Efficient, Oblivious Data Structures for MPC

Marcel Keller and Peter Scholl

Department of Computer Science, University of Bristol
{m.keller,peter.scholl}@bristol.ac.uk

Abstract. We present oblivious implementations of several data structures for secure multiparty computation (MPC) such as arrays, dictionaries, and priority queues. The resulting oblivious data structures have only polylogarithmic overhead compared with their classical counterparts. To achieve this, we give secure multiparty protocols for the ORAM of Shi et al. (Asiacrypt '11) and the Path ORAM scheme of van Dijk et al. (CCS '13), and we compare the resulting implementations. We subsequently use our oblivious priority queue for secure computation of Dijkstra's shortest path algorithm on general graphs, where the graph structure is secret. To the best of our knowledge, this is the first implementation of a non-trivial graph algorithm in multiparty computation with polylogarithmic overhead.
We implemented and benchmarked all of our protocols using the SPDZ protocol of Damgård et al. (Crypto '12), which works in the preprocessing model and ensures active security against an adversary corrupting all but one players. For two parties, the access time for an oblivious array of size 1 million is under 250 ms.

Keywords: Multiparty computation, data structures, oblivious RAM, shortest path algorithm

1 Introduction

In a secure multi-party computation (MPC) protocol, parties wish to perform some computation on their inputs without revealing the inputs to one another. The typical approach to securely implementing an algorithm for MPC is to rewrite the algorithm as a boolean circuit (or arithmetic circuit in some finite field) and then execute each gate of the circuit using addition or multiplication in the MPC protocol. For non-trivial functionalities, however, the resulting circuit can incur a large blow-up compared with the normal runtime. For example, algorithms that use a secret index as a lookup to an array search over the entire array when implemented naïvely in MPC, to avoid revealing which element was accessed. This means that the advantages of using complex data structures such as hash tables and binary trees cannot be translated directly to secure computation programs.

Oblivious RAM (ORAM) allows a client to remotely access their private data stored on a server, hiding the access pattern from the server. Ostrovsky and Shoup first proposed combining MPC and ORAM for a two-server writable PIR protocol [20], and Gordon et al. further explored this idea in a client-server setting, constructing a secure two-party computation protocol with amortized sublinear complexity in the size of the server's input using Yao's garbled circuits [12]. In the latter work, the state of an ORAM client is secret shared between two parties, whilst one party holds the server state (encrypted under the client's secret key). A secure computation protocol is then used to execute each ORAM instruction, which allows a secure lookup to an array of size N in $\mathsf{polylog}(N)$ time, in turn enabling secure computation of general RAM programs.

1.1 Our Contributions

Motivated by the problem of translating complex algorithmic problems to the setting of secure computation, we build on the work of Ostrovsky and Shoup [20] and Gordon et al. [12] by presenting new, efficient data structures for MPC, and applying this to the problem of efficient, secure computation of a shortest path on general graphs using Dijkstra's algorithm. Our contributions are outlined below.

Data structure	Based on	Access complexity	Section
Oblivious array	Demux (Protocol 2)	$O(N)$	3.1
Oblivious dictionary	Trivial ORAM	$O(N \cdot \ell)$	3.2
Oblivious dictionary	Trivial ORAM	$O(N + \ell \cdot \log N)$	3.3
Oblivious array	SCSL ORAM and trivial oblivious dictionary	$O(\log^4 N)$	4.2
Oblivious array	Path ORAM and trivial oblivious dictionary	$\tilde{O}(\log^3 N)$	4.3
Oblivious priority queue	Oblivious array	$O(\log^5 N)$	5.1

Fig. 1. Overview of our oblivious data structures. For the dictionary, ℓ is the maximal size of keys. The $\tilde{O}$ notation hides $\log \log N$ factors.

Oblivious array and dictionary. In the context of MPC, we define an oblivious array as a secret shared array that can be accessed using a secret index, without revealing this index. Similarly, an oblivious dictionary can be accessed by a secret-shared key, which may be greater than the size of the dictionary.

We give efficient, polylogarithmic MPC protocols for oblivious array lookup based on two ORAM schemes, namely the SCSL ORAM of Shi et al. [22] (with an optimization of Gentry et al. [10]) and the Path ORAM scheme of van Dijk et al. [23], and evaluate the efficiency of both protocols. Our approach differs from that of Gordon et al., who consider only a client-server scenario where the server has a very large input. Instead, we use a method first briefly mentioned by Damgård et al. [8], where all inputs are secret shared across all parties, who also initially share the client state of the ORAM. The server's ORAM state is then constructed from the secret shared inputs using MPC and so is secret shared across all parties, but does not need to be encrypted since secret sharing ensures the underlying data is information theoretically hidden. This approach has two benefits: firstly, it scales naturally to any number of parties by simply using any secret-sharing based MPC protocol, and secondly it avoids costly decryption and re-encryption operations within MPC.

Since the benefits of using ORAM only become significant for large input sizes (> 1000), we have also paid particular effort to creating what we call *Trivial ORAM* techniques for searching and accessing data on small input sizes with linear overhead, when full ORAM is less practical. The naive method for searching a list of size N involves performing a secure comparison between every element and the item being searched for, which takes time $O(N \cdot \ell)$ when comparing ℓ-bit integers. In Section 3 we present two $O(N)$ methods for oblivious array and dictionary lookup. These techniques come in useful for implementing the ORAM schemes for large inputs, but could also be used for applications on their own.

Figure 1 gives an overview of our algorithms. As parameters for the complexity we use N for the size and ℓ for the maximal bit length of keys. Note that an oblivious dictionary implies an oblivious array with $\ell = \log N$. Furthermore, we assume that the number of accesses to the more intricate data structures is in $O(N \log(N))$.

Secure priority queue and Dijkstra's algorithm. In Section 5 we use our oblivious array to construct an oblivious priority queue, where secret shared items can be efficiently added and removed from the queue without revealing any information (even the type of operation being performed) and show how to use this to securely implement Dijkstra's shortest path algorithm in time $O(|E| \log^5 |E| + |V| \log^4 |V|)$, when only the number of vertices and edges in the graph is public. The previous best known algorithm [2] for this takes time in $O(|V|^3)$. We also show how to modify the underlying ORAM to implement a priority queue directly, where each priority queue operation essentially takes just one ORAM access (instead of $\log |V|$ accesses). However, we are not able to implement the decrease-key operation using this method, so it cannot be used to improve Dijkstra's algorithm any further.

Novel MPC and ORAM techniques. We introduce several new techniques for MPC and ORAM that are used in our protocols, and may be of use in other applications. In Section 4.3 we describe a new method for *obliviously shuffling* a list of secret shared data points, actively secure against a dishonest majority of corrupted adversaries, using permutation networks. To the best of our knowledge, in the multi-party setting this has previously only been done for threshold adversaries. Section 4.4 describes a method for *batch*

initialization of the SCSL ORAM in MPC using oblivious shuffling and sorting, which saves an $O(\log^3 N)$ factor compared with naively performing an ORAM access for each item, in practice giving a speedup of 10–100 times. Finally, in Section 4.1 we give a new protocol that computes a substring with secret-shared start and end index from a bit string secret-shared in the finite field $\mathsf{GF}(2^n)$, which is equivalent to reducing an integer modulo a secret-shared power for prime fields as presented by Aliasgari et al. [1].

Implementation. We implemented and benchmarked the oblivious array using various ORAM protocols and Dijkstra's algorithm. Our implementation uses the SPDZ protocol of Damgård et al. [7,9], which is in the *preprocessing model*, separating the actual computation from a preprocessing phase where secret random multiplication triples and random bits are generated. The resulting online protocol is actively secure against a dishonest majority, so up to $n-1$ of n parties may be corrupted. We use the MPC compiler and framework of Keller et al. [16] to ensure that the minimal round complexity is achieved for each protocol.

1.2 Related Work

Other than the works already discussed [8,12,20], Gentry et al. [10] describe how to use homomorphic encryption combined with ORAM for reducing the communication cost of ORAM and also for secure computation in a client-server situation. These works are in a similar vein to ours, but our work is the first to thoroughly explore using ORAM for oblivious data structures, and applies to general MPC rather than a specific client-server scenario. We also expect that the access times from our interactive approach are much faster than what could be obtained using homomorphic encryption. Lu and Ostrovsky [18] recently showed how to garble RAM programs for two party computation, which is an elegant theoretical result but does not seem to be practical at this time.

Toft described an oblivious secure priority queue with deterministic operations for use in MPC, without using ORAM [24]. The priority queue is based on a bucket heap, and supports insertion and removal in amortized $O(\log^2 N)$ time, but cannot support the decrease-key operation needed for Dijkstra's algorithm, unlike our main ORAM-based priority queue. Secure variants of shortest path and maximum flow algorithms were presented for use in MPC by Aly et al. [2]. They operate on a secret-shared adjacency matrix without using ORAM, which leads to an asymptotic complexity in $O(|V|^3)$ for Dijkstra's algorithm compared to $O(|E| + |V| \log |V|)$ for the implementation on a single machine. Our solution incurs only an overhead in $O(\log^5 |E| + \log^4 |V|)$ over the classical algorithm. Gentry et al. [10] also show how to modify their ORAM scheme to allow for lookup by key instead of index, by taking advantage of the tree-like structure in recursive ORAM. This was the inspiration for our second priority queue protocol given in Appendix B.

2 Preliminaries

The usual model of secure multiparty computation is the so-called arithmetic black box over a finite field. The parties have pointers to field elements inside the box, and they can order the box to add, multiply, or reveal elements. The box will follow the order if a sufficient number of parties (depending of the MPC scheme used) support it. In the case of the SPDZ protocol that we use for our experiments, only the full set of parties is sufficient. There is a large body of works in this model, some of which we refer to in the next section.

2.1 Building blocks

In this section, we will refer to a few sub-protocols that we use later. All protocols in this paper can be found in Appendix A.

- $b \leftarrow \mathsf{EQZ}([a], \ell)$ computes whether the ℓ-bit value a is zero. Catrina and de Hoogh [4] provide implementations for prime order fields that require either $O(\log \ell)$ rounds or a constant number of rounds.

For fields of characteristic two, EQZ can be implemented by computing the logical OR of the bit decomposition of a. Generally, a simple protocol requires $O(\ell)$ invocations in $O(\log \ell)$ rounds, and $\mathsf{PreMulC}$ by Damgård et al. [6] allows for constant rounds. However, the constant-round version turns out to be slower in our implementation.

- $([b_0], \ldots, [b_{\ell-1}]) \leftarrow \mathsf{PreOR}([a_0], \ldots, [a_{\ell-1}])$ computes the prefix OR of the ℓ input bits. Catrina and de Hoogh [4] presented an implementation that requires $O(\ell)$ invocations in $O(\log \ell)$ rounds, while Damgård et al. [6] showed that a constant-round implementation is feasible. Again, the constant-round implementation is slower in our implementation.
- $\mathsf{IfElse}([c], [a], [b])$ emulates branching by computing $[a] + [c] \cdot ([b] - [a])$. If c is zero, the result is a, if c is one, the result is b. Similarly, $\mathsf{CondSwap}$ in Protocol 1 swaps two secret-shared values depending on a secret-shared bit.
- $[b_0], \ldots, [b_{2^n-1}] \leftarrow \mathsf{Demux}([a_0], \ldots, [a_n])$ computes a vector of bits such that the a-th element is 1 whereas all others are 0 if $(a_0, \ldots, a_n)$ is the bit decomposition of a. We use this for our $O(N)$ oblivious array as well as the position map in Tree ORAM. The implementation of Demux in Protocol 2 is due to Launchbury et al. [17]. It requires $O(2^\ell)$ invocations in $\lceil \log \ell \rceil$ rounds because one call of Demux with ℓ inputs induces 2^ℓ multiplications in one round and two parallel calls to Demux with at most $\lceil \ell/2 \rceil$ inputs.
- $\mathsf{Sort}([x_0], \ldots, [x_{n-1}])$ sorts a list of n values. This can be done using Batcher's odd-even mergesort with $O(n \log^2 n)$ secure comparisons, or more efficiently using the method of Jónsson et al. [15] in $O(n \log n)$, with our oblivious shuffling protocol in Section 4.3. The latter is secure only if the sorted elements are unique.

