3PC-ORAM with Low Latency and Fast Batch Retrieval

Anonymous Author(s)

ABSTRACT

Multi-Party Computation of Oblivious RAM (MPC ORAM) allows n parties to access a secret-shared array of records while protecting privacy of both data and access pattern from $t < n$ colluding parties. MPC ORAM can be used to safe-keep any outsourced information, and it can implement secret-shared memory for general secure computation on large data [7, 16, 19], e.g. a database service with access (and data) privacy. Wang et al. showed *Circuit-ORAM* [24], a version of Path-ORAM of Stefanov et al. [23] with near-optimal Boolean circuit size. Using generic honest-but-curious 2PC or MPC solutions, this yields either a Yao-style 2PC ORAM which matches Path-ORAM round complexity but incurs $\Theta(\kappa)$ blow-up in bandwidth, where κ is the security parameter, or a BGW-style MPC (or GMW-style 2PC/MPC) which asymptotically matches Path-ORAM bandwidth but incurs factor of $\Omega(m \log m)$ increase in round complexity, where m is the logarithm of the array size.

We show a *customized* 3PC Circuit-ORAM protocol, where 3PC stands for MPC for $(t, n) = (1, 3)$, with a novel trade-off between bandwidth and round complexity: Our protocol uses $O(m)$ rounds and $O(m^3\kappa + mD)$ bandwidth, compared to $O(m)$ rounds and $O(m^3\kappa + mD\kappa)$ bandwidth of 2PC Circuit-ORAM [24] and $O(m^2 \log m)$ rounds and $O(m^3 + mD)$ bandwidth of *generic* 3PC Circuit-ORAM, where D is the record size. Concretely, our 3PC Circuit-ORAM has round complexity roughly 2 times, and bandwidth between 24 and 8 that of Path-ORAM, resp. for large and small D, whereas generic 3PC protocol of Araki et al. [1] applied to Circuit-ORAM [24] yields 3PC ORAM with bandwidth resp. between 12 and 8 over Path-ORAM,[1] but with round complexity larger by about two magnitudes.

Our 3PC ORAM also enables *fast batch retrieval*, retrieving a batch of B accesses in $O(\mathrm{b} + m)$ rounds with $O(m^3 + m^2\mathrm{b} + D)$ bandwidth per item. Concretely, for $m = 30$ and $\mathrm{b} = 100$, our retrieval bandwidth ranges from $\approx 20D$ per item for $D = 4$B and $\approx 6D$ for $D = 10$KB,[2] which is comparable to recent *client-server* ORAM's with fast batch processing [8, 10].[3]

1 INTRODUCTION

With the explosive growth in outsourced data it becomes increasingly important to be able to provide access to outsourced data while assuring privacy of both the data *and* access to it, and to do so even if some of the storage components are corrupted. Goldreich and Ostrovsky [14] addressed this by introducing an Oblivious RAM (ORAM), a protocol that allows for access to untrusted outsourced memory while keeping the access pattern secret. Multiple works, e.g. [15, 18, 21–23, 26], worked on ORAM in the client-server setting defined by [14], where the client has a master key which encrypts the outsourced data. However, this setting is limiting for several reasons: Access to the data is limited to a single entity holding the master key, access (and data) privacy is lost when the client is corrupted, and because all trust is based on this master key, the client-server ORAM cannot (by itself) implement shared memory for general secure computation protocol, for example generally-accessible database service which assures access privacy.

A stronger notion which can overcome all these limitations is an SC ORAM, a.k.a. MPC ORAM [7, 16, 20], i.e. an ORAM that can be used within general multi-party secure computation (MPC/SC), where n parties secret-share the stored data, and the protocol allows them to obliviously compute a secret sharing of a data record stored at location N given a secret-sharing of N. MPC ORAM allows n servers to obliviously emulate the RAM functionality, with guaranteed privacy if up to $t < n$ servers collude, and thus MPCORAM can implement RAM access for secure computation of any RAM program, enabling secure computation on large data.

2PC and MPC ORAM Protocols. Gordon et al. [16] proposed the first concrete 2PC ORAM, i.e. MPC ORAM for $n = 2$ and $t = 1$, based on the *Binary Tree* ORAM of Shi et al. [22], utilizing garbled circuits together with a specialized subprotocol for secure computation of a pseudorandom function (PRF) on secret-shared inputs. Gentry et al. [11] showed 2PC ORAM using a homomorphic encryption scheme instead of Yao's garbled circuits (and a modified version of the Binary Tree ORAM). Yao's garbled circuits can be replaced by OT-based circuit computation of GMW [13], which, given precomputation of OT extensions, can reduce both the computation and bandwidth at the cost of increasing round complexity. In a related approach, Keller and Scholl [17] showed a 2PC ORAM (with MPC precomputation) over the arithmetic circuit representation of a Binary Tree ORAM variant, which had low computation and bandwidth overhead but it required significant precomputation, and its round complexity was $\Omega(m^2 \log m)$, where m is the logarithm of the number of records. Wang et al. [25] showed a 2PC ORAM protocol which was based on garbled-circuit technique applied to a Boolean circuit representation of the ORAM client, using a heuristic version of the *Path-ORAM* scheme of Stefanov et al. [23], which reduced the circuit-size of the ORAM client algorithm. Subsequently, Wang et al. [24] presented a 2PC ORAM protocol called *Circuit-ORAM*, which further improved this approach: The client computation in Circuit-ORAM can be implemented with a Boolean circuit of size *linear* in the size of the *path* datastructure retrieved by Path-ORAM, i.e. $O(m^3 + mD)$, where D is the record size. In concrete terms the Boolean circuit for the Circuit-ORAM eviction algorithm has an impressively low overhead of ≤ 5 non-xor gates per bit in the path.

