array elements are not used in the examples of this paper, we will not explore this further. To take such priorities into account, line 6 of Alg. 4 would have to be changed.

The parallel writing protocol is given in Alg. 4. 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. 4 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 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 11 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. A small example of the writing protocol in presented in Fig. 2.

Algorithm 4: 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 priority in $\vec{p}$ is the highest for this position

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 $\quad [\![p'_{n+i}]\!] \leftarrow MAX\_VALUE$
7 $[\![\sigma]\!] \leftarrow \mathsf{sortperm}([\![\vec{j}']\!], [\![\vec{p}']\!])$
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 $[\![\tau]\!] \leftarrow \mathsf{sortperm}([\![\vec{b}]\!])$
. . . . . .
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** $[\![\vec{w}]\!]$

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. 6, we thus

Let $\vec{w} = (1, 4, 9, 16, 25)$. Let the values to be written be as follows, with the following priorities (defining $\vec{j}$, $\vec{v}$ and $\vec{p}$):

j	3	2	4	3
v	5	6	7	8
p	1	2	1	2

In lines 1–6, Alg. 4 prepares the vectors $\vec{j'}$ and $\vec{p'}$. In lines 12–13, it prepares the vector $\vec{v'}$. They are the following, where we let ∞ to denote MAX_VALUE:

j'	3	2	4	3	1	2	3	4	5
p'	-1	-2	-1	-2	∞	∞	∞	∞	∞
v'	5	6	7	8	1	4	9	16	25

The sorting permutation σ is shown below. Also shown are the results of applying (denoted with \$) σ to $\vec{j'}$ (resulting in $\vec{j''}$), to $\vec{p'}$ (not computed at all), and to $\vec{v'}$ (computed as an intermediate value in line 14).

σ	1	2	3	4	5	6	7	8	9
	5	2	6	4	1	7	3	8	9
j''	1	2	2	3	3	3	4	4	5
$\sigma \,\$\, p'$	∞	-2	∞	-2	-1	∞	-1	∞	∞
$\sigma \,\$\, v'$	1	6	4	8	5	9	7	16	25
b	0	0	1	0	1	1	0	1	0

Vector $\vec{b}$, computed in lines 9–10 from $\vec{j''}$, is also shown above. In the rest of Alg. 4, we pick out those elements of $\sigma \,\$\, \vec{v'}$, where the corresponding element of $\vec{b}$ is 0. For this, we sort $\vec{b}$ and apply the resulting permutation τ also to $\sigma \,\$\, \vec{v'}$:

τ	1	2	3	4	5	6	7	8	9
	1	2	4	7	9	3	5	6	8
$\tau \,\$\, b$	0	0	0	0	0	1	1	1	1
$\tau \circ \sigma \,\$\, v'$	1	6	8	7	25	4	5	9	16

The updated vector $\vec{w}$ is equal to the first $|\vec{w}|$ elements of the last row. The update is performed in line 15 of Alg. 4. All values shown in this example are private, i.e. stored in the ABB.

Fig. 2. Example of private writing according to Alg. 4

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 $\vec{w}$. The output of prepareWrite is the pair of oblivious shuffles $(\llbracket\sigma\rrbracket, \llbracket\tau\rrbracket)$. These are input to performWrite together with $\llbracket\vec{v}\rrbracket$ and $\llbracket\vec{w}\rrbracket$.

4.3 Sorting bits

Alg. 4 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. 4 (being equal to the length of $\vec{w}$). The sorting protocol is given in Alg. 5. Here 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 [40].

Algorithm 5: Stable sorting of 0-bits in a bit-vector

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

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

We see that the most complex operations of Alg. 5 are the applications of the oblivious shuffle $\llbracket\tau\rrbracket$. 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 $\llbracket\vec{b}\rrbracket$. 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 $\{1,\ldots,n\}$. The numbers $\{1,\ldots,n\}$ 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 of Alg. 5 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 $1 \le f(m) \le m$). 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.

There exists a sizable body of efficient algorithms for PRAMs. Using our parallel reading and writing protocols, any algorithm for priority-CRCW PRAM 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 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.

5.1 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) [8] 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 $m=n$). In our experiments, all values and indices were elements of $\mathbb{Z}_{2^{32}}$.

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. 3. 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. 4. 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 seconds. This makes 0.5 ms per access. Of course, such efficiency is possible only if the overlying application supports parallelism to this level.

