Three-Party ORAM for Secure Computation

Sky Faber[⋆] Stanislaw Jarecki[⋆⋆] Sotirios Kentros[⋆⋆⋆] Boyang Wei[†]

Abstract. An Oblivious RAM (ORAM) protocol [13] allows a client to retrieve N-th element of a data array D stored by the server s.t. the server learns no information about N. A related notion is that of an *ORAM for Secure Computation* (SC-ORAM) [17], which is a protocol that securely implements a RAM functionality, i.e. given a secret-sharing of both D and N, it computes a secret-sharing of D[N]. SC-ORAM can be used as a subprotocol for implementing the RAM functionality for secure computation of RAM programs [17, 7, 14]. It can also implement a public database service which hides each client's access pattern even if a threshold of servers colludes with any number of clients.

Most previous works used two-party secure computation to implement each step of an ORAM client algorithm, but since secure computation of many functions becomes easier in the three-party honest-majority setting than in the two-party setting, it is natural to ask if the cost of an SC-ORAM scheme can be reduced if one was willing to use three servers instead of two and assumed an honest majority. We show a 3-party SC-ORAM scheme which is based on a variant of the *Binary Tree* Client-Server ORAM of Shi et al. [20]. However, whereas previous SC-ORAM implementations used general 2PC or MPC techniques like Yao's garbled circuits, e.g. [14, 22], homomorphic encryption [11], or the SPDZ protocol for arithmetic circuits [15], our techniques are custom-made for the three-party setting, giving rise to a protocol which is secure against honest-but-curious faults using bandwidth and CPU costs which are comparable to those of the underlying Client-Server ORAM.

Keywords: Oblivious RAM, Secure Computation, Private Information Retrieval

1 Introduction

Oblivious RAM for Secure Computation. An Oblivious RAM (ORAM) is a protocol between *client* and *server* which allows the client who can locally store only small amount of information to write and read from an outsourced memory in such a way that the server cannot tell which memory locations the client accesses, and the cost of each memory access is sublinear in the memory size. Starting from the seminal work of Goldreich and Ostrovsky [13], there have

[⋆] U. California Irvine. Email: fabers@uci.edu
[⋆⋆] U. California Irvine. Email: stasio@ics.uci.edu
[⋆⋆⋆] Salem State U. Email: sotirios.kentros@salemstte.edu
[†] U. California Irvine. Email: boyanw1@uci.edu

been numerous improvements in ORAM techniques, achieving polylogarithmic client storage, computation, bandwidth, and server storage overheads, with the most recent proposal of Path ORAM by Stefanov et al. [21] practical enough to be implemented on secure processors [10, 9, 18].

The above classic formulation of the ORAM problem, which we will call a *client-server ORAM*, provides secure outsourced memory for a *single* client. The client-server ORAM notion can be generalized (and relaxed) by considering still a single client but $n > 1$ servers, and assuring client-access obliviousness only if at most $t < n$ of these servers collude. Such generalization was proposed and realized for $(t, n) = (1, 2)$ by Ostrovsky and Lu [17]. However, one can imagine a stronger notion, namely of a protocol which allows n servers to *emulate* an Oblivious RAM functionality so that a shared memory can be accessed by multiple clients, but their access patterns remain hidden even if up to t of these n servers collude, possibly with any coalition of the clients. Such multi-party ORAM emulator is equivalent to (multi-party) secure computation of the RAM functionality (called SC-ORAM for short), which given the secret-sharing of memory array D and a secret-sharing of a location i (and value v) outputs a secret-sharing of record $\mathsf{D}[i]$ (and writes v at index i in the secret-shared D).

One source of interest in such ORAM emulation is that it can provide oblivious RAM access as subprotocol for secure computation of any RAM program[19, 7, 14]. Recall that secure computation allows a set of parties $S_1, ..., S_n$ to compute some (randomized) function f on their inputs, where each S_i inputs a private value x_i into the computation, in such a way that the protocol reveals nothing else but the final output value $y = f(x_1, ..., x_n)$ to the participants. A standard approach to secure computation is to represent function f as a Boolean circuit [3], an arithmetic circuit [1], or a decision diagram [16]. However, even for very simple functions, each of these representations can be impractically large. This is indeed necessarily so if some of the inputs x_i are very long, i.e. when some of the data involved in the computation of f is large. Consider any information retrieval task, where x_1 is a large database, x_2 is a search term, and f is a search algorithm. The circuit or decision tree representation of f is at least as long as x_1, and therefore secure computation of f using any of the above techniques must take time at least linear in the size of the database.

SC-ORAM Applications. On the other hand each information retrieval problem which has a practical solution does so because it has an efficient RAM program, and as Ostrovsky and Shoup were the first to point out [19], an ORAM emulator can be used to securely compute any RAM program, because each local computation step can be implemented using Yao's garbled circuit technique [24] while each memory access can be handled by the SC-ORAM subprotocol. Examples of such SC-ORAM usage were recently provided by Keller and Scholl [15], who used their SC-ORAM implementation to build MPC implementations of other datastructures, e.g. a priority queue, and then utilized them in MPC computation of various algorithms in the RAM computation model e.g. Dijkstra's shortest path algorithm. In general, SC-ORAM is well suited to secure computation of any information-retrieval algorithm because such algorithms rely very strongly on the RAM model, e.g. by identifying database entries using hash tables of keywords. One application using SC-ORAM in this way could be provision of a shared database resource to *multiple clients* in a way that hides any client's access pattern even if all other clients collude. One can look at such usage of SC-ORAM as providing an alternative to Searchable Symmetric Encryption which requires interaction but hides all patterns of access to the encrypted data. In its most basic form of secure implementation of an array look-up, 3-server SC-ORAM also provides an interactive alternative to 3-server (symmetric) Private Information Retrieval [4, 12], with support for both read and write, and with the added security property that the database itself is private to any (single) server.

Two-Party SC-ORAM Constructions. Apart of pointing out the usefulness of an SC-ORAM protocol, Ostrovsky and Shoup sketched a method for converting any client-server ORAM into a two-party SC-ORAM: One of the two parties can implement the server in the underlying client-server ORAM scheme, while all the work of the client can be jointly computed by the two parties using Yao's garbled circuit technique applied to the circuit representation of each step of the client's algorithm in any client-server ORAM scheme. This idea was also considered by Damgard et al. [7], and it was further developed by Gordon et al. [14] who showed an optimized two-party ORAM emulation protocol based on the *Binary Tree* client-server ORAM of Shi et al. [20], utilizing a novel subprotocol gadget for secure computation of a pseudorandom function (PRF) on secret-shared inputs. Gentry et al. [11] showed several space and computation saving modifications of the Binary Tree client-server ORAM of Shi et al., together with a very different two-party ORAM emulation protocol for it, which used a customized homomorphic encryption scheme instead of Yao's garbled circuits for the two-party computation of the ORAM client's algorithm. Other modifications of the Binary Tree ORAM were shown in [5], and in the *Path ORAM* proposal of Shi et al. [21], but even though they improve the client-server ORAM, it is not clear if these modifications translate into a faster ORAM emulator.

Indeed, Wang et al. [22] examined the circuit complexity of the client's algorithm in several proposed variants of the Binary Tree client-server ORAM, including [20, 5], and they concluded that the client's algorithm in the original scheme of [20] has by far the smallest circuit. They also showed a set of modifications to the Binary Tree ORAM, building on the Path ORAM modifications, which result in roughly a factor of 10 reduction in the circuit representation of the client algorithm in a client-server ORAM, hence speeding up the two-party ORAM emulation protocol of [14] by the same factor. The SC-ORAM protocol for a database D containing 2^m records of size d each requires secure computation of a circuit of asymptotic size $\mathrm{O}\left(m^3 + dm\right)$, resulting in $\mathrm{O}\left(\kappa(m^3 + dm)\right)$ bound on protocol bandwidth where κ is a cryptographic security parameter. For m between 20 and 29 and $d = 4B$ this comes to between $4.6M$ and $13M$ gates, and its secure evaluation requires (on-line) as many hash or block cipher operations as the number of non-xor gates in the circuits. (In a work concurrent to ours Wang et al. [23] showed further reductions in the ORAM client circuit size, reporting $350K$ *and* gates for $m = 20$ and $d = 4B$.)

Multi-Party SC-ORAM and Our Contribution. Secure computation of many functionalities can be implemented with easier tools in the multi-party setting with honest majority than in the two-party setting. (In fact, assuming secure channels any function can be securely computed without further cryptographic assumptions [1, 2].) Thus even as the search for a minimal circuit ORAM continues, one can ask if secure ORAM emulation can be made significantly easier by moving from the 2PC to the MPC setting. Keller and Scholl [15] showed one way to design such multi-party SC-ORAM protocols, using arithmetic circuit representation of an ORAM client and implementing it with the SPDZ MPC protocol of Damgard et al. [8, 6]. Their implementation achieved significantly faster *on-line* times than the 2PC SC-ORAM implementation of [22]: [15] report 250 millisecond wall clock per access for $m = 20$ and $d = 5B$ for 2 machines with direct connection in the online stage, while Wang et al. [22] report 30 seconds of just CPU time for $m = 24$ and $d = 4B$. Moreover, the implementation of [15] is secure against malicious adversaries while that of [22] works only against honest-but-curious faults. On the other hand, this on-line speed-up comes at the cost of intensive precomputation required by the SPDZ MPC protocol, which [15] estimated at between 100 and 800 minutes for $m = 20$.

In this work we explore a different possibility for SC-ORAM design, specific to the setting of three parties with a single corrupted party, with security against honest-but-curious faults. The 3-party SC-ORAM protocol we propose uses a variant of the Binary-Tree ORAM as the underlying data-structure. The *access* part of the proposed SC-ORAM is based on the following observation: If P_1 and P_2 secret-share an array of (keyword,value) pairs (k, v) (this will be a path in the Binary-Tree ORAM) and a searched-for keyword k^* (this will be the searched-for address prefix), then a variant of the Conditional Disclosure of Secret protocol of [12] which we call *Secret-Shared Conditional OT* (SS-COT) allows P_3 to receive value v associated with keyword k^* at the cost roughly equal to the symmetric encryption and transmission of the array. Moreover, while SS-COT reveals the location of pair (k^*, v) in the array to P_3, this leakage can be easily masked if P_1, P_2 first shift the secret-shared array shifted by a random offset. The *eviction* part of the SC-ORAM springs from an observation that instead of performing the eviction computation on all the data in the path retrieved at access, one can use garbled circuit to encode only the procedure determining eviction *movement logic*, i.e. determining which entries in each bucket should be shifted down the path. Then, if P_1, P_2 secret-share the retrieved path, and hence the bits which enter this computation, we can let P_3 evaluate this garbled circuit and learn the positions of the entries to be moved if (1) the eviction moves a constant number of entries in each bucket in a predictable way, e.g. one step down the path, and (2) P_1, P_2 randomly permute the entries in each bucket, so that P_3 always computes a fixed number of randomly distributed distinct indexes for each bucket. Computation of this movement logic uses only two input bits (appropriate direction bit and a full/empty flag) and 17 non-xor gates per bucket entry, so the garbled circuit is much smaller than if it coded the whole eviction procedure. Finally, the secret-shared data held by P_1, P_2 can be moved according to the movement matrix held by P_3 in another OT/CDS variant we call *Secret-Shared Shuffle OT* which uses only xor's and whose bandwidth is roughly four times the size of the secret-shared path.

Assuming constant record sizes the bandwidth of the resulting SC-ORAM protocol is $\mathrm{O}\left(w(m^3 + \kappa m^2)\right)$ where w is the bucket size in the underlying Binary-Tree ORAM. Since the best exact bound on overflow probability we can give requires $w = \Omega(\lambda + m)$ where λ is a statistical security parameter, and since $m < \kappa$, this asymptotic bound is essentially the same as that of the 2-server SC-ORAM of [22]. However, the exact numbers for bandwidth and computation cost (measured in the number of block cipher or hash operations) are much lower, and this is because of two factors: First, even though we still use garbled circuits, the circuits involved have dramatically smaller complexity than in the 2PC implementations (see Table 1 below). Secondly, the cost of all operations *outside* the garbled circuits is a small factor away from the cost of transmission and decryption of server data in the underlying Binary-Tree ORAM algorithm. Concretely, the non-GC bandwidth is under $9P$ where P is the (total) size of tree *paths* retrieved by the Binary-Tree ORAM, and computation is bounded by symmetric encryption of roughly $20P$ bits, whereas for the underlying Binary-Tree ORAM both quantities are $2P$. Finally, stochastic evidence suggests that it suffices that $w = \Omega(\sqrt{\lambda + m})$, which for concrete parameters of $m = 36$ and $\lambda = 40$ reduces the required w, and hence all our protocol costs, by a further factor between 3 and 4.

