5

Oblivious Data Structures

Standard circuit-based execution is not well-suited to programs relying on random access to memory. For example, executing a simple array access where the index is a private variable (we use the $< z >$ notation to indicate that the variable z is private, with its semantic value protected by MPC),

```
a[<i>] = x
```

requires a circuit that scales linearly in the size of the array a. A natural circuit consists of N multiplexers, as shown in Figure 5.1. This method, where every element of a data structure is touched to perform an oblivious read or an update, is known as *linear scan*. For practical computations on large data structures, it is necessary to provide sublinear access operations. However, any access that only touches a subset of the data potentially leaks information about protected data in the computation.

In this chapter, we discuss several extensions to circuit-based MPC designed to enable efficient applications using large data structures. One strategy for providing sublinear-performance data structures in oblivious computation is to design data structures that take advantage of predictable access patterns. Indeed, it is not necessary to touch the entire data structure if the parts that are accessed do not depend on any private data (Section 5.1). A more general strategy, however, requires providing support for arbitrary memory access with sublinear

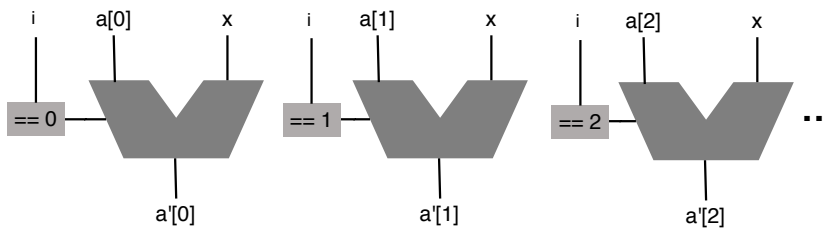

Figure 5.1: A single array access requiring N multiplexers.

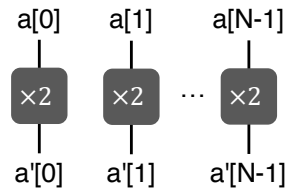

Figure 5.2: Oblivious array update with predictable access pattern.

cost. This cannot be achieved within a general-purpose MPC protocol, but can be achieved by combining MPC with oblivious RAM (Sections 5.2–5.5).

5.1 Tailored Oblivious Data Structures

In some programs the access patterns are predictable and known in advance, even though they may involve private data. As a simple example, consider this loop that doubles all elements of an array of private data:

```
for (i = 0; i < N; i++) {
    a[i] = 2 * a[i]
}
```

Instead of requiring N linear scan array accesses for each iteration (with $\Theta(N^2)$ total cost), the loop could be unrolled to update each element directly, as shown in Figure 5.2. Since the access pattern required by the algorithm is completely predictable, there is no information leakage in just accessing each element once to perform its update.

Most algorithms access data in a way that is not fully and as obviously predictable as in the above example. Conversely, usually it is done in a way that

is not fully data-dependent. That is, it might be *a priori* known (i.e., known independently of the private inputs) that some access patterns are guaranteed to never occur in the execution. If so, an MPC protocol that does not include the accesses that are known to be impossible regardless of the private data may still be secure. Next, we describe oblivious data structures designed to take advantage of predictable array access patterns common in many algorithms.

Oblivious Stack and Queue. Instead of implementing a stack abstraction using an array, an efficient oblivious stack takes advantage of the inherent locality in stack operations—they always involve the top of the stack. Since stack operations may occur in conditional contexts, though, the access pattern is not fully predictable. Hence, an oblivious stack data structure needs to provide conditional operations which take as an additional input a protected Boolean variable that indicates whether the operation actually occurs. For example, the <stack>.condPush(<b>, <v>) operation pushes v on the stack when the semantic value of b is true, but has no impact on the semantic state of the stack when b is false.

A naïve implementation of condPush would be to use a series of multiplexers to select for each element of the resulting stack either the current element, stack[i] when b is false, or the previous element, stack[i − 1] (or pushed value v, for the top element) when b is true. As with a linear scan array, however, this implementation would still require a circuit whose size scales with the maximum current size of the stack for each stack operation.

