3PC ORAM with Low Latency, Low Bandwidth, and Fast Batch Retrieval

Stanislaw Jarecki Boyang Wei

University of California, Irvine

Abstract. Multi-Party Computation of Oblivious RAM (MPC ORAM) implements secret-shared random access memory in a way that protects access pattern privacy against a threshold of corruptions. MPC ORAM enables secure computation of any RAM program on large data held by different entities, e.g. MPC processing of database queries on a secret-shared database. MPC ORAM can be constructed by any (client-server) ORAM, but there is an efficiency gap between known MPC ORAM's and ORAM's. Current asymptotically best MPC ORAM is implied by an "MPC friendly" variant of *Path-ORAM* [26] called *Circuit-ORAM*, due to Wang et al [27]. However, using garbled circuit for Circuit-ORAM's client implies MPC ORAM which matches Path-ORAM in rounds but increases *bandwidth* by $\Omega(\kappa)$ factor, while using GMW or BGW protocols implies MPC ORAM which matches Path-ORAM in bandwidth, but increases *round complexity* by $\Omega(\log n \log\log n)$ factor, where κ is a security parameter and n is memory size.

In this paper we bridge the gap between MPC ORAM and client-server ORAM by showing a specialized 3PC ORAM protocol, i.e. MPC ORAM for 3 parties tolerating 1 fault, which uses only symmetric ciphers and asymptotically matches client-server Path-ORAM in round complexity and for large records also in bandwidth.

Our 3PC ORAM also allows for fast pipelined processing: With postponed clean-up it processes $b = O(\log n)$ accesses in $O(b + \log n)$ rounds with $O(D + \mathsf{poly}(\log n))$ bandwidth per item, where D is record size.

1 Introduction

MPC ORAM. Multi-Party Computation Oblivious Random Access Memory (MPC ORAM), or Secure-Computation ORAM (SC ORAM), is a protocol which lets m parties implement access to a secret-shared memory in such a way that both memory records and the accessed locations remain hidden, and this security guarantee holds as long as no more than t out of m parties are corrupted. Applications of MPC ORAM stem from the fact that it can implement random memory access subprocedure within secure computation of any RAM program. Classic approaches to secure computation [29, 17, 3, 8] express computation as a Boolean or arithmetic circuit, thus their size, and consequently efficiency, is inherently lower-bounded by the size of their inputs. In practice this eliminates the possibility of secure computation involving large data, including such fundamental computing

The shortened version of this paper appears in ACNS'18 proceedings [21]

functionality as search and information retrieval. MPC ORAM makes such computation feasible because it generalizes secure computation from circuits to RAM programs: All RAM program instruction can be implemented using circuit-based MPC, since they involves only local variables, while access to (large) memory can be implemented with MPC ORAM.

As an application of MPC of RAM program, and hence of MPC ORAM, consider an MPC Database, i.e. an MPC implementation of processing of database queries over a secret-shared database. A typical database implementation would hash a searched keyword to determine an address of a hash table page whose content is then matched against the queried keyword. Standard MPC techniques can implement the hashing step, but the retrieval of the hash page is a random access to a large memory. Implementing this RAM access via garbled circuits requires $\Omega(nD\kappa)$ bandwidth, where n is the number of records, D is the record size, and κ is the cryptographic security parameter, which makes such computation unrealistic even for 1MB databases. By contrast, using MPC ORAM can cost $O(\mathsf{poly}(\log n)D\kappa)$ and hence, in principle, can scale to large data.

Inefficiency Gap in MPC ORAM Constructions. The general applicability of MPC ORAM to MPC of any RAM program motivates searching for efficient MPC ORAM realizations. As pointed out in [23, 10], any ORAM with its client implemented with an MPC protocol yields MPC ORAM. This motivates searching for an ORAM with an MPC-friendly client, i.e. a client which can be efficiently computed using MPC techniques [19, 16, 22, 28, 27]. Indeed, the recent *Circuit-ORAM* proposal of Wang et al. [27] exhibits a variant of *Path-ORAM* of Stefanov et al. [26] whose client has a Boolean circuit of an asymptotically optimal size, i.e. a constant factor of the data which Path-ORAM client retrieves from the server, and which forms an input to its computation.

Still, in spite of the circuit-size optimality of Circuit-ORAM,[1] applying generic honest-but-curious MPC protocols to it yields MPC ORAM solutions which are two orders of magnitude more expensive than Path-ORAM:[2] Using Yao's garbled circuit [29] on Circuit-ORAM yields a 2PC ORAM of [27] which has (asymptotically) the same round complexity as Path-ORAM, but its bandwidth, both online and in offline precomputation, is larger by $\Omega(\kappa)$ factor. Alternatively, applying GMW [17] or BGW [3] to the Boolean circuit for Circuit-ORAM yields 2PC or MPC ORAM which asymptotic preserves Path-ORAM bandwidth, but its round complexity is larger by $\Omega(\log n \log\log n)$ factor (compare footnote 3).

Our Contribution: 3PC ORAM with Low Latency *and* Bandwidth. We show that the gap between MPC ORAM and client-server ORAM can be bridged by exhibiting a 3PC ORAM, i.e. MPC for $m=3$ servers with $t=1$ fault, which uses *customized*, i.e. non-generic, 3PC protocols and *asymptotically matches Path-ORAM* in rounds, and, for records size $D = \Omega(\kappa \log^2 n)$, bandwidth. Specifically,

[1] In this paper we call the client-server ORAM implicit in [27] "Circuit-ORAM", and its garbled-circuit 2PC implementation, also shown in [27], "2PC Circuit-ORAM".

[2] We use Path-ORAM as a client-server baseline for these comparisons because Path-ORAM has the most "MPC-friendly" client, hence most MPC ORAM's emulate securely either Path-ORAM or its predecessor, *Binary-Tree ORAM* [25]. (The recent 2PC ORAM of [12] is an exception, discussed below.)

our 3PC ORAM securely emulates the Circuit-ORAM client in 3PC setting, using $O(\log n)$ rounds and $O(\kappa \log^3 n + D \log n)$ bandwidth (see Fig. 1). We note that the 3PC setting of (t, m)=$(1, 3)$ gives weaker security than 2PC setting of (t, m)=$(1, 2)$, but it was shown to enable lower-cost solutions to many secure computation problems compared to both 2PC or general (t, m)-MPC (e.g. [5, 1]) and for that reason it's often chosen in secure computation implementations (e.g. [6, 4]). Here we show that 3PC benefits extend to MPC ORAM.

	rounds	bandwidth
Path-ORAM ($^{\text{client-server}}_{\text{baseline}}$) [26]	$O(\log n)$	$O(\log^3 n + D \log n)$
2PC Circuit-ORAM [27]+[29]	$O(\log n)$	$O(\kappa \log^3 n + \kappa D \log n)$
2PC SQRT-ORAM [30]	$O(\log n)$	$O(\kappa D \sqrt{n \log^3 n})$
2PC FLORAM [12]	$O(\log n)$	$O(\sqrt{\kappa D n \log n})$
generic 3PC Circ.-ORAM [27]+[1]	$O(\log^2 n \log \log n)$	$O(\log^3 n + D \log n)$
3PC ORAM of [14]	$O(\log n)$	$O(\kappa \lambda \log^3 n + \lambda D \log n)$
Our 3PC Circuit-ORAM	$O(\log n)$	$O(\kappa \log^3 n + D \log n)$

Fig. 1: Round and bandwidth comparisons, for n: array size, D: record size, κ: cryptographic security parameter, λ: statistical security parameter.

We show the benefits of our 3PC ORAM contrasted with previous 2PC and 3PC ORAM approaches in Fig. 1. In the 3PC setting we include a *generic 3PC Circuit-ORAM*, which results from implementing Circuit-ORAM with the generic 3PC protocol of Araki et al. [1], which is the most efficient 3PC instantiation we know of either the BGW or the GMW framework.[3] The second 3PC ORAM we compare to is Faber et al. [14], which uses non-generic 3PC techniques, like we do, but it emulates in 3PC with a less efficient Binary-Tree ORAM variant than Circuit-ORAM, yielding 3PC ORAM with bandwidth worse than ours by $\Omega(\lambda)$ factor. Regarding 2PC ORAM, several 2PC ORAM's based on Binary-Tree ORAM variants were given prior to Circuit-ORAM [19, 16, 22, 28], but we omit them from Fig. 1 because Circuit-ORAM outperforms them [27]. We include two recent alternative approaches, 2PC ORAM of [30] based on Square-Root ORAM of [18], and 2PC FLORAM of [12] based on the Distributed Point Function (DPF) of [20]. However, both of these 2PC ORAM's use $O(\sqrt{n})$ bandwidth, and [12] also uses $O(n)$ local computation, which makes them not scale well for large n's.[4] Restricting the comparison to poly$(\log n)$ MPC ORAM, our 3PC ORAM offers the following trade-offs:
(1) Compared to the *generic* 3PC Circuit-ORAM [1] applied to Circuit-ORAM, we increase bandwidth from $O(\log^3 n + D \log n)$ to $O(\kappa \log^3 n + D \log n)$ but reduce

[3] Using the BGW-style MPC over an *arithmetic* circuit for Circuit-ORAM, as was done by Keller and Scholl for another Path-ORAM variant [22], should also yield a bandwidth-competitive 3PC ORAM, but with round complexity at least $\Omega(\log^2 n)$.

[4] 2PC ORAM cost of [12] has stash linear scan $O(T\kappa \log n)$ and amortized re-init $O(nD/T)$. Picking $T = O(\sqrt{nD/\kappa \log n})$ we get $O(\sqrt{\kappa D n \log n})$. In [12] this is rendered as $O(\sqrt{n})$ *overhead*, assuming $D = \Omega(\log n)$ and omitting κ. [12] also show $O(1)$-round 2PC ORAM, but at the price of increased bandwidth and computation.

round complexity from $O(\log^2 n \log\log n)$ to $O(\log n)$;
(2) Compared to the *generic* garbled circuit 2PC [29] applied to Circuit-ORAM, we weaken the security model, from (t, m)=$(1, 2)$ to (t, m)=$(1, 3)$, but reduce bandwidth from $O(\kappa\log^3 n + \kappa D\log n)$ to $O(\kappa\log^3 n + D\log n)$.

Thus for medium-sized records, $D = \Omega(\kappa\log^2 n)$, our 3PC ORAM asymptotically matches client-server Path-ORAM in all aspects, and beats 2PC Circuit-ORAM by $\Omega(\kappa)$ factor in bandwidth, without dramatic increase in round complexity incurred using generic 3PC techniques. In concrete terms, our round complexity is 50x lower than the generic 3PC Circuit-ORAM,[5] and, for $D > 1$KB, our bandwidth is also >50x lower than 2PC Circuit-ORAM. Our protocol is also competitive for small record sizes, e.g. $D = 4B$: First, our bandwidth is only about 2x larger than the generic 3PC Circuit-ORAM; Second, our bandwidth is lower than the 2PC Circuit-ORAM by a factor between 10x and 20x for $20 \leq \log n \leq 30$.

Fast System Response and Batch Retrieval. Another benefit of our 3PC ORAM is a fast system response, i.e. the time we call a *Retrieval Phase*, from an access request to the retrieval of the record. In fact, our protocol supports fast retrieval of a *batch* of requests, because the expensive post-processing of each access (i.e. the Circuit-ORAM eviction procedure) can be postponed for a batch of requests, allowing all of them to be processed at a smaller cost. Low-bandwidth batch retrieval with postponed eviction was recently shown for client-server Path-ORAM variants [24, 11] (see also [15]), and our protocol allows MPC ORAM to match this property in the 3PC setting.

Specifically, our protocol processes $b = O(\log n)$ requests in $3b + 3h$ rounds, using $3D + O(\log^2 n \log\log n)$ bandwidth per record, and to the best of our knowledge no other MPC ORAM allows batch-processing with such costs. After retrieving b requests the protocol must perform all evictions, using $6b$ rounds and $O(b(\kappa\log^3 n + D\log n))$ total bandwidth, but this can be postponed for any batch size that benefits the higher-level MPC application. Concretely, for $\log n \leq 30$, the per-record bandwidth for $b \leq 4\log n$ is only $\leq 3D + 10$KB.

Brief Overview of our 3PC ORAM. We sketch the main ideas behind our 3PC protocol that emulates Circuit-ORAM ORAM. Observe that Circuit-ORAM client, like a client in any Binary-Tree ORAM variant, performs the following steps: (1) locate the searched record in the retrieved tree path, (2) post-process that record (free-up its current location, update its labels, and add the modified record to the path root), (3) determine the *eviction map*, i.e. the permutation on positions in the retrieved path according to which the records will be moved in eviction, and (4) move the records on the path according to the eviction map. The main design principle in our 3PC emulation of Circuit-ORAM is to implement steps (1), (2), and (4) using *customized* asymptotically bandwidth-optimal *and* constant-round protocols (we explain some of the challenges involved in Section 2), and leave the eviction map computation step as in 2PC Circuit-ORAM, implemented with *generic* constant-round secure computation, namely garbled circuits.

[5] We estimate that the circuit depth of the Circuit-ORAM client, and hence the round complexity of the generic 3PC Circuit-ORAM, is > 1000 even for $n = 2^{20}$, compared to $\approx$15 rounds in our 3PC ORAM and $\approx$8 in the client-server Path-ORAM.

Circuit-ORAM computes the eviction map via data-dependent scans, which we do not know how to implement in constant rounds without the garbled circuit overhead. However, computation of the eviction map involves only on metadata, and is independent of record payloads. Hence even though using garbled circuits in this step takes $O(\kappa)$ bandwidth per input bit, this is upper-bounded by the cost of bandwidth-optimal realization of the data movement step (4) already for $D \approx 140$B.

Secondly, we utilize the 3PC setting in the retrieval phase, to keep its bandwidth especially low, namely $O(D + \log^2 n \log\log n)$. The key ingredient is a 3-party *Secret-Shared* PIR (SS-PIR) gadget, which computes a secret-sharing of record $\mathsf{M}[\mathrm{N}]$ given a secret-sharing of array M and of address N. We construct SS-PIR from any 2-server PIR [13] whose servers' responses form an xor-sharing of the retrieved record, which is the case for many 2-PIR schemes [9, 2, 20]. Another component is a one-round bandwidth-optimal compiler from 3PC SS-PIR to 3PC *Keyword* SS-PIR, which retrieves shared value given a sharing of keyword and of (keyword,value) list. With a careful design we use *only three rounds* for the retrieval and post-processing steps, which allows pipelined processing of a batch of accesses using *only three rounds* per tree.

Roadmap. We overview the technical challenges of our construction in Section 2. We present our 3PC ORAM protocol in Section 3, argue its security in Section 4, and discuss our prototype performance in Section 5. All specialized sub-protocols our protocol requires are deferred to Appendix A. The full security argument, the specification of garbled circuits we use, and further prototype performance data, are all included in Appendices B-E.

2 Technical Overview

Overview of Path ORAM [26]. Our 3PC Circuit-ORAM is a 3PC secure computation of Circuit-ORAM of [27] (see footnote 1), which is a variant of Path-ORAM of Shi et al. [26]. We thus start by recalling Path-ORAM of [26], casting it in terms which are convenient in our context. Let M be an array of n records of size D each. Server S keeps a binary tree of depth $\log n$, denoted tree, shown in Fig. 2, where each node is a bucket of a small constant size w, except the root bucket (a.k.a. a *stash*) which has size $s = O(\log n)$. Each tree bucket is a list of *tuples*, which are records with four fields, fb, lb, adr, and data. For each address $\mathrm{N} \in \{0,1\}^{\log n}$, record $\mathsf{M}[\mathrm{N}]$ is stored in a unique tuple T in tree s.t. $\mathsf{T}.(\mathsf{fb}, \mathsf{lb}, \mathsf{adr}, \mathsf{data}) = (1, \mathrm{L}, \mathrm{N}, \mathsf{M}[\mathrm{N}])$ where fb is a full/empty tuple status bit and L is a *label* which defines a tree leaf assigned at random to address N.

Data-structure tree satisfies an invariant that a tuple with label L lies in a bucket on the path from the root to leaf L, denoted $\mathsf{tree}.\mathsf{path}(\mathrm{L})$. To access address N, client C uses a (recursive) *position map* $\mathsf{PM} : \mathrm{N} \to \mathrm{L}$ (see below) to find leaf L corresponding to N, sends L to S to retrieve $\mathsf{path} = \mathsf{tree}.\mathsf{path}(\mathrm{L})$, searches path for $\mathsf{T} = (1, \mathrm{L}, \mathrm{N}, \mathsf{M}[\mathrm{N}])$ with fields $(\mathsf{fb}, \mathsf{adr})$ matching $(1, \mathrm{N})$, assigns new random leaf L' to N, adds a modified tuple $\mathsf{T}' = (1, \mathrm{L}', \mathrm{N}, \mathsf{M}[\mathrm{N}])$ to the root bucket in path (In case of *write* access C also replaces $\mathsf{M}[\mathrm{N}]$ in T' with a new entry), and erase old T

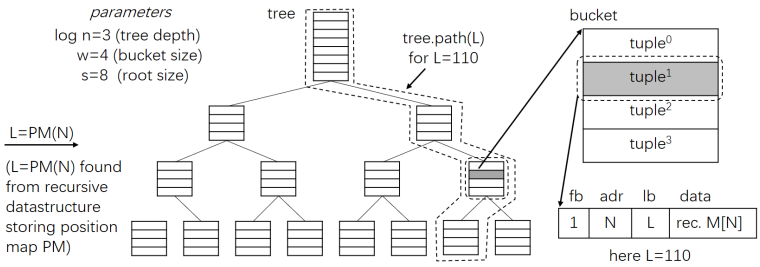

Fig. 2: Path ORAM (*final*) tree

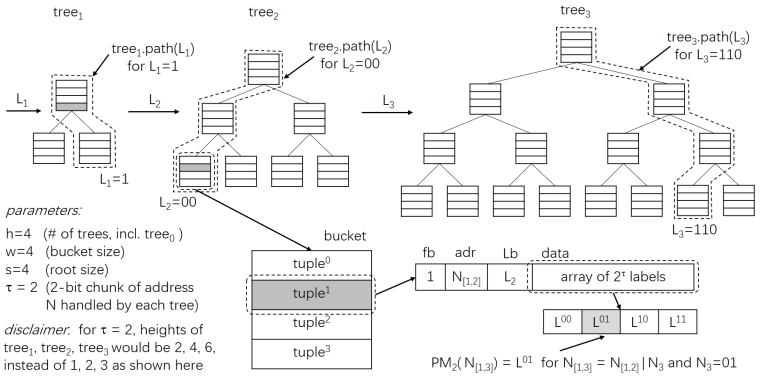

Fig. 3: Path ORAM recursive access

from path by flipping T.fb to 0. Finally, to avoid overflow, C evicts tuples in path as far down as possible without breaking the invariant or overflowing any bucket.

Position map $\mathsf{PM} : \mathrm{N} \to \mathrm{L}$ is stored using the same data-structure, with each tuple storing labels corresponding to a batch of 2^τ consecutive addresses, for some constant τ. Since such position map has only $2^{\log n}/2^\tau = 2^{\log n - \tau}$ entries, this recursion results in $h = (\log n/\tau) + 1$ trees $\mathsf{tree}_0, , .., \mathsf{tree}_{h-1}$ which work as follows (see Fig. 3): Divide N into τ-bit blocks $\mathrm{N}_1, ..., \mathrm{N}_{h-1}$. The top-level tree, tree_{h-1} contains the records of M as described above, shown in Fig. 2, while for $i < h-1$, tree_i is a binary tree of depth $d_i = i\tau$ which implements position map PM_i that matches address prefix $\mathrm{N}_{[1,...,i+1]} = \mathrm{N}_1|...|\mathrm{N}_{i+1}$ to leaf L_{i+1} assigned to this prefix in tree_{i+1}. Access algorithm ORAM.Access traverses this data-structure by sequentially retrieving the labels assigned to each prefix of the searched-for address, using an algorithm we denote ORAM.ML. For i from 0 to $h-1$, algorithm ORAM.ML retrieves $\mathrm{L}_{i+1} = \mathsf{PM}_i(\mathrm{N}_1|...|\mathrm{N}_{i+1})$ from tree_i using the following steps: (1) it identifies path $\mathsf{path} = \mathsf{tree}_i.\mathsf{path}(\mathrm{L}_i)$ in tree_i using label L_i, (2) it identifies tuple T in path s.t. $\mathsf{T.adr} = \mathrm{N}_1|...|\mathrm{N}_i$, and (3) it returns $\mathrm{L}_{i+1} = \mathsf{T.data}[\mathrm{N}_{i+1}]$.

Circuit-ORAM vs. Path-ORAM. Circuit-ORAM (see footnote 1) follows the same algorithm as Path-ORAM except (1) the eviction procedure is *restricted* in that it moves only selected tuples down the path in path, as we discuss further below; and (2) it performs the eviction on two paths in each tree per access. Our 3PC emulation of Circuit-ORAM also runs twice per each tree per access, but

since the second execution is limited to eviction, for simplicity of presentation we omit it in all discussion below, except when we report performance data.

Top-Level Design of 3PC Circuit-ORAM. The client algorithm in all variants of Binary-Tree ORAM, which includes Path-ORAM and Circuit-ORAM, consists of the following phases:

1. *Retrieval*, which given $\mathsf{path} = \mathsf{tree.path}(\mathrm{L})$ and address prefix N, locates tuple $\mathsf{T} = (1, \mathrm{L}, \mathrm{N}, \mathsf{data})$ in path and retrieves next-level label (or record) in data;
2. *Post-Process*, which removes T from path, injects new labels into T, and re-inserts it in the root (= stash);
3. *Eviction*, which can be divided into two sub-steps:
   (a) *Eviction Logic*: An eviction map EM is computed, by function denoted Route, on input label L and the *metadata* fields $(\mathsf{fb}, \mathsf{lb})$ of tuples in path,
   (b) *Data Movement*: Permute tuples in path according to map EM.

Our 3PC ORAM is a secure emulation of the above procedure, with the Eviction Logic function Route instantiated as in Circuit-ORAM, and it performs all the above steps on the *sharings* of inputs tree and N, given label L as a public input known to all parties. With the exception of the next-level label recovered in Retrieval, all other variables remain secret-shared. Our implementation of the above steps resembles the 3PC ORAM emulation of Binary-Tree ORAM by [14] in that we use garbled circuit for Eviction Logic, and specialized 3PC protocols for Retrieval, Post-Process, and Data Movement. However, our implementations are different from [14]: First, to enable low-bandwidth batch processing of retrieval we use different sharings and protocols in Retrieval and Post-Process. Second, to securely "glue" Eviction Logic and Data Movement we need to mask mapping EM computed by Eviction Logic and implement Data Movement given this masked mapping. We explain both points in more detail below.

Low-Bandwidth 3PC Retrieval. The Retrieval phase realizes a *Keyword Secret-Shared PIR* (Kw-SS-PIR) functionality: The parties hold a *sharing* of an array of (keyword,value) pairs, and a *sharing* of a searched-for keyword, and the protocol must output a *sharing* of the value in the (keyword,value) pair that contains the matching keyword. In our case the address prefix $\mathrm{N}_{[1,i]}$ is the searched-for keyword and path is the array of the (keyword,value) pairs where keywords are address fields adr and values are payload fields data.

The 3PC implementation of Retrieval in [14] has $O(\ell D)$ bandwidth where $\ell = O(\log n)$ is the number of tuples in path, and here we reduce it to $3D + O(\ell \log \ell)$ as follows: First, we re-use the *Keyword Search* protocol $\mathsf{KSearch}$ of [14] to create a secret-sharing of index j of a location of the keyword-matching tuple in path. This subprotocol reduces the problem to finding an index where a secret-shared array of length ℓ contains an all-zero string, which has $\Theta(\ell \log \ell)$ communication complexity. Our $\mathsf{KSearch}$ implementation has $2\ell(c + \log \ell)$ bandwidth where 2^{-c} is the probability of having to re-run $\mathsf{KSearch}$ because of collisions in ℓ pairs of $(c + \log \ell)$-bit hash values. The overall bandwidth is optimal for $c \approx \log \log \ell$, but we report performance numbers for $c = 20$.