ORAM	Circuit Size (Asymptotic Bounds)	Circuit Size (gates) $m = 20$	$m = 29$	Number of Inputs $m = 20$	$m = 29$
Path-SC ORAM	$\tilde{O}\left(m^3 + dm\right)\omega(1)$	37.2 M	111.7 M	0.2 M	0.3 M
SCORAM	N/A (heuristic)	4.6 M	13.0 M	0.3 M	0.9 M
3PORAM ($w = 128$)	$O\left(m^3\right)\omega(1)$	96.9K	213.9K	11.5K	25.3K
3PORAM ($w = 32$)	$O\left(m^3\right)\omega(1)$	28.5K	62.6K	3.4K	7.4K

Table 1. Comparison of Circuit Size between this proposal (without optimizations) and the SC-ORAM scheme [22]. All numbers are reported as function of array size $|\mathsf{D}| = 2^m$ for statistical security paramenter $\lambda = 80$. The first 3PORAM estimation uses bucket size $w = 128$ mandated by the strict bound implied by Lemma 2, while the second one uses bucket size $w = 32$ derived from the Markov Chain approximation. (We note Wang et al. [23] recently exhibited further reductions in ORAM circuit, e.g. reporting $350K$ *and* gates for $m = 20$. See Section 5 for further discussion.)

We tested a preliminary Java implementation of our scheme, without making use of several possible optimizations, on three lowest-tier Amazon servers (t2.micro) connected through a LAN. Regarding the overall bandwidth, for a concrete value $m = 18$ (the largest size for which [22] give bandwidth data), $\lambda = 80$, and $d = 4B$, our scheme instantiated with $w = 128$, which satisfies exact security bounds for these m, λ values, uses $1.18MB$ bandwidth, a factor of 40 less than $45MB$ reported by [22]. Using the stochastic bound $w = \Omega(\sqrt{\lambda + m})$ which for the above case of m, λ is satisfied by $w = 32$ (see Section 5) would further decrease the bandwidth by a factor of about 2.5. Regarding computational costs, for $m = 36$ and $d = 4B$, using $w = 32$ justified by the stochastic evidence, our implementation takes 320 milliseconds per access in the on-line phase and 1.3 seconds in the pre-computation phase. The SC-ORAM scheme of [22] reported only local execution times for m up to 26 while we tested our scheme on Amazon EC2 servers communicating over LAN for m up to 36, but for a conservative statistical security parameter $\lambda = 80$ the combined CPU cost of our implementation using $w = 128$ is a factor of about 50 less than that of [22], e.g. it is under 600 msec for $m = 24$ while that of [22] is 30 seconds.

In summary, we see our contributions as three-fold: First, we provide an immediate improvement to any application of SC-ORAM which can be done in the setting of three parties with an honest-majority. Secondly, the techniques we explore can be utilized in a different context, e.g. for a different "secure-computation friendly" eviction strategy for a Binary-Tree ORAM's. Finally, the proposed protocol leaves several avenues for further improvements in 3-party SC-ORAM both on the level of system implementation and algorithm design.

Technical Overview. We base our implementation on the Shi *et. al.* [20] hierarchical two party ORAM, and use a combination of three-party OT's and secure computation (using Yao garbled circuits [24]) in order to ensure privacy in the three party setting. Our protocol follows the same technical approach of two-party SC-ORAM schemes, i.e. of providing a secure computation protocol for access and eviction algorithms in a client-server ORAM. However, the existence of a third party allows us to greatly reduce the cost of this secure computation. Our main observation is that in Binary Tree access and eviction algorithms, like that of Gordon *et. al.* [14], there is a separation in the role played by the input bits of the access or eviction circuit. Part of the bits are used to implement the logic of the circuit, but the majority are data that do not participate in the output of the logic and are, at best, just being moved between some locations based on the output of the logic. We exploit this separation in the three-party setting, by isolating the bits necessary for the logic, using Yao's garbled circuit to securely compute the logic only on those bits, and then use several variants of the (three-party) Oblivious Transfer (OT) protocol to move data to the locations pointed out by the output of the circuits. Since all these variants of OT can be implemented at a cost similar to just the *secure transmission* of the data the OT operates on, this leads to dramatic reductions in the cost of the resulting secure computation protocol. In addition, in the *access* protocol, as opposed to the *eviction*, we avoid using garbled circuits entirely, as the entire logic comes down to finding an index where two lists of n bitstrings contain a matching entry, which we implement using a three-party variant of *Conditional OT* which takes a single interaction round and costs roughly as much as encryption and transmission of these n bitstrings.

We make several modifications in the Binary-Tree ORAM of Shi *et. al.* [20] to make it more efficient for the type of operations we are interested in. We use ideas from Gentry *et. al.* [11] and Stefanov *et. al.* [21]. In particular, we make the ORAM trees more shallow, as in Gentry *et. al.* [11] by increasing how many entries in the ORAM will be mapped to each leaf in expectation and increasing the total capacity (in terms of entries) of the leaf nodes. To be more precise, for a tree that has a total capacity of 2^m entries and a capacity in each node of w, instead of having 2^m leafs in the ORAM tree, we have $\frac{2^m}{w}$ leafs instead. In order to ensure that overflow does not occur in the leafs of the tree we increase capacity of leaf nodes to $4w$. With this change we achieve linear overhead in terms of storage needed for the ORAM, meaning that now the total entries that can be stored in the ORAM are $\mathrm{O}\left(2^m\right)$, in contrast with the $\mathrm{O}\left(w \cdot 2^m\right)$ entries that the Shi *et. al.* [20] ORAM had (note that for most settings $w = \mathrm{O}\left(m\right)$). In addition, we observe that for internal nodes it is not necessary to increase their capacity, since the overflow of internal nodes is mandated by a difference probabilistic process that the one of leaf nodes. In contrast with the approach used in Gentry *et. al.* [11], by only increasing the capacity of leaf nodes, we avoid doubling the bandwidth needed by the ORAM protocol (which is what happened in Gentry *et. al.* [11], since they increase the capacity of all nodes, whether they are leafs or internal nodes).

We adopt the idea of eviction through a single path introduced by Gentry *et. al.* [11]. The main problem we identified in the single path eviction, is that both Gentry *et. al.* [11] and Stefanov *et. al.* [21] evict all entries in all nodes of the path, as far down in the path as they can go. Although this is easy to do in a client-server ORAM where the client retrieves the whole path and performs all operations in the clear text data, in the setting of secure computation on the secret shared data, such eviction is very costly. For this reason, we modify the eviction to only evict at most two items from each node to the next node in the path, provided such items exist. This operation is limited enough to allow for simple garbled circuits. Moreover, it is an oblivious operation in the sense that always two entries are evicted to the next level (we evict empty entries if appropriate entries do not exist), which allows for its simple 3-party implementation. We choose not to increase the fun-out of nodes as Gentry et al. [11] do, since this would complicate both our circuits and the rest of the protocols. We also choose to avoid using the overflow cache used in Path ORAM of Stefanov et al. [21] in order to decrease the total space requirements for their ORAM, deciding instead to experiment at first with a design which maximally simplifies the eviction logic and the associated garbled circuits.

Lastly we briefly explain why it seems difficult to construct a (3-server) SC-ORAM scheme with competitive efficiency based on the *two-server* Client-Server ORAM of Lu and Ostrovsky [17]. Indeed, since in either 3-server or 2-server setting of SC-ORAM we rely on non-collusion between any two parties, we could use a two-server version of the underlying client-server ORAM. Because the two-server ORAM of [17] achieves $O(m)$ amortized overhead per query, the asymptotic running time of the SC-ORAM protocol based on this two-server ORAM could also be only linear in m, which would beat (at least asymptotically) anything based on the $O(m^3)$ single-server ORAM schemes of [20, 11, 21]. However, the Lu-Ostrovsky two-server ORAM has some features which adversely affect the practicality of the resulting SC-ORAM protocol. It is a hierarchical construction in the spirit of [13] with $O(m)$ levels where the i-th level contains

$O(2^{i-1})$ encrypted memory entries. After every 2^t RAM accesses, the construction re-shuffles the first 2^t levels of the hierarchy, incurring $O(2^t)$ cost, which makes the running time of each access highly uneven. The scheme has other "MPC unfriendly" properties, e.g. the client's retrieval algorithm, which has to be emulated with secure computation, acts differently at a given level depending on whether the item has been found in a higher level. Also, the scheme seems to need oblivious computation of a PRF with secret-shared inputs (and possibly also outputs), and the currently best protocol for such OPRF evaluation uses $O(t)$ exponentiations for a t-bit domain [14].

2 Baseline Client-Server ORAM Protocol

We describe the Client-Server ORAM which forms the basis of our three-party SC-ORAM protocol presented in Section 4. The scheme explained below is a variant of the *Binary Tree* client-server ORAM scheme of Shi et al. [20], with some optimizations adopted from a variant given by Gentry et al. [11]. The design principle behind our variant of the Binary Tree ORAM is two-fold: First, we want the client's algorithm to be "secure computation friendly". Secondly, we want to do so without increasing the parameters of the Binary Tree ORAM scheme (e.g. tree depth, tree size, tuple size, bucket size, etc.), as this would also negatively affect the efficiency of the resulting three-party ORAM emulator protocol.

ORAM Forest. Let D be an array of $|\mathsf{D}| \leq 2^m$ records, where $\mathsf{D}[\mathrm{N}]$ for every m-bit address N is a bitstring of fixed length d. For a given security parameter λ and cryptographic security parameter κ, the ORAM protocol needs only $\mathrm{O}(\kappa)$ persistent storage and $\mathrm{O}(m \cdot d)$ transient storage for client C, and $\mathrm{O}(2^m \cdot d) = \mathrm{O}(|D|)$ persistent storage of server S. Given m, d, λ, κ, an ORAM implementation is parametrized by two additional parameters $w, 2^\tau$ where $w = \max(\lambda, m)$ and τ is an integer divisor of m. Let $h = \log_{2^\tau} 2^m = m/\tau$. Server S stores an *ORAM Forest*, $\mathrm{OTF} = (\mathrm{OT}_0, \mathrm{OT}_1, \ldots, \mathrm{OT}_h)$. Each *ORAM Tree* OT_i for $i > 0$ is a binary tree of height $\mathrm{d}_i = i\tau - \log w$ (if $i\tau \leq \log w$ then $\mathrm{d}_i = 0$). Let $\mathrm{N} = [\mathrm{N}^{(1)}|\ldots|\mathrm{N}^{(h)}]$ be the parsing of N into τ-bit segments and let $\mathrm{N}^i = [\mathrm{N}^{(1)}|\ldots|\mathrm{N}^{(i)}]$ be N's prefix of length τi. The last ORAM tree OT_h implements a look-up table F_h s.t. $\mathrm{F}_h(\mathrm{N}) = \mathsf{D}[\mathrm{N}]$, but the efficient retrieval of $\mathrm{F}_h(\mathrm{N})$ from OT_h is possible only given a *label* $\mathrm{L}^h \in \{0,1\}^{\mathrm{d}_h}$ which defines the leaf (or path, see below) in OT_h where value $\mathrm{F}_h(\mathrm{N})$ is stored. The way this label L^h can be found is that each ORAM tree OT_i for $i < h$ implements a look-up table F_i which maps N^i to label $\mathrm{L}^{i+1} \in \{0,1\}^{\mathrm{d}_{i+1}}$, and it is an invariant of OTF that for each i, $\mathrm{L}^{i+1} = \mathrm{F}_i(\mathrm{N}^i)$ defines a leaf (or path, see below) in OT_{i+1} which contains values of F_{i+1} on arguments $\mathrm{N}^{i+1} = \mathrm{N}^i|\mathrm{N}^{(i+1)}$ for all $\mathrm{N}^{(i+1)} \in \{0,1\}^\tau$. Therefore the ORAM *access* algorithm on input N proceeds recursively: The base tree OT_0 is a single vertex which contains values $\mathrm{L}^1 = \mathrm{F}_0(\mathrm{N}^1)$ for all $\mathrm{N}^1 \in \{0,1\}^\tau$, so the algorithm first retrieves $\mathrm{L}^1 = \mathrm{F}_0(\mathrm{N}^1)$ from OT_0, then using L^i it retrieves $\mathrm{L}^{i+1} = \mathrm{F}_i(\mathrm{N}^{i+1})$ from OT_i for $i = 1, \ldots, h-1$, and finally using L^h it retrieves $\mathrm{F}_h(\mathrm{N}) = \mathsf{D}[\mathrm{N}]$ from OT_h. For notational convenience we can think of an ORAM forest OTF as implementing function $\mathrm{F_{OTF}} : \{0,1\}^m \rightarrow \{0,1\}^{\mathrm{d}_1} \times \{0,1\}^{\mathrm{d}_2} \times \ldots \times \{0,1\}^{\mathrm{d}_h} \times \{0,1\}^d$, where $\mathrm{F_{OTF}}(\mathrm{N}) = (\mathrm{F}_0(\mathrm{N}^1), \mathrm{F}_1(\mathrm{N}^2), \ldots, \mathrm{F}_{h-1}(\mathrm{N}^h), \mathrm{F}_h(\mathrm{N}^h))$.