The garbled-circuit technique provides low-round 2PC computation but it incurs a factor of security parameter κ blow-up in bandwidth, so even if the ORAM circuit size is linear in path size, the resulting 2PC ORAM has a factor of $O(\kappa)$ larger bandwidth than Path-ORAM. This bandwidth increase can be avoided using GMW-style 2PC/MPC [13] or BGW-style MPC [2, 5], which give 2PC/MPC protocols with bandwidth which is linear in the circuit

[1]
[2]
[3]

Anonymous Submission to ACM CCS 2017, Dallas, Texas
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

size. To the best of our knowledge, the most bandwidth-efficient generic MPC protocol which utilizes inexpensive local computation, e.g. only symmetric key operations, is the protocol by Araki et al. [1] customized for the 3PC setting of $(t, n) = (1, 3)$, whose bandwidth is only 3 bits per non-xor gate in the Boolean circuit.[4] In concrete terms, applying the *generic* 3PC computation of [1] to Circuit-ORAM yields a 3PC ORAM protocol, with the factor of only between 7.5 and 12 increase over the Path-ORAM bandwidth,[5] resp. for large and small records. However, because *all* the bandwidth-efficient MPC protocols which work for general circuits and utilize only symmetric crypto have round complexity which is linear in the circuit *depth*, measured in layers of non-xor gates, the resulting generic 3PC Circuit-ORAM protocol has large round complexity. In asymptotic terms it is $\Omega(m^2 \log m)$, the depth of the Circuit-ORAM Boolean circuit, and in our estimates it needs > 1000 rounds for $m = 20$. Because of low computation and bandwidth complexity, the 3PC protocol of Araki et al. has excellent throughput [1], but it is not clear (at least for now) how this can be effectively utilized in the context of 3PC ORAM because it is not easy to parallelize access even to the client-server Path-ORAM, see e.g. [4], these difficulties would be compounded in the case of MPC ORAM, and in any event not every higher-level RAM program can be efficiently parallelized.

The only attempt we know of MPC ORAM which matches the $O(m)$ round complexity of Path-ORAM without incurring the bandwidth blow-up factor of $\Omega(\kappa)$ of the Yao-based 2PC/MPC ORAM, is a 3PC ORAM of Faber at al. [9]. The 3PC ORAM of [9] is based on a variant of Binary-Tree ORAM, and achieves bandwidth which is linear in the size of the underlying ORAM path for $D = \Omega(m^2\kappa)$, but their Binary-Tree ORAM uses $O(\lambda + m)$-sized buckets instead of $O(1)$ buckets of Path-ORAM, where λ is a statistical security parameter, resulting in the factor of $\Omega(\lambda)$ blow-up in bandwidth over Path-ORAM, assuming $\lambda \geq m$.

Contribution 1: Low-Latency 3PC ORAM. We show a *customized* 3PC Circuit-ORAM protocol, i.e. a customized 3PC computation of the Circuit-ORAM of Wang et al. [24], which uses $O(m)$ rounds and $O(m^3\kappa + mD)$ bandwidth. Table 1 summarizes asymptotic complexity comparisons of the proposed protocol versus prior work, but the most important aspect of these comparisons is that our costs asymptotically match those of (client-server) Path-ORAM in round complexity, but not in bandwidth, which is $O(m^3 + mD)$ for Path-ORAM. Still, our protocol offers a bandwidth-vs.-rounds trade-off compared to the *generic* 3PC Circuit-ORAM yielded by 3PC of [1] applied to Circuit-ORAM of [24], which has $\Omega(m^2 \log m)$ rounds and $O(m^3 + mD)$ bandwidth. Moreover, whereas the *decrease* in round complexity of our 3PC Circuit-ORAM compared to the generic 3PC Circuit-ORAM is dramatic in practice, the *increase* of bandwidth we incur is rather small.

Concretely, our protocol takes $4h + 5$ rounds[6] compared to $2h$ for Path-ORAM, where $h = O(m)$ is the number of Path-ORAM trees,[7] and our bandwidth is a factor between 8 and 24 over Path-ORAM,[8] respectively for large and small records.[9] While this bandwidth is larger than that of the generic 3PC Circuit-ORAM, it is larger by a factor between 1.1 (for 1KB records) and 2 (for 4B records),[10] while the round complexity reduction is from *im*practical to as practical as (client-server) Path-ORAM. This is confirmed by our prototype implementation in Java running on Amazon EC2 c4.2xlarge servers, which on a EC2 LAN takes about 40 milliseconds per access for $D = 4$ Bytes, and only 10% more for $D = 1$ KB, for $m = 30$.

	bandwidth	rounds
Path-ORAM [23]	$O(m^3 + mD)$	$O(m)$
2PC Circ.-ORAM [24]	$O(m^3\kappa + mD\kappa)$	$O(m)$
3PC ORAM of [9]	$O(m^3\lambda\kappa + mD\lambda)$	$O(m)$
3PC Circ.-ORAM [1, 24]	$O(m^3 + mD)$	$O(m^2 \log m)$
our 3PC Circ.-ORAM	$O(m^3\kappa + mD)$	$O(m)$

Figure 1: Bandwidth and round complexity comparisons

Contribution 2: Fast Batch Access Processing. Our 3PC Circuit-ORAM has an additional benefit of retrieving a batch of b accesses using only $4(\mathrm{b} + h)$ rounds and $O(m^2(m + \mathrm{b}) + D)$ bandwidth per item. After processing b accesses the protocol will incur 5b) rounds and $O(\mathrm{b}(m^3 + mD))$ of "clean up" costs, but these can be postponed, for any parameter b that benefits the higher-level RAM application. This fast batch processing comes from the three-phase structure of Path-ORAM: The *Retrieval* phase identifies the searched-for tuple on the retrieved tree path, and either retrieves the handle, called a *label*, for processing the next-level tree (in the case of a non-final Path-ORAM tree level) or the searched-for record. Second, a *Post-Processing* phase removes the retrieved tuple from the path, adds new random labels to it and re-inserts it in the root bucket (a.k.a. the *stash*). Third, an *Eviction* phase performs the eviction along the retrieved path (including the stash).