Secondly, we use a *Secret-Shared PIR* (SS-PIR) protocol, which creates a fresh sharing of the j-th record given the shared array and the shared index j. We

implement SS-PIR in two rounds from any 2-server PIR [13] whose servers' PIR responses form an xor-sharing of the retrieved record. Many 2-PIR's have this property, e.g. [9, 2, 20], but we exemplify this generic construction with the simplest form of 2-server PIR of Chor et al. [9] which has $3\ell + 3D$ bandwidth. This is not optimal in ℓ, but in our case $\ell \leq 150 + b$ where b is the number of accesses with postponed eviction, the optimized version of SS-PIR sends only $\approx \ell + 3D$ bits *online*, and KSearch already sends $O(\ell \log \ell)$ bits. Our generic 2-PIR to 3PC-SS-PIR compiler is simple (a variant of it appeared in [20]) but the 3-round 3PC Kw-SS-PIR protocol is to the best of our knowledge novel.

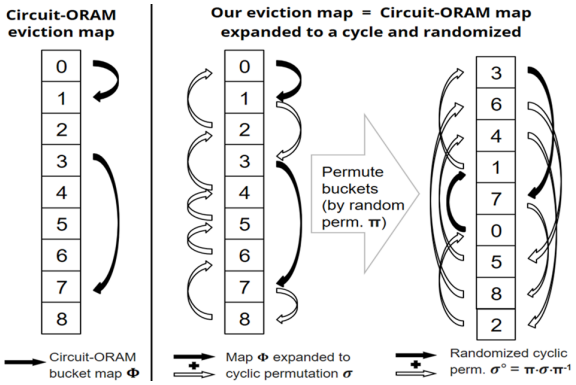

Fig. 4: Randomization of Circuit ORAM's Bucket Map

Efficient 3PC Circuit-ORAM Eviction. In Eviction we use a simple Data Movement protocol, with 2 round and $\approx 2|\mathsf{path}|$ bandwidth. With three parties denoted as $(\mathsf{C}, \mathsf{D}, \mathsf{E})$, our protocol creates a two-party (C, E)-sharing of $\mathsf{path}' = \mathsf{EM}(\mathsf{path})$ from a (C, E)-sharing of path *if* party D holds eviction map EM in the clear. Naively outputting $\mathsf{EM} = \mathsf{Route}(\mathsf{path})$ to party D is insecure, as eviction map is correlated with the ORAM access pattern, so the question is whether EM can be *masked* by some randomizing permutation known by C and E. [14] had an easy solution for its binary tree ORAM variant because its algorithm Route outputs a *regular* EM, that buckets on every except the last level of the retrieved path always move two tuples down to the next level, so all [14] needed to do is to randomly permute tuples on each bucket level of path, and the resulting new EM' on the permuted path leaks no information on EM. By contrast, Circuit-ORAM eviction map is *non-regular* (see Fig. 4): Its bucket level map Φ of EM can move a tuple by variable distance and can leave some buckets untouched, both of which are correlated with the density of tuples in path, and thus with ORAM access pattern.

Thus our goal is to transform the underlying Circuit-ORAM eviction map $\mathsf{EM} = (\Phi, \mathsf{t})$ into a map whose distribution does not depend on the data (Φ describes the bucket-level movement, while t is an array containing one tuple index from each bucket that will be moved). We do so in two steps. First, we add an extra empty tuple to each bucket and we modify Circuit-ORAM algorithm Route to *expand* function $\Phi : \mathsf{Z}_d \to \mathsf{Z}_d \cup \{\perp\}$ into a cyclic permutation σ on Z_d (d is the depth of path, Z_d is the set $\{0, ..., d-1\}$), by adding spurious edges to Φ in the deterministic way illustrated in Fig. 4. Second, we apply two types of masks to

the resulting output (σ, t) of Route, namely a random permutation π on Z_d and two arrays (δ, ρ), each of which contains a random tuple index in each bucket. Our Eviction Logic protocol will use (π, δ, ρ) to mask (σ, t) by computing $(\sigma^\circ, \mathsf{t}^\circ)$ s.t. $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$ (permutation composition) and $\mathsf{t}^\circ = \rho \oplus \pi(\mathsf{t} \oplus \delta)$. And now we have a masked eviction map $\mathsf{EM}_{\sigma^\circ, \mathsf{t}^\circ}$ that can be revealed to party D but does not leak information on $\mathsf{EM}_{\sigma,\mathsf{t}}$ or $\mathsf{EM}_{\Phi,\mathsf{t}}$.

3 Our Protocol: 3PC Emulation of Circuit-ORAM

Protocol Parties. We use $\mathsf{C}, \mathsf{D}, \mathsf{E}$ to denote the three parties participating in 3PC-ORAM. We use x^{P} to denote that variable x is known only to party $\mathsf{P} \in \{\mathsf{C}, \mathsf{D}, \mathsf{E}\}$, $x^{\mathsf{P}_1\mathsf{P}_2}$ if x is known to P_1 and P_2, and x if known to all parties.

Shared Variables, Bitstrings, Secret-Sharing. Each pair of parties $\mathsf{P}_1, \mathsf{P}_2$ in our protocol is initialized with a shared seed to a Pseudorandom Generator (PRG), which allows them to generate any number of shared (pseudo)random objects. We write $x^{\mathsf{P}_1\mathsf{P}_2} \xleftarrow{\$} \mathsf{S}$ if P_1 and P_2 both sample x uniformly from set S using the PRG on a jointly held seed. We use several forms of secret-sharing, and here introduce four of them which are used in our high level protocols $\mathsf{3PC\text{-}ORAM.Access}$ and $\mathsf{3PC\text{-}ORAM.ML}$ (Alg. 1 & 2):

$$\langle x \rangle = (x_1^{\mathsf{DE}}, x_2^{\mathsf{CE}}, x_3^{\mathsf{CD}}) \text{ for } x_1, x_2, x_3 \xleftarrow{\$} \{0,1\}^{|x|} \text{ where } x_1 \oplus x_2 \oplus x_3 = x$$

$$\langle x \rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2} = (x_1^{\mathsf{P}_1}, x_2^{\mathsf{P}_2}) \text{ for } x_1, x_2 \xleftarrow{\$} \{0,1\}^{|x|} \text{ where } x_1 \oplus x_2 = x$$

$$\langle x \rangle_{\mathsf{shift}}^{\mathsf{P}_1\mathsf{P}_2-\mathsf{P}_3} = (x_{12}^{\mathsf{P}_1\mathsf{P}_2}, x_3^{\mathsf{P}_3}) \text{ for } x \in \mathsf{Z}_m, x_{12}, x_3 \xleftarrow{\$} \mathsf{Z}_m \text{ s.t. } x_{12} + x_3 = x \bmod m$$

$$\langle x \rangle_{\mathsf{shift}} = (\langle x \rangle_{\mathsf{shift}}^{\mathsf{CD}-\mathsf{E}}, \langle x \rangle_{\mathsf{shift}}^{\mathsf{CE}-\mathsf{D}}, \langle x \rangle_{\mathsf{shift}}^{\mathsf{DE}-\mathsf{C}})$$

Integer Ranges, Permutations. We define Z_n as set $\{0, ..., n-1\}$, and perm_n as the set of permutations on Z_n. If $\pi, \sigma \in \mathsf{perm}_n$ then π^{-1} is an inverse permutation of π, and $\pi \cdot \sigma$ is a composition of σ and π, i.e. $(\pi \cdot \sigma)(i) = \pi(\sigma(i))$.

Arrays. We use $\mathsf{array}^m[\ell]$ to denote arrays of ℓ bitstrings of size m, and we write $\mathsf{array}[\ell]$ if m is implicit. We use $x[i]$ to denote the i-th item in array x. Note that $x \in \mathsf{array}^m[\ell]$ can also be viewed as a bitstring in $\{0,1\}^{\ell m}$.

Permutations, Arrays, Array Operations. Permutation $\sigma \in \mathsf{perm}_\ell$ can be viewed as an array $x \in \mathsf{array}^{\log \ell}[\ell]$, i.e. $x = [\sigma(0), ..., \sigma(\ell-1)]$. For $\pi \in \mathsf{perm}_\ell$ and $y \in \mathsf{array}[\ell]$ we use $\pi(y)$ to denote an array containing elements of y permuted according to π, i.e. $\pi(y) = [y_{\pi^{-1}(0)}, ..., y_{\pi^{-1}(\ell-1)}]$.

Garbled Circuit Wire Keys. If variable $x \in \{0,1\}^m$ is an input/output in circuit C, and $\mathsf{wk} \in \mathsf{array}^\kappa[m, 2]$ is the set of wire key pairs corresponding to this variable in the garbled version of C, then $\{\mathsf{wk} : x\} \in \mathsf{array}^\kappa[m]$ denotes the wire-key representation of value x on these wires, i.e. $\{\mathsf{wk} : x\} = \{\mathsf{wk}[x[i]]\}_{i=1}^m$. If the set of keys is implicit we will denote $\{\mathsf{wk} : x\}$ as $\overline{x}$.

3PC ORAM Protocol. Our 3PC ORAM protocol, $\mathsf{3PC\text{-}ORAM.Access}$, Alg 1, performs the same recursive scan through data-structure $\mathsf{tree}_0, ..., \mathsf{tree}_{h-1}$ as the client-server Path-ORAM (and Circuit-ORAM) described in Section 2, included

Algorithm 1 3PC-ORAM.Access: 3PC Circuit-ORAM

Params: Address size $\log n$, address chunk size τ, number of trees $h = \frac{\log n}{\tau} + 1$
Input: $\langle \mathsf{OM}, \mathrm{N}, \mathsf{rec}' \rangle$, for $\mathsf{OM} = (\mathsf{tree}_0, ..., \mathsf{tree}_{h-1})$, $\mathrm{N} = (\mathrm{N}_1, ..., \mathrm{N}_{h-1})$
Output: $\langle \mathsf{rec} \rangle$: record stored in OM at address N

1: $\{\langle \mathrm{L}'_i \rangle \xleftarrow{\$} \{0,1\}^{i \cdot \tau}\}_{i=1}^{h-1}$; $\langle \mathrm{N}_0, \mathrm{N}_h, \mathrm{L}'_0, \mathrm{L}'_h \rangle := \perp$; $\mathrm{L}_0 := \perp$
2: **for** $i = 0$ **to** $h-1$ **do**
3PC-ORAM.ML: $\mathrm{L}_i, \langle \mathsf{tree}_i, (\mathrm{N}_0|...|\mathrm{N}_i), \mathrm{N}_{i+1}, \mathrm{L}'_i, \mathrm{L}'_{i+1}, * \mathsf{rec}' \rangle$
$\longrightarrow$ L_{i+1} (* $\langle \mathsf{rec} \rangle$ instead of L_{i+1}), $\langle \mathsf{tree}_i \rangle$

Algorithm 2 3PC-ORAM.ML: Main Loop of 3PC Circuit-ORAM

Param: Tree level index i. path depth d (number of buckets). Bucket size w.
Input: L_i, $\langle \mathsf{tree}, \mathrm{N}, \Delta\mathrm{N}, \mathrm{L}'_i, \mathrm{L}'_{i+1} \rangle$ (* $\langle \mathsf{rec}' \rangle$)
Output: (1) $\mathrm{L}_{i+1} = \mathsf{T.data}[\Delta\mathrm{N}]$ for tuple T on $\mathsf{tree.path}(\mathrm{L}_i)$ s.t. $\mathsf{T.(fb|adr)} = 1|\mathrm{N}$ (* $\langle \mathsf{rec} \rangle := \langle \mathsf{T.data} \rangle$)
(2) $\langle \mathsf{tree.path}(\mathrm{L}) \rangle$ modified by eviction, with $\mathsf{T.lb} := \mathrm{L}'_i$ and $\mathsf{T.data}[\Delta\mathrm{N}] := \mathrm{L}'_{i+1}$ (* $\mathsf{T.data} := \mathsf{rec}'$)
Offline: pick $(\pi, \delta, \rho)^{\mathsf{CE}}$, for $\pi \xleftarrow{\$} \mathsf{perm}_d$, $\delta, \rho \xleftarrow{\$} \mathsf{array}^{\log(w+1)}[d]$

Retrieval of Next Label/Record

$\langle \mathsf{path} \rangle := \langle \mathsf{tree.path}(\mathrm{L}_i) \rangle$

1: KSearch: $\langle \mathsf{path.(fb|adr)}, 1|\mathrm{N} \rangle \rightarrow \langle j \rangle_{\mathsf{shift}}$ ▷ $\mathsf{path}[j].\mathsf{(fb|adr)} = 1|\mathrm{N}$
2: 3ShiftPIR: $\langle \mathsf{path.data} \rangle, \langle j \rangle_{\mathsf{shift}} \rightarrow \langle X \rangle$ (* $\langle \mathsf{rec} \rangle := \langle X \rangle$) ▷ $X = \mathsf{path}[j].\mathsf{data}$
3: 3ShiftXorPIR: $\langle \mathsf{path.data}, \Delta\mathrm{N} \rangle, \langle j \rangle_{\mathsf{shift}} \rightarrow \mathrm{L}_{i+1}$(*skip) ▷ L_{i+1}=$\mathsf{path}[j].\mathsf{data}[\Delta\mathrm{N}]$

Post-Process of Found Tuple

4: ULiT: $\langle X, \mathrm{N}, \Delta\mathrm{N}, \mathrm{L}'_i, \mathrm{L}'_{i+1}$ (* rec')$\rangle, \mathrm{L}_{i+1} \rightarrow \langle \mathsf{T} \rangle$
▷ $X[\Delta\mathrm{N}] := \mathrm{L}'_{i+1}$ (* $X := \mathsf{rec}'$), $\mathsf{T} = (1, \mathrm{N}, \mathrm{L}'_i, X)$
5: FlipFlag: $\langle \mathsf{path.fb} \rangle, \langle j \rangle_{\mathsf{shift}} \rightarrow \langle \mathsf{path.fb} \rangle$ ▷ $\mathsf{path}[j].\mathsf{fb} := 0$
$\langle \mathsf{path} \rangle := \langle \mathsf{path.append\text{-}to\text{-}root(T)} \rangle$

Eviction

6: GC(Route): $\mathrm{L}_i, \delta^{\mathsf{CE}}, \langle \mathsf{path.(fb, lb)} \rangle \rightarrow (\overline{\sigma}, \mathsf{t}')^{\mathsf{D}}, \mathsf{wk}^{\mathsf{E}}$
▷ $\overline{\sigma} = \{\mathsf{wk} : \sigma\}$ and $\mathsf{t}' = \mathsf{t} \oplus \delta$ for expanded Circ-ORAM eviction map (σ, t)
7: PermBuckets: $\overline{\sigma}^{\mathsf{D}}, \pi^{\mathsf{CE}}, \mathsf{wk}^{\mathsf{E}} \rightarrow \sigma^{\circ \mathsf{D}}$ ▷ $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$
8: PermTuples: $\mathsf{t}'^{\mathsf{D}}, (\pi, \rho)^{\mathsf{CE}} \rightarrow \mathsf{t}^{\circ \mathsf{D}}$ ▷ $\mathsf{t}^\circ = \rho \oplus \pi(\mathsf{t}')$
9: XOT: $\langle \mathsf{path} \rangle, (\pi, \delta, \rho)^{\mathsf{CE}}, (\sigma^\circ, \mathsf{t}^\circ)^{\mathsf{D}} \rightarrow \langle \mathsf{path}' \rangle$ ▷ $\mathsf{path}' = \mathsf{EM}_{\sigma,\mathsf{t}}(\mathsf{path})$
$\langle \mathsf{tree.path}(\mathrm{L}_i) \rangle := \langle \mathsf{path}' \rangle$

*: On top-level ORAM tree, i.e. $i = h - 1$. ▷: Comments.

	rounds	bandwidth
KSearch	2	$\approx 2\ell(c+\log \ell)$
3ShiftPIR	2	$3\ell + 3\|\mathsf{data}\|$ for $\|\mathsf{data}\| = 2^\tau \|\mathrm{L}\|$
3ShiftXorPIR	2	$3 \cdot 2^\tau \ell + 6\|\mathrm{L}\|$
ULiT	2	$\approx 4\|\mathsf{data}\|$ $(+4\|\mathsf{data}\|$ offline)
FlipFlag	2	4ℓ
GC(Route)	1	$2\|x\|\kappa$ $(+4\|\mathsf{circ}\| + 2\|x\|)\kappa$ offline)
PermBuckets	2	$3d\log d$ $(+d^2(\kappa + 2\log d) + 3d\log d$ offline)
PermTuples	2	$2d(w+1)$ $(+d(w+1)$ offline)
XOT	3	$4\|\mathsf{path}\| + 2\ell\log(\ell)$ $(+2\|\mathsf{path}\|$ offline)

Fig. 5: Round and bandwidth for subprotocols of Alg. 2, for ℓ the number of tuples on path and x the circuit input size ($\approx \ell(d + \log n) + d\log(w+1)$)

for reference as Alg. 23 in Appendix B, except it runs on inputs in $\langle\cdot\rangle$ secret-sharing format, i.e. sharings of ORAM trees, $\langle \mathsf{tree}_0 \rangle, ..., \langle \mathsf{tree}_{h-1} \rangle$, sharing of address $\langle \mathrm{N} \rangle$, and sharing of a new record $\langle \mathsf{rec}' \rangle$ if $\mathsf{instr} = \mathsf{write}$. The main loop of 3PC-ORAM.Access, i.e. protocol 3PC-ORAM.ML, Alg. 2, also follows the corresponding client-server algorithm ORAM.ML, included for reference as Alg. 24 in Appendix B, except that apart of the current-level leaf label L which is known to all parties, all its other inputs are secret-shared as well.

Protocol 3PC-ORAM.ML calls subprotocols whose round/bandwidth specifications are stated in Fig. 5. (We omit computation costs because they are all comparable to link-encryption of communicated data). The low costs of these subprotocols are enabled by different forms of secret-sharings, e.g. xor versus additive, or 2-party versus 3-party, and by low-cost (or no cost) conversions between them. For implementations of these protocols we refer to Appendix A.

Three Phases of 3PC-ORAM.ML: Protocol 3PC-ORAM.ML computes on sharing $\langle \mathsf{path} \rangle$ for $\mathsf{path} = \mathsf{tree.path}(\mathrm{L})$ and it contains the same three phases as the client-server Path-ORAM, but implemented with specialized 3PC protocols:

(1) Retrieval runs protocol KSearch to compute "shift" (i.e. additive) sharing $\langle j \rangle_{\mathsf{shift}}$ of index for tuple $\mathsf{T} = \mathsf{path}[j]$ in path s.t. $\mathsf{path}[j].\mathsf{adr} = \mathrm{N}$ and $\mathsf{path}[j].\mathsf{fb} = 1$, i.e. it is the unique (and non-empty) tuple pertaining to address prefix N; Then it runs protocol 3ShiftPIR to extract sharing $\langle X \rangle$ of the payload $X = \mathsf{path}[j].\mathsf{data}$ of this tuple, given sharings $\langle \mathsf{path} \rangle$ and $\langle j \rangle_{\mathsf{shift}}$; In parallel to 3ShiftPIR it also runs protocol 3ShiftXorPIR to publicly reconstruct the next-level label stored at position ΔN in this tuple's payload, i.e. $\mathrm{L}_{i+1} = (\mathsf{path}[j].\mathsf{data})[\Delta\mathrm{N}]$, given sharing $\langle \mathsf{path} \rangle$ and $\langle \Delta\mathrm{N} \rangle$. This construction of the Retrieval emulation allows for presenting protocols 3ShiftPIR and 3ShiftXorPIR (see resp. Alg. 9 and 11 in Appendix A) as *generic* SS-PIR constructions from a class of 2-Server PIR protocols. However, a small modification of this design achieves better round *and* on-line bandwidth parameters, see an *Optimizations and Efficiency Discussion* paragraph below.

(2) Post-Process runs the Update-Label-in-Tuple protocol ULiT to form sharing $\langle \mathsf{T} \rangle$ of a new tuple using sharing $\langle X \rangle$ of the retrieved tuple's payload, sharings $\langle \mathrm{N} \rangle$ and $\langle \Delta\mathrm{N} \rangle$ of the address prefix and the next address chunk, and sharings

$\langle \mathrm{L}'_i \rangle, \langle \mathrm{L}'_{i+1} \rangle$ of new labels; In parallel to ULiT it also runs protocol FlipFlag to flip the full/empty flag to 0 in the old version of this tuple in path, which executes on inputs the sharings $\langle \mathsf{path.fb} \rangle$ of field fb of tuples in path and on the "shift" sharing $\langle j \rangle_{\mathsf{shift}}$; Once ULiT terminates the parties can insert $\langle \mathsf{T} \rangle$ into sharing of the root bucket in path. At this point the root bucket has size $s{+}1$ (or $s{+}b$ if we postpone eviction for a batch of b accesses).

(3) Eviction emulates Circuit-ORAM eviction on sharing $\langle \mathsf{path} \rangle$ involved in retrieval (or another path because 3PC-ORAM.Access, just like client-server Circuit-ORAM, performs eviction on two paths per access). It uses the generic garbled circuit protocol GC(Route) to compute the Circuit-ORAM eviction map (appropriately masked), and then runs protocols PermBuckets, PermTuples, and XOT to apply this (masked) eviction map to the secret-shared $\langle \mathsf{path} \rangle$. We discuss the eviction steps in more details below.

Eviction Procedure. As we explain in Section 2, we split Eviction into Eviction Logic, which uses garbled circuit subprotocol to compute the eviction map EM, and Eviction Movement, which uses special-purpose protocols to apply EM to the shared path, which in protocol 3PC-ORAM.ML will be $\langle \mathsf{path} \rangle$. However, recall that revealing the eviction map to any party would leak information about path density, and consequently the access pattern. We avoid this leakage in two steps: First, we modify the Circuit-ORAM eviction logic computation Route, so that when it computes bucket-level map Φ and the tuple pointers array t, which define an eviction map $\mathsf{EM}_{\Phi,\mathsf{t}}$, the algorithm scans through the buckets once more to *expand* the partial map Φ into a complete cycle σ over the d buckets (see Fig. 4). (We include the modified Circuit-ORAM algorithm Route in Appendix D.) Second, the garbled circuit computation GC(Route), see Step 6, Alg. 2, does not output (σ, t) to D in the clear: Instead, it outputs $\mathsf{t}' = \mathsf{t} \oplus \delta$ where δ is a random array, used here as a one-time pad, and the *garbled wire encoding* of the bits of $\sigma = [\sigma(1), ..., \sigma(d)]$, i.e. the output wire keys $\{\mathsf{wk} : \sigma\} = \mathsf{wk}[i][\sigma[i]]\}_{i=1}^{d \log d}$.

Recall that we want D to compute $(\sigma^\circ, \mathsf{t}^\circ)$, a masked version of (σ, t), where $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$ and $\mathsf{t}^\circ = \rho \oplus \pi(\mathsf{t} \oplus \delta)$, for π a random permutation on Z_d and δ, ρ random arrays, all picked by C and E. This is done by protocol PermBuckets, which takes 2 on-line rounds to let D translate $\{\mathsf{wk} : \sigma\}$ into $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$ given wk held by E and π held by C, E, and (in parallel) PermTuples, which takes 2 rounds to let D translate $\mathsf{t}' = \mathsf{t} \oplus \delta$ into $\mathsf{t}^\circ = \rho \oplus \pi(\mathsf{t}')$ given π, ρ held by C, E. Then C, E permute $\langle \mathsf{path} \rangle_{\mathsf{xor}}^{\mathsf{C-E}}$ (implied by $\langle \mathsf{path} \rangle$, because $\langle x \rangle = (x_1^{\mathsf{DE}}, x_2^{\mathsf{CE}}, x_3^{\mathsf{CD}}) \rightarrow (x_1^{\mathsf{E}}, x_2^{\mathsf{E}}, x_3^{\mathsf{C}}) = \langle x \rangle_{\mathsf{xor}}^{\mathsf{C-E}}$) by $\Pi = \tilde{\rho} \cdot \ddot{\pi} \cdot \tilde{\delta}$ where $\ddot{\pi}$, $\tilde{\delta}$, and $\tilde{\rho}$ are permutations on $\ell = d \cdot (w{+}1)$ tuples in the path induced by π, δ, ρ:

- $\pi \in \mathsf{perm}_d$ defines $\ddot{\pi} \in \mathsf{perm}_\ell$ s.t. $\ddot{\pi}(j,t) = (\pi(j), t)$, i.e. $\ddot{\pi}$ moves position t in bucket j to position t in bucket $\pi(j)$;
- $\delta \in \mathsf{array}^{\log(w+1)}[d]$ defines $\tilde{\delta} \in \mathsf{perm}_\ell$ s.t. $\tilde{\delta}(j,t) = (j, t \oplus \delta)$, i.e. $\tilde{\delta}$ moves position t in bucket j to position $t \oplus \delta[j]$ in bucket j; same for ρ and $\tilde{\rho}$;

Now use protocol XOT in 2 round and $\approx 2|\mathsf{path}|$ bandwidth to apply map $\mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ}$ held by D to $\langle \Pi(\mathsf{path}) \rangle_{\mathsf{xor}}^{\mathsf{C-E}}$. The result is $\langle \mathsf{path}^\circ \rangle_{\mathsf{xor}}^{\mathsf{C-E}}$ for $\mathsf{path}^\circ = (\mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ} \cdot \Pi)(\mathsf{path})$, and when C, E apply Π^{-1} to it they get $\langle \mathsf{path}' \rangle_{\mathsf{xor}}^{\mathsf{C-E}}$ for $\mathsf{path}' = (\Pi^{-1} \cdot \mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ} \cdot \Pi)(\mathsf{path})$. Finally $\langle \mathsf{path}' \rangle$ can be reconstructed from $\langle \mathsf{path}' \rangle_{\mathsf{xor}}^{\mathsf{C-E}}$ in 1 round and

$2|\mathsf{path}|$ bandwidth (see Appendix A for secret-sharing conversions and reasoning), and can then be injected into $\langle \mathsf{tree} \rangle$.

