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 Stefanov 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 most 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 online access time for an oblivious array of size one million is under 100 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 [25], 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 [16]. 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 [25] and Gordon et al. [16] 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.

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. [27] (with an optimization of Gentry et al. [14]) and the Path ORAM scheme of Stefanov et al. [28], 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. [10], 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. Furthermore, Gordon et al. only achieve passive security, while our solution naturally provides active security when using an adequate MPC scheme such as SPDZ. Their approach of letting the server store the memory encrypted under a one-time pad for which the client generates the keys does not seem to extend to active security without losing efficiency.

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

Data structure	Based on	Access complexity	Section
Oblivious array	Demultiplexing [21]	$O(N)$	3.1
Oblivious dictionary	Trivial ORAM	$O(N \cdot \ell)$	3.2
	Trivial ORAM	$O(N + \ell \cdot \log N)$	3.3
Oblivious array	SCSL ORAM	$O(\log^4 N)$	4.2
	Path ORAM	$O(\log^3 N)$	4.3
Oblivious priority queue	Oblivious array	$O(\log^4 N)$	5.1
	Modified Path ORAM	$O(\log^3 N)$	Full ver. [19]

Fig. 1. Overview of our oblivious data structures. For the dictionary, ℓ is the maximal size of keys.

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. Note that the complexities may appear slightly higher than expected, due to the overhead of secure comparison in MPC. In a standard word model of computation, a $\log N$-bit comparison costs 1 operation, whereas in MPC this requires $O(\log N)$ operations, leading to the extra $O(\log N)$ factor seen in many of our data structures. 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, when choosing parameters for ORAM schemes we assume that the number of accesses to these 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 $O(|E| \log^4 |E| + |V| \log^3 |V|)$, when only the number of vertices and edges in the graph is public. The previous best known algorithm [2] for this takes time in $O(|V|^3)$. In the full version [19], 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), but we have not implemented this variant.

Secure stable matching With our oblivious array, it is straightforward to implement the preference matrix used in the Gale-Shapley algorithm for stable matching. The resulting protocol again has polylogarithmic overhead compared to the complexity of Gale-Shapley, which is $O(N^2)$ for N agents of both kinds. Previous work by Franklin et al. [12] achieved $O(N^4)$ for two-party computation using semi-homomorphic encryption. We also implemented the case of every agent only having a constant number of preferences. In this case, secure stable matching takes quasi-linear time in the worst case.

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.

Implementation. We implemented and benchmarked the oblivious array and Dijkstra's algorithm using various ORAM protocols. Our implementation uses the SPDZ protocol of Damgård et al. [9,11], 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. [20] to ensure that the minimal round complexity is achieved for each protocol.

1.2 Related Work

Other than the works already discussed [10,16,25], Gentry et al. [14] 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.

Gentry et al. [15] 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 [29]. 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 ORAM-based priority queue.

In a recent, independent work, Wang et al. [31] consider oblivious data structures in the classical ORAM model. Their techniques are very similar to our oblivious priority queue shown in the full version [19], but the general method does not directly translate to the MPC setting due to the use of branching and a client-side cache.

Brickell and Shmatikov [4] were the first to consider graph algorithms in an MPC setting. Their solution only works for two parties and achieves only passive security because they rely on local computation by the two parties. Furthermore, the result is always public whereas our protocols allow for further secure computation without publishing the result.

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. More recently, Blanton et al. [3] presented an oblivious algorithm for finding shortest paths in a graph with complexity in $O(|V|^2)$. The same complexity is achieved in another recent work by Liu et al. [22]. All solutions do not come close to $O(|E| + |V| \log |V|)$ for the implementation on a single machine except for the case of dense graphs. Our solution incurs only an overhead in $O(\log^4|E| + \log^3|V|)$ over the classical algorithm for arbitrary graphs, improving upon previous solutions when the graph is relatively sparse.

Gentry et al. [14] 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 the full version [19].

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.

- $b \leftarrow \mathsf{EQZ}([a], \ell)$ computes whether the ℓ-bit value a is zero. Catrina and de Hoogh [5] 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. [7] 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 [5] presented an implementation that requires $O(\ell)$ invocations in $O(\log \ell)$ rounds, while Damgård et al. [7] 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, CondSwap 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. Our implementation of Demux is due to Launchbury et al. [21]. 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. [18] 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. [27] (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) can be 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. [14] 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. By experimenting with probabilities calculated from the Chernoff bound, we found that the access complexity is actually minimized for bucket size $4k$, and choosing k between 12 and 24 (depending on the ORAM size) gives overflow probability at most 2^{-20} when the number of accesses is in $\tilde{O}(N)$.