Recall that Path-ORAM Retrieval has to be processed sequentially across h ORAM trees, but Post-Processing and Eviction can be performed in parallel on all trees. Our 3PC-Circuit-ORAM processes Retrieval using 4 rounds and $O(m^2)$ for non-final tree levels and $O(m^2 + D)$ on the final level. Consequently, the Retrieval stage, from receiving the secret-shared input N to returning the (secret-shared) record stored at N (and/or writing into that record) takes $4h = O(m)$ rounds and $O(m^3 + D)$ bandwidth, while the Post-Processing and Eviction stage, which requires 6 rounds and the bulkload $O(m^3\kappa + mD)$ of bandwidth, can be postponed till after the ORAM response computation.[11]

This processing structure allows also for efficient *pipelining* of processing a batch of accesses in a way that minimizes the system response time. Note that process A which services an ORAM access on the i-th ORAM tree can trigger *both* process B which

[4] The protocol of [1] follows a longer line of works, notably [3], which can be seen as constant-factor improvements over BGW [2] customized to the $(t, n) = (1, 3)$ setting, or over GMW [13], which using OT with correlated randomness, implied in this setting, already yields 3PC with 6 bits off-line and 6 bits on-line per non-xor gate.

[5]

[6]

[7] In practice we observe optimal bandwidth for $h = m/6$ for m between 20 and 30, unless the record size is very large in which case h can be slightly smaller.

[8]

[9] Moreover, in a round-complexity-optimized protocol variant we get $3h + 5$ rounds, at an insignificant overall increase in bandwidth, but a three times increase in the *Retrieval* phase, which affects batch processing, see Contribution 2.

[10]

[11] Even though the Retrieval has much lower bandwidth than Eviction, its higher round complexity affects the overall latency: In our prototype testing on EC2 LAN the retrieval takes roughly 25% of the total latency, e.g. 10 milliseconds for $m = 30$.

services the same access on the $(i{+}1)$-st ORAM tree *and* process C which services the next ORAM access on the i-th tree. Process B can start when the Retrieval phase of process A returns the next-tree label, while process C needs process A to complete also the Post-Processing phase, which makes i-th ORAM tree ready for processing the next access, with the stash extended by the modified retrieved tuple. However, our 3PC ORAM can piggyback the Post-Process phase in process A with the Retrieval in A and C in a way that makes each process still take only 4 rounds.[12] Consequently, our 3PC ORAM responds to b accesses, for any b, using $4(h + \mathsf{b} - 1)$ rounds and $O(\mathsf{b}(m^2(m + \mathsf{b}) + D))$ bandwidth, and then "cleans up" by processing the Eviction phase of these b accesses, using $\mathsf{b}5 + 1$ rounds and $O(\mathsf{b}(m^3 + mD))$ bandwidth. In concrete terms, the bandwidth used to respond to $\mathsf{b} < 4m$ access requests is, per item roughly 20 times the record size for small records like 4B, decreasing to roughy 7 times for 10KB records.[13] As several recent works argue in the context of Client-Server ORAM [8, 10],[14] lowering the latency of system response in processing a batch of memory accesses improves performance of systems built on top of an ORAM. To give a simple example, it is useful to deliver quickly the first 20 responses to a database query, and perform the (more expensive) datastructure cleaning when the human user (or the higher-level application) processes these responses.

Roadmap. We overview our 3PC-ORAM protocol construction in Section 2, and after explaining our notation in Section 3, we present our 3P-Circuit-ORAM protocol in Section 4. In Section 5 we discuss comparisons between implementation choices and describe testing results of our prototype implementation.

2 PROTOCOL OVERVIEW

Note on Notation. Throughout the paper we use notation x^{P} to denote that variable x is held only by party P, $x^{\mathsf{P}_1\mathsf{P}_2}$ if x is held by P_1 and P_2, and $x^{\mathsf{P}_1\mathsf{P}_2\mathsf{P}_3}$ if x is held by P_1, P_2, and P_3.

Overview of Path ORAM of [23]. Our 3PC Circuit-ORAM is a 3PC secure computation of Circuit-ORAM of [24], which is a variant of Path-ORAM of Shi et al. [23]. Hence we will start recalling Path-ORAM of [23], casing it in terms which will be convenient for our 3PC protocol. Let M be an array of 2^m records of size D each. Server S keeps a binary tree datastructure, denoted tree, where each node is a bucket of a small constant size w, except the root bucket which is a bucket of size $s = O(m)$.[15] (This root bucket plays the role of the *stash* in [23] and can be cashed on the Client C, but it simplifies our terminology to equate it with the tree root.) The tree buckets contain *tuples*, which are records with four fields, fb, lb, adr, and data, and each entry M[N] in array M is stored by unique tuple Tup s.t. $\mathsf{Tup.(fb, lb, adr, data)} = (1, \mathsf{L}, \mathsf{N}, \mathsf{M[N]})$ where fb is a bit indicating full/empty tuple status, here assigned to 1, and L is a label identifying the tree leaf assigned at random to address N.

Datastructure tree satisfies an invariant that a tuple with label L resides in some bucket on the path from the root to leaf L, denoted tree.path(L). To access address N, C uses a (recursive) *position map* $\mathsf{N} \rightarrow \mathsf{L}$ (see below) to find leaf L corresponding to N, sends L to S to retrieve path = tree.path(L), finds tuple $\mathsf{Tup} = (1, \mathsf{L}, \mathsf{N}, \mathsf{M[N]})$ with the address field matching N in path, assigns new random leaf nL to N, flips bit fb in the existing Tup copy in path to 0 to erase that entry, and adds a modified tuple $(1, \mathsf{nL}, \mathsf{N}, \mathsf{M[N]})$ to the root bucket in path (in case of *write* access also replaces M[N] with a new record). To avoid overflow, C evicts all tuples in path as far down as possible without breaking the invariant above (and without overflowing any bucket). For example, if tree has depth 3 and the retrieved path is path(111) then tuples with label 0★★ must stay at the root, those with label 10★ can go to depth 1, those with label 110 can go to depth 2, and those with label 111 can go to depth 3. After eviction, C sends the modified path back to S.