2.2 Tree ORAM Overview

The ORAM schemes we use for our MPC protocols have the same underlying structure as the recursive Tree ORAM of Shi et al. [22] (SCSL), where N ORAM entries are stored in a complete binary tree with N leaves and depth $D = \lceil \log N \rceil$, encrypted under the client's secret key. Nodes in the tree are *buckets*, which each hold up to Z entries encrypted under the client's secret key, where the choice of Z affects statistical security. Each entry within a bucket consists of a triple (a, d_a, L_a), where a is an address in $\{0, \ldots, N-1\}$, d_a is the corresponding data, and L_a is a leaf node currently identified with a. The main invariant to ensure correctness is that at any given time, the entry (a, d_a, L_a) lies somewhere along the path from the root to the leaf node L_a.

The client stores a *position map*, which is a table mapping every address a to its corresponding leaf node L_a. To access the entry at address a, the client simply requests all buckets on the path to L_a, decrypts them and identifies the matching entry. If a write is being performed, the value is updated with a new value. Next, the client chooses a new leaf L'_a uniformly at random, updates the entry with L'_a and places the entry in the root node bucket of the tree. The path is then re-encrypted and sent back to the server.

Since a new leaf mapping is chosen at random every time an entry is accessed, the view of the server is identical at every access. Note that buckets are always padded to their full capacity, even if empty. To distribute the entries on the tree, an *eviction* procedure is used, which pushes entries further down the tree towards the leaves to spread out the capacity more evenly.

The original eviction method of Shi et al. is as follows:

- Client chooses at random two buckets from each level of the tree except the leaves, and requests these and each of their two children from server.
- For each chosen bucket, push an entry down into one of its child buckets, choosing the child based on the entry's corresponding leaf node.
- Re-encrypt the buckets and their children before sending them to the server.

Shi et al. showed that if the bucket size Z is set to k, the probability that any given bucket will overflow (during a single ORAM access) is upper bounded by 2^{-k}.

Reducing client storage via recursion. The basic ORAM described above requires a linear amount of client-side storage, due to the position map. To reduce this, the position map can itself be recursively implemented as an ORAM of size N/χ, by packing χ indices into a single entry. If, say, $\chi = 2$ then recursing this $\log_\chi N$ times results in a final position map of a single node. However, the communication and computation cost is increased by a factor of $O(\log N/\log \chi)$. Note that the entry size must be at least χ times as large as the index size, to be able to store χ indices in each entry.

Gentry et al. ORAM optimizations. Gentry et al. [10] proposed two modifications to the SCSL ORAM to improve the parameter sizes. The first of these is to use a shallower tree, with only N/k instead of N leaves, for an ORAM of size N. The expected size of each leaf bucket is now k, so to avoid overflow it can be shown using a Chernoff bound that it suffices to increase the bucket size of the leaves to $2k$.

Gentry et al. also suggest k to be between 50 and 80, but do not justify their choices any further. If we choose the bucket size as $(1+\delta)k$ for an arbitrary δ, the overflow probability per access is $2^{-(1+\delta)k}$ for an internal bucket by the analysis by Shi et al. and $2^{-\delta^2 k/2}$ for a leaf bucket by Chernoff bound. Summing up we get an overall failure probability of

$$(2^{-(1+\delta)k} + 2^{-\delta^2 k/2})\frac{n}{2k}M \quad (1)$$

for M accesses. On the other hand, the access complexity for one tree is $(1+\delta)k\log(n/k)$. We found that $\delta = 3$ approximately minimizes the access complexity for a fixed overall failure probability. In our implementation, we set k such that (1) is less that 2^{-20} for $M = 100N\log N$ accesses. The number of accesses is motivated by our implementation of Dijkstra's algorithm where the oblivious array holding the priority queue heap is accessed $10N\log N$ times. This choice of k is in $O(\log N)$, which results in an access complexity for one tree in $O(\log^2 N)$ and an overall complexity in $O(\log^3 N)$.

The second optimization of Gentry et al. is to use higher degree trees to make the tree even shallower, which requires a more complex eviction procedure. We did not experiment with this variant, since working with higher degree trees and the new eviction method in MPC does not seem promising to us. When we refer to our implementation of the SCSL ORAM, we therefore mean the protocol with the first modification only.

Path ORAM. Path ORAM is another variant of tree ORAM proposed by van Dijk et al. [23]. It uses a small piece of client side storage, called the 'stash', and a new eviction method, which allows the bucket size Z to be made as small as 4.

For an ORAM query, the client first proceeds as usual by looking up the appropriate leaf node in the position map and requesting the path from the root to this leaf. The accessed entry is found on this path, reassigned another random leaf, and then every entry in the path and the stash is pushed as far down towards the leaf as possible, whilst maintaining the invariant that entries lie on the path towards their assigned leaves. In the case of bucket overflow, entries are placed in the client-stored stash. The proven overflow bound for Path ORAM is not currently very tight, so for our implementation we chose parameters based on simulation results instead of a formal analysis, again looking to achieve an overall overflow probability of 2^{-20} for our applications.

3 Oblivious Data Structures with Linear Overhead

The general model of our oblivious array and dictionary protocols is to secret share both the server and client state of an ORAM between all parties. This means that there is no distinction between client and server anymore because both roles are replaced by the same set of parties.

Since secret sharing hides all information from all parties, the server memory does not need to be encrypted; any requests from the client to the server that are decrypted in the original ORAM protocol are implemented by revealing the corresponding secret shared data. For server memory accesses, the address

must be available in public, which allows the players to access the right shares. Computation on client memory is implemented as a circuit, as required for MPC. This means that any client memory access depending on secret data must be replaced by a scan of the whole client memory, similarly to a Trivial ORAM access. Therefore, ORAM schemes with small client memory are most efficient in our model.

In this section, we outline our implementations based on *Trivial ORAM*, where all access operations are conducted by loading and storing the entire list of entries, and thus have linear overhead. In the context of MPC, the main cost factor is not memory accesses[1], but the actual computation on them. Nevertheless, the two figures are closely related here.

In our experiments (Section 6) we found that, for sizes up to a few thousand, the constructions in this section prove to be more efficient than the ones with better asymptotic complexity despite the linear overhead.

3.1 Oblivious Array

A possible way for obliviously searching an array of size N was proposed by Launchbury et al. [17]. It involves expanding the secret-shared index $i < N$ to a vector of size N that contains a one in the i-th position and zeroes elsewhere. The inner product of this index vector and the array of field elements gives the desired field element. The index vector can likewise be used to replace this field element by a new one while leaving the other field elements intact. The index expansion corresponds to a bit decomposition of the input followed by the Demux operation in Protocol 2. This procedure has cost in $O(N)$.

Storing an empty flag in addition to the array value proves useful for some applications. Therefore, we require that the players store a list of tuples $([v_i], [e_i])_{i=0}^{N-1}$ containing the values and empty flags. Figure 7 shows the combined read and write operation for such an oblivious array of size N. It takes a secret bit $[w]$ indicating whether a new value $[v']$ and empty flag $[e']$ should be written the array or not. The reduction to a read-only or write-only operation or an operation without empty flags is straightforward.

3.2 Oblivious Dictionary

We will use the dictionary in this section as a building block for implementing Tree ORAM in MPC. Previous work refers to the Trivial ORAM used in this section as non-contiguous ORAM, meaning that the index of an entry can be any number, in particular greater than the size of the array.

For simplification, we assume that the dictionary consists of index-value pairs (u, v) where both are field elements of the underlying field. The extension to several values is straightforward. In addition to one index-value pair per entry we store a bit e indicating whether the entry is empty. We will see shortly that this proves to be useful. In summary, the players store a list of tuples of secrets-shared elements $([u_i], [v_i], [e_i])_{i=0}^{N-1}$ for a dictionary of size N. Initially, e_i must be 1 for all i.

The Tree ORAM construction requires the dictionary to provide the following operations:

- $\mathsf{ReadAndRemove}([u])$ returns and removes the value associated with index u if it is present in the dictionary, and a default entry otherwise.
- $\mathsf{Add}([u], [v], [e])$ adds the entry (u, v, e) to the dictionary assuming that no entry with index v exists.
- $\mathsf{Pop}()$ returns and removes a non-empty entry $(u, v, 0)$ if the dictionary is not empty, and the default empty entry $(0, 0, 1)$ otherwise. Shi et al. [22] showed how to implement Pop using $\mathsf{ReadAndRemove}$ and Add, but for Trivial ORAM a dedicated implementation is more efficient.

In MPC, $\mathsf{ReadAndRemove}$ is the most expensive part because it contains a comparison for every entry. Protocol 4 shows how to implement it minimizing the number of rounds. Essentially, it computes an index vector as in the previous section using comparison instead of Demux. The complexity is dominated by the N calls to the equality test EQZ which cost $O(N \cdot \ell)$ invocations in $O(\log \ell)$ or $O(1)$ rounds for ℓ-bit keys, depending on the protocol used for equality testing.

[1] In fact, just accessing the share of an entry comes at no cost because the shares are stored locally.

The Add procedure shown in Protocol 5 highlights the utility of storing bits indicating whether an entry is empty. If this would be done by means of a special index value, Add would have to invoke N equality tests just like ReadAndRemove. Avoiding the equality tests makes the complexity independent of the bit length of the indices because the index vector is computed using PreOR on the list of empty flags. The protocol requires $O(N)$ invocations in $O(\log N)$ or $O(1)$ rounds, depending on the implementation of PreOR being used.

Complementary to Add, the Pop procedure in Protocol 6 starts with finding the first non-empty entry. As Add, the protocol uses PreOR on the list of empty flags and requires $O(N)$ invocations in the loop and PreOR; the round complexity is constant for the loop and $O(\log N)$ or constant for PreOR.

Security. All protocols in this section do not reveal anything. Therefore, they are as secure as the underlying MPC scheme providing the arithmetic blackbox.

3.3 Oblivious Dictionary in $O(N)$

The complexity of ReadAndRemove in the oblivious dictionary can be reduced to only $O(N+\ell\cdot\log N)$ instead of $O(N\cdot\ell)$, at the cost of increasing the round complexity from $O(1)$ to $O(\log N)$.

The protocol (Protocol 8 in Appendix A.2) starts by subtracting the item to be searched for from each input, and builds the *multiplication tree* from these values, which is a complete binary tree where each node is the product of its two children. The original values lie at the leaves, and at each level a single node will be zero, directing us towards the leaf where the item to be found lies. Now we can traverse the tree in a binary search-like manner, at each level obliviously selecting the next element to be compared until we reach the leaves. We do this by computing a bit vector indicating the position of a node which is equal to zero on the current level and then use this to determine the comparison to be performed next. Note that only a single call to EQZ is needed at each level of the tree, but a linear number of multiplications are needed to select the item on the next level.

The main caveat here is that, since the tree of multiplications can cause values to grow arbitrarily, we need to be able to perform EQZ on *any* field element. In $\mathsf{GF}(2^n)$ this works as usual, and in $\mathsf{GF}(p)$ this can be done by a constant-round protocol by Damgård et al. [6].

4 Oblivious Array with Polylogarithmic Overhead

In this section we give polylogarithmic oblivious array protocols based on the SCSL ORAM and Path ORAM schemes described in Section 2.2. These both have the same recursive Tree ORAM structure, so first we describe how to implement the position map, which is the same for both schemes.