We now explain how F_i values are stored in binary tree OT_i. Let $\{0,1\}^{<m}$ denote the binary strings of length from 0 to $m-1$. The nodes of OT_i for $i > 0$ are formed as follows: Each *internal node*, indexed by $j \in \{0,1\}^{<\mathrm{d}_i}$, stores a *bucket* B_j, while each *leaf node*, indexed by $j \in \{0,1\}^{\mathrm{d}_i}$, is a set of *four* buckets $(\mathrm{B}_{j00}, \mathrm{B}_{j01}, \mathrm{B}_{j10}, \mathrm{B}_{j11})$. Tree OT_0 is an exception because it consists of a single root node $\mathrm{B_{root}}$. (Constant 4 is chosen somewhat arbitrarily and can be adjusted, see Section 5.) Each bucket B_j is stored at node j in OT_i encrypted under the master key held by C. Each bucket is an array of w *tuples* of the form $\mathrm{T}^i = (\mathrm{fb}_i, \mathrm{N}^i, \mathrm{L}^i, \mathrm{A}^i)$ where $\mathrm{fb}_i \in \{0,1\}$, $\mathrm{N}^i \in \{0,1\}^{i\tau}$, $\mathrm{L}^i \in \{0,1\}^{\mathrm{d}_i}$, and A^i for $i < h$ is an array containing 2^τ labels $\mathrm{L}^{i+1} \in \{0,1\}^{\mathrm{d}_{i+1}}$, while A^h is a record in D. The above invariant is maintained if for every N (or, if D is a sparse array, only for those N's for which $\mathsf{D}[\mathrm{N}]$ is non-empty), there is a sequence of labels $(\mathrm{L}^1, \ldots, \mathrm{L}^h)$ (assume $\mathrm{L}^0 = 0$ and $\mathrm{N}^{(0)} = 0^\tau$) s.t. each OT_i contains a unique tuple of the form $\mathrm{T} = (1, \mathrm{N}^{(i)}, \mathrm{L}^i, \mathrm{A}^i)$ for some A^i and (1) this tuple is contained in some bucket along the path from the root to the leaf L^i in OT_i; and (2) if $i < h$ then $\mathrm{A}^i\left[\mathrm{N}^{(i+1)}\right] = \mathrm{L}^{i+1}$, and $\mathrm{A}^h = \mathsf{D}[\mathrm{N}]$. Observe that $(\mathrm{L}^1, \ldots, \mathrm{L}^h, \mathsf{D}[\mathrm{N}]) = \mathrm{F_{OTF}}[\mathrm{N}]$.

Access Procedure. To access location N in D, the client C performs the following loop sequentially for each $i = 0, ..., h$, given the recursively obtained label L^i: C sends L^i to the server S, who retrieves and sends to C the encrypted path $\overline{\mathrm{P}_{\mathrm{L}^i}}$ in tree OT_i from the root to the leaf L^i. C decrypts $\overline{\mathrm{P}_{\mathrm{L}^i}}$ using its master key into a corresponding plaintext path $\mathrm{P}_{\mathrm{L}^i}$ which is a sequence of buckets $(\mathrm{B}_1, \ldots, \mathrm{B}_n)$ for $n = \mathrm{d}_i + 4$ (recall that a leaf contains 4 buckets), finds the unique tuple T^i in this bucket sequence of the form $\mathrm{T}^i = (1, \mathrm{N}^{(i)}, \mathrm{L}^i, \mathrm{A}^i)$ and computes either label $\mathrm{L}^{i+1} = \mathrm{A}^i\left[\mathrm{N}^{(i+1)}\right]$ if $i < h$, or, if $i = h$, outputs A^i as the record $\mathsf{D}[\mathrm{N}]$.

Note that protocol reveals the vector of labels $(\mathrm{L}^1, \ldots, \mathrm{L}^h)$ to S. Therefore after each access C picks new random labels $((\mathrm{L}^1)', \ldots, (\mathrm{L}^h)')$, where $(\mathrm{L}^i)'$ is a random bitstring of length d_i, and OTF needs to be updated so that $\mathrm{F_{OTF}}(\mathrm{N}) = ((\mathrm{L}^1)', \ldots, (\mathrm{L}^h)', \mathsf{D}[\mathrm{N}])$. To do this, C erases the tuple $\mathrm{T}^i = (1, \mathrm{N}^{(i)}, \mathrm{L}^i, \mathrm{A}^i)$ in the bucket in which it was found (by flipping the fb field to 0), replaces L^i with $(\mathrm{L}^i)'$, sets $\mathrm{A}^i\left[\mathrm{N}^{(i+1)}\right]$ to $(\mathrm{L}^{i+1})'$, and inserts this modified tuple $(\mathrm{T}^i)'$ into the root bucket B_1. C then re-encrypts the buckets and sends the new encrypted path $(\overline{\mathrm{P}_{\mathrm{L}^i}})'$ to S to insert in place of $\overline{\mathrm{P}_{\mathrm{L}^i}}$ in OT_i.

Constrained Eviction Strategy. The above procedure works except for the fact that the root bucket fills up after w accesses. To ensure that this does not happen (with overwhelming probability), an eviction step is interjected into the access protocol before C re-encrypts $\overline{\mathrm{P}_{\mathrm{L}^i}}$ and sends it back to S. The aim of an eviction process is to move each tuple $\mathrm{T} = (\mathrm{N}^{(i)}, \mathrm{L}, \mathrm{A})$ in an internal node of tree OT_i down towards its "destination leaf" L. Since in access C reads only the tuples in path $\mathrm{P}_{\mathrm{L}^i}$, this will only be done to the tuples found in the internal buckets of this path. Moreover, because we want our eviction strategy to be *secure-computation friendly*, i.e. to be as easy to compute securely in our three-party setting as possible, we restrict this eviction principle in two ways: we will attempt to move *at most two* tuples down in every path, and we will move them only one bucket down. Both of these two restrictions make no sense in the case of a client-server ORAM, where C sees all the buckets in $\mathrm{P}_{\mathrm{L}^i}$ in the cleartext and can move all the tuples in this path as far down as they can go. However, in the context of the multi-party SC-ORAM protocol these constraints make the data movement pattern in the eviction process more predictable, and hence more easily implemented via a secure computation protocol which *does not* implement the whole eviction as a single (securely-computed) circuit.

Technically, this "constrained" eviction strategy works as follows: Consider a bucket B_j corresponding to an internal node in $\mathrm{P}_{\mathrm{L}^i}$, i.e. for $j \leq \mathrm{d}_i$. We say that tuple $\mathrm{T} = (\mathrm{fb}, \mathrm{N}^{(i)}, \mathrm{L}, \mathrm{A})$ in B_j is *moveable* down the path towards leaf L^i if $\mathrm{fb} = 1$ and the j-th bit in its label field L matches the j-th bit of leaf L^i. In every bucket B_j for $j \leq \mathrm{d}_i$ we choose two random tuples which are moveable down towards leaf L^i. If there are no such tuples then we choose two random empty tuples instead (if they exist), and if there is only one then the second one is chosen as a random empty tuple (if it exists). In addition, we choose two random empty tuples (if they exist) among the $4w$ tuples contained in the four buckets B_j contained in the leaf node in $\mathrm{P}_{\mathrm{L}^i}$, i.e. for $j = \mathrm{d}_i + 1, \ldots, \mathrm{d}_i + 4$. Then, for each $i \leq \mathrm{d}_i$, we take the two chosen tuples in bucket B_j and move them to the two spaces vacated in bucket B_{j+1} (except for $j = \mathrm{d}_i$ where the two chosen spaces in the level below can be in any of the buckets $\mathrm{B}_{\mathrm{d}_i+1}, \ldots, \mathrm{B}_{\mathrm{d}_i+4}$).

Eviction fails in case of a bucket overflow i.e. if (1) some internal bucket in $\mathrm{P}_{\mathrm{L}^i}$ does not contain two tuples which are either empty or moveable; or (2) the four buckets corresponding to the leaf node do not contain two empty tuples. As we argue in Section 5, both probabilities are negligible for $w = \mathsf{O}(m)$, assuming the number *accesses* to D is polynomial in $|\mathsf{D}|$.

Notation. In the 3-party SC-ORAM secure protocol for this Client-Server ORAM we will use notation $|\mathrm{P}^i_{\mathrm{L}}|$ to denote the length of any path in tree OT_i, and $|\mathrm{T}^i|$ to denote the length of any tuple in such tree. Note that $|\mathrm{P}^i_{\mathrm{L}}| = (\mathrm{d}_i{+}4)\cdot w \cdot |\mathrm{T}^i|$ and $|\mathrm{T}^i| = 1{+}i\cdot\tau{+}\mathrm{d}_i{+}2^\tau\cdot\mathrm{d}_{i+1}$ if $i < h$ while $|\mathrm{T}^h| = 1{+}h\cdot\tau{+}\mathrm{d}_h{+}d$ because $\mathrm{T}^i = (\mathrm{fb}, \mathrm{N}^i, \mathrm{L}^i, \mathrm{A}^i)$, $|\mathrm{N}^i| = i \cdot \tau$, $|\mathrm{L}^i| = \mathrm{d}_i$, and A^i for $i < h$ is an array holding 2^τ next-level leaf labels of length d_{i+1}, while $\mathrm{A}^h = \mathsf{D}[\mathrm{N}]$ and $|\mathsf{D}[\mathrm{N}]| = d$.

3 Three-Party Protocol Building Blocks

Our SC-ORAM protocols Access, $\mathsf{PostProcess}$, and $\mathsf{Eviction}$ of Section 4 rely on several variants of Oblivious Transfer (OT) or Conditional Disclosure of Secrets (CDS) protocols which we detail here. The efficiency of our SC-ORAM protocol relies on the fact that all these OT variants, including the OT variant employed in Yao's Garbled Circuit (GC) protocol, have significantly cheaper realizations in the 3-party setting. All presented protocols assume secure channels, although in many instances encryption overhead can be eliminated with simple protocol changes, e.g. using pairwise-shared keys in PRG's and PRF's.

Notation. Let κ denote the cryptographic security parameter, which will assume is both the key length and the block length of a symmetric cipher. Let G^ℓ be a

PRG which outputs ℓ-bit strings given a seed of length κ. Let $\mathsf{F}^{\ell}_{\mathsf{k}}$ be a PRF which maps domain $\{0,1\}^{\kappa}$ onto $\{0,1\}^{\ell}$, for k randomly chosen in $\{0,1\}^{\kappa}$. We will write G and F_{k} when $\ell = \kappa$. In our implementation both F and G are implemented using counter-mode AES. If party A holds value a and party B holds value b s.t. $a \oplus b = v$ then we call pair (a, b) an "A/B secret-sharing" of v and denote it as $(\mathrm{s_A}[v], \mathrm{s_B}[v])$. Whenever we describe an intended output of some protocol as A/B secret-sharing of value v, we mean this to be a *random* xor-sharing of v i.e. pair $(r, r \oplus v)$ for r random in $\{0,1\}^{|v|}$. Let $s[j]$ denote the j-th bit of bitstring s, and let $[n]$ denote integer range $\{1, ..., n\}$.

3-Party Variants of Oblivious Transfer. We use several variants of the Oblivious Transfer problem in our three-party setting, namely Secret-Shared Conditional OT, $\mathsf{SS\text{-}COT}^{[N]}$, Secret-Shared Index OT, $\mathsf{SS\text{-}IOT}^{[2^{\tau}]}$, Shuffle OT, $\mathsf{XOT}\begin{bmatrix}N\\k\end{bmatrix}$, Secret-Shared Shuffle OT, $\mathsf{SS\text{-}XOT}\begin{bmatrix}N\\k\end{bmatrix}$, and Shift OT, Shift. We explain the functionality and our implementations of these OT variants below. The common feature of all our implementations is that they require one or two messages both in the pre-computation phase and in the online phase (except of Secret-Shared Shuffle OT which sends four messages), and the computational cost of each protocol for each party, both in pre-computation and on-line, is within a factor of 2 of the cost of secure transmission of the sender's inputs. We stress that all protocol we present form *secure computation* protocols of the corresponding functionalities assuming an *honest-but-curious adversary*, *secure channels*, and a *single corrupted player*. In each case the security proof is a straightforward simulation argument.