Eviction Correctness. We claim that the eviction protocol described above implements mapping $\mathsf{EM}_{\sigma,\mathsf{t}}$ applied to path, i.e. that (note that $(\tilde{x})^{-1} = \tilde{x}$):

$$\mathsf{EM}_{\sigma,\mathsf{t}} = \Pi^{-1} \cdot \mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ} \cdot \Pi = (\tilde{\delta} \cdot \ddot{\pi}^{-1} \cdot \tilde{\rho}) \cdot (\mathsf{EM}_{\pi\sigma\pi^{-1},\rho\oplus\pi(\mathsf{t}\oplus\delta)}) \cdot (\tilde{\rho} \cdot \ddot{\pi} \cdot \tilde{\delta}) \quad (1)$$

Consider the set of points $S = \{(j, \mathsf{t}[j]) \,|\, j \in \mathsf{Z}_d\}$ which are moved by the left hand side (LHS) permutation $\mathsf{EM}_{\sigma,\mathsf{t}}$. To argue that eq. (1) holds we first show that the RHS permutation maps any point $(j, \mathsf{t}[j])$ of S in the same way as the LHS permutation:

$$\begin{array}{rlcl}
(j,\mathsf{t}[j]) & \xrightarrow{(\tilde{\rho}\cdot\ddot{\pi}\cdot\tilde{\delta})} & (\pi(j), \rho[\pi(j)] \oplus \mathsf{t}[j] \oplus \delta[j]) & = \quad (\pi(j), \mathsf{t}^\circ[\pi(j)]) \\
& \xrightarrow{\mathsf{EM}_{\pi\sigma\pi^{-1},\mathsf{t}^\circ}} & (\pi\sigma\pi^{-1}(\pi(j)), \mathsf{t}^\circ[\pi\sigma\pi^{-1}(\pi(j))]) & = \quad (\pi\sigma(j), \mathsf{t}^\circ[\pi\sigma(j)]) \\
& = & (\pi\sigma(j), \rho[\pi\sigma(j)] \oplus \mathsf{t}[\sigma(j)] \oplus \delta[\sigma(j)]) & \\
& \xrightarrow{\tilde{\rho}} & (\pi\sigma(j), \mathsf{t}[\sigma(j)] \oplus \delta[\sigma(j)]) & \xrightarrow{\ddot{\pi}^{-1}} \quad (\sigma(j), \mathsf{t}[\sigma(j)] \oplus \delta[\sigma(j)]) \\
& \xrightarrow{\tilde{\delta}} & (\sigma(j), \mathsf{t}[\sigma(j)]) &
\end{array}$$

It remains to argue that RHS is an identity on points not in S, just like LHS. Observe that set S' of tuples moved by $\mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ}$ consists of the following tuples:

$$(k, \mathsf{t}^\circ[k]) = (k, \rho[k] \oplus \mathsf{t}[\pi^{-1}(k)] \oplus \delta[\pi^{-1}(k)]) = (\pi(j), \rho[\pi(j)] \oplus \mathsf{t}[j] \oplus \delta[j])$$

Also note that:

$$(\tilde{\rho}\cdot\ddot{\pi}\cdot\tilde{\delta})(j, \mathsf{t}[j]) = (\tilde{\rho}\cdot\ddot{\pi})(j, \mathsf{t}[j]\oplus\delta[j]) = \tilde{\rho}(\pi(j), \mathsf{t}[j]\oplus\delta[j]) = (\pi(j), \rho[\pi(j)]\oplus\mathsf{t}[j]\oplus\delta[j])$$

which means that $S' = \Pi(S)$, so if $(j,t) \notin S$ then $\Pi(j,t) \notin S'$, hence $(\mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ} \cdot \Pi)(j,t) = \Pi(j,t)$, and hence $\Pi^{-1} \cdot \mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ} \cdot \Pi$ and $\mathsf{EM}_{\sigma,\mathsf{t}}$ are equal on $(j,t) \notin S$.

Optimizations and Efficiency. As mentioned above, we can improve on both bandwidth and rounds in the Retrieval phase of 3PC-ORAM.ML shown in Alg. 2. The optimization comes from an observation that our protocol KSearch (see Alg. 6, App. A) takes just one round to compute shift-sharing $\langle j \rangle^{\mathsf{DE-C}}_{\mathsf{shift}}$ of index j, and its second round is a resharing which transforms $\langle j \rangle^{\mathsf{DE-C}}_{\mathsf{shift}}$ into $\langle j \rangle_{\mathsf{shift}}$. This round of resharing can be saved, and we can re-arrange protocols 3ShiftPIR and 3ShiftXorPIR (shown as Alg. 9 and 11 in App. A) so they use only $\langle j \rangle^{\mathsf{DE-C}}_{\mathsf{shift}}$ as input and effectively piggyback creating the rest of $\langle j \rangle_{\mathsf{shift}}$ in such a way that the modified protocols, denoted resp. 3ShiftPIR-Mod and 3ShiftXorPIR-Mod (shown as Alg. 12 and 13 in App. A) take 2 rounds, which makes the whole Retrieval take only 3 rounds, hence access protocol 3PC-ORAM.Access takes $3h$ rounds in Retrieval(Protocols 3ShiftPIR-Mod and 3ShiftXorPIR-Mod also use resp. 2ℓ and $2{\cdot}2^\tau\ell$ less bandwidth than 3ShiftPIR and 3ShiftXorPIR.)

Surprisingly, the modified Retrieval *and Post-Processing* phases together take only 3 rounds, amortized over the tree traversal, which enables pipelined processing of b accesses in $3b+3h$ rounds (with postponed eviction). Very briefly, this is because (1) the 2-round protocol FlipFlag can start after the first round of Retrieval (and thus terminates in round 3) because KSearch produces FlipFlag's input $\langle j \rangle^{\mathsf{DE-C}}_{\mathsf{shift}}$

in round 1; and (2) protocol ULiT has 2 rounds, but its first round can be computed in parallel to 1st round of KSearch because it needs only ΔN as an input, and while its 2nd round requires L_{i+1} which is output only in round 3 by 3ShiftXorPIR-Mod (other inputs of ULiT are available before), this 2nd round of ULiT can execute *in parallel* with the 1st round of Retrieval instance for the next access request on the same tree, and this is because the 1st round of retrieval consists of KSearch which takes only fb, adr fields of the tuples in path as inputs while the 2nd round of ULiT works only on the data field of tuple T.[6]

Eviction takes 6 rounds, which can run in parallel on all trees per access, and $O(\kappa \log^3 n + D \log n)$ bandwidth, which in practice is about 100x more than Retrieval and Post-Processing, but it can be postponed for a batch of accesses.

4 Security

Protocol 3PC-ORAM of Section 3 is a three-party secure computation of an Oblivious RAM functionality, i.e. it can implement RAM for any 3PC protocol in the RAM model. To state this formally we define a Universally Composable (UC) Oblivious RAM functionality $\mathsf{F}_{\mathsf{ORAM}}$ for 3-party computation (3PC) in the framework of Canetti [7], and we argue that our 3PC ORAM realizes $\mathsf{F}_{\mathsf{ORAM}}$ in the setting of $m = 3$ parties with *honest majority*, i.e. only $t = 1$ party is (statically) corrupted, assuming *honest-but-curious* (HbC) adversary, i.e. corrupted party follows the protocol. We assume secure pairwise links between the three parties. Since we have static-corruptions, HbC adversary, and non-rewinding simulators, security holds even if communication is asynchronous.

3PC ORAM Functionality. Functionality $\mathsf{F}_{\mathsf{ORAM}}$ is parametrized by address and record sizes, resp. $\log n$ and D, and it takes command Init, which initializes an empty array $\mathsf{M} \in \mathsf{array}^D[n]$, and $\mathsf{Access}(\mathsf{instr}, \langle \mathrm{N}, \mathsf{rec}' \rangle)$ for $(\mathsf{instr}, \mathrm{N}, \mathsf{rec}') \in \{\mathsf{read}, \mathsf{write}\} \times \{0,1\}^{\log n} \times \{0,1\}^D$, which returns a *fresh* secret-sharing $\langle \mathsf{rec} \rangle$ of record $\mathsf{rec} = \mathsf{M}[\mathrm{N}]$, and if $\mathsf{instr} = \mathsf{write}$ it also assigns $\mathsf{M}[\mathrm{N}] := \mathsf{rec}'$. Technically, $\mathsf{F}_{\mathsf{ORAM}}$ needs each of the three participating parties to make the call, where each party provides their part of the sharing, and $\mathsf{F}_{\mathsf{ORAM}}$'s output $\langle \mathsf{rec} \rangle$ is also delivered in the form of a corresponding share to each party. However, in the HbC setting all parties are assumed to follow the instructions provided by an *environment* algorithm Z, which models higher-level protocol which utilizes $\mathsf{F}_{\mathsf{ORAM}}$ to implement oblivious memory access. Hence we can simply assume that Z sends Init and $\mathsf{Access}(\mathsf{instr}, \langle \mathrm{N}, \mathsf{rec}' \rangle)$ to $\mathsf{F}_{\mathsf{ORAM}}$ and receives $\langle \mathsf{M}[\mathrm{N}] \rangle$ in return.

Security of our 3PC ORAM. Since our protocol is a three-party secure *emulation* of Circuit-ORAM, we prove that it securely realizes $\mathsf{F}_{\mathsf{ORAM}}$ in the $(t, m) = (1, 3)$ setting *if* Circuit-ORAM defines a secure Client-Server ORAM, which implies security of 3PC-ORAM by the argument for Circuit-ORAM security given in [27]. We note that protocol 3PC-ORAM.Access of Section 3 implements only procedure Access. Procedure Init can be implemented by running 3PC-ORAM.Access with $\mathsf{instr} = \mathsf{write}$ in a loop for N from 0 to $n-1$ (and arbitrary rec's), but this requires

[6] We include Fig. 10 in Appendix E.1 to visualize the dependencies between subprotocols of 3PC-ORAM.ML, both in single and pipelined execution.

some adjustments in 3PC-ORAM.Access and 3PC-ORAM.ML to deal with initialization of random label assignments and their linkage. We leave the specification of these (straightforward) adjustments to the full version, and our main security claim, stated as Corollary 1 below, assumes that Init is executed by a trusted-party.

We defer the proof of Corollary 1 to Appendix C. Very briefly, the proof uses UC framework, arguing that each protocol securely realizes its intended input/output functionality *if* each subprotocol it invokes realizes its idealized input/output functionality. All subprotocols executed by protocol 3PC-ORAM.ML of Section 3 are accompanied with brief security arguments which argue precisely this statement. As for 3PC-ORAM.ML, its security proof, given in Appendix C, is centered around two facts argued in Section 3, namely that our way of implementing Circuit-ORAM eviction map, with D holding $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$ and $\mathsf{t}^\circ = \rho \oplus \pi(\mathsf{t} \oplus \delta)$ and E, C holding π, ρ, δ is (1) correct, because $\mathit{\Pi}^{-1} \cdot \mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ} \cdot \mathit{\Pi} = \mathsf{EM}_{\sigma,\mathsf{t}}$ for $\mathit{\Pi} = \tilde{\rho} \cdot \ddot{\pi} \cdot \tilde{\delta}$, and (2) it leaks no information to either party, because random π, ρ, δ induce random $\sigma^\circ, \mathsf{t}^\circ$ in D's view.

Corollary 1 (from Appendix C) *Assuming secure initialization,* 3PC-ORAM.Access *is a UC-secure realization of 3PC ORAM functionality* $\mathsf{F}_{\mathsf{ORAM}}$.

5 Performance Evaluation

We tested a Java prototype of our 3PC Circuit-ORAM, with garbled circuits implemented using the ObliVM library by Wang [27], on three AWS EC2 c4.2xlarge servers, with communication links encrypted using AES-128. Each c4.2xlarge instance is equipped with eight Intel Xeon E5-2666 v3 CPU's (2.9 GHz), 15 GB memory, and has 1 Gbps bandwidth. (However, our tested prototype utilizes multithreading only in parallel Eviction, see below.)

In the discussion below we use the following acronyms:
- cust-3PC: our 3PC Circuit-ORAM protocol;
- gen-3PC: generic 3PC Circuit-ORAM using 3PC of Araki et al. [1];
- 2PC: 2PC Circuit-ORAM [27];
- C/S: the client-server Path-ORAM [26].

Wall Clock Time. Fig. 6 shows online timing of cust-3PC for small record sizes ($D = 4$B) as a function of address size $\log n$. It includes Retrieval wall clock time (WC), End-to-End (Retrieval+PostProcess+Eviction) WC, and End-to-End WC with *parallel* Eviction for all trees, which shows 60% reduction in WC due to better CPU utilization. Note that Retrieval takes about 8 milliseconds for $\log n = 30$ (i.e. 2^{30} records), and that Eviction takes only about 4-5 times longer. Recall that Retrieval phase has $3h$ rounds while Eviction has 6, which accounts for much smaller CPU utilization in Retrieval.

CPU Time. We compare total and online CPU time of cust-3PC and 2PC in Fig. 7 with respect to memory size n, for $D = 4$B.[7] Since 2PC implementation [27] does

[7] We include CPU comparisons only with 2PC Circuit-ORAM, and not SQRT-ORAM [30] and DPF-ORAM [12], because the former uses the same Java ObliVM GC library while the latter two use the C library Obliv-C. Still, note that for $n = 30$, the on-line

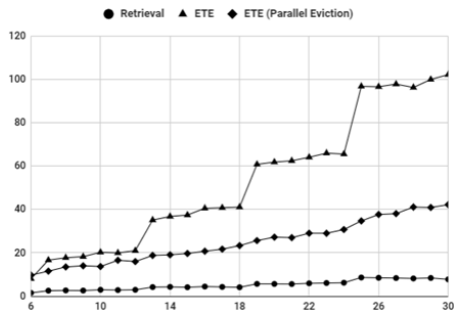

Fig. 6: Our 3PC-ORAM Online Wall-Clock Time(ms) vs $\log n$ for $D = 4$B

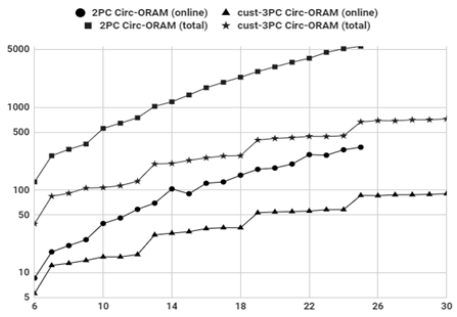

Fig. 7: CPU Time (ms) vs $\log n$, for $D = 4$B

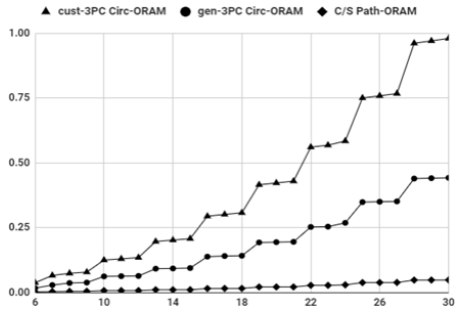

Fig. 8: Online bndw.(MB) vs $\log n$ for D=4B

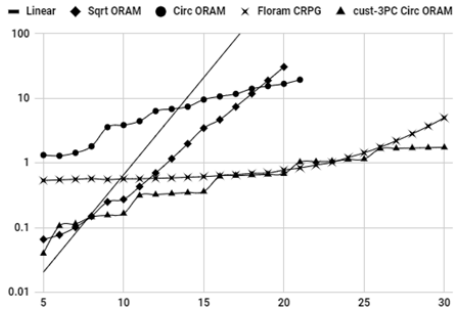

Fig. 9: Comparison with 2PC-ORAM's in online+offline bndw.(MB) vs $\log n$ for D=4B

not provide online/offline separation, we approximate 2PC online CPU time by its garbled circuit evaluation time, because 2PC costs due to OT's can be pushed to precomputation. As Fig. 7 shows, the cust-3PC CPU costs are between 6x and 10x lower than in 2PC, resp. online and total, already for $\log n = 25$, and the gap widens for higher n. In Appendix E.2 we include CPU time comparison with respect to D, which shows CPU ratio of 2PC over cust-3PC grows to ≈ 25 for $D \geq 10$KB.

Bandwidth Comparison with Generic 3PC. Timing results depend on many factors (language, network, CPU, and more), and bandwidth is a more reliable predictor of performance for protocols using only light symmetric crypto. In Fig. 8 we compare online bandwidth of cust-3PC, gen-3PC, and C/S, as a function of the address size $\log n$, for $D = 4$B. We see for small records our cust-3PC is only a factor of 2x worse than the optimal-bandwidth gen-3PC (which, recall, has completely impractical round complexity). In Appendix E.2 we show that as D grows, cust-3PC beats gen-3PC in bandwidth for $D \geq 1$KB.

Bandwidth Comparison with 2PC ORAMs. In Fig. 9 we compare total bandwidth of cust-3PC and several 2PC ORAM schemes, including 2PC, the DPF-based FLORAM scheme of [12], the 2PC SQRT-ORAM of [30], and a trivial linear-scan scheme. Our cust-3PC bandwidth is competitive to FLORAM for all n's,

computation due to FSS evaluation and linear memory scans contributes over 1 sec to wall-clock in [12], while our on-line wall-clock comes to 40 msec.

but for $n \geq 24$ the $O(\sqrt{n})$ asymptotics of FLORAM takes over. Note also that FLORAM uses $O(n)$ local computation vs. our $O(\log^3 n)$, so in the FLORAM case bandwidth comparison does not suffice. Indeed, for $n = 2^{30}$ and $D = 4$B, [12] report > 1 sec overall processing time on LAN vs. 40 msec for us.

For further discussions of bandwidth and CPU time with respect to record size D, and cust-3PC CPU time component, refer to Appendix E.2.

References

1. T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 805–817, 2016.
2. A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers computation in private information retrieval: Pir with preprocessing. *J. Cryptol.*, Mar. 2004.
3. 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.
4. D. Bogdanov, L. Kamm, and B. Kubo. Students and taxes: a privacy-preserving study using secure computation. In *Proceedings on Privacy Enhancing Technologies (PET)*, pages 117–135, 2016.
5. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, pages 192–206, 2008.
6. P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft. *Secure Multiparty Computation Goes Live*, pages 325–343. Springer Berlin Heidelberg, 2009.
7. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42Nd IEEE Symposium on Foundations of Computer Science*, FOCS '01, Washington, DC, USA, 2001. IEEE Computer Society.
8. D. Chaum, C. Crépeau, and I. Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 11–19, 1988.
9. B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, Nov. 1998.
10. I. Damgard, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious ram without random oracles. In *Theory of Cryptography*, pages 144–163, 2011.
11. S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. *Onion ORAM: A Constant Bandwidth Blowup Oblivious RAM*, pages 145–174. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
12. J. Doerner and A. Shelat. Scaling oram for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 523–535, New York, NY, USA, 2017. ACM.
13. Z. Dvir and S. Gopi. 2 serverr pir with subpolynomial communication. *J. ACM*, 63(4):39:1–39:15, Sept. 2016.

14. S. Faber, S. Jarecki, S. Kentros, and B. Wei. *Three-Party ORAM for Secure Computation*, pages 360–385. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.
15. C. W. Fletcher, M. Naveed, L. Ren, E. Shi, and E. Stefanov. Bucket oram: Single online roundtrip, constant bandwidth oblivious ram. *IACR Cryptology ePrint Archive*, 2015:1065, 2015.
16. 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.
17. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.
18. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.
19. 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.
20. Y. Ishai, E. Kushilevitz, S. Lu, and R. Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In *Proceedings of the RSA Conference on Topics in Cryptology - CT-RSA 2016 - Volume 9610*, pages 90–107, New York, NY, USA, 2016. Springer-Verlag New York, Inc.
21. S. Jarecki and B. Wei. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018*, 2018.
22. 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.
23. 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.
24. L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious ram. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 415–430, Berkeley, CA, USA, 2015. USENIX Association.
25. E. Shi, T. H. H. Chan, E. Stefanov, and M. Li. *Oblivious RAM with O((logN)3) Worst-Case Cost*, pages 197–214. Springer Berlin Heidelberg, 2011.
26. 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 *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security*, CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.
27. X. Wang, H. Chan, and E. Shi. Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 850–861, New York, NY, USA, 2015. ACM.
28. X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. Scoram: Oblivious ram for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 191–202, New York, NY, USA, 2014. ACM.
29. 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.

30. S. Zahur, X. Wang, M. Raykova, A. Gascn, J. Doerner, D. Evans, and J. Katz. Revisiting square-root oram efficient random access in multi-party computation. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, IEEE '16, 2016.

A Auxiliary Three-Party Protocols

In this section we specify all sub-protocols used in the 3PC-ORAM protocol 3PC-ORAM.ML, Alg. 2, of Section 3, together with the round/bandwidth characteristics of their implementation. In subsection A.2 we also include modified versions of protocols 3ShiftPIR, Alg. 9, and 3ShiftXorPIR, Alg. 11, namely protocols 3ShiftPIR-Mod, Alg. 12, and 3ShiftXorPIR-Mod, Alg. 13. As we explain in Section 3, using these protocols results in reducing the round complexity of 3PC-ORAM.ML from 4 to 3 per ORAM tree in the retrieval and post-processing phase.

Types of Secret-Sharing. In Alg. 3 we list the types of secret-sharing used in all our protocols. Random sharings of the first three types can be chosen non-interactively by random sampling each sharing component. First four sharings are xor-homomorphic, e.g. for any shared variables $\langle x\rangle, \langle y\rangle$ and constant c, we write $\langle x \oplus y\rangle$ and $\langle x \oplus c\rangle$ for sharing of $x \oplus y$ and $x \oplus c$ locally computed by all players. We can transform one sharing to another via either local transformations, denoted Extract, Alg. 4, or via 1-round protocols, denoted Reshare, Alg. 5. All Reshare protocols output *fresh* sharings of the target type, while the non-interactive transformations Extract are deterministic.

Additional Tools and Notation. In the description of some of the protocols in this section we will find it helpful to use shortcuts which we list below.

Sometimes we need to randomize a secret-sharing with a fresh *zero-sharing*, i.e. a random secret-sharing of a zero. We will use a two-party zero-sharing, $\langle 0^m\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2} = (x_1^{\mathsf{P}_1}, x_2^{\mathsf{P}_2})$, generated by sampling $r^{\mathsf{P}_1\mathsf{P}_2} \xleftarrow{\$} \{0,1\}^m$ and setting $(x_1, x_2) := (r, r)$, and a three-party zero-sharing, $\langle 0^m\rangle_{\mathsf{xor}} = (x_1^{\mathsf{C}}, x_2^{\mathsf{D}}, x_3^{\mathsf{E}})$, generated by sampling $r_1^{\mathsf{DE}} \xleftarrow{\$} \{0,1\}^m$, $r_2^{\mathsf{CE}} \xleftarrow{\$} \{0,1\}^m$, $r_3^{\mathsf{CD}} \xleftarrow{\$} \{0,1\}^m$, and setting $(x_1, x_2, x_3) := (r_2 \oplus r_3, r_1 \oplus r_3, r_1 \oplus r_2)$.