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 Stefanov et al. [28]. 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. Note that we cannot use the ORAM scheme we are trying to implement for the client memory as this would introduce a circularity. Given the overhead for accessing the client memory, 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. [21], which we refer to as demultiplexing. 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 demultiplexing operation. This procedure has cost in $O(N)$.

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

Storing an empty flag in addition to the array value proves useful for some applications. Thus, we require that the players store a list of tuples $([v_i], [e_i])_{i=0}^{N-1}$ containing the values and empty flags.

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. [27] showed how to implement Pop using ReadAndRemove and Add, but for Trivial ORAM a dedicated implementation is more efficient.

In MPC, ReadAndRemove is the most expensive part because it contains a comparison for every entry. Essentially, it computes an index vector as in the previous section using comparison instead of demultiplexing. The complexity is dominated by the N calls to an equality test 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.

Our implementations of Add and Pop make use of the bits indicating whether an entry is empty. This way, we avoid comparing every index in finding the first non-empty or empty entry, respectively. Both protocols require $O(N)$ invocations in $O(\log N)$ or $O(1)$ rounds, depending on the exact implementation.

Theorem 1. *An algorithm using arrays but no branching other than conditional writes to arrays can be securely implemented in MPC with linear overhead.*

Proof (Sketch). Implement arrays as described above and use a circuit for conditional writes. Since there is no branching otherwise, this effectively results in a circuit that can be securely executed using the arithmetic blackbox provided by MPC schemes.

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 (see the full version [19]) 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 an equality test 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 an equality test 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. [7].

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 [5], 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, see the full version [19] for a similar protocol. 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. [7] 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 demultiplexing 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 latter returns a value indicating whether the searched index was found in a particular bucket.

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 a conditional swapping circuit. See the full version [19] for algorithmic descriptions of ReadAndRemove and Add.

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. [27] 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.

Theorem 2. *An algorithm using arrays but no branching other than conditional writes to arrays can be securely implemented in MPC with polylogarithmic overhead and negligible probability of an incorrect result.*

Proof (Sketch). Implement arrays as described above and use a circuit for conditional writes. Since there is no branching otherwise, this effectively results in a circuit that can be securely executed using the arithmetic blackbox provided by MPC schemes. The ORAM scheme accounts for the polylogarithmic overhead and the negligible failure probability.

For a complete simulation, we connect the ORAM and the MPC simulation in the following way: The ORAM simulation outputs the random paths used in ReadAndRemove. We input these values to the simulation of the arithmetic blackbox as they are revealed in the MPC protocol. Further random values are revealed by statistically secure algorithms. We sample those value according to the relevant distributions and input them to the MPC simulation as well. The indistinguishability of the resulting transcript follows using a hybrid argument with the two simulators.

4.3 Path ORAM

The Path ORAM access protocol is initially the same as in the Shi et al. and Gentry et al. schemes (with smaller bucket size), but differs in its eviction procedure. For eviction a leaf ℓ is chosen, either at random or in a deterministic bit-reversed order as in [14], 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.

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.
- For each level k in the path, compute a bit determining whether entry E goes to the bucket on level k or not, and bit indicating whether E ends up in the stash. To do this we maintain bits $u_{k,i}$ for $i = 0$ to $\lfloor \log_2 Z \rfloor$ that keep track of the size of bucket k, updating these as we go.

Now for each entry in the path, we have a list of bits, one of which is set to one to indicate the level we need to obliviously place the entry. At this point it might seem that we could just obliviously shuffle the entries and then reveal each level, but this is insecure. This is because in addition to the real entries in the path there are (an unknown number of) empty entries, which haven't been assigned a level. To be able to safely shuffle and reveal the real entries' levels, we first must assign a level to each empty entry, ensuring that every space in every bucket has had an entry assigned to it, so revealing the distribution of all levels gives no information.

To assign the empty entries with levels we use oblivious sorting: first the entries are sorted to separate the real and empty entries, and then the bucket positions are sorted to separate those that have already been used from the free ones. By aligning these two sorted lists, the free bucket levels can be assigned to the empty entries. Correctly aligning the two lists is actually slightly more involved than this due to the stash and some details have been omitted.

Parameters. In our implementation, we chose deterministic eviction with bucket size 2 and stash size 24. We estimate this gives an overflow probability of 2^{-75} for a single access, which should be comfortable for most applications. See the full version [19] for a complete description and extensive simulation results used to choose parameters.