Algorithm 1 Secret-Shared Conditional OT Protocol $\mathsf{SS\text{-}COT}^{[N]}(\mathrm{S}, \mathrm{R}, \mathrm{H})$

```
Input: S's input $(m_1, ..., m_N)$ and $(a_1, ..., a_N)$, H's input $(b_1, ..., b_N)$.
Output: R outputs pairs $(t, m_t)$ s.t. $a_t = b_t$.
Parameters: $\ell$ and $\ell'$ s.t. $|m_t| = \ell$ and $|a_t| = |b_t| = \ell' \leq \kappa$ for all $t$.
Pre-computation phase: $\mathrm{S}, \mathrm{H}$ share PRF $\mathsf{F}$ keys $\mathsf{k}, \mathsf{k}'$ and $\kappa$-bit random nonces $r_1, ..., r_N$.
1: S sends $\{(e_t, v_t) = (\mathsf{G}^{\ell}(\mathsf{F}^{\kappa}_{\mathsf{k}}(x_t)) \oplus m_t, \mathsf{F}^{\kappa}_{\mathsf{k}'}(x_t))\}_{t=1}^{N}$ to R where $x_t = r_t \oplus [a_t | 0^{\kappa - \ell'}]$.
2: H sends $\{(p_t, w_t) = (\mathsf{F}^{\kappa}_{\mathsf{k}}(y_t), \mathsf{F}^{\kappa}_{\mathsf{k}'}(y_t))\}_{t=1}^{N}$ to R where $y_t = r_t \oplus [b_t | 0^{\kappa - \ell'}]$.
3: R outputs $(t, m_t)$ where $m'_t = e_t \oplus \mathsf{G}^{\ell}(p_t)$ for each $t$ s.t. $v_t = w_t$.
```

Secret-Shared Conditional OT, $\mathsf{SS\text{-}COT}^{[N]}(\mathrm{S}, \mathrm{R}, \mathrm{H})$, is a protocol where S inputs two lists, $(m_1, \ldots, m_N)$ and $(a_1, \ldots, a_N)$, H inputs a single list $(b_1, \ldots, b_N)$, and the protocol's goal is for R to output all pairs (t, m_t) s.t. $a_t = b_t$. This is a very close variant of the Conditional Disclosure of Secrets protocol of [12], and it can be implemented e.g. using modular arithmetic in a prime field. In Alg 1 we provide an alternative design which uses fewer (pseudo)random bits, and hence requires fewer PRG ops in pre-computation, but uses block ciphers in the on-line phase. (The algorithm proposed here was faster in our implementation even in the on-line stage.) S and H share two PRF keys k, k', and for each t helper H sends to R a pair $(p_t, w_t) = (\mathsf{F}_{\mathsf{k}}(b_t), \mathsf{F}_{\mathsf{k}'}(b_t))$, while S sends (e_t, v_t) where e_t is an

Algorithm 2 Shuffle OT Protocol $\mathsf{XOT}\left[\begin{smallmatrix}N\\k\end{smallmatrix}\right]$ (S, R, I)

```
Input: S's input $(m_1, ..., m_N)$ and I's input $(i_1, ..., i_k)$ and $(\delta_1, ..., \delta_k)$.
Output: R's output $(z_1, ..., z_k)$ s.t. $z_\sigma = m_{i_\sigma} \oplus \delta_\sigma$ for all $\sigma$.
Parameters: Let $|m_t| = \ell$ for all $t$.
Pre-computation phase: I and S pick a random permutation $\pi$ on $(1, ..., N)$ and a
sequence of $\ell$-bit random pads $r_1, ..., r_N$.
1: S sends $(a_1, ..., a_N) = (m_{\pi(1)} \oplus r_1, ..., m_{\pi(N)} \oplus r_N)$ to R.
2: I sends $(j_1,..., j_k) = (\pi^{-1}(i_1),..., \pi^{-1}(i_k))$ and $(p_1,..., p_k) = (r_{j_1} \oplus \delta_1,..., r_{j_k} \oplus \delta_k)$ to R.
3: R outputs $(z_1, ..., z_k) = (a_{j_1} \oplus p_1,..., a_{j_k} \oplus p_k)$.
   (Note that $z_\sigma = (m_{\pi(j_\sigma)} \oplus r_{j_\sigma}) \oplus (r_{j_\sigma} \oplus \delta_\sigma) = m_{i_\sigma} \oplus \delta_\sigma$ because $\pi(j_\sigma) = i_\sigma$.)
```

xor of message m_t and $\mathsf{G}(\mathsf{F}_\mathsf{k}(a_t))$ while $v_t = \mathsf{F}_{\mathsf{k}'}(a_t)$. For each t receiver R checks if $v_t = w_t$, and if so then it concludes that $a_t = b_t$ and outputs $m_t = e_t \oplus \mathsf{G}(p_t)$. To protect against collisions in (short) a_t, b_t values both within each protocol instance and across protocol instances each a_t and b_t is xor-ed by respectively S and H by a pre-shared one-time κ-bit random nonce r_t, with all nonces derived via a PRG on a seed shared by S and H.

Secret-Shared Index OT, $\mathsf{SS\text{-}IOT}^{[2^\tau]}(\mathrm{S}, \mathrm{R}, \mathrm{H})$, is a close variant of the Secret-Shared Conditional OT, where S holds a list of messages $(m_0, \ldots, m_{N\text{-}1})$ for $N = 2^\tau$ and an index share $j_\mathrm{S} \in \{0,1\}^\tau$ while H holds the other share $j_\mathrm{H} \in \{0,1\}^\tau$, and the aim of the protocol is for R to output (j, m_j) s.t. $j = j_\mathrm{S} \oplus j_\mathrm{H}$. Our protocol for $\mathsf{SS\text{-}IOT}^{[2^\tau]}$ executes similarly to $\mathsf{SS\text{-}COT}^{[N]}$ except H sends only two values, $(p, v) = (\mathsf{F}_\mathsf{k}(j_\mathrm{H}), \mathsf{F}_{\mathsf{k}'}(j_\mathrm{H}))$ and S's messages are computed as $e_t = \mathsf{G}(\mathsf{F}_\mathsf{k}(j_\mathrm{S} \oplus t))$ and $v_t = \mathsf{F}_{\mathsf{k}'}(j_\mathrm{S} \oplus t)$. Finally, to avoid correlations across protocol instances players H and S xor their PRF inputs with a single pre-shared random κ-bit nonce r.

Shuffle OT, $\mathsf{XOT}\left[\begin{smallmatrix}N\\k\end{smallmatrix}\right]$ (S, R, I), is a protocol between sender S, receiver R, and *indicator* I, where S inputs a sequence of messages $m_1, ..., m_N$, I inputs a sequence of indexes $i_1, ..., i_k$ and a sequence of masks $\delta_1, ..., \delta_k$, and the protocol lets R output a sequence of messages $m_{i_1} \oplus \delta_1, ..., m_{i_k} \oplus \delta_k$, without leaking anything else about S's and I's inputs. See Alg. 2 for an implementation of this protocol.

Secret-Shared Shuffle OT, $\mathsf{SS\text{-}XOT}\left[\begin{smallmatrix}N\\k\end{smallmatrix}\right]$ (A, B, I), involves indicator I and two parties A and B. It is a close variant of the Shuffle OT above, where I holds indexes $i = (i_1, ..., i_k)$, the pads $\delta_1, ..., \delta_k$ are all set to zero, and both inputs $m_1, ..., m_k$ and outputs $m_{i_1}, ..., m_{i_k}$ are secret-shared by A and B. We implement this protocol with two instances of $\mathsf{XOT}\left[\begin{smallmatrix}N\\k\end{smallmatrix}\right]$. The indicator I first chooses a sequence of random masks $\delta = (\delta_1, ..., \delta_k)$, and inputs i, δ into both instances, where the first instance runs on A's input $(\mathrm{s_A}[m_1], ..., \mathrm{s_A}[m_n])$, and lets B output $(\mathrm{s_A}[m_{i_1}] \oplus \delta_1, ..., \mathrm{s_A}[m_{i_k}] \oplus \delta_k)$, while the second instance runs on B's inputs $(\mathrm{s_B}[m_1]$, and lets A output $(\mathrm{s_B}[m_{i_1}] \oplus \delta_1, ..., \mathrm{s_B}[m_{i_k}] \oplus \delta_k)$. It's easy to see that these outputs form a randomized A/B secret-sharing of $(m_{i_1}, ..., m_{i_k})$.

Shifting a Secret-Shared Sequence, $\mathsf{Shift}(\mathrm{A}, \mathrm{B}, \mathrm{H})$. As the access protocol traverses the forest of ORAM trees $\mathrm{OTF} = (\mathrm{OT}_0, \mathrm{OT}_1, ..., \mathrm{OT}_h)$, D and E recover the secret-sharing of path $\mathrm{P}_{\mathrm{L}^i}$, for $i = 1, ..., h$, and make several modifications to it. In particular, the buckets in the path are rotated by a random shift σ_i known to D and E. In the eviction protocol on this retrieved path we need a sub-protocol Shift to reverse this shift by transforming the secret-sharing of this path, which is a sequence of buckets, to a (fresh) secret-sharing of the same buckets but rotated back by σ_i positions. An inexpensive implementation of this task relies on the fact that in our three-party setting player D can act as a "helper" party and create, in pre-computation, *correlated* random inputs for E and C, which allows for an on-line protocol which consists of a few xor operations and a transmission of a single $|\mathrm{P}_{\mathrm{L}^i}|$-bit message from C to E. Since this protocol is a very close variant of protocol $\mathsf{SS\text{-}IOT}^{[N]}$ given above we omit its description.

Yao's Garbled Circuit on Secret-Shared Inputs. The last component used in our ORAM construction is protocol $\mathsf{GC}[\mathsf{F}](\mathrm{A}, \mathrm{B}, \mathrm{R})$, a Yao's garbled circuit solution for secure computation of an arbitrary function [24], executing on public inputs a circuit of function F, where the inputs X to this circuit are secret-shared between A and B, i.e. A inputs $\mathrm{s}_{\mathrm{A}}[X]$ and B inputs $\mathrm{s}_{\mathrm{B}}[X]$, and the protocol lets R compute $\mathsf{F}(X)$. We stress that even though we do use Yao's garbled circuit evaluation as a subprotocol in our SC-ORAM scheme, we use it sparingly, and the computation involved is comparable, for realistic m values, to the necessary cost of decryption of paths $\mathrm{P}_{\mathrm{L}^i}$ retrieved by the underlying Binary-Tree Client-Server ORAM scheme. The protocol is a simple modification of the delivery of the input-wire keys in Yao's protocol, adopted to the setting where the input X is secret-shared by parties A and B, while the third party R will compute the garbled circuit and get the $\mathsf{F}(X)$. Let $n = |X|$ and let κ be the bitlength of the keys used in Yao's garbled circuit. In the off-line stage either A or B, say party A, prepares the garbled circuit for function F and sends it to R, and then for each input wire key pair (K_i^0, K_i^1) created by Yao's circuit garbling procedure, A picks random Δ_i in $\{0,1\}^\kappa$, computes $(A_i^0, A_i^1) = (\Delta_i, K_i^0 \oplus K_i^1 \oplus \Delta_i)$ and $(B_i^0, B_i^1) = (K_i^0 \oplus \Delta_i, K_i^1 \oplus \Delta_i)$, and sends (B_i^0, B_i^1) to B. (To optimize pre-computation A can send to B a random seed from which $\{K_i^0, K_i^1, \Delta_i\}_{i=1}^n$ can be derived via a PRG.) In the on-line phase, for each $i = 1, ..., n$, party A on input bit $a = \mathrm{s}_{\mathrm{A}}[X_i]$ sends A_i^a to R, while party B on input bit $b = \mathrm{s}_{\mathrm{B}}[X_i]$ sends B_i^b. For each $i = 1, ..., n$, party R computes $K_i = A_i \oplus B_i$ for A_i, B_i received respectively from A and B, and then runs Yao's evaluation procedure inputting keys $K_1, ..., K_n$ into the garbled circuit received for F. Observe that $A_i \oplus B_i = K_i^v$ for $v = a \oplus b$, and hence if a, b is the XOR secret-sharing of the i-th input bit, i.e. if $a \oplus b = X_i$, then $K_i = K_i^0$ if $X_i = 0$ and $K_i = K_i^1$ if $X_i = 1$. The protocol is secure thanks to the random pad Δ_i, because for every X_i and every possible sharing (a, b) of X_i, values (A_i, B_i) sent to R are distributed as two random bitstrings s.t. $A_i \oplus B_i = K_i^v$ for $v = X_i$.

4 Three-Party SC-ORAM Protocol

We describe our three-party SC-ORAM protocol, which is a three-party secure computation of the Client-Server ORAM of Section 2. We refer to the three parties involved as C, D, and E. The basic idea for the protocol is to secret-share the datastructure OTF between two servers D and E, and have these two parties implement the Server's algorithm of the Client-Server ORAM scheme of Section 2, while the corresponding Client's algorithm will be implemented with a three-party secure computation involving parties $\mathrm{C}, \mathrm{D}, \mathrm{E}$. In the description below we combine these two conceptually separate parts into a single protocol, but almost all of the protocol implements the three-party computation of the ORAM Client's algorithm, as the Server's side of this Client-Server ORAM consists only of retrieving (the shares of) path $\mathrm{P}_{\mathrm{L}^i}$ from (the shares of) the i-th tree OT_i at the beginning of i-th iteration of the access procedure, and then writing (the shares of) a new path $\mathrm{P}^{\diamond}_{\mathrm{L}^i}$ in place of (the shares of) $\mathrm{P}_{\mathrm{L}^i}$ at the end.