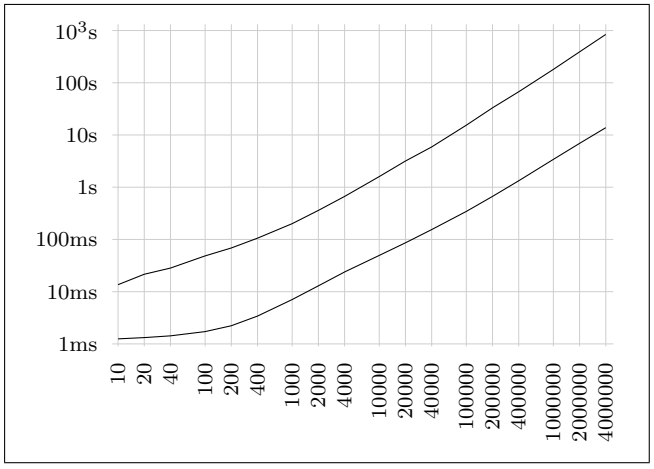

Fig. 3. Times for preparing (upper) and performing (lower) a parallel read, depending on the sum of vector length and the number of reads

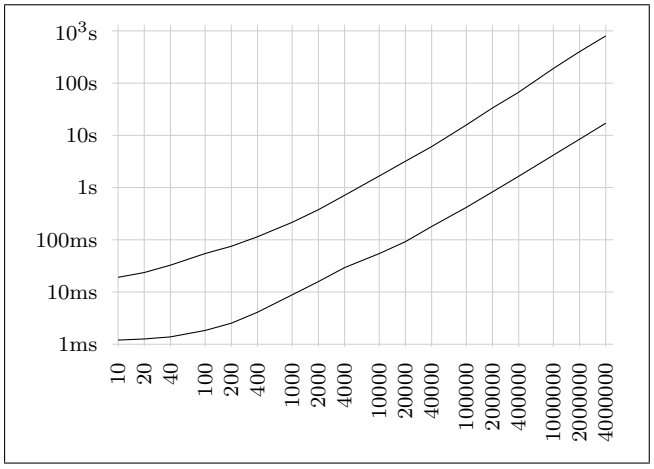

Fig. 4. Times for preparing (upper) and performing (lower) a parallel write, depending on the sum of vector length an the number of writes

5.2 Comparison with Previous Work

We are interested in comparing our techniques with the implementations of ORAM over SMC. The performance of actual implementations has been reported in [29, 34, 42]. But as the underlying SMC protocol sets have been rather different from our Sharemind framework, a fair comparison may be difficult.

Gordon et al. [29] have implemented ORAM on top of garbled circuits, offering security against one honest-but-curious party. Their set-up uses two servers similar to the ones in our cluster, connected to each other with a 1Gbit/s network link. Reading a 128-bit element of an array with length 0.5 million takes about 22 seconds. This is around 4.5 orders of magnitude slower than our result of reading an element of a vector of 2 million 32-bit elements in 0.5 milliseconds.

This comparison assumes that the application using the oblivious access functionality is sufficiently parallel; the timing of our protocols "0.5 ms per read" is valid for $n = 2 \cdot 10^6$ parallel reads from an array of length $m = 2 \cdot 10^6$. If $n \ll m = 2 \cdot 10^6$, then an n-wise parallel read requires around 400 s (by Fig. 3), or $400/n$ seconds per read. If, for example, $n = 10^5$, then our protocol requires 4 ms per read, or only 3.5 orders of magnitude less time than Gordon et al.'s implementation. The parity with their implementation would be obtained at $n \approx 20$.

Obviously, the bottleneck of Gordon et al.'s implementation is the network connection between two servers — for each non-XOR binary Boolean operation in the circuit, three ciphertexts have to be sent from one server to the other. If the authors had instead used SMC protocols based on secret sharing over $\mathbb{Z}_2$, the network communication had been significantly lower. On the other hand, the latency of the network connection might have started to significantly affect the performance. Still, the amount of communication would not have dropped by more than two orders of magnitude — instead of sending a couple of ciphertexts per gate, the communication might have dropped to some bits per gate, but not lower. To match this performance with our protocols, the application using oblivious access must be able to perform at least $n \approx 2000$ reads in parallel.