4.1 Position map in Tree ORAM

The most intricate part of implementing Tree ORAM in MPC is the position map. Since we implement an oblivious array, the positions can be stored as an ordered list without any overhead. Recall that for the ORAM recursion to work it is essential that at least two positions are stored per memory address of the lower-level ORAM. In an MPC setting, there are two ways of achieving this: storing several field elements in the same memory cell and packing several positions per field element. We use both approaches.

To simplify the indexing, we require that both the number of positions per field element and number of field elements per memory cell are powers of two. This allows to compute the memory address and index within a memory cell by bit-slicing. For example, if there are two positions per field element and eight field elements per memory cell, the least significant bit denotes the index within a field element, the next three bits denote the index within the memory cell, and the remaining bits denote the memory address. Because these parameters are publicly known, the bit-slicing is relatively easy to compute with MPC. In prime order fields, one can use Protocol 3.2 by Catrina and de Hoogh [4], which computes the remainder of the division

by a public power of two; in fields of characteristic two one can simply extract the relevant bits by bit decomposition.

The bit-slicing used to extract a position from a field element is more intricate because the index is secret and must not be revealed. For prime-order fields, Aliasgari et al. [1] provide a protocol that allows to compute the modulo operation of a secret number and two raised to the power of a secret number. For fields of characteristic two, Protocol 9 achieves the same. The core idea of both protocols is to compute $[2^m]$ (or $[X^m]$) where m is the integer representation of secret-shared number. The bit decomposition of $[2^m]$ can then be used to mask the bit decomposition of another secret-shared number. Moreover, multiplying with $\mathsf{Inv}([2^m])$ allows to shift a number with the m least significant bits being zero by m positions to the right. The complexity of both versions of Mod2m is $O(\ell)$ invocations in $O(\log \ell)$ rounds or in a constant number of rounds if the protocols by Damgård et al. [6] are used. The latter proved to be less efficient in our experiments.

Finally, if the position map storage contains several field elements per memory cell, one also needs to extract the field element indexed by a secret number. This can be done using Demux from Protocol 2 in the same way as for the oblivious array in Section 3.1.

4.2 SCSL ORAM

Many parts of Tree ORAM are straightforward to implement using the Trivial ORAM procedures from the last section.

For ReadAndRemove, the position is retrieved from the position map and revealed. All buckets on the path can then be read in parallel. Combining the results can be done similarly to the Trivial ORAM ReadAndRemove because the returned $[e]$ reveals whether the searched index was found in a particular bucket. The Add procedure also guarantees that an index appears at most once in the whole ORAM.

The Add procedure starts with adding the entry to the root bucket followed by the eviction algorithm. The randomness used for choosing the buckets to evict from does not have to be secret because the choice has to be made public. It is therefore sufficient to use a pseudorandom generator seeded by a securely generated value in order to reduce communication. However, it is crucial that the eviction procedure does not reveal which child of a bucket the evicted entry is added to. Therefore, we use the conditional swapping procedure in Protocol 1. See Protocols 11 and 12 for the algorithmic descriptions of ReadAndRemove and Add.

Apart from Evict and EvictBucket in Protocols 14 and 13, our protocols also use GetBucket and GetChild. Those access the public structure of the tree. The former returns a reference to the bucket on a given level on the path to a given leaf, and the latter returns a reference to either child of a bucket.

Using ReadAndRemove and Add, it is straightforward to implement the universal access operation that we will later use in our implementation of Dijkstra's algorithm. Essentially, we start by ReadAndRemove and then write back the value just read or the new value depending on the write flag. Similarly, one can construct a read-only or write-only operation.

Complexity. Since the original algorithm by Shi et al. [22] does not involve branching, it can be implemented in MPC with asymptotically the same number of memory accesses. However, the complexity of an MPC protocol is determined by the amount of computation carried out. In ReadAndRemove, the index of every accessed element is subject to a comparison. These indices are $\log N$-bit numbers for an array of size N. Because the access complexity of the ORAM is in $O(\log^3 N)$, the complexity of all comparisons in ReadAndRemove is in $O(\log^4 N)$. It turns out that this dominates the complexity of the SCSL ORAM operations because Add and Pop do not involve comparisons. Furthermore, the algorithms in Section 4.1 have complexity in $O(\log N)$ and are only executed once per index structure and access. This leads to a minor contribution in $O(\log^2 N)$ per access of the oblivious array.

Security. Our protocols only reveal the path of an entry in ReadAndRemove. As in the original ORAM protocol, this information was previously randomly generated in secret, and is only revealed once. Therefore, the revealed path is independent of any secret information. It follows that the security solely relies on the security of the MPC scheme and the protocols presented in Section 2.1, some of which only provide statistical security.

4.3 Path ORAM

The Path ORAM access protocol is initially the same as in the Shi et al. and Gentry et al. schemes, but differs in its eviction procedure. For eviction a random leaf ℓ is chosen, and we consider the path from the root down to the leaf ℓ. For each entry $E = ([i], [\ell_i], [d_i])$ on the path we want to push E as far down the path as it can go, whilst still being on the path towards ℓ_i and avoiding bucket overflow. The high-level strategy for doing this in MPC is to first calculate the final position of each entry in the stash and the path to ℓ, then express this as a permutation and evaluate the resulting, secret permutation by oblivious shuffling. Since oblivious shuffling has only previously been studied in either the two party or threshold adversary setting, we describe a new protocol for oblivious shuffling with $m-1$ out of m corrupted parties.

We use two additional parameters σ and τ that restrict the possible destination buckets of each entry to simplify the allocation procedure. We allow an entry to be pushed no more than σ levels down the path or τ levels back up the path from its current position. If it cannot be placed in any of these buckets then it goes to the stash. In Appendix A.4 we give simulation results suggesting that $\tau = 3, \sigma = 5$ is a reasonable choice that does not increase the stash size by too much.

The eviction protocol for MPC first does the following for each entry $E = ([i], [\ell_i], [d_i])$ in the path to ℓ and the stash:

- Compute the *least common ancestor*, $\mathsf{LCA}(\ell, [\ell_i])$, of each entry's assigned leaf and the random leaf by XORing the bit-decomposed leaves together and identifying the first non-zero bit of this. If the LCA is not between levels $\mathsf{lev}_i - \tau$ and $\mathsf{lev}_i + \sigma$ a bit indicating the stash is set.
- For $k = \mathsf{lev}_i - \tau, \ldots, \mathsf{lev}_i + \sigma$, compute a bit $[b_k]$ determining whether entry E goes to the bucket on level k or not. To do this we maintain bits $u_{k,i}$ that indicate whether the i-th position in bucket k has been filled. This requires a linear scan of size Z for each k, resulting in a complexity of $O(Z(\tau + \sigma))$ for each entry.
- Compute a bit $[b_S]$, determining whether E ends up in the stash.

Finally, we use the bits $[b_0], \ldots, [b_{L-1}]$ for every entry to construct a permutation mapping entries to their positions after eviction, and then obliviously shuffle the entries and reveal the shuffled permutation so it can be applied. This is the most complex part of the process, since the b_i values initially only give positions of *non-empty* entries that are to be evicted. However, to be able to shuffle and safely reveal the permutation, we must also compute (fake) positions for all the empty entries in the path and the stash. To do this, we obliviously sort the entries by their 'empty' bits, and align the empty entries with sorted empty bucket positions.

Parameters and complexity. Our simulations in Appendix A.4 indicate that, as in [23], a stash size of $O(k)$ is needed for overflow probability of 2^{-k} for a single access. As with the SCSL ORAM, we chose the parameters to allow for $100N \log N$ accesses with total failure probability $2^{-\lambda}$, which means the stash size must asymptotically be $O(\lambda + \log N) = O(\log N)$ for our choice of $\lambda = 20$. We empirically estimated this requires a stash size of 48, with $\tau = 3, \sigma = 5$.

Our protocols for oblivious shuffling and bitwise sorting have complexity $O(n \log n)$. In this case $n = |S| + Z \cdot \log N$ so the total complexity of the access protocol is $O(\log N \log \log N)$ (with quite high constants including Z, σ and τ). Adding in the recursion gives an extra $O(\log N)$ factor, for an overall complexity of $\tilde{O}(\log^2 N)$.

Oblivious Shuffling with a Dishonest Majority. To carry out the oblivious shuffling above, we use a protocol based on *permutation networks*, which are circuits with N input and output wires, consisting only of $\mathsf{CondSwap}$ gates with the control bit for each gate hard-wired into the circuit. Given any permutation, there is an algorithm that generates the control bits such that the network applies this permutation to its inputs. The Waksman network [25] is an efficient permutation network with a recursive structure, with $O(N \log N)$ gates and depth $O(\log N)$. It has been used previously in cryptography for two-party oblivious shuffling [13], and also in fully homomorphic encryption [11] and private function evaluation [19].

Computing the control bits requires iterating through the orbits of the permutation, which seems expensive to do in MPC when the permutation is secret shared. Instead, for m players we use a composition of m permutations, as follows:

- Each player generates a random permutation, and locally computes the control bits for the associated Waksman network [2]
- Each player inputs (i.e. secret shares) their control bits.
- For each value $[b]$ input by a player, open $[b] \cdot [b-1]$ and check this equals 0, to ensure that b is a bit.
- The resulting permutation is evaluated by composing together all the Waksman networks.

Step 3 ensures that all permutation network values input by the players are bits, even for malicious parties, so the composed networks will always induce a permutation on the data. As long as at least one player generates their input honestly, the resulting permutation will be random, as required. The only values that are revealed outside of the arithmetic black box are the bits in step 3, which do not reveal information other than identifying any cheaters. For m players, the complexity of the protocol for an input of size n is $O(mn \log n)$.

4.4 Batch initialization of SCSL ORAM

The standard way of filling up an ORAM with N entries is to execute the access protocol once for each entry. This can be quite expensive, particularly for applications where the number of ORAM accesses is small relative to the size of the input. Below we outline a new method for initializing the SCSL ORAM with N entries in MPC in time $O(N \log N)$, saving a factor of $O(\log^3 N)$ over the standard method. Note that this technique could also be used for standard ORAM (not in MPC), and then the complexity becomes $O(N)$, which seems optimal for this task. See Appendix A.6 for details.

Recall that each entry to be added to the ORAM must be assigned a secret leaf, chosen uniformly and independently at random. As a first attempt, we might consider obliviously shuffling the entries, revealing the leaves and then placing them in the appropriate leaf bucket. Although correct, this reveals how many entries were assigned to a particular bucket, which could leak information during future accesses. To avoid this we create dummy entries, so that the total number of entries is the number of positions available in all of the leaf buckets, i.e. $Z \cdot \lceil N/k \rceil$, where Z is the bucket size. We then use a protocol to assign leaves to all the dummy entries so that exactly Z entries in total are assigned to each leaf. This ensures that after obliviously shuffling the combined dummy and real entries, it is safe to reveal the leaf assigned to each entry, since the distribution of these leaves will always be fixed. Finally every entry can be placed directly in its leaf bucket. The standard Chernoff bound analysis ensures with high probability that no leaf bucket will overflow. One interesting aspect of the protocol requires the randomly chosen leaves to be sorted in order to process them. This would normally cost $O(N \log^2 N)$ in MPC, but instead we point out that it's possible to sample sorted values from the correct distribution using secure floating point arithmetic techniques in only $O(N \log N)$.

5 Applications

In this section, we will use the oblivious array from the previous section to construct an oblivious priority queue, which we then will use in a secure multiparty computation of Dijkstra's shortest path algorithm.

5.1 Oblivious Priority Queue

A simple priority queue follows the design of a binary heap: a binary tree where every child has a higher key than its parent. The key-value pairs are put in an oblivious array, with the root at index 0, the left and

[2] Note that generating the control bits randomly does not produce a random permutation, since for a circuit with m gates, there are 2^m possible paths and so any permutation will occur with probability $k2^{-m}$ for some integer k. However, for a uniform permutation we require $k2^{-m} = \frac{1}{n!}$.

right child of the root at indices 1 and 2, respectively, and so on. In order to be able to decrease the key of a value, we maintain a second array that links the keys to the respective indices in the first one.