If $\sigma \in \mathsf{perm}_m$ and we permute σ by $\pi \in \mathsf{perm}_m$, then the result, $[\rho] = \pi(\sigma)$, encodes permutation $\rho = \sigma \cdot \pi^{-1}$, because $\rho(i) = \rho[i] = \sigma[\pi^{-1}(i)] = \sigma(\pi^{-1}(i)) = (\sigma \cdot \pi^{-1})(i)$. More generally, consider a 1-1 function $\sigma : \mathsf{Z}_k \to \mathsf{Z}_m$ for $m \geq k$, and a relation in $\mathsf{Z}_m \times \mathsf{Z}_k$ defined as $\sigma^{-1} = \{(j, i) \text{ s.t. } (i, j) \in \sigma\}$. If $m > k$ then $\pi = \sigma^{-1}$ is not a function, but if $x \in \mathsf{array}[m]$ then $\pi(x)$ denotes an array $y \in \mathsf{array}[k]$ s.t. $y[i] = x[\pi^{-1}(i)] = x[\sigma(i)]$ for $i \in \mathsf{Z}_k$.

We use additional shortcuts, for $a \in \mathsf{array}^\ell[m]$, $t \in \mathsf{Z}_m$, $p \in \{0,1\}^k$, $v \in \{0,1\}^\ell$:

- $a_{\mathsf{shift}[t]}$ denotes $b \in \mathsf{array}^\ell[m]$ s.t. $b[i] = a[i + t \bmod m]$ for $i \in \mathsf{Z}_m$;
- $a_{\mathsf{rot}[p]}$ denotes array $b \in \mathsf{array}^\ell[m]$ s.t. $b[i \oplus p] = a[i]$ for $i \in \mathsf{Z}_m$;
- $a^{\mathsf{xor}[\,v\ @\ t]}$ denotes $b \in \mathsf{array}^\ell[m]$ s.t. $b[t] = a[t] \oplus v$ and $b[i] = a[i]$ for $i \neq t$;
- ind_t^m denotes $x \in \mathsf{array}^1[m]$ s.t. $x[t] = 1$ and $x[i] = 0$ for all $i \neq t$;
- $t_{\mathsf{bit}[i]}$ denotes the i-th bit of binary representation of t, for $i \in \log(m)$.

Algorithm 3 Types of Secret-Sharing

- $\langle x \rangle \;=\; (x_1^{\mathsf{DE}}, x_2^{\mathsf{CE}}, x_3^{\mathsf{CD}})$ for $x_1, x_2, x_3 \xleftarrow{\$} \{0,1\}^{|x|}$ s.t. $x_1 \oplus x_2 \oplus x_3 = x$
- $\langle x \rangle_{\mathsf{xor}} \;=\; (x_1^{\mathsf{C}}, x_2^{\mathsf{D}}, x_3^{\mathsf{E}})$ for $x_1, x_2, x_3 \xleftarrow{\$} \{0,1\}^{|x|}$ s.t. $x_1 \oplus x_2 \oplus x_3 = x$
- $\langle x \rangle_{\mathsf{xor}}^{\mathsf{P}_1\text{–}\mathsf{P}_2} \;=\; (x_1^{\mathsf{P}_1}, x_2^{\mathsf{P}_2})$ for $x_1, x_2 \xleftarrow{\$} \{0,1\}^{|x|}$ s.t. $x_1 \oplus x_2 = x$
- $\langle x \rangle_{\mathsf{2,1\text{-}xor}} \;=\; (\, \langle x \rangle_{\mathsf{2,1\text{-}xor}}^{\mathsf{CD}\text{–}\mathsf{E}}\,,\; \langle x \rangle_{\mathsf{2,1\text{-}xor}}^{\mathsf{DE}\text{–}\mathsf{C}}\,,\; \langle x \rangle_{\mathsf{2,1\text{-}xor}}^{\mathsf{EC}\text{–}\mathsf{D}}\,)$,
  where $\langle x \rangle_{\mathsf{2,1\text{-}xor}}^{\mathsf{P}_1\mathsf{P}_2\text{–}\mathsf{P}_3} = (x_{12}^{\mathsf{P}_1\mathsf{P}_2}, x_3^{\mathsf{P}_3})$ for $x_{12}, x_3 \xleftarrow{\$} \{0,1\}^{|x|}$ s.t. $x_{12} \oplus x_3 = x$
- $\langle x \rangle_{\mathsf{shift}} \;=\; (\, \langle x \rangle_{\mathsf{shift}}^{\mathsf{CD}\text{–}\mathsf{E}}\,,\; \langle x \rangle_{\mathsf{shift}}^{\mathsf{DE}\text{–}\mathsf{C}}\,,\; \langle x \rangle_{\mathsf{shift}}^{\mathsf{EC}\text{–}\mathsf{D}}\,)$, for $x \in \mathsf{Z}_\ell$,
  where $\langle x \rangle_{\mathsf{shift}}^{\mathsf{P}_1\mathsf{P}_2\text{–}\mathsf{P}_3} = (x_{12}^{\mathsf{P}_1\mathsf{P}_2}, x_3^{\mathsf{P}_3})$ for $x_{12}, x_3 \xleftarrow{\$} \mathsf{Z}_\ell$ s.t. $x_{12} + x_3 = x \bmod \ell$

Algorithm 4 Extract on input $\langle x \rangle = (x_1^{\mathsf{P}_1\mathsf{P}_2}, x_2^{\mathsf{P}_2\mathsf{P}_3}, x_3^{\mathsf{P}_3\mathsf{P}_1})$

$\longrightarrow\;\; \langle x \rangle_{\mathsf{xor}}^{\mathsf{P}_1\text{–}\mathsf{P}_2} = ((x_1 \oplus x_3)^{\mathsf{P}_1}, {x_2}^{\mathsf{P}_2})$
$\longrightarrow\;\; \langle x \rangle_{\mathsf{xor}} \;\;\;\;\;\; = ({x_1}^{\mathsf{P}_1}, {x_2}^{\mathsf{P}_2}, {x_3}^{\mathsf{P}_3})$
$\longrightarrow\;\; \langle x \rangle_{\mathsf{2,1\text{-}xor}} = (({x_1}^{\mathsf{P}_1\mathsf{P}_2}, (x_2 \oplus x_3)^{\mathsf{P}_3}), ({x_2}^{\mathsf{P}_2\mathsf{P}_3}, (x_1 \oplus x_3)^{\mathsf{P}_1}), ({x_3}^{\mathsf{P}_3\mathsf{P}_1}, (x_1 \oplus x_2)^{\mathsf{P}_2}))$

Algorithm 5 Reshare: Interactive Sharing Transformation

(i) $\langle x \rangle_{\mathsf{xor}}^{\mathsf{P}_1\text{–}\mathsf{P}_2} = (x_1^{\mathsf{P}_1}, x_2^{\mathsf{P}_2}) \;\;\longrightarrow\;\; \langle x \rangle = (z_1^{\mathsf{P}_2\mathsf{P}_3}, z_2^{\mathsf{P}_1\mathsf{P}_3}, z_3^{\mathsf{P}_1\mathsf{P}_2})$
Pick $z_1^{\mathsf{P}_2\mathsf{P}_3}, z_2^{\mathsf{P}_1\mathsf{P}_3} \xleftarrow{\$} \{0,1\}^{|x|}$;
P_1 and P_2 exchange $(x_1 \oplus z_2)$ and $(x_2 \oplus z_1)$, set $z_3 := (x_1 \oplus z_2) \oplus (x_2 \oplus z_1)$

(ii) $\langle x \rangle_{\mathsf{xor}} = (x_1^{\mathsf{P}_1}, x_2^{\mathsf{P}_2}, x_3^{\mathsf{P}_3}) \;\;\longrightarrow\;\; \langle x \rangle = (z_1^{\mathsf{P}_1\mathsf{P}_2}, z_2^{\mathsf{P}_2\mathsf{P}_3}, z_3^{\mathsf{P}_3\mathsf{P}_1})$
Generate random m-bit zero-sharing, $(s_1^{\mathsf{P}_1}, s_2^{\mathsf{P}_2}, s_3^{\mathsf{P}_3}) \xleftarrow{\$} \langle 0^m \rangle_{\mathsf{xor}}$;
Each P_i sets $z_i := x_i \oplus s_i$ and sends z_i to $\mathsf{P}_{(i+1 \bmod 3)}$

(iii) $\langle x \rangle_{\mathsf{shift}}^{\mathsf{P}_2\mathsf{P}_3\text{–}\mathsf{P}_1} = (x_{23}^{\mathsf{P}_2\mathsf{P}_3}, x_1^{\mathsf{P}_1}) \;\rightarrow\; \langle x \rangle_{\mathsf{shift}} = (\langle x \rangle_{\mathsf{shift}}^{\mathsf{P}_1\mathsf{P}_2\text{–}\mathsf{P}_3}, \langle x \rangle_{\mathsf{shift}}^{\mathsf{P}_2\mathsf{P}_3\text{–}\mathsf{P}_1}, \langle x \rangle_{\mathsf{shift}}^{\mathsf{P}_3\mathsf{P}_1\text{–}\mathsf{P}_2})$
If ${x_{12}}^{\mathsf{P}_1\mathsf{P}_2} \xleftarrow{\$} \mathsf{Z}_m$, P_1 sends $\delta = x_1 - x_{12}$ to P_3, and P_3 sets $x_3 := x_{23} + \delta \pmod m$, then $\langle x \rangle_{\mathsf{shift}}^{\mathsf{P}_1\mathsf{P}_2\text{–}\mathsf{P}_3} := (x_{12}^{\mathsf{P}_1\mathsf{P}_2}, x_3^{\mathsf{P}_3})$; ($\langle x \rangle_{\mathsf{shift}}^{\mathsf{P}_3\mathsf{P}_1\text{–}\mathsf{P}_2}$ computed likewise.)

(iv) $\langle x \rangle_{\mathsf{xor}}^{\mathsf{P}_1\text{–}\mathsf{P}_2} = (x_1^{\mathsf{P}_1}, x_2^{\mathsf{P}_2})\;,\; z_1^{\mathsf{P}_1} \;\;\longrightarrow\;\; z_2^{\mathsf{P}_2}$ s.t. $(z_1^{\mathsf{P}_1}, z_2^{\mathsf{P}_2}) = \langle x \rangle_{\mathsf{xor}}^{\mathsf{P}_1\text{–}\mathsf{P}_2}$
P_1 sends $\delta = x_1 \oplus z_1$ to P_2 who sets $z_2 := x_2 \oplus \delta$
Note: Protocol Reshare *type (iv) is* deterministic *given input z_1, but* InsertLbl*, Alg. 15, invokes it on random z_1, making (z_1, z_2) a fresh sharing of x.*

Bandwidth: (i): $2|x|$, (ii): $3|x|$, (iii): $2|x|$, (iv): $|x|$; Rounds: 1 (for each protocol);
Security: (i): Message $x_2 \oplus z_1$ received by P_1 can be computed from P_1's input and output as $x_1 \oplus z_2 \oplus z_3$, likewise for P_2; (ii) Sharing (z_1, z_2, z_3) is fresh by security of zero-sharing, and each party receives only its output; (iii) Sharing (x_{23}, x_1) is fresh, and value δ received by P_3 can be computed from P_3's input and output; (iv) P_2 can compute message δ from its input and output.

A.1 Protocols for Retrieval

Algorithm 6 Protocol $\mathsf{KSearch}$ (from [14]) (Step 1 in Alg 2)

Param: Security parameter κ, round-complexity parameter λ,
PRF $\mathsf{F} : \{0,1\}^\kappa \to \{0,1\}^\lambda$, array size ℓ, record size $c \le \kappa$

Input: $\langle u, v\rangle_{\mathsf{xor}}^{\mathsf{D}-\mathsf{E}}$ for $u \in \mathsf{array}^c[\ell]$, $v \in \{0,1\}^c$ s.t. $u[i] = v$ for *exactly one* $i \in \mathsf{Z}_\ell$

Output: $\langle i\rangle_{\mathsf{shift}}$ for i as above, i.e. unique i s.t. $u[i] = v$

Offline: $\mathsf{k}^{\mathsf{DE}} \xleftarrow{\$}$ PRF F keyspace (can be re-used for multiple protocol instances)

1: $r^{\mathsf{DE}} \xleftarrow{\$} \mathsf{array}^\kappa[\ell]$, $s^{\mathsf{DE}} \xleftarrow{\$} \mathsf{Z}_\ell$; $(a^{\mathsf{D}}, b^{\mathsf{E}}) := \langle z\rangle_{\mathsf{xor}}^{\mathsf{D}-\mathsf{E}}$ where $\langle z\rangle_{\mathsf{xor}}^{\mathsf{D}-\mathsf{E}}$ is locally transformed from $\langle u, v\rangle_{\mathsf{xor}}^{\mathsf{D}-\mathsf{E}}$ s.t. $z[j] = u[j+s \bmod \ell] \oplus v$ for all j
D sends array x to C s.t. $x[j] = \mathsf{F}_\mathsf{k}(\, r[j] \oplus (a[j] \,|\, 0^{\kappa-c})\,)$ for all j
E sends array y to C s.t. $y[j] = \mathsf{F}_\mathsf{k}(\, r[j] \oplus (b[j] \,|\, 0^{\kappa-c})\,)$ for all j

2: Let $\langle i\rangle_{\mathsf{shift}}^{\mathsf{DE}-\mathsf{C}} := (s^{\mathsf{DE}}, t^{\mathsf{C}})$ for unique t s.t. $x[t] = y[t]$
(if $x[t] = y[t]$ for ≥ 2 t's, C asks D, E to re-run from step 1 with fresh r^{DE})

3: Reshare: $\langle i\rangle_{\mathsf{shift}}^{\mathsf{DE}-\mathsf{C}} \;\to\; \langle i\rangle_{\mathsf{shift}}$

Bandwidth: $\approx(1 + 2^{-\lambda+1})2\ell\lambda$; Rounds: 2 (*with* $\le 2^{-\lambda+\ln \ell}$ *re-run probability*);
Security: By randomness of PRF pre-pad r and PRF property of F, each pair $x[j], y[j]$ of entries in x, y received by C is indistinguishable from a random pair of λ-bit values except for unique i in Z_ℓ s.t. $a[i] = b[i]$, where $x[i] = y[i]$ is distributed as a single random λ-bit value.
Note: This holds over multiple executions with same PRF key k *due to freshness of* r.

Algorithm 7 Protocol SSPIR (from [9]) (Used in Alg. 10 and 8)

Input: $x^{\mathsf{P}_1\mathsf{P}_2} \in \mathsf{array}^m[n]$, $t^{\mathsf{P}_3} \in \mathsf{Z}_n$

Output: $\langle x[t]\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$

Pick $a_1^{\mathsf{P}_1\mathsf{P}_3} \xleftarrow{\$} \{0,1\}^n$, $r^{\mathsf{P}_1\mathsf{P}_2} \xleftarrow{\$} \{0,1\}^m$;

1: P_3 sends $a_2 = a_1 \oplus \mathsf{ind}_t^n$ to P_2 (Note that $a_2 = a_1$ except $a_2[t] = a_1[t] \oplus 1$)

2: P_1 sets $z_1 := r \oplus \mathsf{XORSelect}(x, a_1)$, and P_2 sets $z_2 := r \oplus \mathsf{XORSelect}(x, a_2)$, where $\mathsf{XORSelect}(x, a) = \bigoplus_i$ s.t. $_{a[i]=1}\ x[i]$

Output $\langle x[t]\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2} = (z_1^{\mathsf{P}_1}, z_2^{\mathsf{P}_2})$.

Bandwidth: n; Rounds: 1;
Security: This is the basis of security of PIR of Chor et al. [9]: P_2's received message a_2 is a random string because a_1 is a one-time pad. Moreover, the secret-sharing of $x[t]$ is random because r is a one-time pad.

Algorithm 8 Protocol $\mathsf{ShiftPIR}$ (Used in Alg. 9)

Input: $x^{\mathsf{P}_1\mathsf{P}_2} \in \mathsf{array}^\ell[m]$, $\langle i\rangle_{\mathsf{shift}}^{\mathsf{P}_1\mathsf{P}_2-\mathsf{P}_3} = (s^{\mathsf{P}_1\mathsf{P}_2}, t^{\mathsf{P}_3})$, for $i \in \mathsf{Z}_m$

Output: $\langle x[i]\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$

1: P_1 and P_2 set $x' := x_{\mathsf{shift}[s]}$, i.e. $x'[j] = x[j + s \bmod m]$ for all j

2: SSPIR: $(x')^{\mathsf{P}_1\mathsf{P}_2}, t^{\mathsf{P}_3} \;\to\; \langle x'[t]\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$ $\quad(= \langle x[i]\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2})$

Bandwidth: m; Rounds: 1; Security: No message sent besides SSPIR.

Algorithm 9 Protocol 3ShiftPIR (Step 2 in Alg 2)

Input: $\langle x \rangle = (x_1^{\mathsf{DE}}, x_2^{\mathsf{EC}}, x_3^{\mathsf{CD}})$, $\langle i \rangle_{\mathsf{shift}}$, for $x \in \mathsf{array}^{\ell}[m]$, $i \in \mathsf{Z}_m$

Output: $\langle x[i] \rangle$

1: ShiftPIR: $x_1^{\mathsf{DE}}, \langle i \rangle_{\mathsf{shift}}^{\mathsf{DE-C}} \rightarrow \langle x_1[i] \rangle_{\mathsf{xor}}^{\mathsf{D-E}} = (d_1^{\mathsf{D}}, e_1^{\mathsf{E}})$
ShiftPIR: $x_2^{\mathsf{EC}}, \langle i \rangle_{\mathsf{shift}}^{\mathsf{EC-D}} \rightarrow \langle x_2[i] \rangle_{\mathsf{xor}}^{\mathsf{E-C}} = (e_2^{\mathsf{E}}, c_1^{\mathsf{C}})$
ShiftPIR: $x_3^{\mathsf{CD}}, \langle i \rangle_{\mathsf{shift}}^{\mathsf{CD-E}} \rightarrow \langle x_3[i] \rangle_{\mathsf{xor}}^{\mathsf{C-D}} = (c_2^{\mathsf{C}}, d_2^{\mathsf{D}})$
Note: $(d_1 \oplus e_1) \oplus (e_2 \oplus c_1) \oplus (c_2 \oplus d_2) = x_1[i] \oplus x_2[i] \oplus x_3[i] = x[i]$
2: Reshare: $\langle x[i] \rangle_{\mathsf{xor}} = \big((c_1 \oplus c_2)^{\mathsf{C}}, (d_1 \oplus d_2)^{\mathsf{D}}, (e_1 \oplus e_2)^{\mathsf{E}}\big) \longrightarrow \langle x[i] \rangle$

Bandwidth: $3(m + \ell)$; Rounds: 2;
Security: No message is sent besides secure computation ShiftPIR and Reshare, which outputs random shares to each participant.

Algorithm 10 Protocol ShiftXorPIR (Used in Alg. 11)

Input: $x^{\mathsf{P_1P_2}}$, $\langle i_1 \rangle_{\mathsf{shift}}^{\mathsf{P_1P_2-P_3}} = (s_1^{\mathsf{P_1P_2}}, t_1^{\mathsf{P_3}})$, $\langle i_2 \rangle_{\mathsf{2,1\text{-}xor}}^{\mathsf{P_1P_2-P_3}} = (s_2^{\mathsf{P_1P_2}}, t_2^{\mathsf{P_3}})$,
for $x \in \mathsf{array}^{\ell}[n, m]$, $i_1 \in \mathsf{Z}_n$, $i_2 \in \mathsf{Z}_m$

Output: $\langle x[i_1][i_2] \rangle_{\mathsf{xor}}^{\mathsf{P_1-P_2}}$

1: P_1 and P_2 set $x' \in \mathsf{array}^{\ell}[nm]$ s.t. for all $(j_1, j_2) \in \mathsf{Z}_n \times \mathsf{Z}_m$,
$x'[j_1 \cdot m + j_2] = x[j_1 + s_1 \bmod n][j_2 \oplus s_2]$
P_3 computes $t = t_1 \cdot m + t_2$ (over integers)
2: SSPIR: $x'^{\mathsf{P_1P_2}}, t^{\mathsf{P_3}} \rightarrow \langle x'[t] \rangle_{\mathsf{xor}}^{\mathsf{P_1-P_2}}$ $(= \langle x[i_1][i_2] \rangle_{\mathsf{xor}}^{\mathsf{P_1-P_2}})$

Bandwidth: nm; Rounds: 1; Security: No message sent besides SSPIR.

Algorithm 11 Protocol 3ShiftXorPIR (Step 3 in Alg 2)

Input: $\langle x \rangle = (x_1^{\mathsf{DE}}, x_2^{\mathsf{CE}}, x_3^{\mathsf{CD}})$, $\langle i_1 \rangle_{\mathsf{shift}}$, $\langle i_2 \rangle_{\mathsf{2,1\text{-}xor}}$,
for $x \in \mathsf{array}^{\ell}[n, m]$, $i_1 \in \mathsf{Z}_n$, $i_2 \in \mathsf{Z}_m$

Output: $x[i_1][i_2]$

Generate $(\delta_c{}^{\mathsf{C}}, \delta_d{}^{\mathsf{D}}, \delta_e{}^{\mathsf{E}}) \xleftarrow{\$} \langle 0^{\ell} \rangle_{\mathsf{xor}}$
1: ShiftXorPIR: $x_1^{\mathsf{DE}}, \langle i_1 \rangle_{\mathsf{shift}}^{\mathsf{DE-C}}, \langle i_2 \rangle_{\mathsf{2,1\text{-}xor}}^{\mathsf{DE-C}} \rightarrow \langle x_1[i_1][i_2] \rangle_{\mathsf{xor}}^{\mathsf{D-E}}$
ShiftXorPIR: $x_2^{\mathsf{EC}}, \langle i_1 \rangle_{\mathsf{shift}}^{\mathsf{EC-D}}, \langle i_2 \rangle_{\mathsf{2,1\text{-}xor}}^{\mathsf{EC-D}} \rightarrow \langle x_2[i_1][i_2] \rangle_{\mathsf{xor}}^{\mathsf{E-C}}$
ShiftXorPIR: $x_3^{\mathsf{CD}}, \langle i_1 \rangle_{\mathsf{shift}}^{\mathsf{CD-E}}, \langle i_2 \rangle_{\mathsf{2,1\text{-}xor}}^{\mathsf{CD-E}} \rightarrow \langle x_3[i_1][i_2] \rangle_{\mathsf{xor}}^{\mathsf{C-D}}$
2: Denote $\langle x_1[i_1][i_2] \rangle_{\mathsf{xor}}^{\mathsf{D-E}} = (d_1^{\mathsf{D}}, e_1^{\mathsf{E}})$, $\langle x_2[i_1][i_2] \rangle_{\mathsf{xor}}^{\mathsf{E-C}} = (e_2^{\mathsf{E}}, c_1^{\mathsf{C}})$,
$\langle x_3[i_1][i_2] \rangle_{\mathsf{xor}}^{\mathsf{C-D}} = (c_2^{\mathsf{C}}, d_2^{\mathsf{D}})$
3: Each party P_t, for $t = c, d, e$ broadcasts v_t where
$v_c = c_1 \oplus c_2 \oplus \delta_c$, $v_d = d_1 \oplus d_2 \oplus \delta_d$, $v_e = e_1 \oplus e_2 \oplus \delta_e$
Output $x[i_1][i_2] := v_c \oplus v_d \oplus v_e$.

Bandwidth: $3nm + 6\ell$; Rounds: 2;
Security: By security of zero-sharing $(\delta_c, \delta_d, \delta_e)$, the broadcast values (v_c, v_d, v_e) are distributed as random xor-sharing of output $x[i_1][i_2]$. The rest is secure computation ShiftXorPIR which outputs random shares to each participant.