Liu et al. [42] have also implemented ORAM on top of garbled circuits, offering security against one honest-but-curious party. They do not report running times, but the number of block cipher operations that the garbling party executes. Also, they do not report performance numbers for a single ORAM operation, but only for larger applications that perform oblivious reads and writes. On the other hand, they report using the same methods for encoding ORAM as Gordon et al. [29], hence we believe that the actual access times should be similar as well. Liu et al. [42] improve the use of ORAM in larger SMC applications; their techniques could also be used in conjunction of our oblivious array access protocols, as long as the applications are sufficiently parallelizable.

Keller and Scholl's [34] implementation is probably closest to ours: their ORAM implementation runs on top of SPDZ protocol set [19, 35], based on additively secret-sharing the values among an arbitrary number of servers. In their implementation, they have used two servers similar to our own, connected with a 1Gbit/s network link. Through an expensive offline preprocess-

ing stage and a constant-communication post-execution check, the SPDZ protocol set achieves security against a malicious adversary corrupting all but one server. The secret-sharing has to take place over a field that is sufficiently large for the probability of wrongly passing the post-execution check to be negligible. Keller and Scholl have used $GF(2^{40})$ to measure the access times of their ORAM over SMC implementation.

The online phase of the protocol to multiply two shared field elements is highly efficient in SPDZ, each party having to send just two values to every other party. It is even more efficient than the multiplication protocol in the current implementation of Sharemind, where each party sends a total of five values to the other two computing parties. Hence it may seem that for some SMC application, using an ABB implementation based on the online phase of two-party SPDZ is in general more efficient than the implementation of Sharemind (with three parties).

Unfortunately, the comparison is not that simple. While in this setting, the multiplication protocol of the online phase of SPDZ may be slightly faster than the multiplication protocol of Sharemind, the protocol set of the latter is much larger and contains optimized protocols for many other operations useful in SMC applications, while in SPDZ the other operations have to be realized as compositions of many multiplications [17]. In particular, comparison operations are used in ORAM protocols. In Sharemind, an inequality comparison over $\mathbb{Z}_{2^{32}}$ is around an order of magnitude slower than the multiplication [8]. For values additively shared over $GF(2^k)$, a comparison can be computed with the help of an arithmetic circuit with k multiplications (and a number of free operations) with multiplicative depth $\lceil \log k \rceil$. Hence we guess that the communication costs of the comparison operations in our implementation of Sharemind and in Keller and Scholl's implementation of the online phase of SPDZ differ by no more than a couple of times.

Keller and Scholl [34] report the execution time for the read of one 40-bit element of a million-element array in their implementation of ORAM over SMC to be around 50 ms. This is around two orders of magnitude slower than in our implementation, when reading in parallel $n = 2 \cdot 10^6$ elements form an array of length $m = 2 \cdot 10^6$. Despite all the differences in underlying protocol sets, we believe this to be a basically fair comparison. Again, the performance of our protocol drops if the available parallelism is smaller. To match the performance of [34], we need $n \gtrapprox 10^4$.

Note that none of the implementations in [29, 34, 42] would benefit from parallelization. They all implement the Path ORAM technique [51] which does not support parallel application.

One may wonder how our oblivious reading (and writing) protocols compare with a basic $O(m)$-overhead protocol where the private index $[\![j]\!]$ is first expanded to a characteristic vector $[\![b_0]\!], \ldots, [\![b_{m-1}]\!]$ with $b_i \in \{0, 1\}$ and $b_i = 1$ iff $i = j$. The scalar product of $[\![\vec{b}]\!]$ with the array $[\![\vec{w}]\!]$, requiring m multiplications, would then be the value we seek. Our experiments with Sharemind have shown that on our cluster, we can perform around 4 million 32-bit multiplications per second. To read from an array with $m = 2{\cdot}10^6$ elements, the computation of the scalar product would require 0.5 seconds — around three magnitudes more than our oblivious array access. Additionally, the characteristic vector must be computed; this has similar complexity [39]. We believe that our oblivious read protocol is faster if $n \gtrapprox 1000$.

A fair comparison of our results with previously proposed protocols is further complicated by the prepare/perform phases of our protocols, if the application making use of oblivious data access protocols is such, that the shuffle(s) computed by a single invocation of a prepare-protocol (with complexity $O((m+n)\log(m+n))$) can be used by many perform-protocols (with complexity $O(m+n)$). This would allow the amortization of the expensive parts of our oblivious data access protocols and make them more competitive wrt. ORAM-based protocols.