The usage of ORAM requires that we set an upper limit on the size of the queue. We store the actual size of the heap in a secret-shared variable, which is used to access the first free position in the heap. Depending on the application, the size of the heap can be public and thus stored in clear. However, our implementation of Dijkstra's algorithm necessitates the size to be secret. This also means that the bubbling-up and -down procedures have to be executed on the tree of maximal size. Protocol 17 demonstrates the bubble-up procedure with Heap and Index as the arrays mentioned above and $[s]$ as the secret-shared size. The bubble-down procedure works similarly, additionally deciding whether to proceed with the left or the right child. Note that unlike the presentation in Sections 3 and 4, we allow tuples as array values here. This extension is straightforward to implement. Furthermore, we require a universal access operation like Protocol 7. If such an operation is not available yet, it can implemented by reading first and then either writing the value just read or a new value depending on the write flag.

Our application also requires a procedure that combines insertion and decrease-key (depending on whether the value already is in the queue) and that can be controlled by a secret flag deciding whether the procedure actually should be executed or not. The combination is straightforward since both employ bubbling up, and the decision flag can be implemented using the universal access operation mentioned above. Figure 18 shows the update procedure with Heap, Index, and $[s]$ as in Protocol 17.

For a priority queue of maximal size N, both bubbling up and down run over $\log N$ levels accessing both Heap and Index, whose size is in $O(N)$. Using the SCSL ORAM therefore results in a complexity in $O(\log^5 N)$ per access operation. In Appendix B we show how to modify the Tree ORAM structure to reduce the overhead to $O(\log^4 N)$ per access, but are unable to implement the decrease-key operation in this manner. Therefore, we cannot use it for Dijkstra's algorithm.

5.2 Secure Dijkstra's algorithm

In this section we show how to apply the oblivious priority queue to a secure variant of Dijkstra's algorithm, where both the graph structure and the source vertex are initially secret shared across all parties. In our solution, the only information known by all participants is the number of vertices and edges. It is straightforward to have public upper bounds instead while keeping the actual figures secret. However, the upper bounds then determine the running time. This is inevitable in the setting of MPC because the parties are aware of the amount of computation carried out.

In the usual presentation [5], Dijkstra's algorithm contains a loop nested in another one. The outer loop runs over all vertices, and the inner loop runs over all neighbours of the current vertex. Directly implementing this in MPC would reveal the number of neighbours of the current vertex and thus some of the graph structure. On the other hand, executing the inner loop once for every other vertex incurs a performance punishment for graphs other than the complete graph. As a compromise, one could execute the inner loop as many times as the maximum degree of the graph, but even that would reveal some extra information about the graph structure. Therefore, we replaced the nested loops by a single loop that runs over all edges (twice in case of an undirected graph). Clearly, this has the same complexity as running over all neighbours of every vertex. The key idea of our variant of Dijkstra's algorithm is to execute the body of the outer loop for every edge but ignoring the effects unless the current vertex has changed. For this, the write flag of the universal oblivious array access operation plays a vital role.

We now explain the data structure used by our implementation. Assume that the N vertices are numbered by 0 to $N-1$. We use two oblivious arrays to store the graph structure. The first is a list of neighbours ordered by vertex: it starts with all neighbours of vertex 0 (in no particular order) followed by all neighbours of vertex 1 and so on. In addition, we store for every neighbour a bit indicating whether this neighbour is the last neighbour of the current vertex. The length of this list is twice the number of edges for an undirected graph. We will refer to this array as Edges from now on. In Index, we store for every vertex the index of its first neighbour in Edges. A third array, Dist, is used to store the results: the distance from the source to every vertex. For the sake of simpler presentation, we omit storing the predecessor on the shortest path.

Finally, we use the priority queue Q to store unprocessed vertices with the shortest encountered distance so far. Protocol 19 shows our algorithm.

The main loop accesses a priority queue of size N_V and arrays of size N_E and N_V. Furthermore, we have to initialize a priority queue and an array of size N_V. Using the SCSL ORAM everywhere, this gives a complexity in $O(N_V \log^4 N_V + N_E(\log^5 N_V + \log^4 N_E))$. More concretely, for sparse graphs with $N_E \in O(N_V)$ (such as cycle graphs) the complexity is in $O(N_V \log^5 N_V)$, whereas for dense graphs with $N_E \in O(N_V^2)$ (such as complete graphs), the complexity is in $O(N_V^2 \log^5 N_V)$. The most efficient implementation of Dijkstra's algorithm on a single machine has complexity in $O(N_E + N_V \log N_V)$, and thus the penalty of using MPC is in $O(\log^5 N_V + \log^4 N_E)$.

6 Experiments

We implemented our protocols for oblivious arrays using the system described by Keller et al. [16], and ran experiments on two directly connected machines running Linux on an eight-core i7 CPU at 3.1 GHz. The access times for different implementations of oblivious arrays containing one element of $\mathsf{GF}(2^{40})$ per index are given in Figure 2. Most notably, the linear constructions are more efficient for sizes up to a few thousand. The performance in MPC of Path ORAM and SCSL ORAM are similar; although Path ORAM is asymptotically slightly better, the constants are quite high. It's also worth noting that the proven statistical security bound for Path ORAM is currently not very tight, so our parameters were chosen according to simulations, unlike for the SCSL ORAM.

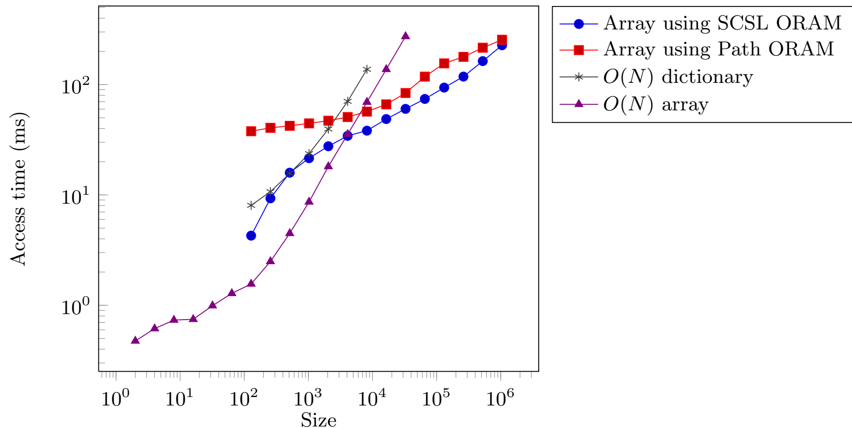

Fig. 2. Oblivious data structure access times.

6.1 Dijkstra's Algorithm

We have benchmarked Dijkstra's algorithm on cycle graphs of varying size. The edges of a cycle graph form a cycle passing every vertex exactly once. We chose it as a simple example of a graph with low degree. Figure 3 shows the timings of our implementation of the algorithm without ORAM techniques by Aly et al. [2] as well as our implementation using the oblivious array in Section 3.1 and the SCSL ORAM in Section 4.2. In all cases, we used MPC over a finite field modulo a 128-bit prime to allow for integer arithmetic. We generated our estimated figures using a fully functional secure protocol but running the main loop only a fraction of the times necessary.

Our algorithms perform particularly well in comparison to the one by Aly et al. because cycle graphs have very low degree. For complete graphs, Figure 5 in Appendix C shows a different picture. The overhead for

using ORAM is higher than the asymptotic advantage for all graph sizes that we could run our algorithms on.

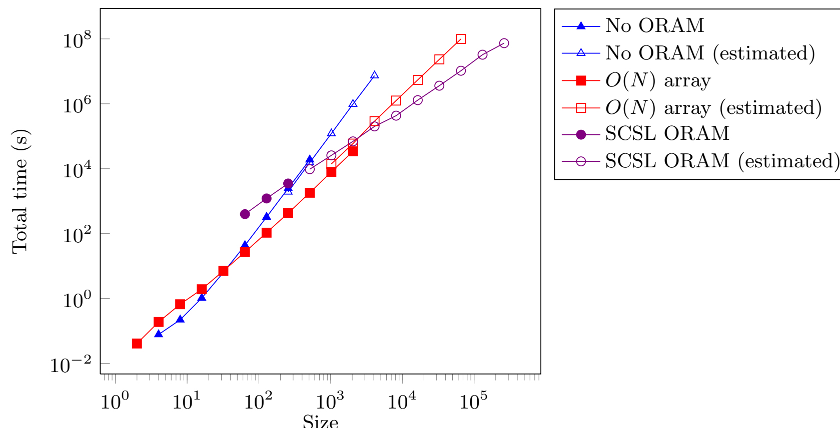

Fig. 3. Dijkstra's algorithm on cycle graphs

6.2 Batch initialization

We implemented the batch initialization protocol for the SCSL ORAM in MPC from Section 4.4. Figure 6 in Appendix C compares performance of this with initialization using N iterations of the access protocol. For $N = 2^{14}$ the batch initialization is around 10 times faster, with the improvement increasing rapidly with N. We have not implemented the method for sampling N pre-sorted random numbers, so instead use Batcher's odd-even sorting network for this, giving complexity $O(N \log^3 N)$ instead of the possible $O(N \log N)$. When this is implemented and scaled to larger sizes we expect at least another order of magnitude of improvement.

Acknowledgements

We would like to thank Nigel Smart for various comments and suggestions.

References

1. Aliasgari, M., Blanton, M., Zhang, Y., and Steele, A. Secure computation on floating point numbers. In *NDSS* (2013), The Internet Society.
2. Aly, A., Cuvelier, E., Mawet, S., Pereira, O., and Vyve, M. V. Securely solving simple combinatorial graph problems. In *Financial Cryptography* (2013), A.-R. Sadeghi, Ed., vol. 7859 of *Lecture Notes in Computer Science*, Springer, pp. 239–257.
3. Bentley, J. L., and Saxe, J. B. Generating sorted lists of random numbers. *ACM Trans. Math. Softw. 6*, 3 (Sept. 1980), 359–364.
4. Catrina, O., and de Hoogh, S. Improved primitives for secure multiparty integer computation. In *SCN* (2010), J. A. Garay and R. D. Prisco, Eds., vol. 6280 of *Lecture Notes in Computer Science*, Springer, pp. 182–199.
5. Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001.
6. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J. B., and Toft, T. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC* (2006), S. Halevi and T. Rabin, Eds., vol. 3876 of *Lecture Notes in Computer Science*, Springer, pp. 285–304.

7. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., and Smart, N. P. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS* (2013), J. Crampton, S. Jajodia, and K. Mayes, Eds., vol. 8134 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.
8. Damgård, I., Meldgaard, S., and Nielsen, J. B. Perfectly secure oblivious RAM without random oracles. In *Theory of Cryptography*. Springer, 2011, pp. 144–163.
9. Damgård, I., Pastro, V., Smart, N. P., and Zakarias, S. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology – CRYPTO 2012* (2012), R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*, Springer, pp. 643–662.
10. Gentry, C., Goldman, K. A., Halevi, S., Julta, C., Raykova, M., and Wichs, D. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies* (2013), Springer, pp. 1–18.
11. Gentry, C., Halevi, S., and Smart, N. Fully homomorphic encryption with polylog overhead. *Advances in Cryptology–EUROCRYPT 2012* (2012), 465–482.
12. Gordon, S. D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., and Vahlis, Y. Secure two-party computation in sublinear (amortized) time. In *ACM Conference on Computer and Communications Security* (2012), T. Yu, G. Danezis, and V. D. Gligor, Eds., ACM, pp. 513–524.
13. Huang, Y., Evans, D., and Katz, J. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS* (2012), The Internet Society.
14. Johansson, T., and Nguyen, P. Q., Eds. *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings* (2013), vol. 7881 of *Lecture Notes in Computer Science*, Springer.
15. Jónsson, K. V., Kreitz, G., and Uddin, M. Secure multi-party sorting and applications. *IACR Cryptology ePrint Archive 2011* (2011), 122.
16. Keller, M., Scholl, P., and Smart, N. P. An architecture for practical actively secure MPC with dishonest majority. In Sadeghi et al. [21], pp. 549–560.
17. Launchbury, J., Diatchki, I. S., DuBuisson, T., and Adams-Moran, A. Efficient lookup-table protocol in secure multiparty computation. In *ICFP* (2012), P. Thiemann and R. B. Findler, Eds., ACM, pp. 189–200.
18. Lu, S., and Ostrovsky, R. How to garble ram programs. In Johansson and Nguyen [14], pp. 719–734.
19. Mohassel, P., and Sadeghian, S. S. How to hide circuits in MPC an efficient framework for private function evaluation. In Johansson and Nguyen [14], pp. 557–574.
20. Ostrovsky, R., and Shoup, V. Private information storage. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (1997), ACM, pp. 294–303.
21. Sadeghi, A.-R., Gligor, V. D., and Yung, M., Eds. *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (2013), ACM.
22. Shi, E., Chan, T.-H. H., Stefanov, E., and Li, M. Oblivious RAM with o ((logn) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011*. Springer, 2011, pp. 197–214.
23. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C. W., Ren, L., Yu, X., and Devadas, S. Path ORAM: an extremely simple oblivious RAM protocol. In Sadeghi et al. [21], pp. 299–310.
24. Toft, T. Secure datastructures based on multiparty computation. Cryptology ePrint Archive, Report 2011/081, 2011. `http://eprint.iacr.org/`.
25. Waksman, A. A permutation network. *Journal of the ACM (JACM) 15*, 1 (1968), 159–163.

A Protocols

We start this section with more protocols from previous work.

- $([a_0], \ldots, [a_{\ell-1}]) \leftarrow \mathsf{BitDec}([a])$ computes the bit decomposition of $a \in \mathsf{GF}(2^n)$, that is a list of bits $a_0, \ldots, a_{\ell-1}$ such that $a = \sum_{i=0}^{\ell-1} a_i \cdot X^i$. It is straightforward to implement using ℓ authenticated random bits.
- $[a^{-1}] \leftarrow \mathsf{Inv}([a])$ computes the inverse of a. It is straightforward to implement given $([x], [x^{-1}])$ for a random x, which can be derived from a random triple $([a], [b], [ab])$.
- $[a_0] \leftarrow \mathsf{Mod2}([a], \ell)$ computes a modulo 2 for an ℓ-bit number a. Catrina and de Hoogh [4] give an implementation with constant complexity that requires $O(\ell)$ secret random bits for statistical security.
- $[c] \leftarrow [a] < [b]$ computes a bit indicating whether a is less than b. Catrina and de Hoogh [4] provide two implementations with complexity linear in the number of bits of a and b, one of which has a constant number of rounds.

- $\mathsf{RandomBit}()$ returns a secret random bit. Damgård et al. [7] show how secret random bits can be generated in the offline phase of SPDZ.
- $\mathsf{ClearRandomBit}()$ returns a clear random bit. It is not necessary to use secret random bits for this; it suffices to use a pseudorandom number generator seeded by a securely generated random value.
- $x \leftarrow \mathsf{Reveal}([x])$ reveals a secret number to all players. This is inherent to all MPC schemes.
- $\mathsf{ExtractBit}([x], i, \ell)$ extracts the i-th least significant bit of x. It can be implemented using Mod2m by Catrina et al. [4] for prime order fields as shown in Protocol 3. We call it $\mathsf{Mod2m}^*$ here to avoid confusion with the version of Mod2m where the exponent is secret-shared. For $\mathsf{GF}(2^n)$, it is most efficient to use BitDec.

Protocol 1 $([d_0], [d_1]) \leftarrow \mathsf{CondSwap}([c], [a_0], [a_1])$

Input: Secret field elements $[c], [a_0], [a_1]$ with c being a bit
Output: $([a_0], [a_1])$ if $c = 0$, $([a_1], [a_0])$ if $c = 1$
1: $[b] \leftarrow [d] \cdot ([a_0] - [a_1])$
2: **return** $([a_0] - [b], [a_1] + [b])$

Protocol 2 $([d_0], \ldots, [d_{2^l-1}]) \leftarrow \mathsf{Demux}([a_0], \ldots, [a_{l-1}])$

Input: Bit decomposition of $[a]$
Output: Bit vector $([d_0], \ldots, [d_{2^l-1}])$ such that $d_a = 1$ and $d_i = 0$ for all $i \neq a$
1: **if** $l = 1$ **then**
2:     **return** $(1 - [a_0], [a_0])$
3: $([b_0], \ldots, [b_{2^{\lfloor l/2 \rfloor}-1}] \leftarrow \mathsf{Demux}([a_0], \ldots, [a_{\lfloor l/2 \rfloor - 1}])$
4: $([c_0], \ldots, [c_{2^{l-\lfloor l/2 \rfloor}-1}] \leftarrow \mathsf{Demux}([a_{\lfloor l/2 \rfloor}], \ldots, [a_{l-1}])$
5: **for** $i = 0$ to $2^l - 1$ **do**
6:     $[d_i] = [b_{i \bmod 2^{\lfloor l/2 \rfloor}}] \cdot [c_{i/2^{\lfloor l/2 \rfloor}}]$
7: **return** $([d_0], \ldots, [d_{2^l-1}])$

Protocol 3 $[b] \leftarrow \mathsf{ExtractBit}([x], i, \ell)$

Input: Secret-shared number $[x]$, index i, ℓ bit length of x
Output: Secret-shared bit $[b]$ such that b is the i-th least significant bit of x
1: $[f], [2^{i-1}] \leftarrow \mathsf{Mod2m}^*([x], i - 1, \ell)$
2: $[g] \leftarrow ([x] - [f]) \cdot \mathsf{Inv}([2^{i-1}])$ ▷ x shifted by $i - 1$ bits
3: **return** $\mathsf{Mod2}([g])$

A.1 Oblivious Data Structures with Linear Overhead

Protocol 4 assumes that every index u_i is unique. Therefore, at most one of the found indicators f_i can be one, which implies that the returned tuple is either $([u], [v_i], [0])$ if there exists $f_j = 1$ and $([u], [0], [1])$ otherwise. Furthermore, the update of $[e_i]$ is simpler because $f_i = 1$ implies that $e_i = 0$. If the index is found, the corresponding entry is replaced by $(0, 0, 1)$.

Protocol 4 $([u],[v],[e]) \leftarrow \mathsf{ReadAndRemove}([u],d,l)$

Input: $[u]$ secret-shared index, d default return value, l maximal bit length of indices
Output: ORAM entry $([u],[v],[0])$ if present, $([0],[0],[1])$ otherwise
Require: $([u_i],[v_i],[e_i])_{i=0}^{n-1}$ such that $u_i \neq u_j$ for every i,j with $e_i = e_j = 1$ and $u_i = v_i = 0$ for every i with $e_i = 0$
Ensure: $([u_i],[v_i],[e_i]) = ([0],[0],[1])$ if $u_i = u$ beforehand and unchanged otherwise

1: **for** $i = 0$ to $N-1$ **do**
2: $\quad [f_i] \leftarrow \mathsf{EQZ}([u]-[u_i],l)\cdot(1-[e_i])$ ▷ Determining whether u found at non-empty position i
3: $\quad [g_i] \leftarrow [f_i]\cdot[v_i]$
4: $\quad [u_i] \leftarrow [u_i]\cdot(1-[f_i])$ ▷ Update entry
5: $\quad [v_i] \leftarrow [v_i]-[g_i]$
6: $\quad [e_i] \leftarrow [e_i]+[f_i]$
7: **return** $([u], \sum_{i=0}^{N-1}[g_i], 1-\sum_{i=0}^{N-1}[f_i])$

Protocol 5 $\mathsf{Add}([u],[v],[e])$

Input: Secret-shared index, value, and empty flag $([u],[v],[e])$
Require: $([u_i],[v_i],[e_i])_{i=0}^{n-1}$ such that $u_i \neq u_j$ for every i,j with $e_i = e_j = 1$ and $u_i = v_i = 0$ for every i with $e_i = 0$
Ensure: $([u_i],[v_i],[e_i]) = ([u],[v],[0])$ if i lowest number with $e_i = 0$ beforehand and unchanged otherwise

1: $([b_1],\dots,[b_N]) \leftarrow \mathsf{PreOR}([e_0],\dots,[e_{N-1}])$
2: $b_0 \leftarrow 0$
3: **for** $i = 0$ to $N-1$ **do**
4: $\quad [c_i] \leftarrow [b_{i+1}]-[b_i]$ ▷ Determine whether i is the first empty position
5: $\quad [u_i] \leftarrow \mathsf{IfElse}([c_i],[u],[u_i])$ ▷ Update entry
6: $\quad [v_i] \leftarrow \mathsf{IfElse}([c_i],[v],[v_i])$
7: $\quad [e_i] \leftarrow \mathsf{IfElse}([c_i],[e],[e_i])$

Protocol 6 $([u],[v],[e]) \leftarrow \mathsf{Pop}()$

Output: The first non-empty entry if existing and $([0],[0],[1])$ otherwise
Require: $([u_i],[v_i],[e_i])_{i=0}^{n-1}$ such that $u_i \neq u_j$ for every i,j with $e_i = e_j = 1$ and $u_i = v_i = 0$ for every i with $e_i = 0$
Ensure: $([u_i],[v_i],[e_i]) = ([0],[0],[1])$ if i lowest number with $e_i = 1$ beforehand and unchanged otherwise

1: $([b_1],\dots,[b_N]) \leftarrow \mathsf{PreOR}(1-[e_0],\dots,1-[e_{N-1}])$
2: $b_0 \leftarrow 0$
3: **for** $i = 0$ to $N-1$ **do**
4: $\quad [c_i] \leftarrow [b_{i+1}]-[b_i]$ ▷ Determine whether i is the first non-empty position
5: $\quad [f_i] \leftarrow [c_i]\cdot[u_i]$
6: $\quad [g_i] \leftarrow [c_i]\cdot[v_i]$
7: $\quad [u_i] \leftarrow [u_i]-[f_i]$ ▷ Update entry; equivalent to $[u_i] \leftarrow \mathsf{IfElse}([c_i],[0],[u_i])$
8: $\quad [v_i] \leftarrow [v_i]-[g_i]$ ▷ Equivalent to $[v_i] \leftarrow \mathsf{IfElse}([c_i],[0],[v_i])$
9: $\quad [e_i] \leftarrow [e_i]+[c_i]$
10: **return** $(\sum_{i=0}^{N-1}[f_i], \sum_{i=0}^{N-1}[g_i], 1-\sum_{i=0}^{N-1}[c_i])$

In Protocol 5, since the output of PreOR is monotonically increasing, at most one c_j is one, namely for j corresponding to the first empty entry in the dictionary. Given that, $([u], [v], [e])$ is added as $([u_j], [v_j], [e_j])$ while the other entries remain.

In Protocol 6, similarly to Protocol 5, c_i equals one if the i-th entry is the first non-empty one and zero otherwise. Note that $c_i = 1$ implies that $e_i = 0$, which makes updating e_i cheaper than updating u_i or v_i.