Given this secret-sharing scenario, the task of the three-party SC-ORAM protocol is to securely compute the following two functionalities:

1. The *access* functionality computes the next-tree label $\mathrm{L}^{i+1} = \mathrm{F}_i(\mathrm{N}^{i+1})$ given the D/E secret-sharing of path $\mathrm{P}_{\mathrm{L}^i}$, for $\mathrm{L}^i = \mathrm{F}_{i-1}(\mathrm{N}^i)$ and the D/E secret-sharing of address prefix N^{i+1};
2. The *eviction* functionality computes the D/E secret-sharing of path $\mathrm{P}^{\diamond}_{\mathrm{L}^i}$ output by the eviction algorithm applied to the D/E secret-shared path $\mathrm{P}_{\mathrm{L}^i}$, after the tuple containing the label identified by the access functionality is moved to the root node.

Both tasks can be computed using standard secure computation techniques but the protocol we show beats a generic one by a few orders of magnitude, and comes close to the computation cost of the underlying Client-Server ORAM itself. Note that the i-th iteration of the Client-Server ORAM needs a Server-to-Client transmission and decryption of path $\mathrm{P}_{\mathrm{L}}\mathrm{L}^i$ and then encryption and Client-to-Server transmission of path $\mathrm{P}^{\diamond}_{\mathrm{L}^i}$. Therefore the base-line cost we want the SC-ORAM to come close to are $h+1$ rounds of Client-Server interaction with $2 \cdot |\mathrm{P}^i_{\mathrm{L}}|$ bandwidth and $(2/\kappa) \cdot |\mathrm{P}^i_{\mathrm{L}}|$ block cipher operations for $i = 0, ..., h$. The main idea which allows us to come close to these parameters is that if the inputs to either access or eviction functionalities, secret-shared by two parties, e.g. D and E, are shifted/permuted/rotated/masked in an appropriate way, then the correspondingly shifted/masked outputs of these functionalities can be revealed to the third party, e.g. C.

In the 3-party setting we separate the Client-Server access/eviction protocols into Access, PostProcess, and Eviction. Protocol Access contains all parts of the client-server access which have to be executed *sequentially*, i.e. the retrieval of sequence $\mathrm{F}_{\mathrm{OTF}}(\mathrm{N}) = (\mathrm{L}^1, \mathrm{L}^2, ..., \mathrm{L}^h, \mathsf{D}[\mathrm{N}])$ done by sequential identification (and removal from the OT_i trees) of the tuple sequence $(\mathrm{T}^1, \mathrm{T}^2, ..., \mathrm{T}^h)$ where T^i is defined as path $\mathrm{P}_{\mathrm{L}}\mathrm{L}^i$ of tree OT_i whose address field is equal to N's prefix N^i and whose A field contains label L^{i+1} at position $\mathrm{N}^{(i+1)}$. Protocol PostProcess performs cleaning-up operations on each tuple T^i in this tuple sequence, by modifying its label field from L^i to $(\mathrm{L}^i)'$ and modifying the label held at $\mathrm{N}^{(i+1)}$-th position in the A^i array of this tuple from L^{i+1} to $(\mathrm{L}^{i+1})'$. Importantly, the PostProcess and Eviction protocols can be done *in parallel* for all trees OT_i, which allows for a better CPU utilization in the protocol execution.

Algorithm 3 Protocol $\mathsf{Access}[i]$ - Oblivious Retrieval of Next Label

```
Input: D, E's inputs: label $L^i$ and secret-sharing of $OT_i$ and $N^{i+1} = [N^i|N^{(i+1)}]$;
Output: (1) C outputs $L^{i+1} = A^i[N^{(i+1)}]$ where $A^i$ is the A field of tuple $T^i$ in $P_{L^i}$ whose N field matches $N^i$; (2) C and E output a secret-sharing of $T^i$ and $P^*_{L^i} = \mathsf{Rot}^{[\sigma,\delta,\rho]}(P'_{L^i})$, where $P^*_{L^i}$ is $P_{L^i}$ without tuple $T^i$; (3) D & E output $\sigma, \rho$;
Pre-computation phase: D & E's input: $(\sigma, \delta, \rho, p) \leftarrow [d_i{+}4] \times [w] \times \{0,1\}^\tau \times \{0,1\}^{|P_{L^i}|}$;
Parameters: $n = w(d_i{+}4)$.
1: D retrieves share $s_D[P_{L^i}]$ from $s_D[OT_i]$ and sets $s_D[\mathsf{Rot}^{[\sigma,\delta,\rho]}(P_{L^i})]$ as the result of the three data-rotations using shifts $(\sigma, \delta, \rho)$ applied to $(s_D[P_{L^i}] \oplus p)$. E computes $s_E[\mathsf{Rot}^{[\sigma,\delta,\rho]}(P_{L^i})]$ in the corresponding way.
2: D sends $s_D[\mathsf{Rot}^{[\sigma,\delta,\rho]}(P_{L^i})]$ and $s_D[N^{i+1}] = (s_D[N^i]|s_D[N^{(i+1)}])$ to C.
3: D and E isolate in their shares of $\mathsf{Rot}^{[\sigma,\delta,\rho]}(P_{L^i})$ a vector of shares of pairs $(fb_j, N_j)$ for $j = 1, ..., n$ of fb and N fields of all tuples in this (rotated) path. E also isolates in $s_E[\mathsf{Rot}^{[\sigma,\delta,\rho]}(P_{L^i})]$ shares $(s_E[\mathsf{Rot}^{[\rho]}(A_1)], ..., s_E[\mathsf{Rot}^{[\rho]}(A_n)])$ of the A field of all tuples. The parties then run $\mathsf{SS\text{-}COT}^{[n]}(E, C, D)$ on E's input $(m_1, ..., m_n)$ and $(a_1, ..., a_n)$ and D's input $(b_1, ..., b_n)$ where $m_t = s_E[\mathsf{Rot}^{[\rho]}(A_t) \oplus y]$, $a_t = s_E[fb_t|N_t] \oplus [0|s_E[N^i]]$, and $b_t = s_D[fb_t|N_t] \oplus [1|s_D[N^i]]$. This subprotocol outputs $(j_1, \bar{e})$ for C s.t. $[fb_{j_1}|N_{j_1}] = [1|N^i]$ and $\bar{e} = y \oplus s_E[\mathsf{Rot}^{[\rho]}(A_{j_1})]$. The client computes $z = \bar{e} \oplus \bar{d}$ where $\bar{d}$ is the A field in the $j_1$-th tuple in $s_D[\mathsf{Rot}^{[\sigma,\delta,\rho]}(P_{L^i})]$. (Note that $j_1$-th tuple in $\mathsf{Rot}^{[\sigma,\delta,\rho]}(P_{L^i})$ is equal to $T^i$, hence $A_{j_1} = A^i$ and $z \oplus y = \mathsf{Rot}^{[\rho]}(A^i)$.)
4: Parties run $\mathsf{SS\text{-}IOT}^{[2^\tau]}(E, C, D)$ on E's input $(y_0, \ldots, y_{2^\tau\text{-}1})$ and $s_E[N^{(i+1)}]$ and D's input $s_D[N^{(i+1)}] \oplus \rho_i$, which outputs pair $(j_2, y_{j_2})$ for C.
5: Each party computes its output as follows:
   – C outputs $L^{i+1} = y_{j_2} \oplus z_{j_2}$ where $z_{j_2}$ is $j_2$-th $d_{i+1}$-bit segment in $z$;
   – C and E form $(s_C[T^i], s_E[T^i])$ as $((1, s_D[N^i], 0^{d_i}, z), (0, s_E[N^i], L^i, y))$;
   – C and E form secret-sharing of $P^*_{L^i}$ by C setting its share to $s_D[\mathsf{Rot}^{[\sigma,\delta,\rho]}(P_{L^i})]$ but with the $j_1$-th tuple modified by flipping bit fb and setting its other bits at random, and E setting its share to $s_E[\mathsf{Rot}^{[\sigma,\delta,\rho]}(P_{L^i})]$;
   – D and E output $(\sigma, \rho)$.
```

Access Protocol. Protocol Access runs on D/E secret-sharing of searched-for address N and the ORAM forest OTF, and it's goal is to compute a D/E secret-sharing of record $\mathsf{D}[N]$. Protocol Access creates two additional outputs, for each $i = 0, \ldots, h$ (with some parts skipped in the edge cases of $i = 0$ and $i = h$): (1) C/E secret-sharing of the path P_{L^i} in OT_i, modified in the way we explain below, and with the tuple T^i defined above removed; and (2) whatever information needed for the $\mathsf{PostProcess}$ protocol to modify T^i into $(T^i)'$ which will be inserted into the root of OT_i in protocol $\mathsf{Eviction}$.

Protocol Access proceeds by executing loop $\mathsf{Access}[i]$ *sequentially*, see Alg. 3, for $i = 0, \ldots, h$. The inputs to $\mathsf{Access}[i]$ are: (1) D/E secret-sharing of OT_i; (2) D/E secret-sharing of address prefix $N^{i+1} = [N^i|N^{(i+1)}]$; (3) Leaf label L^i as the input of D and E (with N^0, $N^{(h+1)}$, and L^0 all empty strings). Its outputs are: (1) C's output the next leaf label $L^{i+1} = F_i(N^{i+1})$, for $i \neq h$, or the C/E secret-sharing of record $r = \mathsf{D}[N]$, for $i = h$; (2) C/E secret-sharing of tuple T^i defined above; and (3) C/E secret-sharing of path $\mathsf{Rot}^{[\sigma_i,\delta_i,\rho_i]}(P'_{L^i})$ which results from rotating the data in $\mathrm{P}_{\mathrm{L}^i}$ by three random shifts $(\sigma_i, \delta_i, \rho_i)$ known to E and D (and of removing T^i from $\mathrm{P}_{\mathrm{L}^i}$).

Data-Rotations and Conditional OT's. We first explain how E and D perform the three data-rotations on the secret-shared path $\mathrm{P}_{\mathrm{L}^i}$ retrieved from the (shares of) the i-th level ORAM tree OT_i (and randomized by D and E xor-ing the shares of $\mathrm{P}_{\mathrm{L}^i}$ retrieved from OT_i by a pre-agreed random pad). E and D pick three values during pre-processing, $\sigma_i, \delta_i, \rho_i$, at random in ranges resp. $\{1, ..., \mathrm{d}_i + 4\}$, $\{1, ..., w\}$, and $\{0,1\}^\tau$. The data-rotation defined by σ_i is performed on the bucket level, i.e. the $\mathrm{d}_i + 4$ buckets in path $\mathrm{P}_{\mathrm{L}^i}$ (recall that there are d_i internal nodes containing a bucket each and that the leaf node contains 4 buckets) are rotated clock-wise by σ_i positions. The data-rotation defined by δ_i is performed on the level of tuples within each bucket, i.e. in each of the $\mathrm{d}_i + 4$ buckets in $\mathrm{P}_{\mathrm{L}^i}$ the sequence of w tuples held in that bucket is rotated clock-wise by δ_i positions. Finally, the bit-vector ρ_i defines τ flips which will be applied to the array A in each of the $(\mathrm{d}_i + 4) \cdot w$ tuples in the path. Namely, the A field in each tuple in the path is treated as a τ-dimensional cube whose content is flipped along the j-th dimension if the j-th bit in ρ_i is 1. Such τ flips define a permutation on elements of A where an element at position t moves to position $t \oplus \rho_i$, for each $t \in \{0,1\}^\tau$. Note that E and D can perform all these data-rotations locally on their shares of the path $\mathrm{P}_{\mathrm{L}^i}$. We use $\mathsf{Rot}^{[\sigma_i, \delta_i, \rho_i]}(\mathrm{P}_{\mathrm{L}^i})$ to denote the resulting tree, and we use $\mathsf{Rot}^{[\rho_i]}(\mathrm{A})$ to denote the result of the permutation defined by $\rho_i \in \{0,1\}^\tau$ on field A as explained above. After applying these data-rotations to $\mathrm{P}_{\mathrm{L}^i}$ the parties run protocols $\mathsf{SS\text{-}COT}^{[n]}$ and $\mathsf{SS\text{-}IOT}^{[2^\tau]}$ described in Section 3, with E as the sender, D as the helper, and C as the receiver in both protocols. The goal of protocol $\mathsf{SS\text{-}COT}^{[n]}$, for $n = (\mathrm{d}_i + 4) \cdot w$, is two-fold: (1) to let C compute the index $j_1 \in \{1, ..., n\}$ where path $\mathsf{Rot}^{[\sigma_i, \delta_i, \rho_i]}(\mathrm{P}_{\mathrm{L}^i})$ contains the unique tuple T^i defined above (i.e. the tuple that contains the searched-for address prefix N^i); and (2) to create a C/E secret-sharing of this tuple. The goal of $\mathsf{SS\text{-}IOT}^{[2^\tau]}$ is to let C compute the $\mathrm{N}^{(i+1)}$-th entry in the A field of this secret-shared tuple T^i, because that field contains the next-tree label $\mathrm{L}^{i+1} = \mathrm{F}_i(\mathrm{N}^{i+1})$.