A.2 Protocols for Reduced-Round Retrieval

Algorithm 12 Protocol 3ShiftPIR-Mod (Step 2 in Alg 2)

Input: $\langle x\rangle = (x_1^{\mathsf{DE}}, x_2^{\mathsf{EC}}, x_3^{\mathsf{CD}})$, $\langle i\rangle_{\mathsf{shift}}^{\mathsf{DE-C}} = (s_1^{\mathsf{DE}}, t_1^{\mathsf{C}})$, for $x \in \mathsf{array}^{\ell}[n]$, $i \in \mathsf{Z}_n$

Output: $\langle x[i]\rangle$

Offline: Pick $a_{12}^{\mathsf{CD}}, a_{23}^{\mathsf{DE}}, a_{32}^{\mathsf{DE}} \xleftarrow{\$} \{0,1\}^n$

D picks $t_2 \xleftarrow{\$} \mathsf{Z}_n$ and sends $a_{21} := a_{23} \oplus \mathsf{ind}_{t_2}^n$ to C

E picks $t_3 \xleftarrow{\$} \mathsf{Z}_n$ and sends $a_{31} := a_{32} \oplus \mathsf{ind}_{t_3}^n$ to C

On input s_1^{DE}:

1: D sends $\delta_{12} = s_1 - t_2 \bmod n$ to C

E sends $\delta_{13} = s_1 - t_3 \bmod n$ to C

On input $\langle x\rangle, t_1^{\mathsf{C}}$:

1: C sets $s_2 := t_1 + \delta_{12} \bmod n$, $s_3 = t_1 + \delta_{13} \bmod n$, and $a_{13} := a_{12} \oplus \mathsf{ind}_{t_1}^n$; C sends s_3 to D and (s_2, a_{13}) to E

C sets $c_1 := \mathsf{XORSelect}((x_2)_{\mathsf{shift}[s_2]}, a_{21})$

C sets $c_2 := \mathsf{XORSelect}((x_3)_{\mathsf{shift}[s_3]}, a_{31})$

D sets $d_1 := \mathsf{XORSelect}((x_1)_{\mathsf{shift}[s_1]}, a_{12})$

D sets $d_2 := \mathsf{XORSelect}((x_3)_{\mathsf{shift}[s_3]}, a_{32})$

E sets $e_1 := \mathsf{XORSelect}((x_1)_{\mathsf{shift}[s_1]}, a_{13})$

E sets $e_2 := \mathsf{XORSelect}((x_2)_{\mathsf{shift}[s_2]}, a_{23})$

2: Reshare: $\langle x[i]\rangle_{\mathsf{xor}} = \big((c_1 \oplus c_2)^{\mathsf{C}}, (d_1 \oplus d_2)^{\mathsf{D}}, (e_1 \oplus e_2)^{\mathsf{E}}\big) \longrightarrow \langle x[i]\rangle$

Correctness: Observe that $d_1 \oplus e_1 = (x_1)_{\mathsf{shift}[s_1]}[t_1] = x_1[s_1 + t_1]$, because $a_{12} \oplus a_{13} = \mathsf{ind}_{t_1}^n$, and therefore $\mathsf{XORSelect}(z, a_{12}) \oplus \mathsf{XORSelect}(z, a_{13}) = z[t_1]$ for any z, e.g. $z = (x_1)_{\mathsf{shift}[s_1]}$. Likewise $c_1 \oplus e_2 = x_2[s_2 + t_2]$ and $c_2 \oplus d_2 = x_3[s_3 + t_3]$.
Note that $s_1 + t_1 = i$, but also
$s_2 + t_2 = (t_1 + \delta_{12}) + t_2 = (t_1 + (s_1 - t_2)) + t_2 = t_1 + s_1 = i$ and
$s_3 + t_3 = (t_1 + \delta_{13}) + t_3 = (t_1 + (s_1 - t_3)) + t_3 = t_1 + s_1 = i$.
It follows that $d_1 \oplus e_1 = x_1[i]$, $c_1 \oplus e_2 = x_2[i]$, and $c_2 \oplus d_2 = x_3[i]$.
Consequently, $(c_1 \oplus c_2) \oplus (d_1 \oplus d_2) \oplus (e_1 \oplus e_2) = x_1[i] \oplus x_2[i] \oplus x_3[i] = x[i]$.

Bandwidth: on-line: $\approx n + 3\ell$, off-line: $\approx 2n$ (assuming s_1^{DE} known off-line) ;
Rounds: 2;
Security: Party C receives only δ_{21} and δ_{31}, but these are random in Z_n because of one-time pads t_2 and t_3. These one-time pads were used also in computing a_{21} and a_{31} (resp. by D and E) but nevertheless a_{21}, a_{31} are uniform random strings in C's view because of onetime pads a_{23} and a_{32}. Party D receives $s_3 = t_1 + \delta_{13}$ from C, but $t_1 + \delta_{13} = t_1 + (s_1 - t_3) = i - t_3$ where t_3 is E's one-time pad, so s_3 is random in Z_n in D's view. Party E receives a_{13} and $s_2 = t_1 + \delta_{12}$ from C, but $t_1 + \delta_{12} = t_1 + (s_1 - t_2) = i - t_2$ where t_2 is D's one-time pad, so s_2 is random in Z_n in E's view. Value $a_{13} = a_{12} \oplus \mathsf{ind}_{t_1}^n$ is also random in E's view because of one-time pad a_{12}. Finally, by correctness of Reshare, the final sharing is a *random* sharing of $x[i]$.

Algorithm 13 Protocol 3ShiftXorPIR-Mod (Step 3 in Alg 2)

Input: $\langle x\rangle = (x_1^{\mathsf{DE}}, x_2^{\mathsf{EC}}, x_3^{\mathsf{CD}})$, $\langle i\rangle_{\mathsf{shift}}^{\mathsf{DE-C}} = (s_1^{\mathsf{DE}}, t_1^{\mathsf{C}})$,
$\langle j\rangle_{2,1\text{-xor}} = ((u_1^{\mathsf{DE}}, v_1^{\mathsf{C}}), (u_2^{\mathsf{CE}}, v_2^{\mathsf{D}}), (u_3^{\mathsf{CD}}, v_3^{\mathsf{E}}))$,
for $x \in \mathsf{array}^{\ell}[n][m]$, $i \in \mathsf{Z}_n$, $j \in \mathsf{Z}_m$

Output: $x[i][j]$

Offline: Pick $a_{12}^{\mathsf{CD}}, a_{23}^{\mathsf{DE}}, a_{32}^{\mathsf{DE}} \xleftarrow{\$} \{0,1\}^{n\times m}$
D picks $t_2 \xleftarrow{\$} \mathsf{Z}_n$ and sends $a_{21} := a_{23} \oplus \mathsf{ind}_{t_2\times m+v_2}^{n\times m}$ to C
E picks $t_3 \xleftarrow{\$} \mathsf{Z}_n$ and sends $a_{31} := a_{32} \oplus \mathsf{ind}_{t_3\times m+v_3}^{n\times m}$ to C

On input $(s_1, u_1)^{\mathsf{DE}}$**:**

1: D sends $\delta_{12} = s_1 - t_2 \bmod n, \rho_{12} = u_1 \oplus v_2$ to C
E sends $\delta_{13} = s_1 - t_3 \bmod n, \rho_{13} = u_1 \oplus v_3$ to C

On input $\langle x\rangle, (t_1, v_1)^{\mathsf{C}}$**:**

1: C sets $s_2 := t_1 + \delta_{12} \bmod n$, $s_3 = t_1 + \delta_{13} \bmod n$, $u_2 = v_1 \oplus \rho_{12}$, $u_3 = v_1 \oplus \rho_{13}$, and $a_{13} := a_{12} \oplus \mathsf{ind}_{t_1\times m+v_1}^{n\times m}$; C sends s_3 to D and (s_2, a_{13}) to E

For every $\mathsf{P} \in \{\mathsf{C}, \mathsf{D}, \mathsf{E}\}, k_{\mathsf{C}} \in \{2,3\}, k_{\mathsf{D}} \in \{1,3\}, k_{\mathsf{E}} \in \{1,2\}, f \in \mathsf{Z}_n, g \in \mathsf{Z}_m$,
P sets $x'_{k_{\mathsf{P}}} \in \mathsf{array}^{\ell}[n][m]$ s.t. $x'_{k_{\mathsf{P}}}[f][g] = x_{k_{\mathsf{P}}}[f + s_{k_{\mathsf{P}}} \bmod n][g \oplus u_{k_{\mathsf{P}}}]$.

C sets $c_1 := \mathsf{XORSelect}(x'_2, a_{21})$, $c_2 := \mathsf{XORSelect}(x'_3, a_{31})$
D sets $d_1 := \mathsf{XORSelect}(x'_1, a_{12})$, $d_2 := \mathsf{XORSelect}(x'_3, a_{32})$
E sets $e_1 := \mathsf{XORSelect}(x'_1, a_{13})$, $e_2 := \mathsf{XORSelect}(x'_2, a_{23})$

2: $\mathsf{C}, \mathsf{D}, \mathsf{E}$ broadcast shares they have among $(c_1, c_2, d_1, d_2, e_1, e_2)$, and compute $x[i][j] = (d_1 \oplus e_1) \oplus (c_1 \oplus e_2) \oplus (c_2 \oplus d_2)$.

Correctness: Observe that
$d_1 \oplus e_1 = \mathsf{XORSelect}(x'_1, a_{12}) \oplus \mathsf{XORSelect}(x'_1, a_{13}) = \mathsf{XORSelect}(x'_1, a_{12} \oplus a_{13}) = \mathsf{XORSelect}(x'_1, \mathsf{ind}_{t_1\times m+v_1}^{n\times m}) = x'_1[t_1][v_1] = x_1[t_1 + s_1 \bmod n][v_1 \oplus u_1] = x_1[i][j]$.
It follows that $c_1 \oplus e_2 = x_2[i][j]$ and $c_2 \oplus d_2 = x_3[i][j]$.
Consequently, $(c_1 \oplus c_2) \oplus (d_1 \oplus d_2) \oplus (e_1 \oplus e_2) = x_1[i][j] \oplus x_2[i][j] \oplus x_3[i][j] = x[i][j]$.

Bandwidth: on-line: $\approx nm + 6\ell$, off-line: $\approx 2nm$ (assuming $(s_1, u_1)^{\mathsf{DE}}$ known off-line) ;
Rounds: 2;
Security: Party C receives $(\delta_{12}, \delta_{13}, \rho_{12}, \rho_{13})$, but these are random because of one-time pads t_2 and t_3 and freshness of v_2 and v_3. The rest follows the same security of 3ShiftPIR-Mod.

A.3 Protocols for PostProcess

Algorithm 14 Protocol ULiT - Update Labels in Tuple (Step 4, Alg 2)

Input: $\langle X, \mathrm{N}, \Delta\mathrm{N}, \mathrm{L}', \mathrm{L}'_{i+1}\rangle, \mathrm{L}_{i+1} = X[\Delta\mathrm{N}]$
where $X \in \mathsf{array}^{\ell}[2^{\tau}]$, $|\Delta\mathrm{N}| = \tau$, $|\mathrm{L}_{i+1}| = |\mathrm{L}'_{i+1}| = \ell$

Output: $\langle \mathsf{T} \rangle$ for $\mathsf{T} = (1|\mathrm{N}|\mathrm{L}'|X)$ with $X[\Delta\mathrm{N}] := \mathrm{L}'_{i+1}$

Offline:

1: $x_1^{\mathsf{CD}}, x_2^{\mathsf{DE}} \xleftarrow{\$} \{0,1\}^{|X|}$
2: Run the offline phase of two InsertLbl instances of step 2, where first instance outputs a_1^{D} and second instance outputs a_2^{D}.
3: D sends $m_e = a_1 \oplus x_1 \oplus x_2$ to E and $m_c = a_2 \oplus x_1 \oplus x_2$ to C.

Online:

1: $\langle \mathsf{xor}\text{-}\mathrm{L}_{i+1} \rangle := \langle \mathrm{L}'_{i+1} \oplus \mathrm{L}_{i+1} \rangle$
Extract: $\langle \Delta\mathrm{N}, \mathsf{xor}\text{-}\mathrm{L}_{i+1} \rangle \rightarrow \langle \Delta\mathrm{N}, \mathsf{xor}\text{-}\mathrm{L}_{i+1} \rangle_{\mathsf{xor}}^{\mathsf{C-D}}$
Extract: $\langle \Delta\mathrm{N}, \mathsf{xor}\text{-}\mathrm{L}_{i+1} \rangle \rightarrow \langle \Delta\mathrm{N}, \mathsf{xor}\text{-}\mathrm{L}_{i+1} \rangle_{\mathsf{xor}}^{\mathsf{E-D}}$
2: InsertLbl: $\langle \Delta\mathrm{N} \rangle_{\mathsf{xor}}^{\mathsf{C-D}}, \langle \mathsf{xor}\text{-}\mathrm{L}_{i+1} \rangle_{\mathsf{xor}}^{\mathsf{C-D}} \rightarrow \langle M \rangle_{\mathsf{xor}}^{\mathsf{D-E}} = (a_1^{\mathsf{D}}, b_1^{\mathsf{E}})$
InsertLbl: $\langle \Delta\mathrm{N} \rangle_{\mathsf{xor}}^{\mathsf{E-D}}, \langle \mathsf{xor}\text{-}\mathrm{L}_{i+1} \rangle_{\mathsf{xor}}^{\mathsf{E-D}} \rightarrow \langle M \rangle_{\mathsf{xor}}^{\mathsf{D-C}} = (a_2^{\mathsf{D}}, b_2^{\mathsf{C}})$
for M which is an all-zero array except $M[\Delta\mathrm{N}] = \mathsf{xor}\text{-}\mathrm{L}_{i+1}$
3: $\langle M \rangle := (x_1^{\mathsf{CD}}, x_2^{\mathsf{DE}}, x_3^{\mathsf{CE}})$ for $x_3^{\mathsf{C}} := m_c \oplus b_2$, $x_3^{\mathsf{E}} := m_e \oplus b_1^{\mathsf{E}}$
Output $\langle \mathsf{T} \rangle := \langle 1|\mathrm{N}|\mathrm{L}'|(X \oplus M) \rangle$

Bandwidth: Online: $\approx 4|X|$, Offline: $\approx 4|X|$;
Rounds: 2 (the first round requires only input $\langle \Delta\mathrm{N} \rangle$, see Alg 15);
Security: Note that by security of InsertLbl, everything the parties receive in the InsertLbl instances can be simulated from their inputs and outputs in these instances.
Security for D: Party D's view includes only its InsertLbl outputs, a_1^{D} and a_2^{D}, which are random strings by security of InsertLbl.
Security for C: Party C receives $m_c = a_2 \oplus x_1 \oplus x_2$ and b_2, but b_2 is random by security of InsertLbl and m_c is random by randomness of x_2.
Security for E: Likewise E receives $m_e = a_1 \oplus x_1 \oplus x_2$ and b_1, but b_1 is random by security of InsertLbl and m_e is random by randomness of x_1..

Algorithm 15 Protocol $\mathsf{InsertLbl}$ - Inserting Label (Used in Alg 14)

Input: $\langle \Delta\mathrm{N}\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$, $\langle \mathrm{L}\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2} = (\mathrm{L}_1^{\mathsf{P}_1}, \mathrm{L}_2^{\mathsf{P}_2})$, for $|\Delta\mathrm{N}| = \tau$, $|\mathrm{L}| = \ell$

Output: $\langle M\rangle_{\mathsf{xor}}^{\mathsf{P}_2-\mathsf{P}_3} = (z_2^{\mathsf{P}_2}, z_3^{\mathsf{P}_3})$,
for $M \in \mathsf{array}^{\ell}[2^{\tau}]$ s.t. $M[\Delta\mathrm{N}] = \mathrm{L}$ and $M[t] = 0^{\ell}$ for $t \neq \Delta\mathrm{N}$

Offline:

1: $(p, a, b)^{\mathsf{P}_1\mathsf{P}_2} \xleftarrow{\$} \mathsf{array}^{\ell}[2^{\tau}]$, $(v, w)^{\mathsf{P}_1\mathsf{P}_2} \xleftarrow{\$} \{0,1\}^{\tau}$
2: $\mathsf{P}_1 : \alpha_1 \xleftarrow{\$} \{0,1\}^{\tau}$, set $u_1 := \alpha_1 \oplus v$, $p^* := p \oplus a_{\mathsf{rot}[u_1]}$
3: $\mathsf{P}_2 : \beta_2 \xleftarrow{\$} \{0,1\}^{\tau}$, set $u_2 := \beta_2 \oplus w$, $z_2 := p \oplus b_{\mathsf{rot}[u_2]}$
4: P_1 sends (u_1, p^*) to P_3; P_2 sends u_2 to P_3; P_2 **outputs** z_2

Online:

1: $\mathsf{Reshare}$: $\langle \Delta\mathrm{N}\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}, \alpha_1^{\mathsf{P}_1} \rightarrow \alpha_2^{\mathsf{P}_2}$ s.t. $(\alpha_1, \alpha_2) = \langle \Delta\mathrm{N}\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$
$\mathsf{Reshare}$: $\langle \Delta\mathrm{N}\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}, \beta_2^{\mathsf{P}_2} \rightarrow \beta_1^{\mathsf{P}_1}$ s.t. $(\beta_1, \beta_2) = \langle \Delta\mathrm{N}\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$
2: P_1 sends $s_1 = b^{\mathsf{xor}[\,\mathrm{L}_1 \,@\, \beta_1 \oplus w]}$ to P_3
i.e. $s_1 = b$ except $s_1[\beta_1 \oplus w] = b[\beta_1 \oplus w] \oplus \mathrm{L}_1$
P_2 sends $s_2 = a^{\mathsf{xor}[\,\mathrm{L}_2 \,@\, \alpha_2 \oplus v]}$ to P_3
i.e. $s_2 = a$ except $s_2[\alpha_2 \oplus v] = a[\alpha_2 \oplus v] \oplus \mathrm{L}_2$
3: P_3 outputs $z_3 := p^* \oplus (s_2)_{\mathsf{rot}[u_1]} \oplus (s_1)_{\mathsf{rot}[u_2]}$

Correctness: Observe that $z_2 = p \oplus b_{\mathsf{rot}[\beta_2 \oplus w]}$ and $p^* = p \oplus a_{\mathsf{rot}[\alpha_1 \oplus v]}$.
Note that $(s_2)_{\mathsf{rot}[u_1]} = (a^{\mathsf{xor}[\,\mathrm{L}_2 \,@\, \alpha_2 \oplus v]})_{\mathsf{rot}[\alpha_1 \oplus v]}$
$= (a_{\mathsf{rot}[\alpha_1 \oplus v]})^{\mathsf{xor}[\,\mathrm{L}_2 \,@\, (\alpha_2 \oplus v) \,\oplus\, (\alpha_1 \oplus v)\,]}$
$= (a_{\mathsf{rot}[\alpha_1 \oplus v]})^{\mathsf{xor}[\,\mathrm{L}_2 \,@\, \Delta\mathrm{N}\,]}$.
Likewise $(s_1)_{\mathsf{rot}[u_2]} = (b_{\mathsf{rot}[\beta_2 \oplus w]})^{\mathsf{xor}[\,\mathrm{L}_1 \,@\, \Delta\mathrm{N}\,]}$.
It follows that
$z_3 = p^* \;\oplus\; (s_2)_{\mathsf{rot}[u_1]} \;\oplus\; (s_1)_{\mathsf{rot}[u_2]}$
$= p \;\oplus\; a_{\mathsf{rot}[\alpha_1 \oplus v]} \;\oplus\; (a_{\mathsf{rot}[\alpha_1 \oplus v]})^{\mathsf{xor}[\,\mathrm{L}_2 \,@\, \Delta\mathrm{N}\,]} \;\oplus\; (b_{\mathsf{rot}[\beta_2 \oplus w]})^{\mathsf{xor}[\,\mathrm{L}_1 \,@\, \Delta\mathrm{N}\,]}$
By xor-ing z_2 and z_3 observer that pad p and rotated pads a and b cancel out and we get $M = z_2 \oplus z_3 = [0, ..., 0]^{\mathsf{xor}[\,\mathrm{L} \,@\, \Delta\mathrm{N}]}$ where $[0, ..., 0]$ is an all-zero array.

Bandwidth: Online: $2 \cdot (2^{\tau}\ell + \tau) \approx 2 \cdot 2^{\tau}\ell$, Offline: $\approx 2^{\tau}\ell$;
Rounds: 2 (the first round requires only input $\langle \Delta\mathrm{N}\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$);
Security:
(1) For $\mathsf{P}_1, \mathsf{P}_2$: Let $(\Delta\mathrm{N}_1^{\mathsf{P}_1}, \Delta\mathrm{N}_2^{\mathsf{P}_2})$ denote input $\langle \Delta\mathrm{N}\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$. P_2 receives $\Delta\mathrm{N}_1 \oplus \alpha_1$ and P_1 receives $\Delta\mathrm{N}_2 \oplus \beta_2$ in in $\mathsf{Reshare}$ in step 1. Bot values are random because α_1, β_2 are randomly chosen resp. by P_1 and P_2, and neither value affects the distribution of protocol outputs (z_2, z_3), because z_2 is uniform by randomness of a, and z_3 is a deterministic function of z_2 and protocol inputs.

(2) For P_3: Values p^*, u_1, u_2, s_1, s_2 sent to P_3 are independently random because of resp. random pads p, v, w, a, b. Sharing (z_1, z_2) is fresh by randomness of p.

Algorithm 16 Protocol $\mathsf{FlipFlag}$ (Step 5, Alg 2)

Input: $\langle \mathsf{fb} \rangle, \langle i \rangle_{\mathsf{shift}}^{\mathsf{DE-C}} = (i_1^{\mathsf{C}}, i_2^{\mathsf{DE}})$, for $\mathsf{fb} \in \mathsf{array}^1[n], i \in \mathsf{Z}_n$

Output: $\langle \mathsf{fb}' \rangle$ s.t fb' is the same as fb except $\mathsf{fb}'[i] = \mathsf{fb}[i] \oplus 1$

1: C creates $a_1 \in \mathsf{array}^1[n]$ s.t. $a_1[i_1] = 1$ and $a_1[j] = 0$ for $j \neq i_1$
2: E creates $a_2 \in \mathsf{array}^1[n]$ s.t. $a_2[j] = 0$ for all j
3: Shift: $\langle a \rangle_{\mathsf{xor}}^{\mathsf{C-E}} = (a_1^{\mathsf{C}}, a_2^{\mathsf{E}}), (s = n - i_2)^{\mathsf{DE}} \rightarrow \langle m \rangle_{\mathsf{xor}}^{\mathsf{C-E}}$
note: $m[i] = a[(i_1 + i_2) + s] = 1$, and $m[j] = 0$ for $j \neq i$
4: $\mathsf{Reshare}$: $\langle m \rangle_{\mathsf{xor}}^{\mathsf{C-E}} \rightarrow \langle m \rangle$
5: $\langle \mathsf{fb}' \rangle := \langle \mathsf{fb} \oplus m \rangle$

Bandwidth: $4n$; Rounds: 2;
Security: Protocol $\mathsf{FlipFlag}$ is secure if protocol Shift is a secure computation of $\langle m \rangle_{\mathsf{xor}}^{\mathsf{C-E}}$ and $\mathsf{Reshare}$ produces fresh secret-sharing $\langle m \rangle$ from $\langle m \rangle_{\mathsf{xor}}^{\mathsf{C-E}}$.

Algorithm 17 Protocol Shift (based on [14]) (Used in Alg 16)

Input: $\langle x \rangle_{\mathsf{xor}}^{\mathsf{C-E}} = (x_1^{\mathsf{C}}, x_2^{\mathsf{E}})$, $s^{\mathsf{DE}} \in \mathsf{Z}_n$, where $x \in \mathsf{array}^{\ell}[n]$

Output: $\langle y \rangle_{\mathsf{xor}}^{\mathsf{C-E}} = (y_1^{\mathsf{C}}, y_2^{\mathsf{E}})$ s.t. $y[t] = x[(t + s) \bmod n]$ for all t

Offline: $p^{\mathsf{CD}}, r^{\mathsf{DE}}, q^{\mathsf{CE}} \xleftarrow{\$} \mathsf{array}^{\ell}[n]$

1: D sends array a to C s.t. $a[t] = (p \oplus r)[(t + s) \bmod n]$
2: C sends $z = x_1 \oplus p$ to E, and outputs $y_1 = a \oplus q$
3: E outputs $y_2 = b \oplus q$ for b s.t. $b[t] = (x_2 \oplus z \oplus r)[(t + s) \bmod n]$

$$y[t] = (y_1 \oplus y_2)[t] = (a \oplus b)[t] = ((p \oplus r) \oplus (x_2 \oplus z \oplus r))[t + s]$$
$$= (p \oplus x_2 \oplus z)[t + s] = (p \oplus x_2 \oplus (x_1 \oplus p))[t + s] = x[t + s]$$

Bandwidth: $2n\ell$; Rounds: 1;
Security: Sharing $\langle y \rangle_{\mathsf{xor}}^{\mathsf{C-E}}$ is fresh by randomness of q, message a received by C is random by randomness of r, message z received by E is random by randomness of p.