A more efficient data structure uses a hierarchical design, dividing the stack representation into a group of buffers where each has some slack space so it is not necessary to touch the entire data structure for each operation. The design in Zahur and Evans (2013), inspired by Pippenger and Fischer (1979), divides the stack into buffers where the level-i buffer has 5×2^i data slots. The top of the stack is represented by level 0. For each level, the data slots are managed in blocks/groups of 2^i slots; thus, for level 1, each data is always added in a block of two data items. For each block, a single bit is maintained that tracks whether the block is currently empty. For each level, a 3-bit counter, t, keeps track of the location of the next empty block available (0–5).

Figure 5.3 depicts an example of a conditional stack which starts off with some data already inserted and two condPush operations are illustrated. The

starting state in the figure depicts a state where none of the t values exceed 3, and hence there is guaranteed sufficient space for two conditional push operations. A multiplexer is used to push the new value into the correct slot based on the t_0 value, similar to the naïve stack circuit design described above. However, in this case, the cost is low since this is applied to a fixed 5-element array. After two conditional push operations, however, with the starting $t_0 = 3$, the level 0 buffer could be full. Hence, it is necessary to perform a shift, which either has no impact (if $t_0 \leq 3$), or pushes one block from level 0 into level 1 (as shown in Figure 5.3). After the shift, two more conditional push operations can be performed. This hierarchical design can extend to support any size stack, with shifts for level i generated for every 2^i `condPush` operations. A similar design can support conditional pop operations, where left shifts are required for level i after every 2^i `condPop` operations. The library implementing the conditional stack keeps track of the number of stack operations to know the minimum and maximum number of possible elements at each level, and inserts the necessary shifts to prevent overflow and underflows.

For all circuit-based MPC protocols, the primary cost is the bandwidth required to execute the circuit, which scales linearly in the number of gates. The cost depends on the maximum possible number of elements at each point in the execution. For a stack known to have at most N elements, k operations access level i at most $\lfloor k/2^i \rfloor$ times since we need a right shift for level i after every 2^i conditional push operations (and similarly, need a left shift after 2^i conditional pop operations).

However, the operations at the deeper levels are more expensive since the size of each block of elements at level i is 2^i, requiring $\Theta(2^i)$ logic gates to move. So, we need $\Theta(2^i \times k/2^i) = \Theta(k)$-sized circuits at level i. Thus, with $\Theta(\log N)$ levels, the total circuit size for k operations is $\Theta(k \log N)$ and the amortized cost for each conditional stack operation is $\Theta(\log N)$.

Other Oblivious Data Structures. Zahur and Evans (2013) also present a similar oblivious hierarchical queue data structure, essentially combining two stacks, one of which only supports push operations and the other that only supports pop operations. Since these stacks only need to support one of the conditional operations, instead of using a 5-block buffer at each level they use a 3-block buffer. Moving data between the two stacks requires an additional

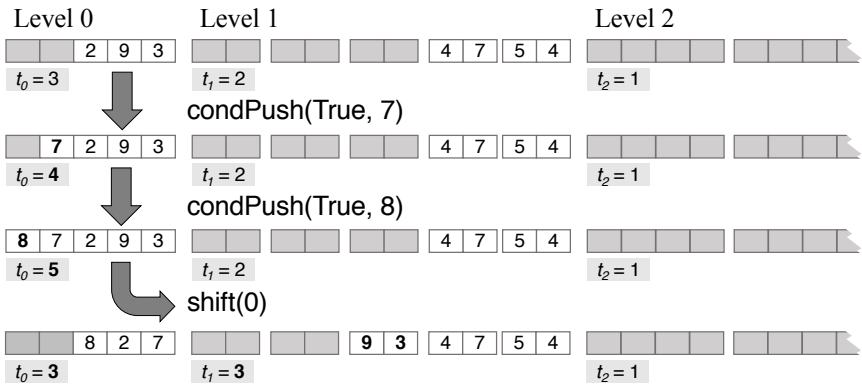

Figure 5.3: Illustration of two conditional push operations for oblivious stack. The shift(0) operation occurs after every two condPush operations.

multiplexer. Similarly to the oblivious stack, the amortized cost for operations on the oblivious queue is $\Theta(\log N)$.