Algorithm 1 ORAM-Access: Client/Server Path ORAM

Param: Address size m, chunk size τ, number of levels $h = \frac{m}{\tau} + 1$

Input: $\mathsf{oram^S} = (\mathsf{tree}_0, ..., \mathsf{tree}_{h-1})$
$\mathsf{N^C} = (\mathsf{N}_1, ..., \mathsf{N}_{h-1})$, $|\mathsf{N}_i| = \tau$ for all i (** and n-rec$^{\mathsf{C}}$)

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

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

For $i = 0$ **to** $h{-}1$ **do:**

ORAM-MainLoop: (for $i = 0$ see footnote "!" in Alg. 2)

$\mathsf{L}_i^{\mathsf{CS}}, \mathsf{tree}_i^{\mathsf{S}}, (\ [\mathsf{N}_0|...|\mathsf{N}_i], \mathsf{N}_{i+1}, \mathsf{nL}_i, \mathsf{nL}_{i+1}, {}^{**}\text{n-rec}\ )^{\mathsf{C}}$
$\longrightarrow\ \mathsf{L}_{i+1}^{\mathsf{CS}}$ (* or rec$^{\mathsf{C}}$), $\mathsf{tree}_i^{\mathsf{S}}$

* On *read* for top-level ORAM tree, ** On *write* for top-level ORAM tree

Position map $\mathsf{N} \rightarrow \mathsf{L}$ is stored using the same datastructure, with one tuple storing leaf values corresponding to a batch of 2^τ consecutive addresses, for constant τ. Since such position map has only $2^m/2^\tau = 2^{m-\tau}$ entries, continuing this recursion results in $h = m/\tau + 1$ trees $\mathsf{tree}_0, , .., \mathsf{tree}_{h-1}$: If N is divided into τ-bit blocks $\mathsf{N}_1, ..., \mathsf{N}_{h-1}$ then tree_i for $i \le h{-}2$ is a binary tree of depth $d_i = i\tau$ which implements a position map that matches address prefix $\mathsf{N}_{[1,...,i+1]}$ with leaf L_{i+1} assigned to it in tree_{i+1}. The last tree, tree_{h-1}, is the top-level tree of depth $d_{h-1} = (h{-}1)\tau = m$ containing the records of M. Algorithm ORAM-Access, Alg 1, traverses this datastructure by sequentially executing the main access loop, ORAM-MainLoop, shown in Alg 2: For $i = 0$ it retrieves L_1 associated with N_1 from tree_0 (this first look-up is simplified because tree_0 is a single tuple holding 2^τ labels corresponding to τ-bit prefixes N_1), for $i = 1$ it uses L_1 to retrieve L_2 associated with $\mathsf{N}_1|\mathsf{N}_2$ from $\mathsf{tree}_1.\mathsf{path}(\mathsf{L}_1)$, for $i = 2$ it uses L_2 to retrieve L_3 associated with $\mathsf{N}_1|\mathsf{N}_2|\mathsf{N}_3$ from $\mathsf{tree}_2.\mathsf{path}(\mathsf{L}_2)$, etc, thus traversing all tree levels until it uses L_{h-1} to find M[N] in $\mathsf{tree}_{h-1}.\mathsf{path}(\mathsf{L}_{h-1})$.

In addition to retrieving M[N] these calls to ORAM-MainLoop must also (i) replace retrieved labels $\mathsf{L}_1, ..., \mathsf{L}_{h-1}$ with fresh random labels $\mathsf{nL}_1, ...\mathsf{nL}_{h-1}$ (step 3), (ii) erase found tuples and place their modified versions at the root (steps 4-5), and (iii) evict the retrieved paths, by first computing the eviction map EM using algorithm PathORAM-Routing (step 6) and then applying it to permute the path elements (step 7).

Generic vs. Non-Generic MPC ORAM Approaches. Existing approaches to MPC ORAM typically choose a generic MPC protocol while modifying the underlying ORAM so that to maintain its performance while minimizing the description of the ORAM

[12]
[13]
[14]
[15]

Algorithm 2 ORAM-MainLoop: Client/Server Path ORAM

Input: L^{CS}, $tree^{S}$, (N, ΔN, nL, nL_{nxt}, **n-rec)^{C}
Output: (1) L^{CS}_{nxt} where L_{nxt} = Tup.data[ΔN] for Tup on tree.path(L) s.t. Tup.(fb|adr) = 1|N (* or rec^{C} where rec = Tup.data)
(2) $tree.path(L)^{S}$ modified by eviction, with Tup.lb := nL and Tup.data[ΔN] := nL_{nxt} (** and Tup.data := n-rec)

S sends path = tree.path(L) to C, who computes the following:

Retrieval of Next Label/Record

1: Tup := retrieve(path.(fb|adr), 1|N) s.t. Tup.(fb|adr) = 1|N

2: L_{nxt} := Tup.data[ΔN] (*,**: replace with rec := Tup.data)

Post-Process

3: Tup.lb := nL, Tup.data[ΔN] := nL_{nxt} (** Tup.data := n-rec)

4: set fb := 0 at record in path where Tup was found in step 1

5: path := path.append-to-root(Tup)

Eviction

6: EM := PathORAM-Routing(L, path.(fb, lb))

7: path′ := DataMovement(EM, path)

C sends path′ to L_{nxt} who inserts it into tree as tree.path(L).

* On *read* for top-level ORAM tree, ** On *write* for top-level ORAM tree
† Partition the path as stash and the rest of the tree path
! In the case of $tree_0$, which is a single tuple, we run only steps 2-3 for Tup := $tree_0$

client in the form required by the chosen MPC protocol. With the exception of Keller and Scholl who use an arithmetic circuit representation of a Path-ORAM variant [17], other works focus on streamlining the Boolean circuit representation of Path-ORAM client [16, 24, 25], with the work of [24] achieving a circuit with remarkably low overhead of < 5 non-xor gates per bit in the path input on which the client computation is performed.[16] However, a generic 2PC computation of this circuit would either incur $O(\kappa)$ blowup in bandwidth, using Yao's garbled circuits, or it would have round complexity lower-bounded by the circuit depth, which in the case of Circuit-ORAM is $\Omega(m^2 \log m)$.[17] The most efficient generic MPC protocol which uses only inexpensive local computation (i.e. symmetric ciphers as opposed to e.g. homomorphic encryption, thus giving the most hope for a practical solution) is the 3PC of Araki et al. [1], which yields 3PC Circuit-ORAM with roughly $\approx 10 \cdot |\text{path}|$ bandwidth but $\Omega(m^2 \log m)$ round complexity.