A.4 Protocols for Eviction

Algorithm 18 Protocol $\mathsf{GC}(\mathsf{circ})$ (Step 6, Alg 2)

Input: $\langle x \rangle_{\mathsf{xor}}^{\mathsf{C-E}}$, for x the input of circuit circ

Output: $((\overline{y}, z)^{\mathsf{D}}, \mathsf{owk}^{\mathsf{E}})$, for $\overline{y} = \{\mathsf{owk} : y\}$, $(y, z) = \mathsf{circ}(x)$, $\mathsf{owk} \xleftarrow{\$} \mathsf{array}^{\kappa}[|y|, 2]$

Offline: E sends to D a garbled version of circ', where $\mathsf{circ}'(x_1, x_2) = \mathsf{circ}(x_1 \oplus x_2)$, which includes the wire-key-to-bit translation table *only* for the output wires corresponding to variable z;
E also sends to C the set of wire keys iwk_1 corresponding to input variable x_1, and retains the set of wire keys iwk_2 corresponding to input variable x_2 and set owk corresponding to output variable y

1: C and E on input $\langle x \rangle_{\mathsf{xor}}^{\mathsf{C-E}} = (x_1^{\mathsf{C}}, x_2^{\mathsf{E}})$, select input wire keys according to their respective input $x_1^{\mathsf{C}}, x_2^{\mathsf{E}}$ and send to D resp. $\{\mathsf{iwk}_1 : x_1\}$ and $\{\mathsf{iwk}_2 : x_2\}$

2: D evaluates the garbled circuit circ' starting given the received sets of wire keys; Because the garbled circuit contains the wire-key-to-bit translation table only for the wires corresponding to variable z, D outputs the z part of the output in the clear, but for variable y it can only output the wire key set $\{\mathsf{owk} : y\}$ corresponding to value y.

Bandwidth: Online: $2|x|\kappa$, for sec. par. κ; Offline: $(4|\mathsf{circ}| + 2|x|)\kappa$; Rounds: 1;
Security: This is a trivial modification of Yao's garbled circuit computation procedure.

Algorithm 19 Protocol $\mathsf{PermTuples}$ (Step 8, Alg 2)

Param: Number of buckets d, bucket size w

Offline Input: $(\pi, \rho)^{\mathsf{CE}}$ for $\pi \in \mathsf{perm}_d$, $\rho \in \mathsf{array}^{w+1}[d]$

Input: $\mathsf{t}^{\mathsf{D}} \in \mathsf{array}^{w+1}[d]$

Output: $\mathsf{t}^{\circ \mathsf{D}} = \rho \oplus \pi(\mathsf{t})$

Offline: $p^{\mathsf{ED}}, r^{\mathsf{EC}} \xleftarrow{\$} \mathsf{array}^{w+1}[d]$; E sends $a = \pi(p \oplus r)$ to D

1: D sends $z = \mathsf{t} \oplus p$ to C.

2: C sends $g = \rho \oplus \pi(z \oplus r)$ to D.

3: D outputs $\mathsf{t}^{\circ} = a \oplus g$.

Correctness: $\mathsf{t}^{\circ} = a \oplus g = \pi(p \oplus r) \oplus \rho \oplus \pi(z \oplus r) = \pi(p \oplus r) \oplus \rho \oplus \pi(t \oplus p \oplus r)$
$= \pi(p \oplus r) \oplus \rho \oplus \pi(t) \oplus \pi(p \oplus r) = \rho \oplus \pi(t)$

Bandwidth: Online: $2|\mathsf{t}| = 2d(w{+}1)$; Offline: $|\mathsf{t}| = d(w{+}1)$; Rounds: 2;
Security: Array z received by C is random because p is a one-time pad known only to D and E. Array a received by D off-line is random because r is a one-time pad known only to C and E, and array g received by D on-line gives no additional information because it can be computed as $g = a \oplus \mathsf{t}^{\circ}$ from a and D's output t°.

Algorithm 20 Protocol $\mathsf{PermBuckets}$ (Step 7, Alg 2)

Param: Path depth d, security parameter κ;
hash function $\mathsf{H}^{\mathsf{DE}} : \{0,1\}^{\log(d)\cdot\kappa} \to \{0,1\}^{\kappa}$.

Input: $\overline{\sigma}^{\mathsf{D}}, \pi^{\mathsf{CE}}, \mathsf{wk}^{\mathsf{E}}$, s.t. $\pi \in \mathsf{perm}_d$, $\mathsf{wk} \in \mathsf{array}^{\kappa}[d, \log(d), 2]$,
and $\overline{\sigma} = \{\mathsf{wk} : \sigma\}$ for some $\sigma \in \mathsf{perm}_d$

Output: $\sigma^{\circ\mathsf{D}}$ s.t. $\sigma^{\circ} = \pi \cdot \sigma \cdot \pi^{-1}$

Offline: (assume pre-generated π^{CE} and wk^{E})

1: E sets $\mathsf{keys} \in \mathsf{array}^{\kappa}[d, d, \log(d)]$ s.t. for each $i, j \in \mathsf{Z}_d$, $k \in \mathsf{Z}_{\log(d)}$, $\mathsf{keys}[i][j][k] = \mathsf{wk}[i][k][j_{\mathsf{bit}[k]}]$, which means $\mathsf{keys}[i][j] = \overline{\sigma}[i]$ for $\sigma[i] = j$.

2: E picks $\mathsf{MK} \xleftarrow{\$} \mathsf{array}^{\log(d)}[d, d]$, and sets $\mathsf{TB} \in \mathsf{array}^{\kappa+\log(d)}[d, d]$ s.t. each $\mathsf{TB}[i] \in \mathsf{array}^{\kappa+\log(d)}[d]$ is a sequence of d (key,value) pairs, each of which binds key $\mathsf{H}(\mathsf{keys}[i][j])$ to value $\pi(j) \oplus \mathsf{MK}[i][j]$, i.e. $\mathsf{TB}[i][j] = (\mathsf{H}(\mathsf{keys}[i][j]), \pi(j) \oplus \mathsf{MK}[i][j])$. In another words, $\mathsf{TB}[i](\cdot)$ is a look-up function s.t. $\mathsf{TB}[i](\mathsf{H}(\mathsf{keys}[i][j])) = \pi(j) \oplus \mathsf{MK}[i][j]$ for $j \in d$.

3: For every $i \in \mathsf{Z}_d$, E picks a random permutation in perm_d and uses it to permute the entries of both $\mathsf{TB}[i]$ *and* $\mathsf{MK}[i]$.

4: E picks $p, r \xleftarrow{\$} \mathsf{array}^{\log(d)}[d]$, and computes $a = \pi(p \oplus r)$.

5: E sends TB, p, a to D and MK, r to C.

Online:

1: D initializes $\mathsf{I} \in \mathsf{array}^{\log(d)}[d]$. For every $i \in \mathsf{Z}_d$, D sets $[\sigma'][i] = \mathsf{TB}[i](\mathsf{H}(\overline{\sigma}[i]))$, and sets $\mathsf{I}[i] = j$ s.t. $\mathsf{H}(\overline{\sigma}[i])$ is the key of $\mathsf{TB}[i][j]$ (key,value) pair. D then sends $z = \sigma' \oplus p$ and I to C.

2: C sets $m \in \mathsf{array}^{\log(d)}[d]$ s.t. $m[i] = \mathsf{MK}[i][\mathsf{I}[i]]$ for every $i \in \mathsf{Z}_d$. C sends $g = \pi(z \oplus r \oplus m)$ to D

3: D outputs σ° s.t. $\sigma^{\circ} = a \oplus g$.

Correctness:

$$\begin{aligned}\sigma^{\circ} &= a \oplus g = \pi(p \oplus r) \oplus \pi(\sigma' \oplus p \oplus r \oplus m) = \pi(\sigma' \oplus m) \\ &= \pi([(\mathsf{TB}[0][\mathsf{I}_0] \oplus \mathsf{MK}[0][\mathsf{I}_0]), ..., (\mathsf{TB}[d-1][\mathsf{I}[d-1]] \oplus \mathsf{MK}[d-1][\mathsf{I}[d-1]])]) \\ &= \pi([\pi(\sigma_0), ..., \pi(\sigma_{d-1})]) = \pi(\pi \cdot \sigma) = \pi \cdot \sigma \cdot \pi^{-1}\end{aligned}$$

Bandwidth: Online: $3d\log(d)$; Offline: $d^2(\kappa + 2\log(d)) + 3d\log(d)$; Rounds: 2;
Security: C's view z and I are indistinguishable from random strings because p is random and unknown to C, and for every $i \in \mathsf{Z}_d$, $\mathsf{TB}[i]$ and $\mathsf{MK}[i]$ are permuted by a random permutation in Z_d. Given D's input a (from pre-computation) and output σ°, D's view g can be simulated as $a \oplus \sigma^{\circ}$. E receives nothing online.

Algorithm 21 Protocol XOT (Step 9, Alg 2)

Input: $\langle \mathsf{path} \rangle, (\pi, \delta, \rho)^{\mathsf{CE}}, (\sigma^\circ, \mathsf{t}^\circ)^{\mathsf{D}}$, for $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}, \mathsf{t}^\circ = \rho \oplus \pi(\delta \oplus \mathsf{t})$

Output: $\langle \mathsf{path}' \rangle$, for $\mathsf{path}' = \mathsf{EM}_{\sigma,\mathsf{t}}(\mathsf{path})$

Offline: D picks $p \xleftarrow{\$} \{0,1\}^{|\mathsf{path}|}$ and executes the first step of the two instances of $\mathsf{HalfXOT}$ below (see step 1, Alg. 22)

$\mathsf{Extract}$: $\langle \mathsf{path} \rangle \to \langle \mathsf{path} \rangle^{\mathsf{C-E}}_{\mathsf{xor}}$. Parties sets $\mathsf{EM}^{\circ\,\mathsf{D}} := \mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ}$, $\mathit{\Pi}^{\mathsf{CE}} := \tilde{\rho} \cdot \ddot{\pi} \cdot \tilde{\delta}$, and $(x_1^{\mathsf{C}}, x_2^{\mathsf{E}}) = \langle \mathsf{path}^\circ \rangle^{\mathsf{C-E}}_{\mathsf{xor}} := \langle \mathit{\Pi}(\mathsf{path}) \rangle^{\mathsf{C-E}}_{\mathsf{xor}}$. (see eq. (1) in Sec. 3)

1: The following two steps are performed in parallel:
$\mathsf{HalfXOT}$: $x_1^{\mathsf{C}}, (\mathsf{EM}^\circ, p)^{\mathsf{D}} \to y_1^{\mathsf{E}}$ ▷ $y_2 = p \oplus \mathsf{EM}^\circ(x_1)$
$\mathsf{HalfXOT}$: $x_2^{\mathsf{E}}, (\mathsf{EM}^\circ, p)^{\mathsf{D}} \to y_2^{\mathsf{C}}$ ▷ $y_1 = p \oplus \mathsf{EM}^\circ(x_2)$
Parties set $\langle y \rangle^{\mathsf{C-E}}_{\mathsf{xor}} := (y_1^{\mathsf{C}}, y_2^{\mathsf{E}})$, ▷ $y = \mathsf{EM}^\circ(x) = \mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ}(\mathit{\Pi}(\mathsf{path}))$
and set $\langle \mathsf{path}' \rangle^{\mathsf{C-E}}_{\mathsf{xor}} := \langle \mathit{\Pi}^{-1}(y) \rangle^{\mathsf{C-E}}_{\mathsf{xor}}$ ▷ $\mathsf{path}' = \mathit{\Pi}^{-1}(y) = \mathsf{EM}_{\sigma,\mathsf{t}}(\mathsf{path})$
2: $\mathsf{Reshare}$: $\langle \mathsf{path}' \rangle^{\mathsf{C-E}}_{\mathsf{xor}} \to \langle \mathsf{path}' \rangle$

Bandwidth: Online: $4|\mathsf{path}| + 2m\log(m)$, $m = \#$ tuples; Offline: $2|\mathsf{path}|$; Rounds: 3;
Security: Because p is a one-time pad known only to D, both y_1 and y_2 are individually random, and by security of $\mathsf{HalfXOT}$, the protocol leaks nothing beyond (locally random) values y_1 to E and y_2 to C. Moreover, by correctness of $\mathsf{HalfXOT}$ we have that (y_1, y_2) is an xor-sharing of $y = \mathsf{EM}^\circ(x)$. Then by eq. (1) in Sec. 3, we have $\mathsf{path}' = \mathit{\Pi}^{-1}(y) = \mathit{\Pi}^{-1} \cdot \mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ} \cdot \mathit{\Pi}(\mathsf{path}) = \mathsf{EM}_{\sigma,\mathsf{t}}(\mathsf{path})$.

Algorithm 22 Protocol $\mathsf{HalfXOT}$ (Steps 1 and 2 in Alg 21)

Param: n, k, ℓ s.t. $k \le n$.

Input: $x^{\mathsf{P}_1}, (\sigma, p)^{\mathsf{P}_2}$ s.t. $x \in \mathsf{array}^\ell[n], p \in \mathsf{array}^\ell[k]$, and $\sigma^{-1} : \mathsf{Z}_k \overset{1-1}{\to} \mathsf{Z}_n$

Output: y^{P_3} s.t. $y = p \oplus \sigma(x)$, i.e. $y[i] = p[i] \oplus x[\sigma^{-1}(i)]$ for $i \in \mathsf{Z}_k$

Offline: $r^{\mathsf{P}_1\mathsf{P}_2} \xleftarrow{\$} \mathsf{array}^\ell[n]$; $\delta^{\mathsf{P}_2\mathsf{P}_3} \xleftarrow{\$} \mathsf{perm}_n$

1: On P_2's input p: P_2 sends $s = p \oplus \delta(r)$ to P_3
2: On P_2's input σ: P_2 sends $\pi = \delta^{-1} \cdot \sigma$ to P_1
3: On P_1's input x (and message π): P_1 sends $a = r \oplus \pi(x)$ to P_3
P_3 outputs $y = s \oplus \delta(a)$

Note that $y = s \oplus \delta(a) = p \oplus \delta(r) \oplus \delta(a) = p \oplus \delta(r \oplus a) = p \oplus \delta(\pi(x)) = p \oplus (\delta \cdot \pi)(x) = p \oplus (\delta \cdot (\delta^{-1} \cdot \sigma))(x) = p \oplus \sigma(x)$
Bandwidth: $n\ell + k\ell + k\log(n)$; Rounds: 2;
Security: P_3's view includes s, a where a is random $n\ell$-bit string because of one-time pad r and s reveals no additional information beyond P_3's output because it can be computed as $s = y \oplus \delta(a)$; P_1's view includes $\pi = \delta^{-1} \cdot \sigma$ but π is a random 1-1 function from Z_k to Z_n because δ is a random permutation in Z_n .

B Algorithms for Client-Server Path-ORAM [26]

For completeness we recall the access algorithm of Client-Server Path-ORAM of Shi et al. [26], which we briefly recall in Section 2. As we explain in Section 2, we call the main Path-ORAM algorithm ORAM.Access and its main loop ORAM.ML, and here we show both algorithms as resp. Alg 23 and Alg 24.

The main point of including these algorithms here is to observe that our 3PC ORAM protocol described in Section 3 is a 3PC emulation of the (client-server) Path-ORAM algorithm, shown here, except that the eviction map computation in step 6 of algorithm ORAM.ML, algorithm PathORAM-Route, is replaced by the eviction computation algorithm Route of Circuit-ORAM [27] (with some necessary modifications described in Section 3). In particular, observe that our top-level protocol, 3PC-ORAM.Access shown as Alg 1 in Section 3, and its main loop 3PC-ORAM.ML, shown as Alg. 2 in Section 3, are 3PC emulations of resp. algorithms ORAM.Access, Alg 23, and ORAM.ML, Alg 24.

Algorithm 23 ORAM.Access: Client/Server Path-ORAM

Param: Address size $\log n$, address chunk size τ, number of trees $h = \frac{\log n}{\tau} + 1$

Input: $\mathsf{OM}^{\mathsf{S}} = (\mathsf{tree}_0, ..., \mathsf{tree}_{h-1})$, $\mathrm{N}^{\mathsf{C}} = (\mathrm{N}_1, ..., \mathrm{N}_{h-1})$, (* $\mathsf{rec}'^{\mathsf{C}}$)

Output: $\mathsf{rec}^{\mathsf{C}}$: record stored in OM at address N

1: $\{\mathrm{L}_i'^{\mathsf{C}} \xleftarrow{\$} \{0,1\}^{i\cdot\tau}\}_{i=1}^{h-1}$; $(\mathrm{N}_0, \mathrm{N}_h, \mathrm{L}_0', \mathrm{L}_h')^{\mathsf{C}} := \perp$; $\mathrm{L}_0 := \perp$

2: **for** $i = 0$ **to** $h-1$ **do** (for $i = 0$ see footnote "!" in Alg. 24)

ORAM.ML: L_i, $\mathsf{tree}_i^{\mathsf{S}}$, ($\mathrm{N}_0|...|\mathrm{N}_i$, N_{i+1}, L_i', L_{i+1}', *rec')$^{\mathsf{C}}$

$\longrightarrow$ L_{i+1} (* $\mathsf{rec}^{\mathsf{C}}$ instead of L_{i+1}), $\mathsf{tree}_i^{\mathsf{S}}$

---

*: On top-level ORAM tree

C Security Proof for the 3PC ORAM Protocol

Client-Server ORAM. Since our protocol is an emulation of client-server ORAM we need to formally define the latter notion. We define client-Server ORAM scheme Σ as a pair of algorithms (Init, Access) parametrized by $\log n, D$, where Init initializes array OM of plaintext in the encrypted data stored on the server, while $\mathsf{Access}(\mathsf{instr}, \mathsf{OM}, \mathrm{N}, \mathsf{rec}')$, for $(\mathsf{instr}, \mathrm{N}, \mathsf{rec}')$ as above, outputs $(\mathsf{rec}, \mathsf{locL}, \mathsf{OM}')$, where rec is the record output by the client, locL is the list of locations in OM touched by the access procedure, and OM′ is identical to OM except at locations locL where the values in OM′ and OM can differ. In Path-ORAM variants like Circuit-ORAM, OM is a serialized list of nodes of trees $\mathsf{tree}_0, , .., \mathsf{tree}_{h-1}$ (see Sec. 2) and locL is the list of the nodes retrieved during access (and any additional nodes retrieved in eviction). Note that locL captures the only information the server learns because the positions of retrieved nodes define labels $\mathrm{L}_1, ..., \mathrm{L}_{h-1}$ received by the server.

Algorithm 24 ORAM.ML: Main Loop of Client/Server Path-ORAM

Input: L, $\mathsf{tree}^{\mathsf{S}}$, $(\mathrm{N}, \Delta\mathrm{N}, \mathrm{L}', \mathrm{L}'_{i+1}, *\ \mathsf{rec}')^{\mathsf{C}}$

Output: (1) L_{i+1} where $\mathrm{L}_{i+1} = \mathsf{T.data}[\Delta\mathrm{N}]$ for T on $\mathsf{tree.path}(\mathrm{L})$ s.t. $\mathsf{T.(fb|adr)} = 1|\mathrm{N}$ (* or $\mathsf{rec}^{\mathsf{C}}$ where $\mathsf{rec} = \mathsf{T.data}$)
(2) $\mathsf{tree.path}(\mathrm{L})^{\mathsf{S}}$ modified by eviction, with $\mathsf{T.lb} := \mathrm{L}'$ and $\mathsf{T.data}[\Delta\mathrm{N}] := \mathrm{L}'_{i+1}$ (* $\mathsf{T.data} := \mathsf{rec}'$)

S sends $\mathsf{path} = \mathsf{tree.path}(\mathrm{L})$ to C, who computes the following:

Retrieval of Next Label/Record

1: $\mathsf{T} := \mathsf{retrieve}(\mathsf{path.(fb|adr)}, 1|\mathrm{N})$ ▷ $\mathsf{T} \in \mathsf{path}$ s.t. $\mathsf{T.(fb|adr)} = 1|\mathrm{N}$

2: $\mathrm{L}_{i+1} := \mathsf{T.data}[\Delta\mathrm{N}]$ (* replace with $\mathsf{rec} := \mathsf{T.data}$)

Post-Process

3: $\mathsf{T.lb} := \mathrm{L}'$, $\mathsf{T.data}[\Delta\mathrm{N}] := \mathrm{L}'_{i+1}$ (* $\mathsf{T.data} := \mathsf{rec}'$)

4: set $\mathsf{fb} := 0$ at position in path where T was found in step 1

5: $\mathsf{path} := \mathsf{path.append\text{-}to\text{-}root}(\mathsf{T})$

Eviction

6: $\mathsf{EM} := \mathsf{PathORAM\text{-}Route}(\mathrm{L}, \mathsf{path.(fb, lb)})$ ▷ EM is an eviction map

7: $\mathsf{path}' := \mathsf{ApplyMovement}(\mathsf{EM}, \mathsf{path})$ ▷ $\mathsf{path}' = \mathsf{EM}(\mathsf{path})$

C sends L_{i+1} and path' to S, who inserts it in tree as $\mathsf{tree.path}(\mathrm{L})$

*: On top-level ORAM tree; ▷: Comments;
!: For $i = 0$, C runs steps 2-3 for $\mathsf{T} := \mathsf{tree}_0$ and sends L_{i+1} and (modified) tree_0 to S

For the sake of definition 1 below, consider experiment Exp_{Σ} which on input $\mathsf{accL} = \{(\mathsf{instr}_i, \mathrm{N}_i, \mathsf{rec}'_i)\}_{i=1}^{q}$ sets $\mathsf{OM} \xleftarrow{\$} \Sigma.\mathsf{Init}$, then sets $(\mathsf{rec}_i, \mathsf{locL}_i, \mathsf{OM}_i) \xleftarrow{\$} \Sigma.\mathsf{Access}(\mathsf{instr}_i, \mathsf{OM}_{i-1}, \mathrm{N}_i, \mathsf{rec}'_i)$ for $i = 1, .., q$, and outputs $\{(\mathsf{rec}_i, \mathsf{locL}_i)\}_{i=1}^{q}$.

Definition 1. *We call a client-server ORAM scheme Σ (statistically)* secure *if there exists an efficient algorithm* SIM *s.t. for any* $\log n, D$ *and list* $\mathsf{accL} = \{(\mathsf{instr}_i, \mathrm{N}_i, \mathsf{rec}'_i)\}_{i=1}^{q}$ *polynomial in security parameter, variables* $\mathsf{View}_{\mathsf{ideal}}$ *and* $\mathsf{View}_{\mathsf{real}}$ *are (statistically) indistinguishable, where* $\mathsf{View}_{\mathsf{ideal}} \xleftarrow{\$} \mathsf{SIM}(\log n, D, q)$ *and* $\mathsf{View}_{\mathsf{real}} = \{\mathsf{locL}_i\}_{i=1}^{q}$ *where* $\{(\mathsf{rec}_i, \mathsf{locL}_i)\}_{i=1}^{q} \xleftarrow{\$} \mathsf{Exp}_{\Sigma}(\mathsf{accL})$.

We call Σ (statistically) correct *if experiment* $\mathsf{Exp}_{\Sigma}(\mathsf{accL})$ *retrieves records* $\{\mathsf{rec}_i\}_{i=1}^{q}$ *which are consistent, except for negligible probability, with the write instruction sequence* $\{(\mathsf{instr}_i, \mathrm{N}_i, \mathsf{rec}'_i) \in \mathsf{accL}$ *s.t.* $\mathsf{instr}_i = \mathsf{write}\}$ *in* accL.

Secure Realization of 3PC ORAM Functionality. We argue that our 3PC ORAM realizes functionality $\mathsf{F}_{\mathsf{ORAM}}$ by arguing that it is a *3PC emulation* of a secure client-server ORAM, namely the Circuit-ORAM. Consider an ORAM scheme Σ and define two functionalities: $\mathsf{F}_{\Sigma.\mathsf{Init}}$, which runs $\mathsf{OM} \xleftarrow{\$} \Sigma.\mathsf{Init}$ and outputs a random sharing $\langle \mathsf{OM} \rangle$, and functionality $\mathsf{F}_{\Sigma.\mathsf{Access}}$, which on Z's input $(\mathsf{instr}, \langle \mathsf{OM}, \mathrm{N}, \mathsf{rec}' \rangle)$ does the following: (1) it reconstructs $\mathsf{OM}, \mathrm{N}, \mathsf{rec}'$ from their

sharings; (2) it runs $(\mathsf{rec}, \mathsf{locL}, \mathsf{OM}') \xleftarrow{\$} \Sigma.\mathsf{Access}(\mathsf{instr}, \mathsf{OM}, \mathrm{N}, \mathsf{rec}')$; (3) it creates fresh sharing $\langle \mathsf{rec} \rangle$; (4) it creates $\langle \mathsf{OM}' \rangle$, which is identical to $\langle \mathsf{OM} \rangle$ for all $i \notin \mathsf{locL}$, but for all $i \in \mathsf{locL}$ sharings $\langle \mathsf{OM}'[i] \rangle$ are fresh; (5) it outputs $(\langle \mathsf{rec}, \mathsf{OM}' \rangle, \mathsf{locL})$ where locL is public, i.e. it is sent to each party. The following proposition is elementary:

Lemma 1. *If Σ is a secure and correct client-server ORAM scheme, and $\Pi = (\Pi.\mathsf{Init}, \Pi.\mathsf{Access})$ where $\Pi.\mathsf{Init}$ securely realizes $\mathsf{F}_{\Sigma.\mathsf{Init}}$ and $\Pi.\mathsf{Access}$ securely realizes $\mathsf{F}_{\Sigma.\mathsf{Access}}$ then Π securely realizes 3PC ORAM functionality $\mathsf{F}_{\mathsf{ORAM}}$.*

Let Σ_{CircO} be the client-server Circuit-ORAM scheme. Theorem 1 below is the key part of the security proof:

Theorem 1. *Protocol* 3PC-ORAM.Access *securely realizes functionality $\mathsf{F}_{\Sigma_{\mathsf{CircO}}.\mathsf{Access}}$.*

We argue theorem 1 below, but first we state its implication together with Lemma 1 and the fact that the client-server ORAM scheme Σ_{CircO} is secure [27]. Corollary 1, restated in a simpler form in Section 4, is the main security claim of our paper.