Note that D and E hold the secret-sharing of N^i and for each $t = 1, ..., n$ they also hold the shares of the address N_t in the t-th tuple in $\mathsf{Rot}^{[\sigma_i, \delta_i, \rho_i]}(\mathrm{P}_{\mathrm{L}^i})$. If D and E form values a_t and b_t as an xor of these two sharings, i.e. $a_t = \mathrm{s_E}[\mathrm{N}^i \oplus \mathrm{s_E}[\mathrm{N}_t]]$ and $b_t = \mathrm{s_D}[\mathrm{N}^i \oplus \mathrm{s_D}[\mathrm{N}_t]]$ then $a_t = b_t$ if and only if $\mathrm{N}_t = \mathrm{N}^i$, i.e. if and only if t points to a unique tuple T^i in (rotated) path $\mathsf{Rot}^{[\sigma_i, \delta_i, \rho_i]}(\mathrm{P}_{\mathrm{L}^i})$ whose address field N equals the searched-for address N^i. Therefore if D and E run the Secret-Shared Conditional OT $\mathsf{SS\text{-}COT}^{[n]}$ on $(a_1, ..., a_n)$ and $(b_1, ..., b_n)$ defined above as their condition-share vectors, then C will compute the index j_1 to the searched-for tuple T^i contained in this path. Moreover, $\mathsf{SS\text{-}COT}^{[n]}$ will also compute the secret-sharing of T^i if E picks a random pad y of length $2^\tau \cdot \mathrm{d}_{i+1}$, and defines the message vector it inputs to $\mathsf{SS\text{-}COT}^{[n]}$ as $(m_1, ..., m_n)$ where m_t is an xor of y with E's share of the A field in the t-th tuple in $\mathsf{Rot}^{[\sigma_i, \delta_i, \rho_i]}(\mathrm{P}_{\mathrm{L}^i})$. Note that the A field in any entry in the rotated path corresponds to array $\mathsf{Rot}^{[\rho_i]}(\mathrm{A})$ where A was the field of the corresponding entry in the original path. Therefore C's output in this $\mathsf{SS\text{-}COT}^{[n]}$ instance will be j_2 together with $\bar{e} = y \oplus \mathrm{s_E}[\mathsf{Rot}^{[\rho_i]}(\mathrm{A}^i)]$

where the searched-for tuple T^i is defined as $(1, \mathrm{L}^i, \mathrm{N}^i, \mathrm{A}^i)$. Finally, D can send to C its share of the whole path $\mathsf{Rot}^{[\sigma_i,\delta_i,\rho_i]}(\mathrm{P}_{\mathrm{L}^i})$, so if C computes z as an xor of $\bar{e}$ with the A field in the j_1-th tuple in $\mathrm{s_D}[\mathsf{Rot}^{[\sigma_i,\delta_i,\rho_i]}(\mathrm{P}_{\mathrm{L}^i})]$ then (z, y) form a C/E secret-sharing of $\mathsf{Rot}^{[\rho_i]}(\mathrm{A}^i)$.

It remains for us to explain how $\mathsf{SS\text{-}IOT}^{[2^\tau]}$ computes an entry in this secret-shared field that corresponds to the next-level address chunk $\mathrm{N}^{(i+1)}$, because that's the entry which contains $\mathrm{L}^{i+1} = \mathrm{F}_i(\mathrm{N}^{i+1})$. Note that E and D hold the secret-sharing of $\mathrm{N}^{(i+1)}$ and that they also hold the bit-vector ρ_i s.t. the entry at t-th position in A^i is located at position $t \oplus \rho_i$ in $\mathsf{Rot}^{[\rho_i]}(\mathrm{A}^i)$. Since L^{i+1} sits at the t-th position in A^i for $t = \mathrm{N}^{(i+1)}$, we will find if we retrieve the j_2-th entry of $\mathsf{Rot}^{[\rho_i]}(\mathrm{A}^i)$ for $j_2 = \mathrm{N}^{(i+1)} \oplus \rho_i$. Note, however, that e.g. $\mathrm{s_D}[\mathrm{N}^{(i+1)}] \oplus \rho_i$ and $\mathrm{s_E}[\mathrm{N}^{(i+1)}]$ form a secret-sharing of j_2, and therefore the Secret-Shared Index OT protocol $\mathsf{SS\text{-}IOT}^{[2^\tau]}$ executed on sharing $(\mathrm{s_D}[\mathrm{N}^{(i+1)}] \oplus \rho_i, \mathrm{s_E}[\mathrm{N}^{(i+1)}])$ and E's data vector $y = (y_0, ..., y_{2^\tau\text{-}1})$, will let C output j_2 together with the j_2-th fragment y_{j_2} of y. Since (z, y) form the secret-sharing of $\mathsf{Rot}^{[\rho_i]}(\mathrm{A}^i)$, C can compute the j_2-th entry of $\mathsf{Rot}^{[\rho_i]}(\mathrm{A}^i)$, i.e. the next-level tree label L^{i+1}, by xor-ing y_{j_2} with j_2-th fragment of $z = (z_0, ..., z_{2^\tau\text{-}1})$.

Security Argument. This protocol is a secure computation of $\mathsf{Access}[i]$ functionality. Note that D and E do not receive any messages in this protocol, while C learns D's fresh random share $\mathrm{s_D}[\mathsf{Rot}^{[\sigma_i,\delta_i,\rho_i]}(\mathrm{P}_{\mathrm{L}^i})]$ of the rotated path, the index j_1 to the location of $\mathrm{T}^i = (1, \mathrm{N}^{i+1}, \mathrm{L}^i, \mathsf{Rot}^{[\rho_i]}(\mathrm{A}^i))$ in this rotated path, string $\bar{e} = y \oplus \mathrm{s_E}[\mathsf{Rot}^{[\rho_i]}(\mathrm{A}^i)]$, the index $j_2 = \mathrm{N}^{(i+1)} \oplus \rho_i$ where L^{i+1} is held in $\mathsf{Rot}^{[\rho_i]}(\mathrm{A}^i)$, and label $\mathrm{L}^{i+1} = \mathrm{F}_i(\mathrm{N}^{i+1})$. This view can be efficiently simulated given only L^{i+1} because (1) D's share of any path retrieved from OT_i is always a fresh random string because D and E randomize the sharing of $\mathrm{P}_{\mathrm{L}^i}$ after retrieving it from OT_i; (2)j_1 is a random integer in $\{1, ..., w \cdot (\mathrm{d}_i + 4)\}$ because the buckets are rotated by random $\sigma_i \in \{1, ..., \mathrm{d}_i + 4\}$ and the tuples within each bucket are rotated by random $\delta_i \in \{1, ..., w\}$; (3) $\bar{e}$ and j_2 are random bitstrings, because so are y and ρ_i; (4) C's view of $\mathsf{SS\text{-}COT}^{[n]}$ and $\mathsf{SS\text{-}IOT}^{[2^\tau]}$ can be simulated from their outputs.

Boundary Cases. Alg. 3 shows protocol $\mathsf{Access}[i]$ for $0 < i < h$. For $i = 0$ tree OT_i contains a single node, shifts σ_0, δ_0 are not used, sub-protocol $\mathsf{SS\text{-}COT}^{[n]}$ is skipped, index j_1 is not used, and the outputs include only j_2 for C, ρ_0 for D and E, and the C/E secret-sharing of T^0 (with L^0 and N^0 set to empty strings). For $i = h$ the $\mathsf{SS\text{-}IOT}^{[2^\tau]}$ sub-protocol is skipped, shift ρ_h and index j_2 are not used, and (z, y) held by C and E form a secret-sharing of record $\mathsf{D}[\mathrm{N}]$.

Post-Process. The post-process protocol $\mathsf{PostProcess}$ transforms the C/E secret-shared tuples $\mathrm{T}^0, ..., \mathrm{T}^h$ output by Access to prepare the inputs for protocol $\mathsf{Eviction}$. It does so by executing a loop $\mathsf{PostProcessT[i]}$ in Alg. 4 *in parallel* for $i = 0, ..., h-1$ (tuple T^h is not part of this step). The goal of post-processing is to replace the L^{i+1} value which sits at the j_2-th position in the A field of the *secret-shared* tuple T^i (where j_2 is an index C learns in $\mathsf{Access}[i]$), with the *secret-shared* value $(\mathrm{L}^{i+1})'$. In other words, we need to inject a secret-shared

Algorithm 4 Protocol $\mathsf{PostProcessT}[i]$ - Inserting New Labels into T^i

```
Input: C's input $s_C[T^i], L^i, L^{i+1}, j_2$;    E's input $s_E[T^i]$;
Input known in pre-computation: C/D secret-sharing of labels $(L^i)'$ and $(L^{i+1})'$,
where E forwards its shares to D;
Output: E/C secret-sharing of tuple $(T^i)' = (1, N^i, (L^i)', A')$ where $A'[j_2] = (L^{i+1})'$
and $A'[t] = A[t]$ for all $t \neq j_2$ where $T^i = (1, N^i, L^i, A)$;
Pre-computation phase: D picks $r_1, ..., r_{2^\tau}$ in $\{0,1\}^{d_{i+1}}$ and $\alpha$ in $\{0,1\}^\tau$, and sends
$\alpha, r_1, ..., r_{2^\tau}$ to C and $s_1, ..., s_{2^\tau}$ to E s.t. $s_\alpha = r_\alpha \oplus s_E[(L^{i+1})']$ and $s_t = r_t$ for $t \neq \alpha$.
1: C sends $\delta = \alpha - j_2 \pmod{2^\tau}$ to E.
2: C outputs $s_C[(T^i)'] = s_C[T^i] \oplus (0, 0^{i\tau}, L^i \oplus s_C[(L^i)'], (c_1|...|c_{2^\tau}))$ where $c_t =$
   $r_{t+\delta \pmod{2^\tau}}$ for all $t \neq j_2$ and $c_t = r_{t+\delta \pmod{2^\tau}} \oplus L^{i+1} \oplus s_C[(L^{i+1})']$ for $t = j_2$.
3: E outputs $s_E[(T^i)'] = s_E[T^i] \oplus (0, 0^{i\tau}, s_E[(L^i)'], (e_1|...|e_{2^\tau}))$ where $e_t = s_{t+\delta \pmod{2^\tau}}$.
```

value into a secret-shared array at a secret position known only to one party. However, we can utilize the fact that this secret-shared value to be injected can be chosen in pre-processing and that E's share of it can be revealed to D. Let $(c, e) = (s_C[(L^{i+1})'], s_E[(L^{i+1})'])$ and let E sends its share e to D in pre-processing. If D pre-computes two $|A|$-long correlated random pads, one for C and one for E, with the known difference e between them at random location α known to C, then $e \oplus c$ can be injected at position j_2 into the C/E secret-sharing of T^i if (1) C sends $\delta = \alpha - j_2 \bmod 2^\tau$ to E, (2) both parties rotate the pads they receives from D counter-clockwise by δ positions, in this way placing the unique pad cells that differ by e at position j_2, (3) both parties xor their shares of T^i with these pads, with C injecting an xor with c at position j_2 into her share. (In addition C will also erase the previous leaf value at position j_2 in A field of T^i by adding L^{i+1} to that xor.)