In this work we show a 3PC ORAM protocol which departs from the generic 2PC/MPC approach, by implementing parts of the ORAM client algorithm using customized protocols, while still using generic secure computation techniques for some other parts. We choose investigate the honest-but-curious 3PC setting of $(t, n) = (1, 3)$ because in this setting we can get particularly efficient implementations of the customized protocols.[18] This "mixed" approach to 3PC ORAM was used before by Faber [9], and both our 3PC ORAM protocol and the one of [9] stem from an observation that variants of Binary Tree ORAM, including Path-ORAM and Circuit-ORAM, can be divided into the following phases:

(1) *Retrieval*, which given path retrieved the ORAM tree and address chunk N, locates the searched-for tuple $(1, L, N, \text{data})$ in path, and retrieves the next-level leaf in data (in case of a non-final tree) or data itself (in case of a final tree);
(2) *Post-Process*, which removes the retrieved tuple from the path, injects new random labels, and re-inserts it in the root (= stash).
(3) *Eviction*, which can be broken down to two sub-phases:
   (a) *Eviction Logic*: Given label-data $(\text{fb}, L, \cdot)$ of each tuple in path and the label which defines this path, compute permutation EM on path path that determines the eviction movement that should be applied to these tuples.
   (b) *Data Movement*: Permute tuples in path according to EM.

The 3PC ORAM of [9] shows how to compute the Retrieval, Post-Process, and the Data Movement stages with $O(1)$ rounds and $O(|\text{path}|)$ bandwidth, and moreover points out that if the Eviction Logic stage is computed using the generic secure computation of a circuit, this circuit computes only on *metadata* in the path, i.e. on the *label* fields in the tuples, and in particular it is independent of record size D. However, [9] show this for a version of Binary Tree ORAM where each bucket in the tree path contains $O(\lambda + m)$ tuples instead of $O(1)$ in Path-ORAM (or Circuit-ORAM), where λ is a statistical security parameter, which blows up the Path-ORAM costs by a factor of $\Omega(\max(m, \lambda))$.

Efficient 3PC for Circuit-ORAM Eviction. Our 3PC ORAM reuses the inexpensive Data Movement protocol of [9], but integrating it with the Eviction Logic computed by Circuit-ORAM poses a challenge. Circuit-ORAM encodes the eviction map EM, i.e. the permutation according to which the tuples in path are moved in the Data Movement phase, using two separate objects, a partial map Φ on d buckets in path, and an array t of d bucket indexes, s.t. if $\Phi(i) = \perp$ then all tuples in the i-th bucket stay put, otherwise a tuple at position $\text{t}[i]$ in the i-th bucket is moved to position $\text{t}[i']$ in the (i')-th bucket number for $i' = \Phi(i)$. In other words, (Φ, t) defines the eviction map $\text{EM}_{\Phi,\text{t}}$ on $Z_n \sim Z_d \times Z_w$ for $n = d \cdot w$ as follows (here we assume the root/stash size is w instead of s, and we refer to Section 4.3 for handling the $s > w$ issue):

$$\text{EM}_{\Phi,\text{t}}(i, j) = \begin{cases} (\Phi(i), \text{t}[\Phi(i)]) & \text{if } \Phi(i) \neq \perp \text{ and } j = \text{t}[i] \\ (i, j) & \text{otherwise} \end{cases}$$

However, the 3PC Data Movement protocol of [9] assumes that one of the three parties, e.g. P_1, gets (Φ, t) computed by the Eviction Logic circuit in the clear. This was secure for [9] because their eviction map is *regular* in the sense that the distribution of (Φ, t) is independent of the path content. By contrast, function Φ computed in Circuit-ORAM is non-regular (see Fig. 2), and releasing it in the clear would leak information on the location of tuples in the path, which is in turn correlated with the pattern of accesses to ORAM.

[16] Constant 5 upper-bounds # gates per bit in the *meta-data* fragments of path, i.e. fields fb, lb, adr, while for the payload data the constant is even lower, and close to 2.
[17] The low circuit complexity of Circuit-ORAM stems from computing data-movement map EM via 3 *linear scans* through path meta-data, and applying this map in a single top-down linear scan. Unfortunately, all these linear scans seem inherently sequential.

[18] Indeed, the 3PC setting has been shown to yield more efficient secure computation in a variety of setting, e.g. [1, 3].

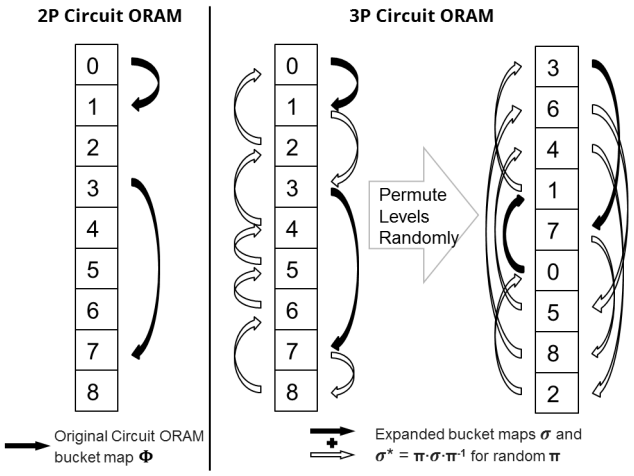

Figure 2: Randomization of Circuit ORAM's Bucket Map

Making Eviction Map Regular. Our task is therefore to transform the underlying eviction map Φ into a map whose distribution does not depend on the data. We do so in two steps. First, we "expand" function $\Phi : \mathsf{Z}_d \rightarrow \mathsf{Z}_d$ into a *cyclic* permutation σ on Z_d by adding *spurious edges* (using an additional fake empty tuple added to each bucket) to the transition map Φ (see Fig. 2). Second, we apply two types of masks to the resulting outputs (σ, t) of the modified Eviction Logic computation: a random permutation π on Z_d and two arrays δ, ρ of d random elements in Z_{w+1} (bucket size becomes w+1 because of adding fake tuples). the modified Eviction Logic will output $(\sigma^\circ, \mathrm{t}^\circ)$ where $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$ and $\mathrm{t}^\circ = \rho \oplus \pi(\mathrm{t} \oplus \delta)$. Array t° is random because so is array ρ, while σ° is random in C_d, a set of cyclic permutations over Z_d, because every cycle $\sigma^\circ \in C_d$ can be obtained from every other $\sigma \in C_d$ by node re-labeling $\pi \in \mathsf{perm}_d$, i.e. $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$. For each pair (σ, σ°) there are exactly d of π's, corresponding to d cycle rotations, which implies that $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$ is uniform in C_d given random π in perm_d.