Protocol 7 $([v], [e]) \leftarrow \mathsf{Access}([u], [v'], [e'], [w])$

Input: $[u]$ secret-shared index, $[v']$ secret-shared replacement value, $[e']$ secret-shared replacement empty flag, $[w]$ secret-shared writing flag

Output: $[v]$ secret-shared value, $[e]$ secret-shared empty flag

Require: $([v_i], [f_i])_{i=0}^{n-1}$ such that $v_i = 0$ for every i with $e_i = 0$

Ensure: $([v_i], [e_i]) = ([v'], [e'])$ if $u = i$ and $w = 1$, unchanged otherwise

1: $[b_0], \ldots, [b_{N-1}] \leftarrow \mathsf{Demux}(\mathsf{BitDec}(u, \lceil \log N \rceil))$ ▷ Computing the index vector
2: $[v] \leftarrow \sum_{i=0}^{N-1} [b_i] \cdot [v_i]$ ▷ Reading
3: $[e] \leftarrow \sum_{i=0}^{N-1} [b_i] \cdot [e_i]$
4: **for** $i = 0$ to $N - 1$ **do** ▷ Writing
5: $\quad [w_i] \leftarrow [w] \cdot [b_i]$
6: $\quad [v_i] \leftarrow \mathsf{IfElse}([w_i], [v'], [v_i])$
7: $\quad [e_i] \leftarrow \mathsf{IfElse}([w_i], [e'], [e_i])$
8: **return** $[v], [e]$

A.2 Oblivious Dictionary in $O(N)$

Protocol 8 describes the $O(N)$ oblivious dictionary described in Section 3.3. Since the values in the multiplication tree M can blow up significantly and wrap around in the field, it is necessary that the comparison algorithm EQZ works on arbitrary secret shared field elements. In $\mathsf{GF}(2^k)$ this is straightforward using the standard protocol based on bit decomposition. In $\mathsf{GF}(p)$, however, we can no longer use the efficient protocol of Catrina and de Hoogh [4], since this requires κ extra bits of room in the field element to achieve statistical security $2^{-\kappa}$. We use the unconditionally secure protocol of Damgård et al. [6] to do this to do this in a constant number of rounds.

Complexity: Assume a secure comparison takes C rounds. Building the initial multiplication tree can be done in $\log_2 N$ rounds. Then there are $C \cdot (\log_2 N + 1)$ rounds for the comparisons, and $2 \log_2 N$ rounds for the inner multiplication loops, plus one final round to compute the result, giving a total of $(C+3)\log_2 N + C + 1 \in O(\log N)$ rounds if a constant-round comparison protocol is used. The total communication cost is $O(N + \max(L, k) \cdot \log N)$ when operating on L-bit numbers in a k-bit field, which is dominated by $O(N)$ for most applications.

Protocol 8 $\mathsf{Find}([A], [x])$

Input: $A = (v_0, \ldots, v_{n-1})$: array of length $n = 2^L$. x: to be searched for in A.
Output: Bits $(B_0, \ldots, B_{n-1}), b$ such that $B_i = 1$ if $A_{B_i} = x$, and $b = 0$ if x is not in A.
Require: $\mathsf{EQZ}([x])$: comparison of an *arbitrary* secret shared field element x with zero.

```
 1: for j = n − 1 to 2n − 1 do
 2:     [M_j] ← [t_j] − [x]                         ▷ Match will equal 0
 3: for j = n − 2 to 0 do
 4:     [M_j] ← [M_{2j+1}] · [M_{2j+2}]              ▷ Build multiplication tree M
 5: [B_0] ← EQZ(M_0)
 6: for j = 1 to L do
 7:     [m] ← 0
 8:     for k = 0 to 2^j − 1 do
 9:         t ← k + 2^j − 1
10:         if k mod 2 = 0 then
11:             [m] ← [m] + [B_{(t−1)/2}] · [M_t]
12:     [b] ← 1 − EQZ([m])
13:     for k = 0 to 2^j − 1 do
14:         t ← k + 2^j − 1
15:         if k mod 2 = 0 then
16:             [B_t] ← [B_{(t−1)/2}] · (1 − [b])
17:         else
18:             [B_t] ← [B_{(t−1)/2}] · [b]
19: return ([B_{n−1}, . . . , [B_{2n−2}]), [B_0]
```

A.3 Oblivious Array with Polylogarithmic Overhead

Note that $\bar{m}$ denotes the canonical integer representation of $m \in \mathsf{GF}(2^n)$ in Protocol 9. Since the bit representation of $X^{\bar{m}}$ is 1 for the $\bar{m}$-th least significant bit and 0 otherwise, $(c_0, \ldots, c_{m-1})$ are 0, and $(c_m, \ldots, c_{l-1})$ are 1. It is then easy to see that the output is as claimed. Furthermore, the complexity of the protocol is $2l+3$ invocations in $\lceil \log l \rceil + 3$ rounds because BitDec requires only one invocation. The logarithmic summand in the round complexity stems from the product in line 3. Adapting the constant-round unbounded fan-in multiplication protocol by Damgård et al. [6], it can be reduced to one round.

Both the protocol for prime-order fields and fields of characteristic two also compute $[2^m]$ and $[X^{\bar{m}}]$, respectively. Computing the inverse of those allows to compute the truncation of $[a]$ by $[m]$ bits. This allows in addition to replace a position by a new one without changing or revealing the other positions stored in

Protocol 9 $([f], [X^{\bar{m}}]) \leftarrow \mathsf{Mod2m}([a], [m], l)$

Input: $[a]$ and $[m]$ secret elements of $\mathsf{GF}(2^n)$, a of bit length l, m of bit length at most $\lceil l \rceil$
Output: $\mathsf{GF}(2^n)$ secret element $[f]$ where the $\bar{m}$ least significant bits are the same as $[a]$, and all other bits are zero
1: $l' \leftarrow \lceil \log l \rceil$
2: $([m_0], \dots, [m_{l'-1}]) \leftarrow \mathsf{BitDec}([m], l')$
3: $[X^{\bar{m}}] \leftarrow \prod_{i=0}^{l'-1}([m_i] \cdot X^{2^i} + (1 - [m_i]))$
4: $([b_0], \dots, [b_{l'}]) \leftarrow \mathsf{BitDec}([X^{\bar{m}}], l)$
5: $[c_0] \leftarrow [b_0]$
6: **for** $i = 1$ to l **do**
7: $\quad [c_i] \leftarrow [c_{i-1}] + [b_i]$
8: $([a_0], \dots, [a_{l-1}]) \leftarrow \mathsf{BitDec}([a], l)$
9: **return** $(\sum_{i=0}^{l-1}[a_i] \cdot (1 - [c_i]) \cdot X^i, [X^{\bar{m}}])$

the same field element. Protocol 10 demonstrates this for the case of one bit, whereupon the generalization to any number of bits is straightforward. It also work for fields of prime order by using 2 instead of X. The correctness of the protocol follows from the fact that d equals a truncated by $\bar{m}$ bits, and that e equals a truncated by $\bar{m} + 1$ bits.

Protocol 10 $([f], [g]) \leftarrow \mathsf{Replace}([a], [m], [b], l)$

Input: $[a]$, $[m]$, and $[b]$ secret elements of $\mathsf{GF}(2^n)$, a of bit length l, m of bit length at most $\lceil l \rceil$, b a bit
Output: Secret $\mathsf{GF}(2^n)$ elements $[f]$ and $[g]$ such that f equals a with the $\bar{m}$-th least significant replaced by b, and g is is the $\bar{m}$-th least significant bits of a
1: $([c], [X^{\bar{m}}]) \leftarrow \mathsf{Mod2m}([a], [m], l))$
2: $[d] \leftarrow ([a] - [c]) \cdot \mathsf{Inv}([X^{\bar{m}}]$
3: $([d_0], \dots, [d_{l-1}]) \leftarrow \mathsf{BitDec}([d]), l)$
4: $[e] \leftarrow ([c] - [d_0]) \cdot X^{-1}$
5: **return** $(([e] \cdot X + [b]) \cdot [X^{\bar{m}}] + [c], [d_0])$

Protocol 11 $([v], [e]) \leftarrow \mathsf{ReadAndRemove}([u])$

Input: $[u]$ secret-shared index
Output: $[v]$ secret-shared value, $[e]$ secret-shared empty flag
Require: Map position map, D depth of tree, tree of buckets
Ensure: $[l^*]$ secret-shared random path to be used in following Add, index u removed
1: $[l^*] \leftarrow \sum_{i=0}^{D-1} \mathsf{RandomBit}() \cdot 2^i$ ▷ D random secret-shared bits; X^i for $\mathsf{GF}(2^n)$
2: $[l] \leftarrow \mathsf{Map.Access}([u], [l^*], [1])$
3: $l \leftarrow \mathsf{Reveal}([l])$
4: **for** $i = 0$ to D **do**
5: $\quad \mathsf{B} \leftarrow \mathsf{GetBucket}(l, i)$ ▷ Bucket on level i and path to leaf l
6: $\quad ([v_i], [l_i]), [e_i] \leftarrow \mathsf{B.ReadAndRemove}([u])$
7: **return** $\sum_{i=0}^{D}[v_i] \cdot (1 - [e_i]), \sum_{i=0}^{D}(1 - [e_i])$

Protocol 12 $\mathsf{Add}([u],[v],[e])$

Input: Secret-shared index, value, and empty flag $([u],[v],[e])$
Require: Tree of buckets, $[l^*]$ secret-shared random path generated by $\mathsf{ReadAndRemove}$
Ensure: Entry (u,v,e) added

1: $\mathsf{B} \leftarrow \mathsf{GetBucket}(\lambda, 0)$ ▷ Root bucket
2: $\mathsf{B.Add}([u],([v],[l^*]),[e])$
3: $\mathsf{Evict}()$

Protocol 13 $\mathsf{EvictBucket}(\mathsf{B}, i)$

Input: Bucket B, level i
Require: Tree depth D
Ensure: Move an entry (if existing) of B on level i to the correct child

1: $[u],([v],[l]),[e] \leftarrow \mathsf{B.Pop}()$
2: $[b] \leftarrow \mathsf{ExtractBit}([l], i, D)$
3: $[u_0],[u_1] \leftarrow \mathsf{CondSwap}([b],[u],[0])$
4: $[v_0],[v_1] \leftarrow \mathsf{CondSwap}([b],[v],[0])$
5: $[l_0],[l_1] \leftarrow \mathsf{CondSwap}([b],[l],[0])$
6: $[e_0],[e_1] \leftarrow \mathsf{CondSwap}([b],[e],[1])$
7: $\mathsf{GetChild}(\mathsf{B},0).\mathsf{Add}([u_0],([v_0],[l_0]),[e_0])$ ▷ Left child of B
8: $\mathsf{GetChild}(\mathsf{B},1).\mathsf{Add}([u_1],([v_1],[l_1]),[e_1])$ ▷ Right child of B

Protocol 14 $\mathsf{Evict}()$

Require: D depth of tree, tree of buckets
Ensure: Run eviction

1: $\mathsf{EvictBucket}(\mathsf{GetBucket}(\lambda, 0))$ ▷ Root bucket
2: **for** $d = 1$ to $D-1$ **do**
3: **for** $i = 1$ to 2 **do**
4: $\mathsf{EvictBucket}(\mathsf{GetBucket}(\sum_{j=0}^{d} \mathsf{ClearRandomBit}() \cdot 2^j, d))$ ▷ X^j for $\mathsf{GF}(2^n)$

A.4 Path ORAM

Protocol 15 computes the *least common ancestor* of two leaf nodes, one of which is secret shared. It is fairly straightforward after bit decomposition, simply identifying the first position where the bits differ.

Protocol 15 $([\ell], [r_0], \ldots, [r_{L-1}]) \leftarrow \mathsf{LCA}(a, [b])$

Output: Bits $[r_i]$ such that $r_i = 1$ iff $i = \mathsf{LCA}(a, b)$.