Data structures designed to provide sublinear-cost oblivious operations can be used for a wide range of memory access patterns, whenever there is sufficient locality and predictability in the code to avoid the need to provide full random access. Oblivious data structures may also take advantage of operations that can be batched to combine multiple updates into single data structure scan. For example, Zahur and Evans (2013) present an associative map structure where a sequence of reads and writes with no internal dependencies can be batched into one update. This involves constructing the new value of the data structure by sorting the original values and the updates into a array, and only keeping the most recent value of each key-value pair. This allows up to N updates to be performed with a circuit of size $\Theta(N \log^2 N)$, with an amortized cost per update of $\Theta(\log^2 N)$.

The main challenge is writing programs to take advantage of predictable memory access patterns. A sophisticated compiler may be able to identify predictable access patterns in typical code and perform the necessary transformations automatically, but this would require a deep analysis of the code and no suitable compiler currently exists. Alternatively, programmers can manually rewrite code to use libraries that implement the oblivious data structures and manage all of the bookkeeping required to carry out the necessary shift

operations. Another strategy for building efficient oblivious data structures is to build upon a general-purpose Oblivious RAM, which is the focus of the rest of this chapter.

5.2 RAM-Based MPC

Oblivious RAM (ORAM) was introduced by Goldreich and Ostrovsky (1996) as a memory abstraction that allows arbitrary read and write operations without leaking any information about which locations are accessed. The original ORAM targeted the client-server setting, where the goal is to enable a client to outsource data to an untrusted server and perform memory operations on that outsourced data without revealing the data or access patterns to the server.

Ostrovsky and Shoup (1997) first proposed the general idea of using ORAM to support secure multi-party computation by splitting the role of the ORAM server between two parties. Gordon *et al.* (2012) were the first to propose a specific method for adapting ORAM to secure computation (RAM-MPC, also often called RAM-SC). In (Gordon *et al.*, 2012), the ORAM state is jointly maintained by both parties (client and server) within a secure computation protocol. The key idea is to have each party store a share of the ORAM's state, and then use a general-purpose circuit-based secure computation protocol to execute the ORAM access algorithms.

An ORAM system specifies an initialization protocol that sets up the (possibly already populated) storage structure, and an access protocol that implements oblivious read and write operations on the structure. To satisfy the oblivious memory goals, an ORAM system must ensure that the observable behaviors reveal nothing about the elements that are accessed. This means the physical access patterns produced by the access protocol for any same-length access sequences must be indistinguishable.

The initialization protocol takes as input an array of elements and initializes an oblivious structure with those elements without revealing anything about the initial values other than the number of elements. Assuming the access protocol is secure, it is always possible to implement the initialization protocol by performing the access protocol once for each input element. The costs of initializing an ORAM this way, however, may be prohibitive, especially as used in secure computation.

The RAM-MPC construction proposed by Gordon *et al.* (2012) implemented an ORAM-based secure multi-party computation. To implement an ORAM access, a circuit is executed within a 2PC that translates the oblivious logical memory location into a set of physical locations that must be accessed to perform the access. The physical locations are then revealed to the two parties, but the ORAM design guarantees that these leak no information about the logical location accessed. Each party then retrieves the data shares stored in those locations, and passes them into the MPC. To complete the access, a logical write occurs, as follows. The circuit executing within the MPC produces new data elements to be written back to each of the physical locations. These locations are output in plaintext, together with the data shares to be written into parties' local physical storage.

Gordon *et al.* (2012) proved that combining semi-honest 2PC with the semi-honest ORAM protocol where ORAM state is split between the two parties and operations are implemented within the 2PC results in a secure RAM-based MPC in the semi-honest model.[1]

5.3 Tree-Based RAM-MPC

The construction of Gordon *et al.* (2012) builds on the tree-based ORAM design of Shi *et al.* (2011). The underlying data structure in the tree-based ORAM storing N elements is a binary tree of height $\log N$, where each node in the tree holds $\log N$ elements. Each logical memory location is mapped to a random leaf node, and the logical index and value of the data element is stored in encrypted form in one of the nodes along the path between the tree root and that leaf. To access a data item, the client needs to map the item's location to its associated leaf node and retrieve from the server all nodes along the path from the root to that node. Each of the nodes along the path is decrypted and scanned to check if it is the requested data element.