Finally, it turns out that P_1 who gets the transformed eviction map defined by $(\sigma^\circ, \mathrm{t}^\circ)$ in the clear can effectively permute the secret-shared path P held by $\mathsf{P}_2, \mathsf{P}_3$ by the correct underlying eviction movement map $\mathsf{EM}_{\sigma,\mathrm{t}}$, using the Data Movement protocol of [9], a 3-party OT variant XOT. Protocol XOT allows P_1 who holds permutation Π to permute the xor-shared array x held by $\mathsf{P}_2, \mathsf{P}_3$ so that $\mathsf{P}_2, \mathsf{P}_3$ hold an xor-sharing of array $\Pi(x)$. Using XOT we implement our eviction movement as follows: First $\mathsf{P}_2, \mathsf{P}_3$ locally transform their secret-sharing of P using π, σ, and ρ, then the resulting secret-shared array is permuted by $\mathsf{EM}_{\sigma^\circ,\mathrm{t}^\circ}$ via protocol XOT, and finally that XOT output is locally post-processed by $\mathsf{P}_2, \mathsf{P}_3$ using π, σ, ρ. This composition of permutations results in a secret-sharing of $\mathsf{EM}_{\sigma,\mathrm{t}}$ applied to P. We discuss how and why this works in more details in Section 4.2.

Low-Bandwidth Retrieval using PIR. Whereas the above discussions refer to implementing 3PC for the Eviction phase of Circuit-ORAM with $O(1)$ rounds, the Eviction Logic circuit linear in the path metadata size, and the Data Movement linear in the path size, the second important cost benefits of our 3PC-Circuit-ORAM compared to the generic approaches are in the Retrieval (and Post-Process) phases. Note that the Retrieval phase implements a functionality which is similar to *Conditional Discovery of Secrets (CDS)* [12]. A CDS protocol can be seen as implementing the following functionality: Two parties hold a secret-sharing of an array of (keyword,value) pairs, and a secret-sharing of a searched-for keyword, and the protocol should output a value in the (keyword,value) pair whose keyword matches the searched-for keyword. This is exactly what is needed in the Retrieval protocol, where the searched-for keyword is the secret-shared address (or address prefix) N, and path is the secret-shared array of the (keyword,value) pairs where keywords are addresses and values are the tuples (or tuple payloads). (The actual Retrieval requires some adjustments because it also needs to perform an index-lookup on the retrieved payload corresponding to the next address chunk but it is a technicality.)

The CDS protocol of [12] or in [9] have $O(nD)$ bandwidth, where D is the size of the value/payload (e.g. taking the last-level ORAM tree as an example). In the 3PC context we can reduce this bandwidth to $O(n \log n + D)$, by implementing the CDS functionality as follows: First, we employ a protocol we call *Conditional Discovery of Index (CDI)*, to create a secret-sharing of an index i of a location of a the matching (keyword,value) pair in path. This can be in fact implemented with CDS directly (as we do here), with "payloads" of size $\log n + c$ for a constant c which controls only the probability of re-running the CDI protocol because of collisions. (Setting $c = 20$ gives the probability of 2^{-20} per access of having to re-run the CDI step.) The second step, is the retrieval of (secret-shared) payload held at the (secret-shared) location i in the (secret-shared) array path. This step can therefore be implemented with secret-sharing variants of PIR. In the 3PC setting we can use the information-theoretic 2-server PIR of Chor et al. [6], where all the inputs and outputs, i.e. the array path, the index i, and the retrieved record $x[i]$, are secret-shared. The 2-server PIR of [6] we use does not have the best known asymptotic performance as a Client-Server PIR, using $O((n \cdot D)^{1/2})$ bandwidth, but we use it for two reasons: First, because it is easy to convert to the secret-sharing variants where inputs and outputs are secret-shared. Second, because in our context D is most often larger than n, in which case the 2-server PIR of [6] already gives an optimal $O(D)$ bandwidth.
[19]

3 NOTATION

Protocol Parties, Shared Variables, Bitstrings, Secret-Sharing. We use C, D, E to denote the three parties participating in 3PC-ORAM. Each pair of parties $\mathsf{P}_1, \mathsf{P}_2$ is initialized with a shared seed to a Pseudorandom Generator (PRG), which allows them to generate any number of shared (pseudo)random objects. We will use $x^{\mathsf{P}_1\mathsf{P}_2} \xleftarrow{\$} S$ to denote P_1 and P_2 generating x at random from set S (or creating a random variable of type S) by using the PRG on this jointly held seed. We use $|x|$ to denote the length of variable x cast as a bitstring. We use sel_i^n to denote a bitstring of length n containing all zeroes except a single one at the i-th position.

Our 3PC-ORAM protocol uses four forms of secret-sharing, where $\langle \cdot \rangle_{2,3}$ is used for storing permanent data tree, stash, and the other sharings are used for temporary variables.