6 Privacy-preserving MST

Let the ABB store the information about the structure of a graph and the weights of its edges. 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 ini-

tially 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. 1 in privacy-preserving manner (indeed, we will present the MST protocol as an extension to the ABB specified in Sec. 4), 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. 2 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. 2.

Algorithm 6: Privacy-preserving checking for stars

Data: Private vector $[\![\vec{F}]\!]$ of length n, where $1 \le F_i \le n$
Result: Private predicate $[\![\overrightarrow{St}]\!]$, indicating which elements of $\{1,\dots,n\}$ belong to stars according to $\vec{F}$

1 $[\![\sigma]\!] \leftarrow$ prepareRead$([\![\vec{F}]\!], n)$
2 $[\![\vec{G}]\!] \leftarrow$ performRead$([\![\vec{F}]\!], [\![\sigma]\!])$
3 **foreach** $i \in \{1,\dots,n\}$ **do** $[\![b_i]\!] \leftarrow [\![F_i]\!] \stackrel{?}{=} [\![G_i]\!]$;
4 $[\![b_{n+1}]\!] \leftarrow$ false
5 **foreach** $i \in \{1,\dots,n\}$ **do**
6 $\quad [\![a_i]\!] \leftarrow [\![b_i]\!] \,?\, (n+1) : [\![G_i]\!]$
7 $[\![\vec{b'}]\!] \leftarrow$ obliviousWrite$([\![\vec{a}]\!], \overrightarrow{\text{false}}, \vec{1}, [\![\vec{b}]\!])$
8 $[\![\vec{p}]\!] \leftarrow$ performRead$([\![\vec{b'}]\!], [\![\sigma]\!])$ // Ignore b'_{n+1}
9 **foreach** $i \in \{1,\dots,n\}$ **do** $[\![St_i]\!] \leftarrow [\![b'_i]\!] \wedge [\![p_i]\!]$;
10 **return** $[\![\overrightarrow{St}]\!]$

As next, we prepare to privately perform the assignment in line 4 of Alg. 2. 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 4 presumes that the number of writes is public. We overcome this dependency by assigning to a dummy position each time Alg. 2 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 6 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 7 of Alg. 6, the oblivious write is performed. The arguments of obliviousWrite are in the same order as in the preamble of Alg. 4: 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 7, all values to be written are equal to false, as in Alg. 2. 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. 4, 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 8 and 9 of Alg. 6 correspond to the assignment in line 6 of Alg. 2. First we compute $St[F[v]]$ for all v (in terms of Alg. 2) 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 9.

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

To adapt Alg. 1 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. 1 does not change if all edges are assumed to be active all the time. In this case, only the stopping criterion of Alg. 1 (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 ($\log_{3/2} n$). We opt for the second solution, as otherwise we may