Complexity. Computing the level that each entry ends up at requires $O(\log^2 N)$ multiplications. Our protocols for oblivious shuffling and bitwise sorting have complexity $O(n \log n)$ where here $n = |S| + Z \cdot \log N$, so the total complexity of the access protocol is $O(\log^2 N)$. Adding in the recursion gives an extra $O(\log N)$ factor, for an overall complexity of $O(\log^3 N)$.

For $\mathsf{ReadAndRemove}$, the complexity is in $O(\log^3 N)$ instead of $O(\log^4 N)$ (for SCSL ORAM) because of the constant bucket size. This results in an overall access complexity in $O(\log^3 N)$.

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 [30] 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 [17], and also in fully homomorphic encryption [13] and private function evaluation [23].

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:

1. Each player generates a random permutation, and locally computes the control bits for the associated Waksman network [2]
2. Each player inputs (i.e. secret shares) their control bits.
3. 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.
4. 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. In the full version of this paper [19], we describe a new method for initializing the SCSL ORAM with N entries 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. In practice our experiments indicate that this improves performance by at least 1-2 orders of magnitude, depending on the ORAM size.

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.

[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 $k2^{-m} = \frac{1}{n!}$.

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 1, the left and right child of the root at indices 2 and 3, 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. 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 for conditional writing. 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. We refer to the full version [19] for exact descriptions of the procedures.

For a priority queue of maximal size N, both bubbling up and down run over $\log N$ levels accessing oblivious arrays of size in $O(N)$. Using Path ORAM therefore results in a complexity in $O(\log^4 N)$ per access operation. In the full version [19], we show how to modify the Tree ORAM structure to reduce the overhead to $O(\log^3 N)$ per access, similarly to the binary search ORAM of Gentry et al. [14] and, very recently, the oblivious priority queue of Wang et al. [31]. Unlike the latter, this priority queue supports decrease key so can be used for Dijkstra's algorithm, but we have not currently implemented this variant.

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 [6], 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, conditional writing to an oblivious array plays a vital role.

In the following, we 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 the twice the number of edges for an undirected graph. In another array, we store for every vertex the index of its first neighbour in the first array. A third array is used to store the results, i.e., 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 a priority queue to store unprocessed vertices with the shortest encountered distance so far. See the full version [19] for 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 Path ORAM everywhere, this gives a complexity in $O(N_V \log^3 N_V + N_E(\log^4 N_V + \log^3 N_E))$. More concretely, for sparse graphs with $N_E \in O(N_V)$ (such as cycle graphs) the complexity is in $O(N_V \log^4 N_V)$, whereas for dense graphs with $N_E \in O(N_V^2)$ (such as complete graphs), the complexity is in $O(N_V^2 \log^4 N_V)$. 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 $O(\log^4 N_V + \log^3 N_E)$.

6 Experiments

We implemented our protocols for oblivious arrays using the system described by Keller et al. [20], and ran experiments on two directly connected machines running Linux on an eight-core i7 CPU at 3.1 GHz. The online 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. For these timings, we have also used the packing strategies described in Section 4.1 at the top level.

To estimate the offline time required to generate the necessary preprocessing data, we used the figures of Damgård et al. [8,9] for the finite fields $\mathsf{GF}(2^n)$ and $\mathsf{GF}(p)$, respectively. The offline time for accessing an oblivious array of size 2^{20} using Path ORAM would be 117 minutes in $\mathsf{GF}(2^n)$ and 32 minutes in $\mathsf{GF}(p)$, for active security with cheating probability 2^{-40}. Note that this could be easily improved using multiple cores, since the offline phase is trivially parallelizable, and also the $\mathsf{GF}(2^n)$ times could be cut further by using the offline phase from TinyOT [24].

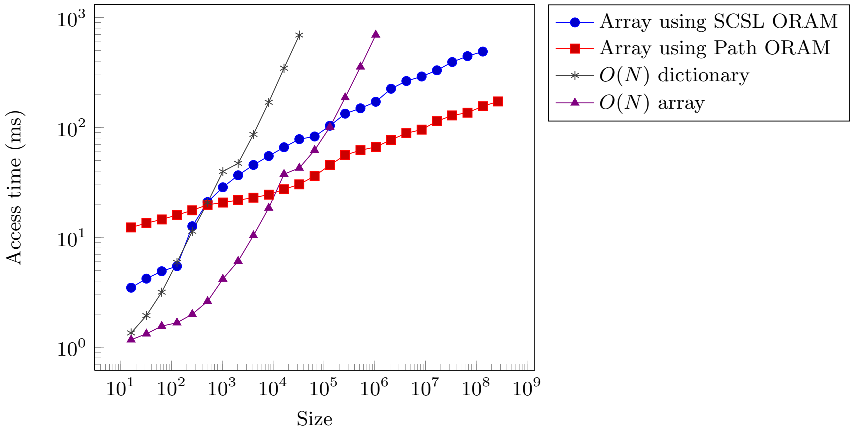

Fig. 2. Oblivious data structure online access times.