[19]

Algorithm 3 Types of Secret-Sharing

- $\langle x\rangle_{2,3}$ is a "2-out-of-3" sharing, $(x_1^{\mathsf{DE}}, x_2^{\mathsf{CE}}, x_3^{\mathsf{CD}})$, for random x_1, x_2, x_3 s.t. $x_1 \oplus x_2 \oplus x_3 = x$.
- $\langle x\rangle_{\mathsf{xor}}$ is a three-party xor-sharing, $(x_1^{\mathsf{C}}, x_2^{\mathsf{D}}, x_3^{\mathsf{E}})$, for random x_1, x_2, x_3 s.t. $x_1 \oplus x_2 \oplus x_3 = x$
- $\langle x\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$ is a two-party xor-sharing, $(x_1^{\mathsf{P}_1}, x_2^{\mathsf{P}_2})$, for random x_1, x_2 s.t. $x_1 \oplus x_2 = x$
- $\langle x\rangle_{2,1\text{-xor}}$: three instances of "2+1" sharing, $\langle x\rangle_{2,1\text{-xor}}^{\mathsf{CD}-\mathsf{E}}$, $\langle x\rangle_{2,1\text{-xor}}^{\mathsf{DE}-\mathsf{C}}$, $\langle x\rangle_{2,1\text{-xor}}^{\mathsf{EC}-\mathsf{D}}$, where $\langle x\rangle_{2,1\text{-xor}}^{\mathsf{P}_1\mathsf{P}_2-\mathsf{P}_3} = (x_{12}^{\mathsf{P}_1\mathsf{P}_2}, x_3^{\mathsf{P}_3})$ for random x_{12}, x_3 s.t. $x_{12} \oplus x_3 = x$
- $\langle x\rangle_{2,1\text{-shift}}$, for $x \in \mathsf{Z}_n$: three instances of "2+1 shift" sharing, $\langle x\rangle_{2,1\text{-shift}}^{\mathsf{CD}-\mathsf{E}}$, $\langle x\rangle_{2,1\text{-shift}}^{\mathsf{DE}-\mathsf{C}}$, $\langle x\rangle_{2,1\text{-shift}}^{\mathsf{EC}-\mathsf{D}}$, where $\langle x\rangle_{2,1\text{-shift}}^{\mathsf{P}_1\mathsf{P}_2-\mathsf{P}_3} = (x_{12}^{\mathsf{P}_1\mathsf{P}_2}, x_3^{\mathsf{P}_3})$ for random x_{12}, x_3 in Z_n s.t. $x_{12} + x_3 = x \bmod n$

Because all our sharings are xor-homomorphic, if two variables are shared, e.g. $\langle x\rangle_{2,3}, \langle y\rangle_{2,3}$, we will write e.g. $\langle x+y\rangle_{2,3}$ to denote the sharing of $x+y$ locally computed by all players by xor-ing their shares of x and y. Likewise if c is a constant known to all players then $\langle x+c\rangle_{2,3}$ is locally computed by designated player(s) xor-ing their share(s) of x by c. Some of our protocols need to transform one sharing to another, and we denote all locally computed re-sharings as Local-Transform, shown in Alg. 4, and all interactive re-sharing procedures as Reshare, shown in Alg. 5.

The interactive re-sharing protocols Reshare type (i), (ii), and (iii), see Alg. 5, form *fresh* sharings of the target type. (The Reshare of type (iv) is an exception, but its outputs are a deterministic function of its inputs.) By contrast, the non-interactive sharing transformations Local-Transform, Alg. 4, are deterministic, so they do not create fresh sharings of the desired type. However, each of these sharings can be refreshed if the parties locally generate a random *zero sharing* of the corresponding type, and the fresh sharing is generated by adding the old sharing with the zero sharing. There are two types of zero-sharing we will need: A two-party xor-sharing of an all-zero string, $\langle 0^n\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2} = (x_1^{\mathsf{P}_1}, x_2^{\mathsf{P}_2})$, can be created simply by sampling $r^{\mathsf{P}_1\mathsf{P}_2} \xleftarrow{\$} \{0,1\}^n$ and setting $(x_1, x_2) := (r, r)$. A three-party xor-sharing of an all-zero string, $\langle 0^n\rangle_{\mathsf{xor}} = (x_1^{\mathsf{C}}, x_2^{\mathsf{D}}, x_3^{\mathsf{E}})$, can be created by sampling $r_1^{\mathsf{DE}} \xleftarrow{\$} \{0,1\}^n$, $r_2^{\mathsf{CE}} \xleftarrow{\$} \{0,1\}^n$, and $r_3^{\mathsf{CD}} \xleftarrow{\$} \{0,1\}^n$, and setting $(x_1, x_2, x_3) := (r_2 \oplus r_3, r_1 \oplus r_3, r_1 \oplus r_2)$.

Algorithm 4 Local-Transform: Local Sharing Transformation

- $\langle x\rangle_{2,3} \longrightarrow \langle x\rangle_{\mathsf{xor}}$
  If $\langle x\rangle_{2,3} = (x_1^{\mathsf{DE}}, x_2^{\mathsf{CE}}, x_3^{\mathsf{CD}})$ then $\langle x\rangle_{\mathsf{xor}} = (x_2^{\mathsf{C}}, x_3^{\mathsf{D}}, x_1^{\mathsf{E}})$
- $\langle x\rangle_{2,3} \longrightarrow \langle x\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$
  If $\langle x\rangle_{2,3} = (x_1^{\mathsf{P}_1\mathsf{P}_2}, x_2^{\mathsf{P}_2\mathsf{P}_3}, x_3^{\mathsf{P}_3\mathsf{P}_1})$ then $\langle x\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2} = ((x_1 + x_3)^{\mathsf{P}_1}, x_2^{\mathsf{P}_2})$
- $\langle x\rangle_{2,3} \longrightarrow \langle x\rangle_{2,1\text{-xor}}$
  If $\langle x\rangle_{2,3} = (x_1^{\mathsf{DE}}, x_2^{\mathsf{CE}}, x_3^{\mathsf{CD}})$ then $\langle x\rangle_{2,1\text{-xor}}^{\mathsf{CD}-\mathsf{E}} = (z_{12}^{\mathsf{CD}}, z_3^{\mathsf{E}})$ e.g. for $(z_{12}, z_3) := (x_3, x_1 \oplus x_2)$. Likewise for $\langle x\rangle_{2,1\text{-xor}}^{\mathsf{DE}-\mathsf{C}}$ and $\langle x\rangle_{2,1\text{-xor}}^{\mathsf{EC}-\mathsf{D}}$

Integer Ranges, Permutations. We use Z_n to denote set $\{0, ..., n-1\}$, and perm_n to denote the set of permutations on Z_n.