A careful reader will notice the following technical difficulty. Performing data look up requires mapping the logical location to the leaf node. However, the size of this map is linear in N, and hence the map cannot be maintained by the client with sublinear storage. The solution is to store the location map

[1] RAM-MPC can also be made to work in the malicious security model (Afshar *et al.*, 2015), but care must be taken to ensure that data stored outside the MPC is not corrupted.

by the server using another instance of tree-based ORAM. Importantly, since the size of the index map is smaller than the size of the data, the size of the second ORAM tree can be smaller than N, as each item in the second tree can store several mappings. To support larger ORAMs, a sequence of look-up trees might be used, where only the smallest tree is stored by the client.

In RAM-MPC, the trees are secret-shared between the two parties. To access an element, the parties execute a 2PC protocol that takes the shares of the logical index and outputs (in shares to each party) the physical index for the next level. A linear scan is used on the reconstructed elements along the search path to identify the one corresponding to the requested logical index. In the final tree, the shares of the requested data element are output to the higher-level MPC protocol. Note that the data in each node could also be stored in an ORAM to avoid the need for a linear scan, but since the bucket sizes are small here and (at least with this ORAM design) there is substantial overhead required to support an ORAM, a simple linear scan is preferred.

To complete the data access procedure, we need to ensure that the subsequent access results in an oblivious access pattern, even if the same element is accessed again. To achieve this in tree-based ORAMs, the accessed logical location is re-mapped to a random leaf node, and the updated data value is inserted into the root node in the tree, ensuring its availability in the subsequent access. To prevent the root node from overflowing, the protocol of Shi *et al.* (2011) uses a balancing mechanism that pushes items down the tree after each ORAM access. Randomly-selected elements are *evicted* and are moved down the tree by updating both of the child nodes of the selected nodes (this is done to mask which leaf-path contains the evicted element).

Intuitively, the access pattern is indistinguishable from a canonical one, and hence an adversary cannot distinguish between two accesses. Indeed, every time an element is accessed, it is moved to a random-looking location. Further, every access retrieves a complete path from the root to a leaf, so as long as the mapping between logical locations and leaves is random and not revealed to the server, the server learns no information about which element is accessed. The scheme does have the risk that a node may overflow as evicted elements are moved down the tree, and not be able to store all of the elements required. The probability of an overflow after k ORAM accesses with each node holding $O(\log(\frac{kN}{\delta}))$ elements is shown by Shi *et al.* (2011) to be less than δ, which

is why the number of elements in each node is set to $O(\log N)$ to make the overflow probability negligible. The constant factors matter, however. Gordon *et al.* (2012) simulated various configurations to find that a binary search on a 2^{16} element ORAM (that is, only 16 operations) could be implemented with less than 0.0001 probability of overflow with a bucket size of 32.

Variations on this design improved the performance of tree-based ORAM for MPC have focused on using additional storage (called a *stash*) to store overflow elements and reduce the sizes of the buckets needed to provide negligible failure probability, as well as on improving the eviction algorithm.

Path ORAM. Path ORAM (Stefanov *et al.*, 2013) added a stash to the design as a fixed-size auxiliary storage for overflow elements, which would be scanned on each request. The addition of a small stash enabled a more efficient eviction strategy than the original binary-tree ORAM. Instead of selecting two random nodes at each level for eviction and needing to update both child nodes of the selected nodes to mask the selected element, Path ORAM performed evictions on the access path from the root to the accessed node, moving elements along this path from the root towards the leaves as much as possible. Since this path is already accessed by the request, no additional masking is necessary to hide which element is evicted. The Path ORAM design was adapted by Wang *et al.* (2014a) to provide a more efficient RAM-MPC design, and they presented a circuit design for a more efficient eviction circuit.

Circuit ORAM. Further advances in RAM-MPC designs were made both by adapting improvements in traditional client-server ORAM designs to RAM-MPC, as well as by observing differences between the costs and design space options between the MPC and traditional ORAM setting and designing ORAM schemes focused on the needs of MPC.