1: $[a]_B \leftarrow \mathsf{BitDec}([a], L)$
2: $[b]_B \leftarrow \mathsf{BitDec}([b], L)$
3: $[c] \leftarrow 1, [\mathsf{lev}] \leftarrow 0$
4: **for** $i \in \{0, \ldots, L-1\}$ **do**
5: $\quad [t] \leftarrow 1 - \mathsf{XOR}(a_i, [b_i])$
6: $\quad [r_i] \leftarrow [c] - [c] \cdot [t]$
7: $\quad [c] \leftarrow [c] \cdot [t]$
return $[r_0], \cdots, [r_{L-1}]$

Given the bits $[r_0], \cdots, [r_{L-1}]$ indicating which level on the eviction path an entry wants to be placed, we must calculate the actual position the entry will end up, depending on the current bucket allocations. Protocol 16 does this for a single entry. The result takes the form of a bit for each possible entry on the path or in the stash, one of which will be set for the given entry to be evicted. For each bucket on the path, we maintain bits $[u_1], \cdots, [u_Z]$, where $u_i = 0$ if the i-th entry in the bucket has already been assigned to. Since buckets are always filled up from the bottom, we can easily use these to compute bits $s_1 = 1 + u_1$, $s_i = u_{i-1} + u_i$ for $i > 1$, so that $s_i = 1$ iff the bucket contains i entries.

Now starting at the deepest level in the path, we must compute a bit $[e]$ indicating whether we *want* to place the entry at this level, and then a bit $[b]$ determining whether we *can* place it here (depending on the bucket's size). Initially set $e, b = 0$, then at level j update them by $e \leftarrow (1-b) \cdot ((1-e) \cdot r_j + e) \cdot (1-\delta)$, where δ is the empty entry flag, and $b \leftarrow e \cdot (1 - u_Z)$. The $((1-e) \cdot r_j + e)$ factor in e ensures that if we wanted to add the entry on the previous level but failed (i.e. $e = 1, b = 0$) then we will try again for this level by setting $e = 1$. Then we multiply each $[s_i]$ by $[b]$ to obtain a bit for each entry of the bucket that determines whether the current evicted entry will be placed there, and update each $[u_i]$ by XORing with $[s_i] \cdot [b]$. Note that the s_i bits can be packed into a single $\mathsf{GF}(2^k)$ element so that only one multiplication by b is needed for each level. When the bits are needed later a bit decomposition can be applied to extract them. Finally a bit is computed determining whether the entry needs to be added to the stash or not, and a set of bits for the stash entries is obtained similarly.

Placing the evicted entries. For every entry to be evicted, the previous algorithm computed a secret shared bit for each position in the path and in the stash, one of which is set to 1, indicating the final position of the entry. Now to complete the eviction process, we need to place each entry in its appropriate position.

Observing that the eviction procedure in Path ORAM is a permutation on all of the elements in the stash and the path, we show how to use permutation networks and sorting networks to reduce the quadratic overhead of the naive eviction method to $O(n \log n)$.

Let P be the $n \times n$ matrix combining all of the $\mathsf{pos}, \mathsf{pos}'$ values from every evicted entry. So the i-th row of P is a bit vector containing a single 1 corresponding to the final position of entry i. We can think of P as a *partial* permutation matrix: for every non-empty entry, P sends the entry to its allocated position after eviction, but all of the empty entries are simply assigned to 0. To reduce the quadratic complexity of the naïve method, we first must convert P into a complete permutation matrix P' by assigning positions to all of the empty entries. Then we obliviously shuffle and reveal the permutation P', apply the resulting clear permutation to the entries and finally reverse the oblivious shuffle to end up with the desired result.

Adding up the rows of P gives us a length n bit vector, whose 0 values correspond to positions which don't have any evicted entries assigned to, so will be empty after eviction. Similarly, adding up the columns of P results in a bit vector whose 0 values correspond to entries which are empty before eviction, so are not

Protocol 16 Calculate the final position of an entry.

Input: Entry $E = (i, [\ell_i], [d_i])$ residing at level lev_i on the path to ℓ.
Output: $Z \times (L+1)$ matrix $[\mathsf{pos}]$, such that $\mathsf{pos}_{j,k} = 1$ iff E is to be placed in the k-th position of the bucket at level j and $s = 1$ if E needs to be placed in the stash.
Require: $[\delta]$, bit indicating if E is empty.

1: $[r_0], \ldots, [r_{L-1}] \leftarrow \mathsf{LCA}(\mathsf{lev}_i, [\ell_i])$
2: $[\mathsf{pos}], [a] \leftarrow 0$
3: $[e], [b] \leftarrow 0$
4: **for** $j = \min(L, \mathsf{lev}_i + \sigma)$ to $\max(0, \mathsf{lev}_i - \tau)$ **do**
5: $\quad [e] \leftarrow (1 - [b]) \cdot ((1 - e) \cdot [r_j] + [e]) \cdot (1 - [\delta])$
6: $\quad [b] \leftarrow [e] \cdot (1 - [u_{j,Z}])$
7: $\quad [s_1] \leftarrow 1 + [u_{j,1}]$
8: $\quad$ **for** $k = 2$ to Z **do**
9: $\quad\quad [s_k] \leftarrow [u_{j,k-1}] + [u_{j,k}]$
10: $\quad$ **for** $k = 1$ to Z **do**
11: $\quad\quad [\mathsf{pos}_{j,k}] \leftarrow [b] \cdot [s_k]$ $\quad$ ▷ parallelizable by packing $\mathsf{pos}_{j,k}$ into one field element
12: $\quad\quad [u_{j,k}] \leftarrow [u_{j,k}] + [\mathsf{pos}_{j,k}]$
13: $\quad [a] \leftarrow [a] + \sum_i [s_i]$ $\quad$ ▷ a set to 1 iff B is added to a bucket on the path
14: $[a'] \leftarrow (1 - [\delta]) - [a]$ $\quad$ ▷ 1 if needs to be added to stash
15: **return** $[\mathsf{pos}], [a']$

assigned to any position. We need to align the zeroes in these two vectors and assign an index from the latter vector to each 0 value of the former. Obliviously sorting both vectors will ensure the 0 values are aligned, then a simple linear scan across both sorted vectors can be used to update the permutation. At this stage we no longer need P' represented as a permutation matrix, but instead just calculate the index that each entry is sent to (for the bit values already computed, the index can be obtained by simply summing up each value accordingly).

Now we have the permutation P', represented as a length n vector of secret shared indices, and wish to apply this permutation to the n evicted entries. To do this we generate a secret, random permutation network, apply this to P' and reveal the result. The entries are permuted by this clear permutation and then fed through the random permutation network backwards. For this stage it is important that P' is indeed a complete permutation: if P was used instead then the result here would have repeated indices, revealing how many empty entries there were during eviction, which would not be secure.

Simulations. We measured the stash size for 100 million accesses after 1 million 'warm-up' writes and calculated the overall probability of each stash size occurring for various choices of the parameters τ, σ. The results shown in Figure 4 are for $N = 2^{16}$, but we found the overflow probabilities did not get any worse with larger sizes. Extrapolating these estimates with linear regression, we chose $\tau = 3, \sigma = 5$ with a stash size of 48 for a 40-bit statistical security level.

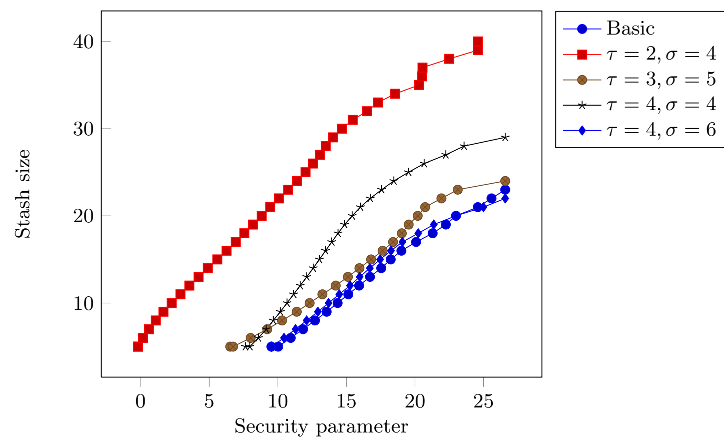

Fig. 4. Estimated security parameter for various stash sizes in Path ORAM. 100 million accesses to an ORAM of size 2^{16}.

A.5 Applications

Protocol 17 BubbleUp($[u]$)

Input: Secret-shared index $[u]$
Require: Heap and Index store a heap with depth up to l
Ensure: Bubble up along a path containing the index u

1: $[b_0], \dots, [b_{l-1}] \leftarrow \mathsf{BitDec}([u] + 1, l)$
2: $[c_1], \dots, [c_l] \leftarrow \mathsf{PreOR}([b_{l-1}], \dots, [b_0])$
3: $[t] \leftarrow [c_1] \sum_{i=1}^{l-1}([c_{i+1}] - [c_i]) \cdot 2^i$ ▷ Levels from u to maximal depth
4: $[j_c] \leftarrow ([u] + 1) \cdot [t] - 1$ ▷ Index of a leaf of the subtree rooted at u
5: **for** $i = 0$ to $l - 1$ **do**
6: $[j_p] \leftarrow ([j_c] - 1) \gg 1$ ▷ Compute parent position; shifting by computing modulo 2
7: $([p_p], [v_p]), [e_p] \leftarrow \mathsf{Heap.Read}([j_p])$ ▷ Read parent
8: $([p_c], [v_c]), [e_c] \leftarrow \mathsf{Heap.Read}([j_c])$ ▷ Read child
9: $[x] \leftarrow (1 - [e_p]) \cdot (1 - [e_c]) \cdot ([p_p] > [p_c])$ ▷ Determining whether to swap parent and child
10: $[p_p], [p_c] \leftarrow \mathsf{CondSwap}([x], [p_p], [p_c])$
11: $[v_p], [v_c] \leftarrow \mathsf{CondSwap}([x], [v_p], [v_c])$
12: $[e_p], [e_c] \leftarrow \mathsf{CondSwap}([x], [e_p], [e_c])$
13: $\mathsf{Heap.Write}([j_p], ([p_p], [v_p]), [e_p])$
14: $\mathsf{Heap.Write}([j_c], ([p_c], [v_c]), [e_c])$
15: $\mathsf{Index.Access}([v_p], [j_p], [x])$ ▷ Only update position if swapping
16: $\mathsf{Index.Access}([v_c], [j_c], [x])$
17: $[j_c] \leftarrow [j_p]$ ▷ One level up for next loop body

Protocol 18 Update($[v], [p'], [w]$)

Input: $[v]$ secret-shared value, $[p']$ secret-shared new priority (at most existing priority), $[w]$ secret-shared writing flag
Require: Heap and Index store a heap of sufficient size
Ensure: Heap contains v with priority p' if $w = 1$ and remains unchanged otherwise

1: $[i], [e] \leftarrow \mathsf{Index.Read}([v])$
2: $[i] \leftarrow \mathsf{IfElse}([e], [i], [s])$ ▷ Use existing position or first available
3: $\mathsf{Index.Access}([v], [s], [e] \cdot [w])$ ▷ Update position if necessary
4: $[s] \leftarrow [s] + [e] \cdot [w]$ ▷ Increase size if necessary
5: $\mathsf{Heap.Access}([i], ([p'], [v]), 0, [w])$ ▷ Update heap
6: $\mathsf{BubbleUp}([i])$ ▷ Restore heap condition

Protocol 19 Dist $\leftarrow$ Dijkstra$([s]$, Edges, Index, $N_V, N_E)$

Input: $[s]$ secret-shared source vertex, Edges and Index as in text, N_V number of vertices, N_E number of edges
Output: Dist containing the distance from the source vertex to every vertex

```
 1: Create Dist with size N_V and default 0
 2: Create priority queue Q with size N_V
 3: Dist.Write([s], 0)
 4: Q.Update([s], 0, 1)
 5: [z] ← 1
 6: [u] ← 0
 7: for i = 0 to 2N_E do
 8:     [u] ← IfElse([z], Q.Pop([z]), [u])          ▷ New vertex if previously at last neighbour
 9:     [j] ← IfElse([z], Index.Read([u]), [j])     ▷ New edge index if previously at last neighbour
10:     [v], [w], [z] ← Edges.Read([j])             ▷ Read neighbour, weight and last-neighbour bit
11:     [j] ← [j] + 1                               ▷ Move on in edge list
12:     [a] ← Dist.Read([u]) + [w]                  ▷ Distance from source to v via u
13:     [d], [f] ← Dist.Read([v])                   ▷ Previously encountered distance to v if at all
14:     [x] ← ([a] < [d]) + [f]                     ▷ Determine whether shorter via u, assuming d = 0 if v not in Dist
15:     Dist.Access([v], [a], [x])                  ▷ Update Dist if shorter
16:     Q.Update([v], [a], [x])                     ▷ Add v to Q or decrease key to a if shorter
17: return Dist
```

A.6 Batch initialization of SCSL ORAM in MPC

At first consider what happens if we assign each entry a secret leaf, chosen uniformly and independently at random, and then obliviously shuffle the entries, reveal their leaves and place them in the appropriate leaf bucket. Although correct, this reveals how many entries were assigned to a particular bucket, which could leak information during future accesses. Our technique for hiding this is similar to the eviction procedure in Path ORAM. We need to compute the number of entries assigned to each leaf bucket, and then create a dummy empty entry for each empty position in the buckets. The real and dummy entries are then combined and shuffled, so it is safe to reveal the leaves.