Corollary 1. *Protocol $\Pi = (\mathsf{Init}, \mathsf{Access})$ where $\Pi.\mathsf{Init}$ is implemented by calling functionality $\mathsf{F}_{\Sigma_{\mathsf{CircO}}.\mathsf{Init}}$ directly while $\Pi.\mathsf{Access}$ runs protocol* 3PC-ORAM.Access*, is a secure realization of 3PC ORAM functionality $\mathsf{F}_{\mathsf{ORAM}}$ in the $\mathsf{F}_{\Sigma_{\mathsf{CircO}}.\mathsf{Init}}$-hybrid world (i.e. assuming secure initialization according to $\mathsf{F}_{\Sigma_{\mathsf{CircO}}.\mathsf{Init}}$).*

Proof of Theorem 1. We argue Theorem 1 in a recursive way, relying on the composition property of UC security [7], which says that in order to argue that protocol Π, which makes calls to sub-protocols $\Pi_1, ..., \Pi_k$, securely realizes functionality F, you need to argue two facts: (1) first, that protocol Π realizes F if sub-protocols $\Pi_1, ..., \Pi_k$ are replaced by calls to idealized functionalities $\mathsf{F}_1, ..., \mathsf{F}_k$, where F_i is an algorithm that implements the intended input/output behavior of sub-protocol Π_i (note that we state the intended input/output behavior of each protocol at the beginning of the Algorithm figure containing that protocol), and (2) for each i protocol Π_i securely realizes functionality F_i. If a protocol Π_i makes calls to other sub-protocols then the above proof procedure is recursively applied to Π_i. Following this methodology, we first argue that protocol 3PC-ORAM.Access securely realizes $\mathsf{F}_{\Sigma_{\mathsf{CircO}}.\mathsf{Access}}$ if subprotocol 3PC-ORAM.ML is replaced by a call to an ideal functionality $\mathsf{F}_{\mathsf{3O\text{-}ML}}$, which implements the desired input/output behavior of 3PC-ORAM.ML as stated in Alg. 2. Then, we argue that protocol 3PC-ORAM.ML securely realizes functionality $\mathsf{F}_{\mathsf{3O\text{-}ML}}$, and we do so using the same procedure as in the case of the top-level protocol 3PC-ORAM.Access, i.e. we first argue that 3PC-ORAM.ML realizes functionality $\mathsf{F}_{\mathsf{3O\text{-}ML}}$ if the calls to each of the subprotocols 3PC-ORAM.ML invokes are replaced by calls to the ideal functionalities which implement the intended input/output behavior of these subprotocols.

Following this recursive methodology we argue that each of these sub-protocols indeed realizes its corresponding idealized functionality F, and we do so (following the same recursive procedure, i.e. replacing any sub-protocol calls by their idealized functionalities) not "centrally", i.e. in the proof below, but by supplementing each sub-protocol figure with a security argument. The goal of each of these arguments is to briefly argue (1) that the protocol outputs are distributed as specified by

F, and (2) that for each party P_i there exists an efficient *simulator* SIM_i s.t. the distribution of messages P_i receives in the subprotocol is indistinguishable from a distribution produced by SIM_i from the inputs to F and outputs of F held by that party. These two components show that a protocol Π securely realizes F, because it shows that for each corrupt party $\mathsf{P}_i \in \{\mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3\}$ and every input $x = (x_1, x_2, x_3)$, the distribution $\mathsf{View}_{\mathsf{real},\mathsf{P}_i} = (y, \mathsf{Tr}_i)$ where $y = (y_1, y_2, y_3)$ are the parties' outputs in a random execution of protocol Π on inputs x and Tr_i is P_i's transcript in that execution, is indistinguishable from $\mathsf{View}_{\mathsf{ideal},\mathsf{P}_i} = (y, \mathsf{Tr}_i)$ where $y \xleftarrow{\$} \mathsf{F}(x)$ and $\mathsf{Tr}_i \xleftarrow{\$} \mathsf{SIM}_i(x_i, y_i)$. Because most of our protocols employ information-theoretic tools, for most protocols we do not explicitly show simulator algorithms, but instead we argue that the messages received by each party in the protocol are either randomly distributed or are determined by (and efficiently computable from) other messages and input/outputs of this party. All of these arguments can be turned into corresponding simulators. We note that our protocols often generate pseudorandom values using a pair-wise shared PRG seeds. The security arguments we provide are simplified by treating all such values as random, but by standard reduction to PRG security they imply computational security of the respective protocol.

Security argument for 3PC-ORAM.Access. It remains for us to argue that 3PC-ORAM.Access securely realizes $\mathsf{F}_{\Sigma_{\mathsf{CircO}}.\mathsf{Access}}$ if subprotocol 3PC-ORAM.ML realizes an ideal functionality $\mathsf{F}_{\mathsf{3O\text{-}ML}}$ which is stated as the input/output requirements of 3PC-ORAM.ML, Alg. 2, and that 3PC-ORAM.ML realizes $\mathsf{F}_{\mathsf{3O\text{-}ML}}$ if subprotocols KSearch, 3ShiftXorPIR, 3ShiftPIR, ULiT, FlipFlag, GC, PermBuckets, PermTuples, and XOT realize the ideal functionalities stated as their input/output requirements.

We thus first argue that protocol 3PC-ORAM.Access securely realizes $\mathsf{F}_{\Sigma_{\mathsf{CircO}}.\mathsf{Access}}$ if calls to subprotocol 3PC-ORAM.ML are replaced by calls to an ideal functionality $\mathsf{F}_{\mathsf{3O\text{-}ML}}$. As stated by the input/output requirements of 3PC-ORAM.ML, Alg. 2, functionality $\mathsf{F}_{\mathsf{3O\text{-}ML}}$ takes input $x = (\mathrm{L}, \langle \mathsf{tree}, \mathrm{N}, \Delta\mathrm{N}, \mathrm{L}', \mathrm{L}'_{i+1}, \mathsf{rec}' \rangle)$ and outputs $y = (\mathrm{L}_{i+1}, \langle \mathsf{tree}' \rangle)$ (or $y = (\langle \mathsf{rec} \rangle, \langle \mathsf{tree}' \rangle)$ if the call pertains to a final tree), where (1) $\mathrm{L}_{i+1} = \mathsf{T.data}[\Delta\mathrm{N}]$ (or $\mathsf{rec} = \mathsf{T.data}$ for a final tree) for a unique T in $\mathsf{tree.path}(\mathrm{L})$ s.t. $\mathsf{T.(fb|adr)} = 1|\mathrm{N}$, and (2) tree' is identical to tree except for $\mathsf{tree.path}(\mathrm{L})$ which is modified by (2a) inserting into the root bucket in $\mathsf{tree.path}(\mathrm{L})$ tuple T modified s.t. $\mathsf{T.lb} = \mathrm{L}'$ and $\mathsf{T.data}[\Delta\mathrm{N}] := \mathrm{L}'_{i+1}$ (or $\mathsf{T.data} = \mathsf{rec}'$ for a final tree and $\mathsf{instr} = \mathsf{write}$), (2b) flipping the full/empty bit to empty at the original location in $\mathsf{tree.path}(\mathrm{L})$ where T is found, and (2c) applying the Circuit-ORAM eviction algorithm to $\mathsf{tree.path}(\mathrm{L})$. Note that if one removes the secret-sharing layer from the inputs and outputs of $\mathsf{F}_{\mathsf{3O\text{-}ML}}$ then this is a deterministic procedure which the client-server Circuit-ORAM algorithm performs for each tree. Note also that protocol 3PC-ORAM.Access follows $\Sigma_{\mathsf{CircO}}.\mathsf{Access}$, i.e. algorithm ORAM.Access, Alg. 23, except that it also operates on secret-shared inputs $\langle \mathsf{OM}, \mathrm{N}, \mathsf{rec}' \rangle$. Therefore the final output $\langle \mathsf{rec} \rangle$, the modified $\langle \mathsf{OM} \rangle$ datastructure, and the location list locL, uniquely defined by labels $\mathrm{L}_1, ..., \mathrm{L}_{h-1}$, are created exactly as by functionality $\mathsf{F}_{\Sigma.\mathsf{Access}}$. Since the only non-secret-shared information protocol 3PC-ORAM.Access reveals are labels $\mathrm{L}_1, ..., \mathrm{L}_{h-1}$, which are implied by the location list locL, there is no further information to simulate.

Security argument for 3PC-ORAM.ML. We argue that protocol 3PC-ORAM.ML securely realizes $\mathsf{F}_{\text{3O-ML}}$ defined above if calls to subprotocols KSearch, 3ShiftXorPIR, 3ShiftPIR, ULiT, FlipFlag, GC, PermBuckets, PermTuples, and XOT are replaced by calls to the ideal functionalities which these procedures implement, stated by the intended input/output behavior of these subprotocols. We will argue this only for the non-boundary case, because the case of the first tree tree_0 and the final tree tree_{h-1} are both simplifications of the processing for the non-boundary case. We need to argue two things: First, that for any $x = (\mathrm{L}, \langle \mathsf{tree}, \mathrm{N}, \Delta\mathrm{N}, \mathrm{L}', \mathrm{L}'_{i+1}, \mathsf{rec}' \rangle)$ the outputs $y = (\mathrm{L}_{i+1}, \langle \mathsf{tree}' \rangle)$ of 3PC-ORAM.ML on input x are distributed as the outputs of $\mathsf{F}_{\text{3O-ML}}(x)$. And secondly, that for every player P_i there is an efficient simulator SIM_i s.t. $\mathsf{Tr}_i \xleftarrow{\$} \mathsf{SIM}_i(x_i, y_i)$ is indistinguishable from P_i's transcript in an instance of 3PC-ORAM.ML which outputs y on input x, and where x_i and y_i are P_i's parts in respectively x and y. First, note that given $\langle \mathsf{tree} \rangle$ in x, the only part of $\langle \mathsf{tree}' \rangle$ in y that matters is the part that is modified by both protocol 3PC-ORAM.ML and functionality $\mathsf{F}_{\text{3O-ML}}$, i.e. path $\mathsf{tree.path}(\mathrm{L})$ uniquely defined by L in x. In other words, we can think of both 3PC-ORAM.ML and $\mathsf{F}_{\text{3O-ML}}$ as outputting $y = (\mathrm{L}_{i+1}, \langle \mathsf{path}' \rangle)$ where $\langle \mathsf{path}' \rangle$ in the case of 3PC-ORAM.ML is defined in step 9, Alg. 2, and in the case of $\mathsf{F}_{\text{3O-ML}}$ it is a fresh sharing of modified $\mathsf{tree.path}(\mathrm{L})$ described in points (2a)-(2c) above.

We first argue that $(\mathrm{L}_{i+1}, \langle \mathsf{path}' \rangle)$ output by 3PC-ORAM.ML is distributed as outputs of $\mathsf{F}_{\text{3O-ML}}$. Let $\mathsf{path} = \mathsf{tree.path}(\mathrm{L})$ for $\mathsf{tree}, \mathrm{L}$ in x. By the input/output functionality of KSearch, see Alg. 6, invoked in step 1 of 3PC-ORAM.ML, the output $\langle j \rangle_{\mathsf{shift}}$ of this call is a fresh sharing of j s.t. $\mathsf{path}[j].(\mathsf{fb}|\mathsf{adr}) = 1|\mathrm{N}$, because by the Path-ORAM invariant there is a unique index j of a tuple in path s.t. $\mathsf{path}[j].(\mathsf{fb}, \mathsf{adr}) = (1, \mathrm{N})$. By the input/output functionalities 3ShiftPIR and 3ShiftXorPIR (in Alg. 9,11), invoked in steps 2 and 3, outputs $\langle X \rangle$ and L_{i+1} of these calls satisfy that $\langle X \rangle$ is a fresh sharing of $X = \mathsf{path}[j].\mathsf{data}$ and $\mathrm{L}_{i+1} = \mathsf{path}[j].\mathsf{data}[\Delta\mathrm{N}] = X[\Delta\mathrm{N}]$. (Observe that path.data in the call to 3ShiftXorPIR is an ℓ by m array containing next-level labels, for ℓ the number of tuples in path and $m = 2^\tau$, the size of the data array field in each tuple.) This already shows that L_{i+1} is computed as in $\mathsf{F}_{\text{3O-ML}}$. By the input/output functionality ULiT (in Alg. 14), invoked in step 4, this call outputs fresh $\langle \mathsf{T} \rangle$ for $\mathsf{T} = (1|\mathrm{N}|\mathrm{L}'|X')$ s.t. $X'[\Delta\mathrm{N}] = \mathrm{L}'_{i+1}$ and payload X' is identical to payload X in all other entries. It follows that T is a new copy of the retrieved tuple $(1|\mathrm{N}|\mathrm{L}|X)$ with the modified labels as in step (2a) of $\mathsf{F}_{\text{3O-ML}}$. Next, the input/output functionality FlipFlag (in Alg. 16), invoked in step 5, ensures that path.fb values are modified so that the free bit is flipper at position j, i.e. that $\mathsf{path}[j].\mathsf{fb}$ becomes $1 \oplus \mathsf{path}[j].\mathsf{fb}$, but since $\mathsf{path}[j].\mathsf{fb}$ was 1 (by the functionality of KSearch) it now becomes 0, as in step (2b) of $\mathsf{F}_{\text{3O-ML}}$. Finally, as we argue in Section 3, steps 6-9 implement the Circuit-ORAM eviction procedure correctly: GC(Route) in step 6 outputs $(\overline{\sigma}, \mathsf{t}')$ for D s.t. $\mathsf{EM}_{\sigma,\mathsf{t}}$ is the Circuit-ORAM eviction map for $\mathsf{t} = \mathsf{t}' \oplus \delta$, and PermBuckets and PermTuples calls in steps 7-8 output σ° and t° for D s.t. $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$ and $\mathsf{t}^\circ = \rho \oplus \pi(\mathsf{t}')$. Finally, equation (1) in Section 3 implies that if $\Pi = \tilde{\rho} \cdot \ddot{\pi} \cdot \tilde{\delta}$ then $\Pi^{-1} \cdot \mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ} \cdot \Pi$, which is a permutation applied to path in call to XOT in step 9, is the same permutation as $\mathsf{EM}_{\sigma,\mathsf{t}}$, hence path' is computed as in step (2c) of $\mathsf{F}_{\text{3O-ML}}$.

It thus remains for us to exhibit simulators SIM_i for each party $\mathsf{P}_i \in \{\mathsf{C}, \mathsf{D}, \mathsf{E}\}$ s.t. $\mathsf{Tr}_i \xleftarrow{\$} \mathsf{SIM}_i(x_i, y_i)$ is indistinguishable from P_i's transcript in an instance of 3PC-ORAM.ML which outputs y on input x. Recall also that we need only to consider 3PC-ORAM.ML in the *hybrid* world where each of its subprotocol calls is replaced with a call to the corresponding functionality. (As explained above, this suffices *if* we argue, as we do, that each of the subprocedure protocols securely realizes its ideal functionality.) First, observe that in steps 1-5 the functionalities called in each step produce random sharings of various values, specifically $\langle i \rangle_{\mathsf{shift}}$, $\langle X \rangle$, $\langle \mathsf{T} \rangle$, and $\langle \mathsf{path.fb} \rangle$, so the views of any party of these outputs can be easily simulated because each party's shares of each of these outputs are randomly distributed. The one exception is step 3, which produces public value $\mathrm{L}_{i+1} = \mathsf{path}[j].\mathsf{data}[\Delta\mathrm{N}]$, but since this is the same value that is output by $\mathsf{F}_{\mathsf{3O\text{-}ML}}$, as argued above, each simulator can trivially reproduce it given $\mathsf{F}_{\mathsf{3O\text{-}ML}}$'s output. In steps 6-8 only E and D receive outputs, namely wk for E and $\overline{\sigma}, \mathsf{t}', \sigma^\circ, \mathsf{t}^\circ$ for D. Value wk is an array of random keys, which the simulator of E's view can trivially produce. By the definition of the input/output behavior of GC(Route) (defined by GC and Route), PermBuckets, and PermTuples, the outputs of D are distributed as $(\overline{\sigma}, \mathsf{t}', \sigma^\circ, \mathsf{t}^\circ) = (\{\mathsf{wk} : \sigma\}, \mathsf{t} \oplus \delta, \pi \cdot \sigma \cdot \pi^{-1}, \rho \oplus \pi(\mathsf{t}'))$, where wk are random keys chosen by E, $\pi \xleftarrow{\$} \mathsf{perm}_d$ and $\delta, \rho \xleftarrow{\$} \mathsf{array}^{\log(w+1)}[d]$ are chosen by C and E, and pair (σ, t), for $\sigma \in \mathsf{perm}_d$ and $\mathsf{t} \in \mathsf{array}^{\log(w+1)}[d]$, forms the expanded Circ-ORAM eviction map computed (deterministically) by circuit Route given leaf L and label-data $\mathsf{path}.(\mathsf{fb}, \mathsf{lb})$ of the path tuples. Variable $\overline{\sigma}$ is random because it is a selection of keys in wk which are distributed as random strings to D regardless of the value of σ. Since σ output by circuit Route is guaranteed to be a *cycle* in perm_d, for every σ the random variable $\sigma^\circ = \pi\sigma\pi^{-1}$, for $\pi \xleftarrow{\$} \mathsf{perm}_d$, is distributed as a *random cycle* in perm_d. The other two values, t' and t°, are random in $\mathsf{array}^{\log(w+1)}[d]$ because of one-time pads δ and ρ. Thus the whole string of values learned by D in these steps can be produced by a simulator. Since calls to XOT in steps 9 produce only random shares to each party, this concludes the argument.

D Routing Circuit Computation

In this section we explain the construction of the routing circuit Route used in the eviction phase of protocol 3PC-ORAM.ML (see Step 6 in Alg. 2, Section 3).

D.1 Main Routing Circuit

Circuit Route determines the eviction map by generating a dp array (PrepareDeepest), computing the eviction array σ from dp (PrepareTarget), and making the eviction map σ into a cycle (MakeCycle). Circuits PrepareDeepest and PrepareTarget are, with minor variations, the same circuits which implemented the original Circuit-ORAM eviction computation CircORAM-Route [27], but circuit MakeCycle is new. Because of some small differences in the implementation of PrepareDeepest and PrepareTarget, the combined size of circuit Route is virtually identical to the size of CircORAM-Route reported in [27]. We discuss these three circuits in the following subsections.

Algorithm 25 Circuit Route (Used in Step 6, Alg 2)

Param: Tree height d, bucket size w.

Input: Full/empty bits $\mathsf{fb} \in \mathsf{array}^1[d, w]$; labels $\mathrm{lbl} \in \mathsf{array}^d[d, w]$; path label $\mathrm{L} \in \{0,1\}^d$; masks $\delta \in \mathsf{array}^{\log(w+1)}[d]$

Output: $\sigma \in \mathsf{perm}_d$ and $\mathsf{t}' \in \mathsf{array}^{\log(w+1)}[d]$, where σ extends Φ into a cycle and $\mathsf{t}' = \delta \oplus \mathsf{t}$ for Eviction Map Φ and Tuple Index t computed as in Circuit-ORAM [27]

1: $(\mathrm{dp}, j_{\mathsf{d}}, j_{\mathsf{e}}, e) :=$ $\mathsf{PrepareDeepest}(\mathrm{L}, \mathsf{fb}, \mathrm{lbl})$
2: $(\Phi, \mathsf{t}', nTop, nBot, eTop, eBot) :=$ $\mathsf{PrepareTarget}(\mathrm{dp}, j_{\mathsf{d}}, j_{\mathsf{e}}, e, \delta)$
3: $\sigma :=$ $\mathsf{MakeCycle}([\Phi], nTop, nBot, eTop, eBot)$

Circuit cost: $[3wd + (2w+5) \cdot log(w) + (d+34) \cdot log(d)] \cdot d \ \rightarrow \ O(d^2 \log(d))$

D.2 Prepare Array dp

Algorithm $\mathsf{PrepareDeepest}$ in Alg. 26, based on the same name algorithm in [27], outputs an array dp where $\mathrm{dp}[i] < i$ is the index of the first bucket in the path which contains a tuple that can be evicted to the i-th bucket. (If no tuples in higher levels can be evicted to the i-th bucket then $\mathrm{dp}[i] = \perp$.) In addition, $\mathsf{PrepareDeepest}$ outputs three other arrays $j_{\mathsf{d}}, j_{\mathsf{e}}, e$, where $j_{\mathsf{d}}[i]$ is the index of the "deepest tuple" in the i-th bucket, i.e. a tuple which could be evicted furthest down from that bucket, $e[i] = 1$ if and only if there is an empty tuple at this level, and $j_{\mathsf{e}}[i]$ is the index of that empty tuple. (If $e[i] = 0$ then $j_{\mathsf{e}}[i]$ is meaningless.)

Algorithm 26 Circuit $\mathsf{PrepareDeepest}$ [27] (Used in Alg. 25)

Param: Tree height d, non-root bucket size w.

Input: Path label $\mathrm{L} \in \{0,1\}^d$, array of full/empty bits $\mathsf{fb} \in \mathsf{array}^1[d, w]$ and labels $\mathrm{lbl} \in \mathsf{array}^d[d, w]$

Output: $\mathrm{dp} \in \mathsf{array}^{\log(d)}[d]$, $e \in \mathsf{array}^1[d]$, $j_{\mathsf{d}}, j_{\mathsf{e}} \in \mathsf{array}^{\log(w+1)}[d]$, s.t. $j_{\mathsf{d}}[i], j_{\mathsf{e}}[i]$ are indexes of deepest/empty tuples in i-th bucket, $e[i] = 1$ if there i-th bucket has an an empty tuple, and $\mathrm{dp}[i] = i'$ s.t. the deepest tuple in (i')-th bucket can move to bucket i ($\mathrm{dp}[i] = \perp$ if no such i' exists)

1: $\mathrm{dp} := [\perp, \perp, ..., \perp]$, $src := \perp$, $goal := -1$
2: **for** $i := 0$ **to** $d-1$ **do** ▷ cycle: d
3: **if** $goal \geq i$ **then** ▷ cost: $log(d)$
4: $\mathrm{dp}[i] := src$ ▷ cost: $log(d)$
5: $(l, j_{\mathsf{d}}[i], j_{\mathsf{e}}[i], e[i]) := \mathsf{FDAE}(i, \mathrm{L}, \mathsf{fb}[i], \mathrm{lbl}[i])$ ▷ cost: Alg 27
6: **if** $l > goal$ **then** ▷ cost: $log(d)$
7: $goal := l$ ▷ cost: $log(d)$
8: $src := i$ ▷ cost: $log(d)$

Circuit cost: $(3w + \log d) \cdot d^2 + (2w \log w + 5 \log d) \cdot d$

Find the Deepest and Empty Tuples. Algorithm FDAE in Alg. 27 (which stands for FindDeepestAndEmpty) is adopted from [27], and it is a sub-procedure of Alg. 26 which finds the "deepest tuple", i.e. a tuple which can be evicted the furthest down the path, in a bucket at level (=depth) i in the path, and outputs its index j_{d} in the bucket together with the target level l'. FDAE also determines if there is an empty tuple in this bucket, and outputs its index j_{e} and a flag e which is set to 1 if an empty tuple was found. If no tuple can be moved down from the i-th bucket then FDAE returns $(j_{\mathsf{d}}, l') = (0, i)$, and if there is no empty tuple then $(j_{\mathsf{e}}, e) = (0, 0)$.

Algorithm 27 Circuit FDAE [27] (Used in Alg. 26)

Param: Tree height d, non-root bucket size w.