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

1 **foreach** $i \in \{1, \ldots, 2m\}$ **do**
2 $[\![\omega'_i]\!] \leftarrow -[\![\omega_{i \bmod m}]\!]$
3 $[\![E'_i]\!] \leftarrow [\![E_{(i+m) \bmod 2m}]\!]$
4 **foreach** $i \in \{1, \ldots, n+1\}$ **do**
5 $[\![F_i]\!] \leftarrow i$
6 $[\![\mathcal{W}_i]\!] \leftarrow (m+1)$
7 **foreach** $i \in \{1, \ldots, m+1\}$ **do** $[\![\mathcal{T}_i]\!] \leftarrow$ false;
8 $[\![\sigma^{\mathrm{e}}]\!] \leftarrow \mathsf{prepareRead}([\![\vec{E}]\!], n)$
9 **for** *iteration_number* $:= 1$ **to** $\lfloor \log_{3/2} n \rfloor$ **do**
10 $[\![\overrightarrow{St}]\!] \leftarrow \mathsf{StarCheck}([\![\vec{F}]\!])$
11 $[\![\vec{F}^{\mathrm{e}}]\!] \leftarrow \mathsf{performRead}([\![\vec{F}]\!], [\![\sigma^{\mathrm{e}}]\!])$ // Ignore F_{n+1}
12 $[\![\overrightarrow{St}^{\mathrm{e}}]\!] \leftarrow \mathsf{performRead}([\![\overrightarrow{St}]\!], [\![\sigma^{\mathrm{e}}]\!])$
13 **foreach** $i \in \{1, \ldots, m\}$ **do**
14 $[\![d_i]\!] \leftarrow [\![F^{\mathrm{e}}_i]\!] \stackrel{?}{=} [\![F^{\mathrm{e}}_{i+m}]\!]$
15 **foreach** $i \in \{1, \ldots, 2m\}$ **do**
16 $[\![a_i]\!] \leftarrow [\![St^{\mathrm{e}}_i]\!] \wedge \neg [\![d_{i \bmod m}]\!] \; ? \; [\![F^{\mathrm{e}}_i]\!] : (n+1)$
17 $([\![\sigma^{\mathrm{v}}]\!], [\![\tau^{\mathrm{v}}]\!]) \leftarrow \mathsf{prepareWrite}([\![\vec{a}]\!], [\![\vec{\omega}']\!], n+1)$
18 $[\![\vec{F}]\!] := \mathsf{performWrite}([\![\sigma^{\mathrm{v}}]\!], [\![\tau^{\mathrm{v}}]\!], [\![\vec{E}']\!], [\![\vec{F}]\!])$
19 $[\![\vec{\mathcal{W}}]\!] := \mathsf{performWrite}([\![\sigma^{\mathrm{v}}]\!], [\![\tau^{\mathrm{v}}]\!], (i \bmod m)_{i=1}^{2m}, [\![\vec{\mathcal{W}}]\!])$
20 $[\![\vec{\mathcal{T}}]\!] := \mathsf{obliviousWrite}([\![\vec{\mathcal{W}}]\!], \overrightarrow{\mathsf{true}}, \vec{1}, [\![\vec{\mathcal{T}}]\!])$
21 $[\![\sigma^{\mathrm{f}}]\!] \leftarrow \mathsf{prepareRead}([\![\vec{F}]\!], n+1)$
22 $[\![\vec{G}]\!] \leftarrow \mathsf{performRead}([\![\vec{F}]\!], [\![\sigma^{\mathrm{f}}]\!])$
23 $[\![\vec{H}]\!] \leftarrow \mathsf{performRead}([\![\vec{G}]\!], [\![\sigma^{\mathrm{f}}]\!])$
24 **foreach** $i \in \{1, \ldots, n\}$ **do**
25 $[\![c^{(1)}_i]\!] \leftarrow i \stackrel{?}{=} [\![G_i]\!]$
26 $[\![c^{(2)}_i]\!] \leftarrow i \stackrel{?}{<} [\![F_i]\!]$
27 $[\![c^{(3)}_i]\!] \leftarrow [\![F_i]\!] \stackrel{?}{=} [\![H_i]\!] \wedge [\![F_i]\!] \stackrel{?}{<} [\![G_i]\!]$
28 $[\![F_i]\!] := \begin{cases} i, & \text{if } [\![c^{(1)}_i]\!] \wedge [\![c^{(2)}_i]\!] \\ [\![F_i]\!], & \text{if } [\![c^{(1)}_i]\!] \wedge \neg [\![c^{(2)}_i]\!] \\ [\![F_i]\!], & \text{if } \neg [\![c^{(1)}_i]\!] \wedge [\![c^{(3)}_i]\!] \\ [\![G_i]\!] & \text{if } \neg [\![c^{(1)}_i]\!] \wedge \neg [\![c^{(3)}_i]\!] \end{cases}$
29 **return** $([\![\mathcal{T}_1]\!], \ldots, [\![\mathcal{T}_m]\!])$

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 $x \bmod m$ 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. 1. 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. 1 — 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 iteration starts very similarly to Alg. 1, 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. 1). In line 9 of Alg. 1, 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 14–16 of Alg. 7: we choose the address of the element to update. If the update should be made then this address is $[\![F^{\mathrm{e}}_i]\!]$. Otherwise, it is the dummy address $n+1$. In lines 17–19 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 20 we update $[\![\vec{\mathcal{T}}]\!]$ similarly to line 13 of Alg. 1.

Compared to Alg. 1, 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. 1, 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 25–28 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 28 is implemented as a composition of ? : operations.

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

Alg. 7 (including Alg. 6) is secure for the reasons given in Sec. 3.1.3 — 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 $O(\log^2|V|)$, because each private reading or writing preparation requires $O(\log|V|)$ rounds.

Awerbuch-Shiloach algorithm (Alg. 1) accesses all of its memory during each of its iterations. 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.