Recall that with our parameter choices in the SCSL ORAM we have N/k leaf buckets, each of size $Z = (1+\delta)k$ for some small constant δ. Suppose we wish to initialize the ORAM with N entries $e_0, \ldots, e_{N-1}$, to be given the indices $0, \ldots, N-1$. The steps for this are as follows:

- Assign each entry e_i a leaf ℓ_i, chosen uniformly at random.
- Sort the N leaves.
- Set $s \leftarrow 0$, $B_0 \leftarrow (0, \ell_0)$, $b_0 \leftarrow 1$.
- For $i = 1$ up to $N-1$, do:
  - If $\ell_i = \ell_{i-1}$ then $s \leftarrow s+1, b_{i-1} \leftarrow 0$
  - Else, $s \leftarrow 0$, $b_{i-1} \leftarrow 1$
  - If $s > 2k$ then abort.
  - Set $B_i = (s, \ell_i)$

The bits b_i indicate the *last* leaf assigned to that bucket, while s_i describes the position of an entry within its leaf bucket. Now we must use these to calculate bits $e_0, \ldots, e_{Z \cdot \lceil N/k \rceil}$ indicating the empty positions in all the leaf buckets.

- Shuffle the list of (ℓ_i, s_i, b_i) tuples.
- Reveal the shuffled b_i bits, and place the corresponding (ℓ_i, s_i) pairs into a new list. s_i now tells us the number of entries assigned to leaf ℓ_i.
- Express each s_i as a unary vector (Protocol B2U by Aliasgari et al. [1]), giving the bits e_j indicating position emptiness.

Next we must sort the pairs (e_i, ℓ_i) by the e_i bits, create empty entries for the leaves where $e_i = 1$, and match up the real entries to be added with the rest of the leaves. Finally all of these $Z \cdot \lceil N/k \rceil = O(N)$ entries are shuffled and the leaves revealed so they can be placed accordingly.

Generating a pre-sorted list of secret values. Instead of generating secret random leaves and then sorting them, which takes $O(N \log^2 N)$ time for $\log N$-bit numbers, it is possible to sample uniformly random numbers in sorted order by taking partial sums of exponentially distributed random numbers [3]. Sampling from the exponential distribution requires computing fixed or floating point logarithms, which can be done in MPC with complexity $O(\log N)$ [1], so this method generates a sorted list in time $O(N \log N)$.

Complexity. Using the above method for generating the sorted random leaves, the complexity of filling up an individual ORAM is $O(N \log N)$. The complexity for SCSL ORAM with recursion is therefore $\sum_{i=1}^{\log N} O(2^i \log 2^i) = O(N \log N)$, so a multiplicative factor of $O(\log^3 N)$ is saved over the naive method.

Application to ordinary ORAM. For ordinary ORAM (not in MPC), the above description can be simplified somewhat, since determining which positions in the leaf buckets have not been assigned can be done with a simple linear scan. Assuming a computation model with $\log N$-bit word size, generating sorted numbers and shuffling become linear operations, so the overall complexity is $O(N)$ (and the same with recursion). This is optimal in terms of communication cost, since clearly every entry must be sent to the server.

B ORAM Priority Queue

Inspired by the work of Gentry et al., which showed how to modify ORAM to perform a secure binary search in just one ORAM access, we now show how to modify ORAM to build an oblivious priority queue. Each priority queue operation essentially costs the same as a single ORAM access, saving a $O(\log N)$ factor on the oblivious priority queue in Section 5.1, although we are not able to support the decrease-key operation required for Dijkstra's algorithm. Asymptotically this has slightly worse overhead than Toft's secure priority queue, which has $O(\log^2 N)$ amortized overhead, the advantage of our approach being that the use of ORAM allows the cost to be non-amortized.

Recall that the standard recursive ORAM consists of $\ell + 1$ individual tree ORAMs, with the tree ORAM on level i of size 2^i. An entry at level i contains two leaf numbers for the next tree at level $i + 1$, with the actual data all stored on level ℓ. Gentry et al. modified this to allow the client to select the next leaf based on a tag instead of the index, enabling a secure binary search on the tags with just one ORAM access.

For an oblivious min-priority queue, the tree ORAM on level i represents the level i nodes of a binary heap. We modify each entry by storing the data for the corresponding node in the binary heap, and also leaf numbers of the node's children (in the next ORAM level), indicating which child is smallest. Each entry in a tree at level $i < \ell$ has the form:

$$(L^*, j, d, p, (L_0, L_1))$$

where j identifies an address interval of size $2^{\ell+1-i}$, d is the data with priority p and L_0, L_1 are the level $i+1$ ORAM leaves corresponding to the left and right children of this node in the binary heap. If the size of a priority queue can be public, the number of ORAM levels can change dynamically as elements are added and removed for increased efficiency.

Recall the standard binary heap implementation of a min-priority queue: the heap is a complete binary tree where the root node always has the lowest priority, and each node has priority greater than or equal to its children, with the following basic operations:

- Insert: add a new node by first placing the node in the next empty position on the bottom level of the tree, and then carrying out a 'bubble-up' operation, moving the node up to the appropriate level based on its priority.
- GetMin: this returns the root node from the tree, then replaces it with the last node in the lowest level before carrying out a 'bubble-down' procedure to restore the heap property.

The key insight to implementing these operations securely is that the bubble-down procedure is very similar to the way of accessing a recursive ORAM, only using a different metric to decide the path to go down. By adding the data described above to each ORAM entry, we can use a recursive-style access for both the bubble-down and bubble-up procedures (since bubble-up can just be performed in reverse, starting at the root).

During the secure GetMin, it would seem that we have to do two separate recursive ORAM accesses: one to read the last node and another to bubble up. This can be avoided if we store a separate copy of the last node and update this during every other operation. Using this we can begin bubbling up straight away, saving a recursive ORAM access.

In detail, the procedures are as follows:

Insert: To insert an element $[x]$ with priority $[p]$, let j be the index in the binary heap corresponding to the next empty position in the bottom ORAM level. Let $j_0, \ldots, j_\ell$ be the bits of j, so at the i-th level in the binary heap, the child L_{j_i} takes us on the path to j. Find in the top ORAM level the list $(L^*, j_0, d_0, p_0, (L_0, L_1))$ and do the following for $i = 1$ up to $\ell - 1$:

1. If $[p] < [p_{i-1}]$ then swap $([x], [p])$ and $([d_{i-1}], [p_{i-1}])$.
2. Let L_i be the leaf in (L_0, L_1) corresponding to the interval containing j, and let L_i^* be a new random leaf.

3. Read the path to leaf L_i in the level $i+1$ ORAM and find an entry of the form $(L_i, j_i, d_i, p_i, (L_0, L_1))$.
4. Replace L_i by L_i^* in both entries on level i and $i+1$.

At the bottom level ℓ, the entry j will be empty, so we just need to set $([d_\ell], [p_\ell]) = ([x], [p])$ and finally write the values $([d_0], [p_0]), \ldots, ([d_\ell], [p_\ell])$ back to their respective ORAM levels (with their new leaf indices, L_i^*) and carry out the eviction procedure for each ORAM.

GetMin: Remove root, look up last node and then bubble-down as in insert. But during bubble down choose next ORAM entry based on priority, i.e. read both leaves L_0, L_1 to obtain two priorities $[p_0], [p_1]$, perform the secure comparison $[b] = [p_1] < [p_0]$ and (using $[b]$ as a selection bit) go to leaf $[L_b]$. This requires two ORAM accesses per level of the heap, so is roughly twice as slow as insertion.

C Figures

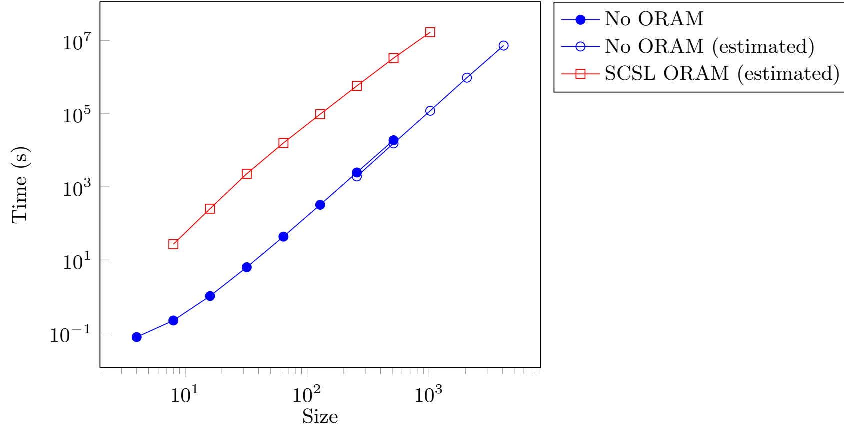

Fig. 5. Dijkstra's algorithm on a complete graph

C.1 Batch initialization

Figure 6 compares performance of batch initialization of a single SCSL ORAM in MPC as described in Section 4.4 with initialization using N iterations of the access protocol. We have not implemented the method for sampling N pre-sorted random numbers, so instead using Batcher's odd even sorting network for this, giving complexity $O(N \log^3 N)$ instead of the possible $O(N \log N)$. Even so, the figure shows a considerable performance advantage over the naive method for for the sizes we tested on, and we expect this to improve another 10-100x for larger sizes when these are implemented. For $N = 2^{14}$, the sorting network takes around 50% of the time, so up to another factor of two could be gained from eliminating this.

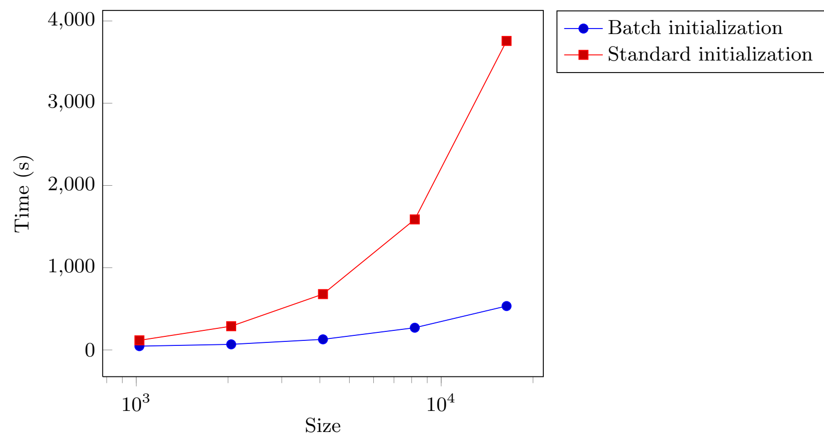

Fig. 6. Batch initialization of SCSL ORAM in MPC.