Input: Level index $i \in \mathsf{Z}_d$, path label $\mathrm{L} \in \{0,1\}^d$, tuples' full/empty bits $\mathsf{fb} \in \mathsf{array}^1[w]$ and labels $\mathrm{lbl} \in \mathsf{array}^d[w]$.

Output: $l' \in \mathsf{Z}_d$, $j_{\mathsf{d}}, j_{\mathsf{e}} \in \mathsf{Z}_{w+1}$, $e \in \{0,1\}$, where l' is the deepest level index, $j_{\mathsf{d}}, j_{\mathsf{e}}$ are indexes of resp. the first deepest tuple and the first empty tuple, and e is a flag indicating whether the bucket contains an empty tuple.

1: $l := 0^i|1^{d-i-1}$; $j_{\mathsf{d}}, j_{\mathsf{e}}, et := 0$
2: **for** $j := 0$ **to** $w-1$ **do** ▷ cycle: w
3: $lz := \mathrm{lbl}[j] \oplus \mathrm{L}$
4: $lz' :=$ set all bits after the first bit 1 in lz to 1 ▷ cost: d
5: **if** $\mathsf{fb}[j] = 1$ **and** $lz' < l$ **then** ▷ cost: d
6: $j_{\mathsf{d}} := j$ ▷ cost: $\log w$
7: $l := lz'$ ▷ cost: d
8: **else if** $\mathsf{fb}[j] = 0$ **and** $e = 0$ **then**
9: $j_{\mathsf{e}} := j$ ▷ cost: $\log w$
10: $e := 1$
11: $l' :=$ number of leading 0s in l ▷ cost: $d \cdot \log(d)$

Circuit cost: $d \cdot (3w + \log d) + 2w \log w$

Step explanations of Alg. 26 FDAE:
Line 1: j_{d} stores the index and l the target-depth of the first "deepest tuple", i.e. the tuple which can go deepest along the path L. j_{d} is initialized as 0, the first tuple in the bucket, and l as the current depth i, kept in a special-purpose unary format as the number of leading zeros. j_{e}, initialized as 0, stores the index of the first empty tuple. Flag e, initialized as 0, indicates if an empty tuple is found.
Lines 3-4: the number of leading zeros in the xor of a tuple's label with leaf L defines the deepest level the tuple can be evicted to.
Line 5: Bitstring lz' is smaller than bitstring l iff lz' has more leading 0's.
Line 6-7: If the new tuple is non-empty, i.e. $\mathsf{fb}[j] = 1$, and lz' has more leading 0's than l, then the new tuple can go deeper then the previously found one, in which case we update index j_{d} and target-depth l.

Line 8-10: If the new tuple is empty, update j_{e} and e.
Line 11: Compute the deepest level number in an integer format by counting the number of leading zeros in l.

D.3 Prepare Arrays σ and t

Algorithm 28 Circuit PrepareTarget (following [27]) (Used in Alg. 25)

Param: Tree height d, non-root bucket size w

Input: $\mathrm{dp} \in \mathsf{array}^{\log(d)}[d]$, $j_{\mathsf{d}}, j_{\mathsf{e}} \in \mathsf{array}^{\log(w+1)}[d]$,
$e \in \mathsf{array}^{1}[d]$, $\delta \in \mathsf{array}^{\log(w+1)}[d]$

Output: $\sigma \in \mathsf{array}^{\log(d)}[d]$, $\mathsf{t}' \in \mathsf{array}^{\log(w+1)}[d]$, $nTop, nBot, eTop, eBot \in \mathsf{Z}_d$

1: $nTop, nBot, eTop, eBot, src, \mathrm{dest} := \perp$
2: $\sigma, \mathsf{t} := [\perp, \perp, ..., \perp]$
3: **for** $i := d-1$ **to** 0 **do**
4: **if** $i = src$ **then** ▷ cost: $log(d)$
5: $\sigma[i] := \mathrm{dest}$ ▷ cost: $log(d)$
6: $\mathsf{t}[i] := j_{\mathsf{d}}[i]$ ▷ cost: $log(w)$
7: $src := \perp$ ▷ cost: $log(d)$
8: **if** $\mathrm{dp}[i] = \perp$ **then** ▷ cost: $log(d)$
9: $\mathrm{dest} := i$ ▷ cost: $log(d)$
10: **else**
11: $\mathrm{dest} := \perp$ ▷ cost: $log(d)$
12: **if** $\mathrm{dp}[i] \neq \perp$ **then**
13: **if** $\mathrm{dest} \neq \perp$ **and** $src = \perp$ **then** ▷ cost: $2log(d)$
14: $\sigma[i] := \mathrm{dest}$ ▷ cost: $log(d)$
15: $\mathsf{t}[i] := j_{\mathsf{e}}[i]$ ▷ cost: $log(w)$
16: **if** $(\mathrm{dest} = \perp$ **and** $e[i] = 1)$ **or** $\sigma[i] \neq \perp$ **then** ▷ cost: $log(d)$
17: $src := \mathrm{dp}[i]$ ▷ cost: $log(d)$
18: $\mathrm{dest} := i$ ▷ cost: $log(d)$
19: $eTop := src$ ▷ cost: $log(d)$
20: **if** $eBot = \perp$ **then** ▷ cost: $log(d)$
21: $eBot := \mathrm{dest}$ ▷ cost: $log(d)$
22: $\mathsf{t}[i] := j_{\mathsf{e}}[i]$ ▷ cost: $log(w)$
23: **if** $\mathsf{t}[i] = \perp$ **then** ▷ cost: $log(w)$
24: $\mathsf{t}[i] := w$ ▷ cost: $log(w)$
25: $nTop := i$ ▷ cost: $log(d)$
26: **if** $nBot = \perp$ **then** ▷ cost: $log(d)$
27: $nBot := i$ ▷ cost: $log(d)$
28: $\mathsf{t}'[i] = \mathsf{t}[i] \oplus \delta[i]$

Circuit cost: $[5log(w) + 18log(d)] \cdot d$

Algorithm PrepareTarget in Alg. 28 is an extended version of the corresponding algorithm in [27], which determines the final eviction pattern. PrepareTarget outputs a σ array which contains the same eviction movement as in [27], plus the eviction jumps filling up the possible gaps. PrepareTarget also outputs an array t where $\mathsf{t}[i]$ is the index of the tuple that will be evicted on level i. Note that each $\mathsf{t}[i]$ is selected from one of $j_{\mathsf{d}}[i], j_{\mathsf{e}}[i]$, or w (fake/empty tuple index) depending on what kind of eviction movement level i is doing. And this t will be finally masked by δ so the real indices will be hidden to D.

Step explanations of Alg. 28 PrepareTarget:
Line 1: $nTop$ stores the index of the top level which has no movement during eviction; $nBot$ stores the index of the bottom level which has no movement; $eTop$ stores the index of the top level which contains a tuple to be evicted; $eBot$ stores the index of the bottom level which a tuple will be evicted to. src indicates the current level which has a tuple that can be evicted; dest stores the current target level which a tuple can be evicted to.
Line 2: $\sigma[i]$ indicates the target level that a tuple at level i will be evicted to; t_i is the index of the tuple that will be evicted at level i.
Line 4-5: If we reach the source level i which contains a tuple to be evicted, then we update $\sigma[i]$ as the destination level where the tuple should go to.
Line 6: We are evicting a full tuple in this case, so we should update $\mathsf{t}[i]$ as the full tuple index $j_{\mathsf{d}}[i]$.
Line 8-11: If $\mathrm{dp}[i] = \perp$ but we are evicting a tuple on level i, then level i can be the tail level of a gap, so we should still keep track of i as a possible eviction destination level.
Line 12: We are at the level where a tuple may be evicted to. This also means this level can be the head level of a gap.
Line 13-15: Because of Line 8-9, we have a gap tail level on record, so we update $\sigma[i]$ to add this gap jump, and thus use an empty tuple index $j_{\mathsf{e}}[i]$ as $\mathsf{t}[i]$.
Line 16-18: If we have an empty spot on this level, or we will also evict a tuple on this level, then we can allow some tuple to be evicted to this level. So we update src and dest so later on we can enter Line 4 section and update σ.
Line 19-21: We update $eTop$ every time we find a new eviction jump; we update $eBot$ only for the first time.
Line 22: At the bottom level where a tuple will be evicted to, we must choose an empty tuple spot to accept the incoming tuple.
Line 23-27: If the current level i does not have any movement during the eviction, then we will set the fake empty tuple index w as $\mathsf{t}[i]$. Also we will update $nTop$ and $nBot$ if necessary.
Line 28: Mask each $\mathsf{t}[i]$ with $\delta[i]$ so the real value is hidden.

D.4 Making the Eviction Map into A Cycle

Algorithm MakeCycle in Alg. 29 adds upwards spurious jumps to the eviction jump array σ output from PrepareTarget and makes the final eviction map as a cycle.

Step explanations of Alg. 29 MakeCycle:

Algorithm 29 Circuit MakeCycle (Used in Alg. 25)

Param: Tree height d.

Input: $\sigma \in \mathsf{array}^{\log(d)}[d]$, $nTop, nBot, eTop, eBot \in \mathsf{Z}_d$

Output: $\sigma \in \mathsf{array}^{\log(d)}[d]$

```
1:  nPrev := ⊥
2:  for i := 0 to d − 1 do
3:      if nTop = ⊥ then                                  ▷ cost: log(d)
4:          if i = eBot then                              ▷ cost: log(d)
5:              σ[i] := eTop                              ▷ cost: log(d)
6:      else if i = eBot then
7:          σ[i] := nBot                                  ▷ cost: log(d)
8:      else if σ[i] = ⊥ then                             ▷ cost: log(d)
9:          if i = nTop then                              ▷ cost: log(d)
10:             if eTop = ⊥ then                          ▷ cost: log(d)
11:                 σ[i] := nBot                          ▷ cost: log(d)
12:             else
13:                 σ[i] := eTop                          ▷ cost: log(d)
14:         else
15:             σ[i] := nPrev                             ▷ cost: log(d)
16:         nPrev := i                                    ▷ cost: log(d)
```

Circuit cost: $11log(d) \cdot d$

Line 1: $nPrev$ keeps track of the one previous level during the linear scan that does not have tuple movement.
Line 3-5: It is possible that after PrepareTarget every level is involved in eviction, so we only have real eviction sequence(RS) and no spurious sequence(SS) ($nTop$ and $nBot$ will be $\perp$). In this case we only do one thing to make the eviction cycle: when we are at the bottom level of RS, evict to the top level of RS.
Line 6-7: When $i = eBot$, which means $eBot \neq \perp$, this is the case where there are both RS and SS. And because we are now at the bottom level of RS, we should evict to the bottom level of SS.
Line 8: This means we are at the level where there is no eviction jump given by output of PrepareTarget. It is possible we have both RS and SS, or just SS (meaning no real eviction will be done on this path).
Line 9: We are at the top level where there is no real eviction.
Line 10-11: When $eTop = \perp$, we do not have RS but only SS. So we only need to do one thing to make the eviction cycle, which is to evict from the top level of SS to the bottom level of SS.
Line 12-13: If $eTop \neq \perp$, then we have both RS and SS. So when we are at the top level of SS, we should evict to the top level of RS.
Line 14-15: When we are at the no-real eviction levels except the top level, we should evict to the one previous no-real eviction level we encountered (because we are doing linear scan from root to leaf now, and we want the eviction direction of

SS to be from leaf to root).
Line 16: Every time we are on a new no-real eviction level, we should record it so we can add eviction jump from the next no-real eviction level to this one.

E Efficiency and Performance

E.1 3PC ORAM Round Complexity

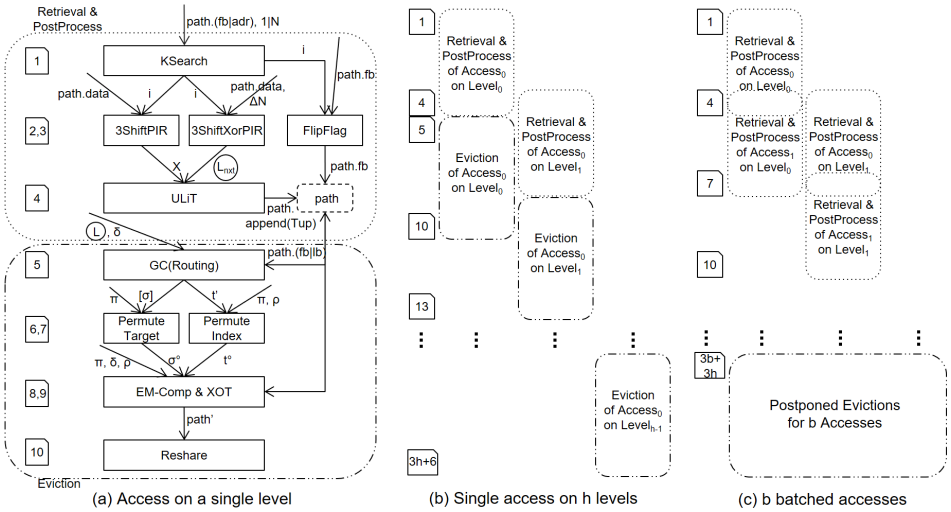

Fig. 10: Accesses and Rounds

In Figure 10 we show pictorially the dependencies between subprotocols of 3PC-ORAM.ML, both in single and in pipelined execution. This figure illustrates the points we make in the *Optimizations and Efficiency* paragraph at the end of Section 3.

First, using part (a) of the figure one can confirm which subprotocols of a single instance of 3PC-ORAM.ML execution can run in parallel, and which block requires as inputs the outputs produced by another. In particular note that protocol FlipFlag can run in parallel to 3ShiftPIR-Mod and 3ShiftXorPIR-Mod.

Secondly, in part (b) we show the parallelism in Retrieval+PostProcess(PP) and Eviction phases of h executions of 3PC-ORAM.ML, for i from 0 to $h-1$, which are part of a single 3PC-ORAM.Access memory access instance. In particular, this shows why a single execution of 3PC-ORAM.Access takes $3h + 6$ rounds.

Third, in part (c) we show how Retrieval+PP phases can be pipelined if protocol 3PC-ORAM.Access services a batch of b accesses with postponed Eviction. As we explain in the aforementioned paragraph in Section 3, the Retrieval+PP phases take 4 rounds, but the next execution of the Retrieval+PP block can start

after 3 rounds *both* for processing the same access on the next tree *and* for processing the next access on the same tree. The former is the case because protocol 3ShiftXorPIR-Mod returns the next-level label in round 3 of the Retrieval+PP phase. The latter is the case because protocol FlipFlag completes in round 3, at which point the retrieved path (and hence the whole tree) has proper sharings of fields fb and adr, and this is all that is needed for the KSearch protocol to start servicing the next access on the same tree. When protocol ULiT terminates in round 4, it does so in parallel to this KSearch instance (hence in Fig. 10 the last round of one Retrieval+PP execution coincides with the first round of the next execution of Retrieval+PP), at which point the data fields of all tuples in the path (and hence in the tree) are properly shared.

E.2 Additional Performance Data

Online Bandwidth. In Fig. 11-12 we compare online bandwidth between 2PC and 3PC Circuit ORAM. Because 2PC's online bandwidth can be estimated by counting the total bits of circuit input wire keys, we used this estimate to sketch 2PC's bandwidth graph so it can be compared with our 3PC's bandwidth for large $\log n$ and D. In Fig. 11, for small $D = 4B$, the bandwidth comparison between 2PC and 3PC is similar to the comparison on CPU time, because circuit input wire keys are the major components of bandwidth, just like circuit evaluation time is the major cost of CPU time. But if we fix $\log n$ and estimate bandwidth with respect to D, we see that as D grows, eventually bandwidth on D will dominate, and thus reflect the factor of κ for the 2PC/3PC bandwidth ratio because 2PC sends wire keys on D while 3PC does not. And this D turning point should be around/before $D = 1$KB according to Fig. 12.

Bandwidth and Comparison to Generic 3PC. In 13 we compare on-line bandwidth of three schemes, cust-3PC, gen-3PC, and C/S as a function of record size D (for $\log n = 30$). To make this comparison clearer Fig. 14 and 15 show the *ratios* between the bandwidth of our 3PC-ORAM protocol and the "base-line" bandwidth cost of client-server Path-ORAM, i.e. bndw(cust-3PC)/bndw(C/S), and for comparison the corresponding ratio for the generic 3PC Circuit-ORAM, i.e. bndw(gen-3PC)/bndw(C/S). Fig. 14 shows these ratios as a function of the address size $\log n$ (for $D = 4$B), and Fig. 15 shows them as a function of record size D (for $\log n = 30$). From Fig. 14 we see clearly that for small records our protocol is factor of 21x over the client-server Path-ORAM, about 2x worse than the optimal-bandwidth generic 3PC protocol (which, recall, has completely impractical round complexity), but in Fig. 15 we see that our bandwidth is only about 5x over client-server Path-ORAM, beating the generic 3PC protocol *even in exact bandwidth* for $D \geq 1$KB.

Low Bandwidth in Retrieval Phase. In Fig. 16 we show the ratio between our End-to-End bandwidth over our Retrieval bandwidth, which asymptotically differs by an $O(\log n)$ factor but in practice for $\log n = 30$ we see it grow from 88x for small D to 300x for $D = 10$KB. Secondly we show the ratio between our Retrieval bandwidth and the Ring-ORAM [24], which is a recent *client-server* Path-ORAM

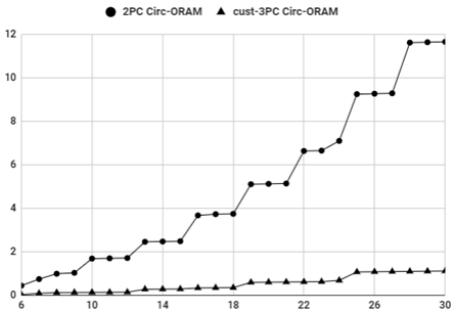

Fig. 11: Online Bandwidth (MB) vs $\log n$ (bits), for $D = 4$ bytes

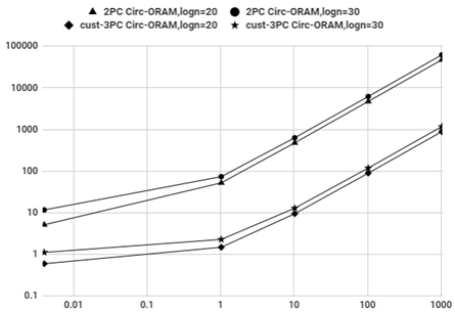

Fig. 12: Online Bandwidth (MB) vs D (KB)

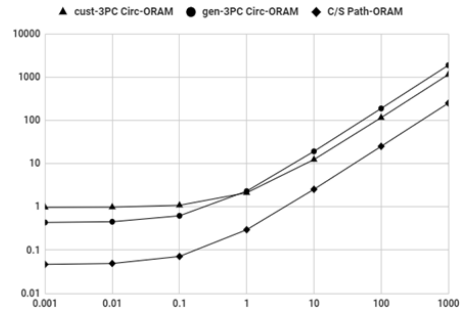

Fig. 13: Online bndw.(MB) vs D(KB) for $\log n$=30

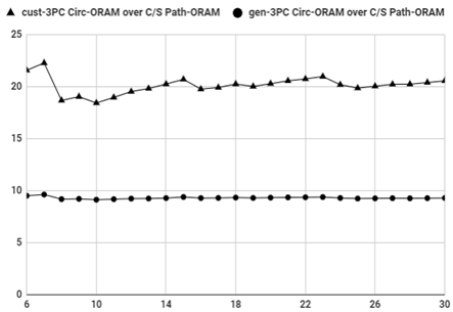

Fig. 14: Online bndw. *ratio* vs $\log n$ for D = 4B

variant with low-bandwidth Retrieval using symmetric key cryptography: We see that our bandwidth is only 4x to 3x over the cost of its client-server counterpart.

Online CPU Time. We compare online CPU time between 2PC and 3PC Circuit ORAM with respect to both record size D and address bit length $\log n$. We measure 2PC's online CPU time by only counting its garbled circuit evaluation time, while counting everything 3PC computed online (garbled circuit, link encryptions, etc.) as its online CPU time.

According to our measurements, when D is fixed to some small number like 4B, the online CPU time ratio between 2PC and 3PC grows as $\log n$ increases (shown in Fig. 7), and 3PC's online CPU time can be about 6 times smaller than 2PC for $\log n = 25$ mainly because 2PC has much larger circuit size. We also concluded that if fixing $\log n$ and comparing CPU time with respect to D, when D becomes larger, eventually D will out-weight other metadata fields in tuples and become the dominating factor in CPU time. For 3PC's CPU time, this turning point will be D between 1-10KB as the log-scale Fig. 17 shows. And based on our calculations, the upper-bound of CPU time ratio between 2PC and 3PC should be around $\frac{4\kappa}{13}$ when D is large, which is about 25 when $\kappa = 80$.

CPU Cost Components. In Fig. 18 we show the online CPU cost components of our 3PC-ORAM. As shown in the graph, when D is small, garbled circuit computation, which stays constant as D changes, dominates the CPU cost. And as D

grows, link encryption/decryption weights more and more as expected. However, what we also see is that there exists huge costs besides link encryptions and circuit computation when D becomes large. We found that the major part of these costs appears to be data handling cost, i.e. data structure conversions, reading/writing link bytes. Then it makes sense that these costs will also grow as D grows. To further improve the online CPU cost, we suppose more efficient languages than Java on this may show better performance.

τ Optimization. Parameter τ affects many things of the recursive tree structure ORAM schemes: number of trees, circuit sizes, bandwidth, message rounds, etc. Picking the best τ is not obvious, because it often involves trade-offs between different measurements (i.e. bandwidth vs circuit size), and different ORAM schemes may have different optimal τ values. To select the best τ for our 3PC-ORAM, we estimated our Online CPU cost as a function of τ. Based on the function curve, we found the estimated optimal τ is between 5 and 6. Though the optimal point is closer to 5 than 6 in the estimate, the actual measurements of our 3PC-ORAM shows that both $\tau = 5$ and 6 can be good choices in practice, where $\tau = 6$ actually has slight advantages on circuit size, CPU time, and Access WC time. We also estimated CPU cost of Circ-ORAM in terms of τ, and found that the optimal τ for Circ-ORAM is 3, which is the value used in Circ-ORAM paper. This verifies our estimates on τ is indeed correct.

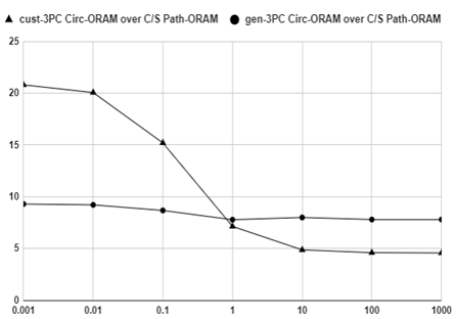

Fig. 15: Online bndw. *ratio* vs D for $\log n$=30

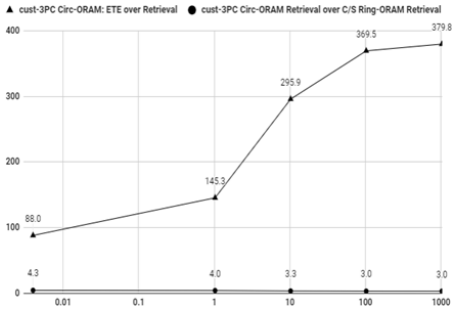

Fig. 16: Online bndw. *ratio* vs $D_{(\text{B})}$ for $\log n$=30

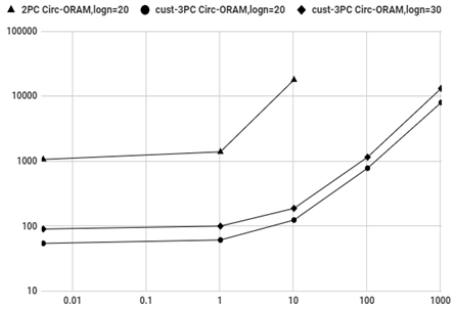

Fig. 17: Online CPU (ms) vs D (KB)

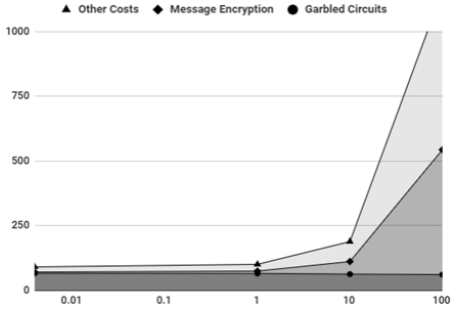

Fig. 18: Online CPU Time Components(ms) vs D(KB)