Eviction Protocol. Protocol $\mathsf{Eviction}$ executes subprotocol $\mathsf{Eviction}[i]$ in Alg. 5 in *parallel* for each $i = 0, ..., h$. (For $i = 0$ protocol $\mathsf{Eviction}[i]$ skips all the steps in Alg 5 except the last one.) Subprotocol $\mathsf{Eviction}[i]$ performs an ORAM eviction procedure on path $\mathrm{P}^*_{\mathrm{L}^i}$, whose C/E secret-sharing is output by protocol Access. The protocol has two parts: First, using Yao's garbled circuit protocol GC (see Section 3) it allows D to identify two tuples in each internal bucket of $\mathrm{P}^*_{\mathrm{L}^i}$ which are either moveable one notch down this path or they are empty (see the eviction algorithm in Section 2). Another instance of GC will similarly find two empty tuples in the four buckets corresponding to the leaf in $\mathrm{P}^*_{\mathrm{L}^i}$. The reason these pairs of indices j_0, j_1 can be leaked to D is that (1) C and E randomly permute the tuples in each bucket in $\mathrm{P}^*_{\mathrm{L}^i}$ before using them in this protocol, and (2) index j_b computed for $b = 0, 1$ for each bucket in $\mathrm{P}^*_{\mathrm{L}^i}$ is defined as the first moveable tuple in that bucket after a random offset λ_b (counting the tuples cyclically), where shifts λ_0, λ_1 are chosen by E independently for each bucket at random in $\{1, ..., w\}$. The circuit computed for every internal bucket takes only $2w$ bits of input (one for bit fb and one for an agreement in the i-th bit of a leaf label in the tuple and the i-th bit of label L^i defining path $\mathrm{P}^*_{\mathrm{L}^i}$), and has only about $16w$ non-xor gates. Once D gets two indexes per each bucket in the path,

Algorithm 5 Protocol $\mathsf{Eviction}[i]$ - Eviction in Path $\mathrm{P}_{\mathrm{L}^i}$ of OT_i

```
Input: C/E secret-sharing of path $\mathrm{P}^*_{\mathrm{L}^i}$ and tuple $(\mathrm{T}^i)'$; $\sigma, \rho$ held by $\mathrm{E}, \mathrm{D}$;
Output: D/E secret-sharing of path $\mathrm{P}^{\diamond}_{\mathrm{L}^i}$ to be inserted into tree $\mathrm{OT}_i$ in place of $\mathrm{P}_{\mathrm{L}^i}$.
Notation: Let $W = \{1, ..., w\}$, $\mathrm{IB} = \{0, ..., \mathrm{d}_i - 1\}$, and $\mathrm{EB} = \{\mathrm{d}_i, \mathrm{d}_i + 1, \mathrm{d}_i + 2, \mathrm{d}_i + 3\}$.
Pre-computation phase: C and E share random permutations $\pi_1, ..., \pi_{\mathrm{d}_i}$ on set $[w]$, a random permutation $\pi_{\mathrm{d}_i+1}$ on set $[4 \cdot w]$, and a random pad $\xi$ of length $|\mathrm{P}_{\mathrm{L}^i}|$;
1: Parties run protocol $\mathsf{Shift}(\mathrm{C}, \mathrm{E}, \mathrm{D})$ on inputs C/E-secret-sharing of $\mathrm{P}^*_{\mathrm{L}^i}$ and on $\mathrm{D}, \mathrm{E}$ input a shift $\sigma$. The protocol outputs a C/E-secret-sharing of path identical to $\mathrm{P}^*_{\mathrm{L}^i}$ but with buckets shifted back by $\sigma$ positions. In addition, for each $j \in \mathrm{IB}$, C and E use $\pi_j$ to permute (their shares of) the tuples in the $j$-th bucket in the resulting path, and they use $\pi_{\mathrm{d}_i+1}$ to permute (their shares of) the tuples in the four buckets corresponding to the leaf node. The resulting path, shared by C and E, is denoted $\mathrm{P}^{**}_{\mathrm{L}^i}$.
2: Let $\mathrm{fb}^j_\ell$ and $\mathrm{L}^j_\ell$ be the fb and L fields of the $\ell$-th tuple in the $j$-th bucket in $\mathrm{P}^{**}_{\mathrm{L}^i}$. For each $j \in \mathrm{IB}$, parties run protocol $\mathsf{GC}[\mathsf{F2FT}](\mathrm{E}, \mathrm{C}, \mathrm{D})$, see Sec 3, on C's inputs $\{\mathrm{s_C}[\mathrm{fb}^j_\ell, \mathrm{L}^j_\ell[j+1]]\}_{\ell \in W}$ and on E's inputs $\{\mathrm{s_E}[\mathrm{fb}^j_\ell], B_j\}_{\ell \in W}$ where $B_j = \mathrm{s_E}[\mathrm{L}^j_\ell[j+1]] \oplus 1 \oplus \mathrm{L}^i[j+1]$. (Note that $\mathrm{s_C}[\mathrm{L}^j_\ell[j+1]] \oplus B_j = 1$ iff the secret-shared value $\mathrm{L}^j_\ell$ and the public value $\mathrm{L}^i$ agree on $(j+1)$-st bit.) For each $j \in \mathrm{IB}$, D defines $\alpha^1_j, \alpha^2_j \in [1, ..., w]$ as the indices of the two output wires of $\mathsf{F2FT}$ on which D received output bit 1 in the $j$-th instance of $\mathsf{GC}[\mathsf{F2FT}]$.
3: The parties run protocol $\mathsf{GC}[\mathsf{F2ET}](\mathrm{E}, \mathrm{C}, \mathrm{D})$ on E's inputs $\{\mathrm{s_E}[\mathrm{fb}^j_\ell]\}_{\ell \in W, j \in \mathrm{EB}}$ and C's inputs $\{\mathrm{s_C}[\mathrm{fb}^j_\ell]\}_{\ell \in W, j \in \mathrm{EB}}$. D defines $\alpha^1_{\mathrm{d}_i}, \alpha^2_{\mathrm{d}_i} \in [1, ..., 4 \cdot w]$ as the indices of the two output wires of $\mathsf{F2ET}$ on which D received output bit 1 in this instance of $\mathsf{GC}[\mathsf{F2ET}]$.
4: D prepares a sequence of $k = w \cdot (\mathrm{d}_i + 4)$ indices $I = (\beta_1, ..., \beta_k)$ s.t.
   $$\beta_{w \cdot j + \ell} = \begin{cases} k+1, & \text{if } j = 0 \text{ and } \ell = \alpha^1_0 \\ k+2, & \text{if } j = 0 \text{ and } \ell = \alpha^2_0 \\ w \cdot (j-1) + \alpha^1_{j-1}, & \text{if } 1 \le j \le \mathrm{d}_i - 1 \text{ and } \ell = \alpha^1_j \\ w \cdot (j-1) + \alpha^2_{j-1}, & \text{if } 1 \le j \le \mathrm{d}_i - 1 \text{ and } \ell = \alpha^2_j \\ w \cdot (\mathrm{d}_i - 1) + \alpha^1_{\mathrm{d}_i - 1}, & \text{if } j \ge \mathrm{d}_i \text{ and } w \cdot (j - \mathrm{d}_i) + \ell = \alpha^1_{\mathrm{d}_i} \\ w \cdot (\mathrm{d}_i - 1) + \alpha^2_{\mathrm{d}_i - 1}, & \text{if } j \ge \mathrm{d}_i \text{ and } w \cdot (j - \mathrm{d}_i) + \ell = \alpha^2_{\mathrm{d}_i} \\ w \cdot j + \ell & \text{otherwise} \end{cases}$$
   and then divides $I$ into $\mathrm{d}_i + 4$ chunks, each of which has $w$ indices, and permutes each chunk with the corresponding $\varrho_r$.
5: C prepares a sequence of $k+2$ shares $(\mathrm{s_C}[a_1], ..., \mathrm{s_C}[a_{k+2}])$ by setting $\mathrm{s_C}[a_{w \cdot j + \ell}] = \mathrm{s_C}[\mathrm{T}^j_\ell]$ where $\mathrm{T}^j_\ell$ is $\ell$-th tuple in $j$-th bucket $\mathrm{B}^j$ in $\mathrm{P}^{**}_{\mathrm{L}^i}$, for $\ell \in W$ and $j \in \mathrm{IB} \cup \mathrm{EB}$, $\mathrm{s_C}[a_{k+1}]$ as $\mathrm{s_C}[(\mathrm{T}^i)']$, and $\mathrm{s_C}[a_{k+2}]$ as 0 concatenated with a random string of $i \cdot \tau + \mathrm{d}_i + 2^\tau \cdot \mathrm{d}_{i+1}$ bits. E prepares a sequence of $k+2$ shares $(\mathrm{s_E}[a_1], ..., \mathrm{s_E}[a_{k+2}])$ in the corresponding way, using its shares of $\mathrm{P}^{**}_{\mathrm{L}^i}$ and $(\mathrm{T}^i)'$.
6: The parties run protocol $\mathsf{SS\text{-}XOT}\left[{k+2 \atop k}\right]$ on C's input $(\mathrm{s_C}[a_1], ..., \mathrm{s_C}[a_{k+2}])$, E's input $(\mathrm{s_E}[a_1], ..., \mathrm{s_E}[a_{k+2}])$, and D's input $I$. C and E set their shares of path $\mathrm{P}^{\diamond}_{\mathrm{L}^i}$ to their output in this $\mathsf{SS\text{-}XOT}\left[{k+2 \atop k}\right]$ protocol xor'ed with string $\xi$.
7: C sends $\mathrm{s_C}[\mathrm{P}^{\diamond}_{\mathrm{L}^i}]$ to D; D and E insert their shares of $\mathrm{P}^{\diamond}_{\mathrm{L}^i}$ into their shares of $\mathrm{OT}_i$.
```

it uses the Secret-Shared Shuffle OT protocol $\mathsf{SS\text{-}XOT}\begin{bmatrix}k+2\\k\end{bmatrix}$ (see Section 3) to randomizes the secret-sharing of all tuples in $\mathrm{P}_{\mathrm{L}^i}$ while (1) moving the secret-shared tuple $(\mathrm{T}^i)'$ prepared by PostProcess into the root bucket, and (2) moving the two chosen tuples in each bucket to the space vacated by the two tuples chosen in the bucket below. Finally, C and E randomize their secret-sharing of the resulting path $\mathrm{P}^{**}_{\mathrm{L}^i}$ by xor-ing their shares with a pre-agreed random pad, C sends its share of $\mathrm{P}^{**}_{\mathrm{L}^i}$ to D, and D and E insert their respective shares of $\mathrm{P}^{**}_{\mathrm{L}^i}$ into their shares of OT_i, in place of the shares of the original path $\mathrm{P}_{\mathrm{L}^i}$ retrieved in the first step of $\mathsf{Access}[i]$.

5 Protocol Analysis

Assuming constant record size the bandwidth of our protocol is $\mathrm{O}\left(w(m^3 + \kappa m^2)\right)$, where w the bucket size of the nodes in our protocol, $|\mathsf{D}| = 2^m$, and κ is the cryptographic security parameter. The $\mathrm{O}\left(wm^3\right)$ term comes from the fact that all our protocols except for the GC evaluation have bandwidth $O(|\mathrm{P}_{\mathrm{L}^i}|)$ where $\mathrm{P}_{\mathrm{L}^i}$ is a path accessed in OT_i. (The online part of our protocol requires 7 such transmissions per each OT_i). Each path $\mathrm{P}_{\mathrm{L}^i}$ in OT_i has length $\mathrm{O}\left(w(\mathrm{d}_i)^2\right)$ where d_i is linear in i, and the summation is then done for i from 1 to $h = \mathrm{O}\left(m\right)$. The $\mathrm{O}\left(w\kappa m^2\right)$ term is the bandwidth for garbled circuits, since the inputs to the circuits for a path have $\mathrm{O}\left(wm\right)$ bits and there are $\mathrm{O}\left(m\right)$ paths retrieved during the traversal of the ORAM forest.

Each party's local cryptographic computation is $\mathrm{O}\left(w\left(m^3/\kappa + m^2\right)\right)$ block cipher or hash operations. Note that the $\mathrm{O}\left(wm^3/\kappa\right)$ factor comes already from secure transmission of data in the Client-Server ORAM, hence this cost seems cryptographically minimal. The GC computation contributes $\mathrm{O}\left(wm^2\right)$ hash function operations, all performed by one party. Since $m < \kappa$, the $\mathrm{O}\left(wm^2\right)$ term could dominate, and indeed we observe that the GC computation occupies a significant fraction of the overall CPU cost.

The performance of the scheme is linear in the bucket size parameter w, and the size of this parameter should be set so that the probability of overflow of any bucket throughout the execution of the scheme is bounded by $2^{-\lambda}$ for the desired statistical security parameter λ. The probability that an internal node overflows and the probability that a leaf node overflows are independent stochastic processes and for this reason we examine them separately. The analytical bounds we give for both cases are not optimal. For the leaf node overflow probability the bound we give in Lemma 1 could be made tight if the number of ORAM accesses N is equal to the number of memory locations 2^m, but for the general case of $\mathsf{N} > 2^m$ we use a simple union bound which adds a N factor. If a tighter analysis could be made, it could potentially reduce the required w by up to $\log(\mathsf{N})$ bits. The bound we give for the internal node overflow probability in Lemma 2 is simplistic and clearly far the optimal. We amend this bound by a discussion of a stochastic model which we used to approximate the eviction process. If this approximation is close to the real stochastic process then the scheme can be instantiated with much smaller bucket sizes than those implied by Lemma 2.

Lemma 1. (Leaf Nodes) *If we have* N *accesses in an ORAM forest with the total capacity for* 2^m *records and with leaf nodes which hold* $4w$ *entries, then the probability that some leaf node overflows at some access is bounded by:*

$$\Pr[\textit{some leaf node in } \mathrm{OTF} \textit{ overflows}] \leq \mathsf{N} \cdot h^2 \cdot \frac{2^m}{w} \cdot 2^{-2w}$$

The proof of this lemma follows from a standard bins-and-balls argument.