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.

For reference, we have also included timings for the offline phase, estimated using the costs given by Damgård et al. [9]. Note that these timings are to be understood as core seconds because the offline phase is highly parallelizable, unlike the online phase. It follows that the wall time of the offline phase can be brought down to the same order of magnitude of the online phase using ten thousand cores.

Our algorithms perform particularly well in comparison to the one by Aly et al. because cycle graphs have very low degree. For complete graphs, the full version [19] shows a different picture. The overhead for using ORAM is higher than the asymptotic advantage for all graph sizes that we managed to run our algorithms on.

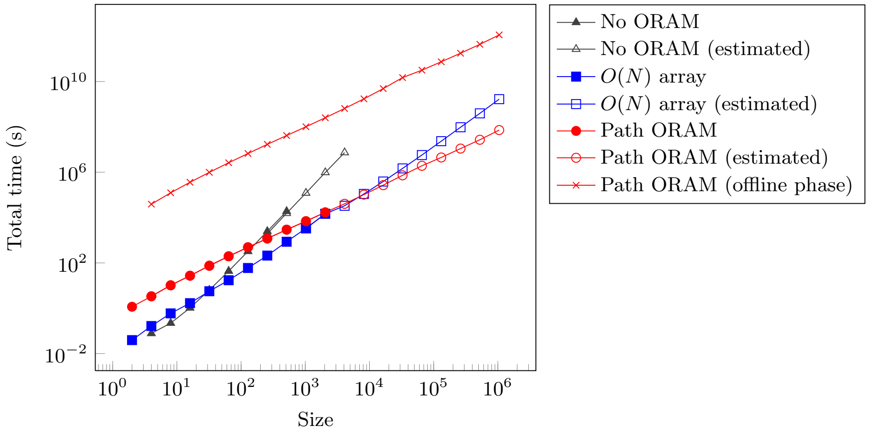

Fig. 3. Dijkstra's algorithm on cycle graphs

6.2 Gale-Shapley Algorithm for the Stable Marriage Problem

Using an oblivious array, it is straightforward to implement the Gale-Shapley algorithm in MPC. We have implemented a worst case example, where the main loop is executed $O(N^2)$ times. However, initialization of the preference matrices costs $O(N^2)$ in any case. With the overhead of the oblivious array, we get an overall complexity in $O(N^2 \log^3 N)$. Running a fully functional program for a limited time, we estimate that computing Gale-Shapley for 8192 pairs takes $9.1 \cdot 10^7$ seconds online and $1.5 \cdot 10^{12}$ seconds offline, using the figures by Damgård et al. [9] for the latter estimate. See the full version [19] for more timings.

Gale-Shapley with limited preferences. It does not always make sense to have full ranking of preferences. For example, it is hard to imagine that a human could ranking a thousand options. Therefore, we investigated the case of every agent only having a top twenty and feeling indifferent about the rest. In this case, both initialization and the worst-case main loop of a slightly modified algorithm become linear in the number of agents. Using the same techniques as above, we estimate that computing Gale-Shapley with 20 preferences for 2^{20} pairs takes $3.5 \cdot 10^7$ seconds online and $7.1 \cdot 10^{11}$ seconds offline.

Acknowledgements

We would like to thank Nigel Smart for various comments and suggestions. This work has been supported in part by EPSRC via grant EP/I03126X.