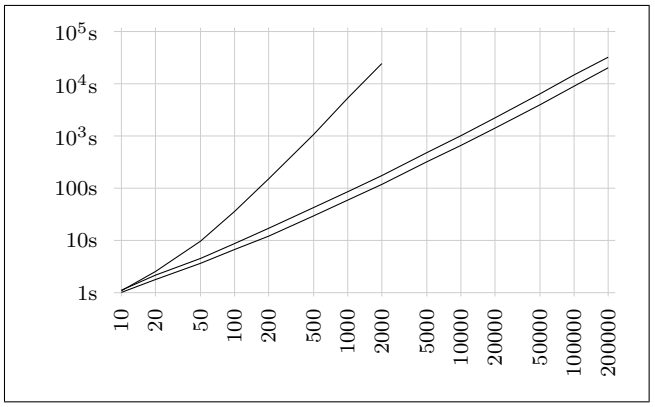

Fig. 5. Running times for the private MST algorithm, depending on the number of vertices n. Number of edges is $m = 3n$ (lower line), $m = 6n$ (middle line), $m = n(n-1)/2$ (upper line)

The results of our performance tests are depicted in Fig. 5. These will serve as the baseline for any further investigations in this direction.

7 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 [38], 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. 6 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.

As we already mentioned in Sec. 5, our oblivious parallel array access protocols may be used to turn any efficient PRAM algorithm into a privacy-preserving computation, as long as the control flow of the algorithm does not depend on private data. We have already applied these protocols in a number of privacy-preserving applications. In [50], parallel algorithms for string matching have been adapted for SMC. We have also implemented a privacy-preserving Bayesian spam filter, and a privacy-preserving database query engine that keeps track, which original rows contributed to the answers of different queries, in order to provide personalised differential privacy [23] to the answers.

Privacy-preserving MST may also find applications in a number of areas, e.g. for privacy-preserving gene expression data clustering [56] or image processing [55]. The likely real-world data sizes for these applications are in the range of the experiments we reported in Fig. 5.

Acknowledgements

This research has been supported by Estonian Research Council through grant No. IUT27-1, by the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS, and by the European Union Seventh Framework Programme (FP7/2007–2013) under grant agreement No. 284731 (UaESMC).

References

[1] M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele. Secure computation on floating point numbers. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[2] A. Aly, E. Cuvelier, S. Mawet, O. Pereira, and M. V. Vyve. Securely solving simple combinatorial graph problems. In A.-R. Sadeghi, editor, *Financial Cryptography*, volume 7859 of *Lecture Notes in Computer Science*, pages 239–257. Springer, 2013.

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

[4] A. Ben-David, N. Nisan, and B. 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.

[5] M. Blanton and E. Aguiar. Private and oblivious set and multiset operations. In H. Y. Youm and Y. Won, editors, *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012*, pages 40–41. ACM, 2012.

[6] M. Blanton, A. Steele, and M. Aliasgari. Data-oblivious graph algorithms for secure computation and outsourcing. In K. Chen, Q. Xie, W. Qiu, N. Li, and W. Tzeng, editors, *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, pages 207–218. ACM, 2013.

[7] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In S. Jajodia and J. López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[8] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.

[9] E. Boyle, K.-M. Chung, and R. Pass. Oblivious parallel ram. Cryptology ePrint Archive, Report 2014/594, 2014. http://eprint.iacr.org/.

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

[11] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, pages 223–239, Washington, DC, USA, 2010.

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

[13] O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In J. Garay and R. De Prisco, editors, *Security and Cryptography for Networks*, volume 6280 of *LNCS*, pages 182–199. Springer, 2010.

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

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

[16] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. In B. Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2001.

[17] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In S. Halevi and T. Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[18] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In S. Jarecki and G. Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.

[19] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[20] I. Damgård, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious ram without random oracles. In Y. Ishai, editor, *TCC*, volume 6597 of *Lecture Notes in Computer Science*, pages 144–163. Springer, 2011.

[21] I. Damgård and J. B. Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In D. Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2003.

[22] D. Demmler, T. Schneider, and M. Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*. The Internet Society, 2015.

[23] H. Ebadi, D. Sands, and G. Schneider. Differential privacy: Now it's getting personal. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 69–81. ACM, 2015.

[24] R. Gennaro, M. O. Rabin, and T. Rabin. Simplified vss and fact-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111, 1998.