To keep this probability below $2^{-\lambda}$ we need that $2w \geq \lambda + \log \mathsf{N} + m + 2\log m$. It is easy to see that if you increase the number of buckets in a leaf node, the constant of this linear relationship (which is roughly $\frac{1}{2}$ for 4 buckets per leaf) decreases rapidly. For example if one uses 6 buckets per leaf, the constant of the linear relationship between w and $m + \log \mathsf{N} + \lambda$ becomes $\frac{1}{6}$, allowing for much smaller buckets. This means that by modifying the number of buckets per leaf, we can ensure that it is the internal nodes that define the size of buckets. We note that increasing the number of buckets per leaf increases the total space for the ORAM forest OTF.

Lemma 2. (Internal Nodes) *If we have* N *accesses and subsequent evictions in an ORAM forest with internal buckets of size* w*, then the probability that some internal bucket overflows at some access is bounded by:*

$$\Pr[\textit{some internal bucket in } \mathrm{OTF} \textit{ overflows}] \leq \mathsf{N} \cdot h \cdot \mathrm{d}_h \cdot w \cdot 2^{-(w-1)}$$

We can prove Lemma 2 by assuming that there exists an internal node that during all accesses and subsequent evictions is on the verge of overflowing (has w or $w-1$ entries in it). We also assume the worst case of each node always receiving exactly two new entries, and we compute the probability that a node is not able to evict two entries, thus causing an overflow.

To keep this probability below $2^{-\lambda}$, the lemma implies that $w - \log w \geq \lambda + \log \mathsf{N} + 2\log m + 1$. For $w < 512$ this can be simplified as $w \geq \lambda + \log \mathsf{N} + 2\log m + 10$. For $\mathsf{N} \leq c \cdot 2^m$ this implies $w \geq \lambda + m + 2\log m + 10 + \log c$.

Stochastic Approximation. The above analysis is pessimistic, since it assumes that there exists a critical bucket that is always full, having w or $w-1$ entries and bounds the probability of such a bucket having a "bad event". It does not explore how difficult it is for a bucket to reach such a state, or how a congested bucket is emptying over time. In order to better understand such behaviors we observe that each internal node can be modeled as a Markov Chain, where the state of the chain counts how many entries are currently in the node. The node is initially empty. Whenever a node is selected in an eviction path it may receive up to two entries depending on whether the parent node was able to evict one or two entries. Moreover the node could evict up to two entries to its child that participates in the eviction path. The root always receives 1 entry and may evict up to two entries. Intuitively since the eviction path is picked at random and each entry is assigned to a random leaf node, each entry in a node in the eviction path can be evicted to the selected child node with probability $\frac{1}{2}$. So for this model we make the following relaxation: Instead of mapping an entry to a leaf node, when it is inserted for the first time in the root, we just let the leaf node be "defined" as the entry is pushed down the tree during eviction. In that sense we abstract entries and the only think we need to care for, is how many entries there exist in a given internal node at a given moment, which is expressed by the state of the Markov Chain.

This model needs one Markov Chain for each internal node. We make the following relaxation: We use one Markov Chain for each level of the tree. A Markov Chain starts empty. At each eviction step, a Markov Chain at level i may receive up to two entries depending on how many entries the Markov Chain in the previous level $i-1$ was able to evict. Moreover the Markov Chain at level i may evict up to two entries to level $i+1$. The Markov Chain for the root (level 0) always receives 1 entry. The state of a Markov Chain keeps tracks of how many entries are in it. At each eviction step an entry can be evicted with probability $\frac{1}{2}$ (the same as the probability we had for the previous model).

The final relaxation we do, is that we remove the direct relationship between a Markov Chain at level i evicting an entry and the Markov Chain at level $i+1$ receiving an entry. We first observe that on expectation at every level the Markov Chain receives at most 1 entry at each eviction step. Intuitively in order to prove this we observe that initially all nodes are empty. The root receives one entry in each eviction step, from there we can use a recursive argument that at any level i a node cannot be evicting more than 1 entry on expectation in each eviction step, which is what the node at level $i+1$ is receiving. Since the eviction probabilities only depend on the current state of a Markov Chain, the worst case for the Markov Chain, is when the variance of the input is maximized. This happens when with probability $\frac{1}{2}$ the node receives 0 entries and with probability $\frac{1}{2}$ the node receives 2 entries (also maximizes the expectation to 1).

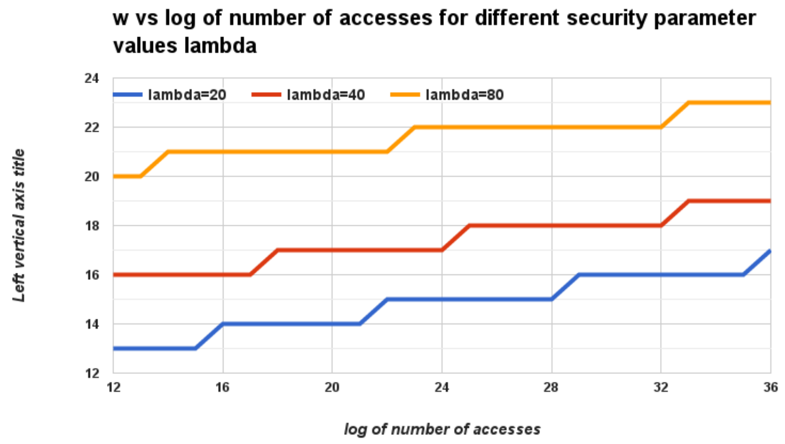

Fig. 1. w for different $\log \mathsf{N}$

We use this last model in order to bound the probability of overflow for internal nodes in our implementation and in order to set bucket sizes. In particular we generate a Markov Chain the has $w+2$ states, w for the bucket size, one empty state and one overflow state. The overflow state is a sink. We compute the probability of being in the overflow state after N accesses assuming the node was initially empty and perform a union bound on the number of nodes in all paths of the ORAM forest OTF. In figure 1 for different statistical security parameters λ equal with $20, 40$ and 80, we show the minimum bucket sizes w for $\log \mathsf{N}$ in the range $12, \ldots 36$ and $m = \mathrm{O}(\log \mathsf{N})$. Generally, we observe that using the Markov Chain based approximation can lead to tighter bounds on the internal node sizes, from which we conjecture that the size of *internal nodes* can be reduced to $\mathrm{O}\left(\sqrt{\lambda + \log \mathsf{N}}\right)$ $(w > 2\sqrt{\lambda + \log \mathsf{N} + 2\log m})$.

6 Implementation and Testing

We built and benchmarked a prototype JAVA implementation of the proposed 3-party SC-ORAM protocol. We tested this implementation on the entry-level Amazon EC2 t2.micro virtual servers, which have one hyperthread on a 2.5GHz CPU and 1GB RAM. Each of the three protocol participants C, D, E where co-located in the same availability zone and connected via a local area network. Here we will briefly show the most important findings, and we defer to the full version of the paper for more detailed performance data.

We measured the performance of the online and offline stages of our protocol separately, but our development effort was focused on optimizing the online stage so the offline timings provide merely a loose upper-bound on the precomputation overhead. We measured both wall clock and CPU times for each execution, where the wall clock time is defined as the maximum of the individual wall clocks, and the CPU time as the sum of the CPU times of the three parties. We tested our prototype for bit-length m of the RAM addresses ranging from 12 to 36, and for record size d ranging from 4 to 128 Bytes. Since the SC-ORAM protocol has two additional parameters, the bucket size w and the bitlength of RAM address segments τ, we tested the sensitivity of the performance to w using w equal to 16, 32, 64, or 128, and for each (m, w, d) tuple we searched for τ that minimize the wall clock (an optimal τ was always between 3 and 6 for the tested cases).

Figure 2 shows the wall clock time of the online stage as a function of the bitlength m of the RAM address space, for the two cases $(w, d) = (16, 4)$ and $(w, d) = (32, 4)$. We found that the CPU utilization in the online phase of our protocol is pretty stable, growing from about 25% for smaller m's to 35% for $m \geq 30$, hence the graph of the CPU costs as function of m has a very similar shape. Our testing showed that the influence of the record size d on the overall performance is very small for d less than $100B$, but higher payload sizes start influencing the running time. Our testing confirms that the running time has clear linear relationship to the bucket size w: The wall clock for $w = 64$ grows by a factor close to 1.8 compared to $w = 32$, and for $w = 128$ by a factor close to 3.5 (for large m and small d). The offline wall clock time grows from 400 msec for $m = 12$ to 1300 msec for $m = 36$ for $w = 32$, but these numbers should be taken only as loose upper bounds on the precomputation overhead of our SC-ORAM. Finally, we profiled the code to measure the percentage of

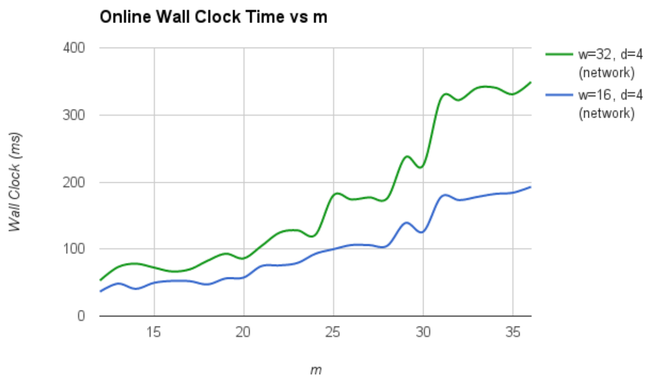

Fig. 2. Online Wall Clock vs RAM address size m

CPU time spent on different protocol components. We found that the fraction of the fraction of the total CPU costs of the online phase spent on Garbled Circuit evaluation decreases from $45\% - 50\%$ for $m = 12$ to 25% for $m = 36$. We also found that only about half of that cost is spent in SHA evaluation, i.e. that the Garbled Circuit evaluation protocol spends only about half its CPU time on decryption of the garbled gates. The fraction of the CPU cost spent on symmetric ciphers, which form the only cryptographic costs of all the non-GC part of our protocol, decreases from the already low figure of 10% for small m's to below 5% for $m = 36$. By contrast, the fraction of the CPU cost spent on handling message passing to and from TCP communication sockets grows from 12% for small m's to 30% for $m = 36$.

References

1. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, New York, NY, USA, 1988. ACM.
2. D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 11–19, New York, NY, USA, 1988. ACM.
3. S. Choi, K. Hwang, J. Katz, T. Malkin, and D. Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line market-places. *Topics in Cryptology–CT-RSA 2012*, pages 416 – 432, 2012/// 2012.
4. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of 36th FOCS*, pages 41–50, 1995.
5. K.-M. Chung, Z. Liu, and R. Pass. Statistically-secure oram with $\tilde{O}(\log^2 n)$ overhead. In P. Sarkar and T. Iwata, editors, *Advances in Cryptology ASIACRYPT 2014*, volume 8874 of *Lecture Notes in Computer Science*, pages 62–81. Springer Berlin Heidelberg, 2014.

6. I. Damgard, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure mpc for dishonest majority - or: Breaking the spdz limits. In *ESORICS*, pages 144–163, 2013.
7. I. Damgard, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious ram without random oracles. In *Theory of Cryptography*, pages 144–163, 2011.
8. I. Damgard, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 20123.
9. C. Fletcher. Ascend: An architecture for performing secure computation on encrypted data. In *MIT CSAIL CSG Technical Memo 508*, 2013.
10. C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, STC '12, pages 3–8, New York, NY, USA, 2012. ACM.
11. C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs. Optimizing oram and using it efficiently for secure computation. In *Privacy Enhancing Technologies*, PETS'13, pages 1–18, 2013.
12. Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. In *STOC*, 1998.
13. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.
14. S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *Computer and Communications Security (CCS)*, CCS '12, pages 513–524, 2012.
15. M. Keller and P. Scholl. Efficient, oblivious data structures for mpc. In P. Sarkar and T. Iwata, editors, *ASIACRYPT*, volume 8874 of *Lecture Notes in Computer Science*, pages 506–525. Springer Berlin Heidelberg, 2014.
16. L. Kruger, S. Jha, E.-J. Goh, and D. Boneh. Secure function evaluation with ordered binary decision diagrams. In *Conference on Computer and Communications Security*, CCS '06, pages 410–420, New York, NY, USA, 2006. ACM.
17. S. Lu and R. Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, pages 377–396, 2013.
18. M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. In *Conference on Computer and Communications Security*, CCS '13, pages 311–324, New York, NY, USA, 2013. ACM.
19. R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 294–303, 1997.
20. E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with $O((\log n)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.
21. E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *Conference on Computer and Communications Security (CCS)*, CCS'13, pages 299–310, 2013.
22. X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. Scoram: Oblivious ram for secure computation. In *Conference on Computer and Communications Security*, CCS '14, pages 191–202, New York, NY, USA, 2014. ACM.
23. X. S. Wang, T.-H. Hubert, and E. Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Eprint IACR Archive*, page 2015/672, 2014.
24. A. C.-C. Yao. Protocols for secure computations (extended abstract). In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, FOCS'82, pages 160–164, 1982.