Wang *et al.* (2014a) argued that the main cost metric for ORAM designs used in circuit-based secure computation should be *circuit complexity*, whereas client-server designs were primarily evaluated based on (client-server) bandwidth metrics. When used within an MPC protocol, the execution costs of an ORAM are dominated by the bandwidth costs of executing the circuits needed to carry out ORAM accesses and updates within the MPC protocol.

Wang *et al.* (2015b) proposed Circuit ORAM, an ORAM scheme designed

specifically for optimal circuit complexity for use in RAM-MPC. Circuit ORAM replaced the complex eviction method of Path ORAM with a more efficient design where the eviction can be completed with a single scan of the blocks on the eviction path, incorporating both the selection and movement of data blocks within a single pass. Their key insight is to perform two metadata scans first, so as to determine which blocks are to be moved, together with their new locations. These scans can be run on the metadata labels, which are much smaller than the full data blocks. After these scans have determined which blocks to move, the actual data blocks can be moved using a single pass along the path from the stash-root to the leaf, storing at most one block of data to relocate as it proceeds.

Because the metadata scans are on much smaller data than the actual blocks, the concrete total cost of the scan is minimized. Wang *et al.* (2015b) proved that with block size of $D = \Omega(\log^2 N)$, Circuit ORAM can achieve statistical failure probability of δ for block by setting the stash size to $O(\log \frac{1}{\delta})$ with circuit size of $O(D(\log N + \log \frac{1}{\delta}))$. The optimizations in Circuit ORAM reduce the effective cost of ORAM (measured by the number of non-free gates required in a circuit) by a factor of over 30 compared to the initial binary-tree ORAM design for a representative 4GB data size with 32-bit blocks.

5.4 Square-Root RAM-MPC

Although the first proposed ORAM designs were hierarchical, early RAM-MPC designs did not adopt these constructions because their implementation seemed to require implementing a pseudo-random function (PRF) within the MPC, and using the outputs of that function to perform an oblivious sort. Both of these steps would be very expensive to do in a circuit-based secure computation circuit, so RAM-MPC designs favored ORAMs based on the binary-tree design which did not require sorting or a private PRF evaluation.

Zahur *et al.* (2016) observed that the classic square-root ORAM design of Goldreich and Ostrovsky (1996) could in fact be adapted to work efficiently in RAM-MPC by implementing an oblivious permutation where the PRF required for randomizing the permutation would be jointly computed by the two parties *outside* of the generic MPC. This led to a simple and efficient ORAM design, which, unlike tree-based ORAMs, has zero statistical failure probability, since there is no risk that a block can overflow. The design maintains a public

set, Used, of used physical locations (revealing no information since logical locations are assigned randomly to physical ones, and only accessed at most once), and an oblivious stash at each level that stores accessed blocks. Since the stash contains private data, it is stored in encrypted form as wire labels within the MPC. Unlike in the tree-based ORAM designs where the stash is used as a probabilistic mechanisms to deal with node overflows, in Square-Root ORAM the stash is used deterministically on each access. Each access adds one element to each of the level stashes, and all of the stashes must be linearly scanned on every access. If an accessed element is found in the stash, to preserve the obliviousness, the look-up continues with a randomly selected element. The size of each stash determines the number of accesses that can be done between reshufflings. Optimal results are obtained by setting the size to $\Theta(\sqrt{(N)})$, hence the name "square-root ORAM". The oblivious shuffling is performed using a Waksman network (Waksman, 1968), which requires $n \log_2 n - n + 1$ oblivious swaps to permute n elements. Using the design from Huang *et al.* (2012a), this can be done with one ciphertext per oblivious swap.