[25] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs. Optimizing oram and using it efficiently for secure computation. In E. D. Cristofaro and M. Wright, editors, *Privacy Enhancing Technologies*, volume 7981 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[26] C. Gentry, S. Halevi, C. Jutla, and M. Raykova. Private Database Access With HE-over-ORAM Architecture. Cryptology ePrint Archive, Report 2014/345, 2014. http://eprint.iacr.org/.

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

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

[29] S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure Two-Party Computation in Sublinear (Amortized) Time. In T. Yu, G. Danezis, and V. 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.

[30] K. Hamada, R. Kikuchi, D. Ikarashi, K. Chida, and K. Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In T. Kwon, M.-K. Lee, and D. Kwon, editors, *ICISC*, volume 7839 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2012.

[31] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. 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.

[32] Y. Huang, D. Evans, and J. Katz. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols? In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

[33] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[34] M. Keller and P. Scholl. Efficient, Oblivious Data Structures for MPC. In P. Sarkar and T. Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, volume 8874 of *Lecture Notes in Computer Science*, pages 506–525. Springer, 2014.

[35] M. Keller, P. Scholl, and N. P. Smart. An architecture for practical actively secure mpc with dishonest majority. In Sadeghi et al. [47], pages 549–560.

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

[37] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In Y. 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.

[38] P. Laud and J. Willemson. Composable oblivious extended permutations. In F. Cuppens, J. García-Alfaro, A. N. Z. Heywood, and P. W. L. Fong, editors, *Foundations and Practice of Security - 7th International Symposium, FPS 2014, Montreal, QC, Canada, November 3-5, 2014. Revised Selected Papers*, volume 8930 of *Lecture Notes in Computer Science*, pages 294–310. Springer, 2014.

[39] J. Launchbury, I. S. Diatchki, T. DuBuisson, and A. Adams-Moran. Efficient lookup-table protocol in secure multiparty computation. In P. Thiemann and R. B. Findler, editors, *ICFP*, pages 189–200. ACM, 2012.

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

[41] H. Lipmaa and T. Toft. Secure equality and greater-than tests with sublinear online complexity. In F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, and D. Peleg, editors, *ICALP (2)*, volume 7966 of *Lecture Notes in Computer Sci-*

ence, pages 645–656. Springer, 2013.

[42] C. Liu, Y. Huang, E. Shi, J. Katz, and M. W. Hicks. Automating efficient ram-model secure computation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 623–638. IEEE Computer Society, 2014.

[43] L. Malka. Vmcrypt: modular software architecture for scalable secure computation. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 715–724. ACM, 2011.

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

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

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

[47] A. Sadeghi, V. D. Gligor, and M. Yung, editors. *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. ACM, 2013.

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

[49] E. Shi, T. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In D. H. Lee and X. 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.

[50] S. Siim. Privacy-Preserving String Matching with PRAM Algorithms. Cryptography Seminar report, University of Tartu, 12 2014. https://courses.cs.ut.ee/MTAT.07.022/2014_fall/uploads/Main/sander-report-f14.pdf.

[51] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Sadeghi et al. [47], pages 299–310.

[52] T. Toft. Secure data structures based on multi-party computation. In C. Gavoille and P. Fraigniaud, editors, *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 291–292. ACM, 2011. Full version in Cryptology ePrint archive, http://eprint.iacr.org/2011/081.

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

[54] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi. SCORAM: Oblivious RAM for Secure Computation. In G. Ahn, M. Yung, and N. Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 191–202. ACM, 2014.

[55] J. Wassenberg, W. Middelmann, and P. Sanders. An efficient parallel algorithm for graph-based image segmentation. In X. Jiang and N. Petkov, editors, *Computer Analysis of Images and Patterns, 13th International Conference, CAIP 2009, Münster, Germany, September 2-4, 2009. Proceedings*, volume 5702 of *Lecture Notes in Computer Science*, pages 1003–1010. Springer, 2009.

[56] Y. Xu, V. Olman, and D. Xu. Clustering gene expression data using a graph-theoretic approach: an application of minimum spanning trees. *Bioinformatics*, 18(4):536–545, 2002.

[57] A. C.-C. Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.

[58] S. Zahur and D. Evans. Circuit structures for improving efficiency of security and privacy tools. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 493–507. IEEE Computer Society, 2013.