References

1. Aliasgari, M., Blanton, M., Zhang, Y., Steele, A.: Secure computation on floating point numbers. In: NDSS. The Internet Society (2013)
2. Aly, A., Cuvelier, E., Mawet, S., Pereira, O., Vyve, M.V.: Securely solving simple combinatorial graph problems. In: Sadeghi, A.R. (ed.) Financial Cryptography. Lecture Notes in Computer Science, vol. 7859, pp. 239–257. Springer (2013)
3. Blanton, M., Steele, A., Aliasgari, M.: Data-oblivious graph algorithms for secure computation and outsourcing. In: Chen, K., Xie, Q., Qiu, W., Li, N., Tzeng, W.G. (eds.) ASIACCS. pp. 207–218. ACM (2013)
4. Brickell, J., Shmatikov, V.: Privacy-preserving graph algorithms in the semi-honest model. In: Roy, B.K. (ed.) ASIACRYPT. Lecture Notes in Computer Science, vol. 3788, pp. 236–252. Springer (2005)
5. Catrina, O., de Hoogh, S.: Improved primitives for secure multiparty integer computation. In: Garay, J.A., Prisco, R.D. (eds.) SCN. Lecture Notes in Computer Science, vol. 6280, pp. 182–199. Springer (2010)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Second Edition. The MIT Press and McGraw-Hill Book Company (2001)
7. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Halevi, S., Rabin, T. (eds.) TCC. Lecture Notes in Computer Science, vol. 3876, pp. 285–304. Springer (2006)
8. Damgård, I., Keller, M., Larraia, E., Miles, C., Smart, N.P.: Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In: Visconti, I., Prisco, R.D. (eds.) SCN. Lecture Notes in Computer Science, vol. 7485, pp. 241–263. Springer (2012)
9. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) ESORICS. Lecture Notes in Computer Science, vol. 8134, pp. 1–18. Springer (2013)
10. Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious RAM without random oracles. In: Theory of Cryptography, pp. 144–163. Springer (2011)
11. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology – CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 643–662. Springer (2012)
12. Franklin, M.K., Gondree, M., Mohassel, P.: Improved efficiency for private stable matching. In: Abe, M. (ed.) CT-RSA. Lecture Notes in Computer Science, vol. 4377, pp. 163–177. Springer (2007)
13. Gentry, C., Halevi, S., Smart, N.: Fully homomorphic encryption with polylog overhead. Advances in Cryptology–EUROCRYPT 2012 pp. 465–482 (2012)
14. Gentry, C., Goldman, K.A., Halevi, S., Julta, C., Raykova, M., Wichs, D.: Optimizing ORAM and using it efficiently for secure computation. In: Privacy Enhancing Technologies. pp. 1–18. Springer (2013)

15. Gentry, C., Halevi, S., Lu, S., Ostrovsky, R., Raykova, M., Wichs, D.: Garbled RAM revisited. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT. Lecture Notes in Computer Science, vol. 8441, pp. 405–422. Springer (2014)
16. Gordon, S.D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., Vahlis, Y.: Secure two-party computation in sublinear (amortized) time. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) ACM Conference on Computer and Communications Security. pp. 513–524. ACM (2012)
17. Huang, Y., Evans, D., Katz, J.: Private set intersection: Are garbled circuits better than custom protocols? In: NDSS. The Internet Society (2012)
18. Jónsson, K.V., Kreitz, G., Uddin, M.: Secure multi-party sorting and applications. IACR Cryptology ePrint Archive 2011, 122 (2011)
19. Keller, M., Scholl, P.: Efficient, oblivious data structures for MPC. Cryptology ePrint Archive, Report 2014/137 (2014), `http://eprint.iacr.org/2014/137`
20. Keller, M., Scholl, P., Smart, N.P.: An architecture for practical actively secure MPC with dishonest majority. In: Sadeghi et al. [26], pp. 549–560
21. Launchbury, J., Diatchki, I.S., DuBuisson, T., Adams-Moran, A.: Efficient lookup-table protocol in secure multiparty computation. In: Thiemann, P., Findler, R.B. (eds.) ICFP. pp. 189–200. ACM (2012)
22. Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.: Automating efficient RAM-model secure computation, `http://www.cs.umd.edu/~liuchang/paper/oakland2014.pdf`
23. Mohassel, P., Sadeghian, S.S.: How to hide circuits in MPC an efficient framework for private function evaluation. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT. Lecture Notes in Computer Science, vol. 7881, pp. 557–574. Springer (2013)
24. Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: Advances in Cryptology–CRYPTO 2012, pp. 681–700. Springer (2012)
25. Ostrovsky, R., Shoup, V.: Private information storage. In: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. pp. 294–303. ACM (1997)
26. Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.): 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013. ACM (2013)
27. Shi, E., Chan, T.H.H., Stefanov, E., Li, M.: Oblivious RAM with O((logN)3) worst-case cost. In: Advances in Cryptology–ASIACRYPT 2011, pp. 197–214. Springer (2011)
28. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C.W., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol. In: Sadeghi et al. [26], pp. 299–310
29. Toft, T.: Secure datastructures based on multiparty computation. Cryptology ePrint Archive, Report 2011/081 (2011), `http://eprint.iacr.org/2011/081`
30. Waksman, A.: A permutation network. Journal of the ACM (JACM) 15(1), 159–163 (1968)
31. Wang, X., Nayak, K., Liu, C., Shi, E., Stefanov, E., Huang, Y.: Oblivious data structures. Cryptology ePrint Archive, Report 2014/185 (2014), `http://eprint.iacr.org/2014/185`