One major advantage of the Square-Root ORAM design is its concrete performance, including initialization. All that is required to initialize is produce a random permutation and obliviously permute all the input data, generating the oblivious initial position map using the same method as the update protocol. Compared to earlier ORAM designs, where initialization was done with repeated writes, this approach dramatically reduces the cost of using the ORAM in practice. Without considering initialization, Square-Root ORAM has a per-access cost that is better than linear scan, once there are more than 32 blocks (for typical block sizes of 16 or 32 bytes), whereas Circuit ORAM is still more expensive than linear scanning up to 2^{11} blocks. Although Square-Root ORAM has asymptotically worse behavior than Circuit ORAM, its concrete costs per access are better for ORAM sizes up to 2^{16} blocks. For such large ORAMs, initialization costs become an important factor — initializing a Square-Root ORAM requires $\Theta(\log N)$ network round trips, compared to $\Theta(N \log N)$ for Circuit ORAM. Initializing a Circuit ORAM with $N = 2^{16}$ blocks would take several days, and its asymptotic benefits would only be apparent for very expensive computations.

5.5 Floram

Doerner and Shelat (2017) observed that even the sublinear-cost requirement, which was an essential design aspect of traditional ORAM systems, was not necessary to be useful in RAM-MPC. Since the cost of secure computation far exceeds the cost of standard computation, ORAM designs that have linear cost "outside of MPC", but reduce the computation performed "inside MPC", may be preferred to sublinear ORAM designs. With this insight, Doerner and Shelat (2017) revisited the Distributed Oblivious RAM approach of Lu and Ostrovsky (2013) and based a highly scalable and efficient RAM-MPC scheme on two-server private information retrieval (PIR). The scheme, known as Floram (*Function-secret-sharing Linear ORAM*), can provide over 100× improvements over Square-Root ORAM and Circuit ORAM across a range of realistic parameters.

Distributed Oblivious RAM relaxes the usual security requirement of ORAM (the indistinguishability of server traces). Instead, the ORAM server is split into two non-colluding servers, and security requirement is that the memory access patterns are indistinguishable based on any single server trace (but allowed to be distinguishable if the traces of both servers are combined). We note that it is not immediately obvious how to use this primitive in constructing two-party MPC, since it requires two non-colluding servers *in addition* to the third player—the ORAM client.

Private information retrieval (PIR) enables a client to retrieve a selected item from a server, without revealing to the server which item was retrieved (Chor *et al.*, 1995). Traditionally, PIR schemes are different from ORAM in that they only provide read operations, and that they allow a linear server access cost whereas ORAM aims for amortized sublinear retrieval cost.

A *point function* is a function that outputs 0 for all inputs, except one:

$$P^{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0, & \text{otherwise.} \end{cases}$$

Gilboa and Ishai (2014) introduced the notion of *distributed point functions* (DPF), where a point function is secret-shared among two players with shares that have sizes sublinear in the domain of the function, hiding the values of both α and β. The output of each party's evaluation of the secret-shared function is a share of the output and a bit indicating if the output is valid:

$y_p^x = P_p^{\alpha,\beta}(x)$ (party p's share output of the function), and $t_p^x = (x = \alpha)$ (a share of 1 if $x = \alpha$, otherwise a share of 0). Gilboa and Ishai (2014) showed how this could be used to efficiently implement two-server private information retrieval, and Boyle *et al.* (2016b) improved the construction.

The Floram design uses secret-shared distributed point functions to implement a two-party oblivious write-only memory (OWOM) and both a two-party oblivious read-only memory (OROM). The ORAM is constructed by composing the OWOM and OROM, but since it is not possible to read from the write-only memory, Floram uses a linear-scan stash to store written elements until it is full, at which point the state of the ORAM is refreshed by converting the write-only memory into oblivious read-only memory, replacing the previous OROM and resetting the OWOM stash. In the OWOM, values are stored using XOR secret sharing. To write to an element, all elements are updated by xor-ing the current value with the output of a generated distributed point function—so, the semantic value of the update is 0 for all elements other than the one to be updated, and the difference between the current value and updated value for the selected element.

Reading. In the OROM, each stored value is masked by a PRF evaluated at its index and the masked value is secret-shared between the two OROMs. To read element i from the OROM, each party obtains k_p^i from the MPC corresponding to its key for the secret-shared DPF. Then, it evaluates $P_{k_p^i}(x)$ on each element of its OROM and combines all the results with xor. For each element other than $x = i$ the output is its share of 0, so the resulting sum is its share of the requested value, v_p^i. This value is input into the MPC, and xor-ed with the value provided by the other party to obtain $R^i = v_1^i \oplus v_2^i$. To obtain the actual value of element i, R^i is xor-ed with the output of $\mathsf{PRF}_k(i)$, computed within the MPC. The PRF masking is necessary to avoid leaking any information when the OROM is refreshed. Each read requires generating a DPF ($O(\log N)$ secure computation and communication), $O(N)$ local work for the DPF evaluation at each element, and constant-size (independent of N) secure computation to compute the PRF for unmasking. In addition, each read requires scanning the stash within the MPC using a linear scan in case the requested element has been updated since the last refresh. Hence, the cost of the scheme depends on how large a stash is needed to amortize the refresh cost.

Refreshing. Once the stash becomes full, the ORAM needs to be refreshed by converting the OWOM into a new OROM and clearing the stash. This is done by having each party generate a new PRF key (k_1 generated by P_1, k_2 generated by P_2) and masking all of the values currently stored in its OWOM with keyed PRF, $W'_p[i] = \mathsf{PRF}_{k_p}(i) \oplus W_p[i]$, for each party, $p \in \{1, 2\}$. The masked values are then exchanged between the two parties. The OROM values are computed by xor-ing the value received from the other party for each cell with their own masked value to produce $R[i] = \mathsf{PRF}_{k_1}(i) \oplus \mathsf{PRF}_{k_2}(i) \oplus W_1[i] \oplus W_2[i]$, where $v[i] = W_1[i] \oplus W_2[i]$. Each party passes in its PRF key to the MPC, so that values can be unmasked within MPC reads by computing $\mathsf{PRF}_{k_1}(i) \oplus \mathsf{PRF}_{k_2}(i)$ within the MPC. This enables the read value to be unmasked for used within the MPC, without disclosing the private index i. Thus, refreshing the stash requires $O(N)$ local computation and communication, and no secure computation. Because the refresh cost is relatively low, the optimal access period is $O(\sqrt{N})$ (with low constants, so their concrete implementation used $\sqrt{N}/8$).

Floram offers substantial performance improvements over Square-Root ORAM and all other prior ORAM constructions used in RAM-MPC, even though its asymptotic costs are linear. The linear-cost operations of the OROM and OWOM are implemented outside the MPC, so even though each access requires $O(N)$ computation, the concrete cost of this linear work is much less that the client computation done within the MPC. In Doerner and Shelat (2017)'s experiments, the cost of the secure computation is the dominant cost up to ORAMs with 2^{25} elements, after which the linear local computation cost becomes dominant. Floram was over able to scale to support ORAMs with 2^{32} four-byte elements with an average access time of 6.3 seconds over a LAN.

Floram also enables a simple and efficient initialization method using the same mechanism as used for refreshing. The Floram design can also support reads and writes where the index is not private very efficiently—the location i can be read directly from the OWOM just by passing in the secret-shared values in location i of each party's share into the MPC. Another important advantage of Floram is that instead of storing wire labels as is necessary for other RAM-MPC designs, which expands the memory each party must store by factor κ (the computational security parameter), each party only needs to store a secret share of the data which is the same as the original size of the data for each the OROM and OWOM.

5.6 Further Reading

Many other data structures have been proposed for efficient MPC, often incorporating ORAM aspects. Keller and Scholl (2014) proposed efficient MPC data structures for arrays, built on top of ORAM designs. Wang *et al.* (2014b) devised oblivious data structures including priority queues that take advantage of sparse and predictable access patterns in many applications, and presented a general pointer-based technique to support efficient tree-based access patterns.

We only touched on the extensive literature on oblivious RAM, focusing on designs for MPC. ORAM continues to be an active research area, with many different design options and tradeoffs to explore. Buescher *et al.* (2018) study various MPC-ORAM designs in application settings and developed a compiler that selects a suitable ORAM for the array-accesses in a high-level program. Faber *et al.* (2015) proposed a three-party ORAM based on Circuit ORAM that offers substantial cost reduction in the three-party, honest majority model. Another new direction that may be useful for MPC ORAM is to allow some amount of limited leakage of the data access pattern to gain efficiency (Chan *et al.*, 2017; Wagh *et al.*, 2018).