Algorithm 5 Reshare: Interactive Sharing Transformation

(i) $\langle x\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2} = (a_1^{\mathsf{P}_1}, a_2^{\mathsf{P}_2}) \longrightarrow \langle x\rangle_{2,3} = (b_1^{\mathsf{P}_2\mathsf{P}_3}, b_2^{\mathsf{P}_1\mathsf{P}_3}, b_3^{\mathsf{P}_1\mathsf{P}_2})$
Pick $b_1^{\mathsf{P}_2\mathsf{P}_3}, b_2^{\mathsf{P}_1\mathsf{P}_3} \xleftarrow{\$} \{0,1\}^{|x|}$. P_1 and P_2 exchange $(a_1 \oplus b_2)$ and $(a_2 \oplus b_1)$ and set $b_3 := (a_1 \oplus b_2) \oplus (a_2 \oplus b_1)$.

(ii) $\langle x\rangle_{\mathsf{xor}} = (x_1^{\mathsf{C}}, x_2^{\mathsf{D}}, x_3^{\mathsf{E}}) \longrightarrow \langle x\rangle_{2,3} = (y_1^{\mathsf{CD}}, y_2^{\mathsf{DE}}, y_3^{\mathsf{CE}})$
Generate a random n-bit zero-sharing (see text above), $(s_1^{\mathsf{C}}, s_2^{\mathsf{D}}, s_3^{\mathsf{E}}) = \langle 0^n\rangle_{\mathsf{xor}}$, and let C send $y_1 = x_1 \oplus s_1$ to D, D send $y_2 = x_2 \oplus s_2$ to E, and E send $y_3 = x_3 \oplus s_3$ to C.

(iii) $\langle x\rangle_{2,1\text{-shift}}^{\mathsf{DE}-\mathsf{C}} = (a_{23}^{\mathsf{DE}}, a_1^{\mathsf{C}}) \rightarrow \langle x\rangle_{2,1\text{-shift}}$
We get $\langle x\rangle_{2,1\text{-shift}}^{\mathsf{EC}-\mathsf{D}} = (b_{13}^{\mathsf{CE}}, b_2^{\mathsf{D}})$ for $x \in \mathsf{Z}_n$, if $b_{13}^{\mathsf{CE}} \xleftarrow{\$} \mathsf{Z}_n$, C sends $\delta = a_1 - b_{13}$ to D, D sets $b_2 := a_{23} + \delta$ (all mod n). $\langle x\rangle_{2,1\text{-shift}}^{\mathsf{CD}-\mathsf{E}}$ is computed likewise.

(iv) $\langle x\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2} = (x_1^{\mathsf{P}_1}, x_2^{\mathsf{P}_2})$, $z_1^{\mathsf{P}_1} \rightarrow z_2^{\mathsf{P}_2}$ s.t. $(z_1^{\mathsf{P}_1}, z_2^{\mathsf{P}_2}) = \langle x\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$
P_1 sends $\Delta = x_1 \oplus z_1$ to P_2 and P_2 sets $z_2 := x_2 \oplus \Delta$.

Bandwidth: (i): $2|x|$, (ii): $3|x|$, (iii): $2|x|$, (iv): $|x|$; Rounds: 1 (for each protocol); Security: (i): Message $a_2 \oplus b_1$ received by P_1 can be computed from P_1's input and output as $a_1 \oplus b_2 \oplus b_3$, likewise for P_2; (ii) Sharing (y_1, y_2, y_3) is fresh by security of zero-sharing, and each party receives only its output; (iii) Sharing (b_{13}, b_2) is fresh, and value δ received by D can be computed from D's input and output; (iv) P_2 can compute message Δ from its input x_2 and output z_2..

For $\pi, \sigma \in \mathsf{perm}_n$, we use $\pi \cdot \sigma$ for denote permutation composition, i.e. $(\pi \cdot \sigma)(i) = \pi(\sigma(i))$, and π^{-1} for an inverse permutation of π.

Arrays. We use array$[n]$ to denote arrays of n elements, and array$[n](w)$ if the n elements are all w-bit strings. For $x \in$ array$[n]$ and $i \in \mathsf{Z}_n$ we use either x_i or $x[i]$ to denote the i-th item in array x. An array x of type array$[n](w)$ can be typecast as a bit-string in $\{0,1\}^{nw}$. Moreover, a bit-string $x \in \{0,1\}^{nw}$ can be typecast as an array in array$[n](w)$, in which case x_i (or $x[i]$) denotes the i-th chunk of w bits in x. For example, if $x, y \in \{0,1\}^{nw}$ are cast as arrays in array$[n](w)$ then $z = x \oplus y$ denotes an array in array$[n](w)$ where $z_i = x_i \oplus y_i$ for all $i \in \mathsf{Z}_n$. We denote 2-dimensional n by m arrays as array$[n, m]$, and if $x \in$ array$[n, m]$ then $x[i] = x_i$ is the i-th row of x, and $x[i, j] = x_{i,j} = (x_i)_j$ is an element in the i-th row and j-th column in x. If $x, y \in$ array$[n, m]$ then $z = x \oplus y$ means $z_{i,j} = x_{i,j} \oplus y_{i,j}$ for $(i, j) \in \mathsf{Z}_n \times \mathsf{Z}_m$.

Array Operations. We use shortcuts for the following array operations: If $a \in$ array$[n]$ and $\Delta \in \mathsf{Z}_n$ then $a_{\mathsf{shift}[\Delta]}$ denotes array b s.t. $b[i] = a[i + \Delta \bmod n]$ for $i \in \mathsf{Z}_n$. If $p \in \{0,1\}k$ then rot$[p]$ is a permutation in perm_n for $n = 2^k$ s.t. rot$[p](i) = p \oplus i$ for all $i \in \mathsf{Z}_n$. Furthermore, if $a \in$ array$[n]$ for $n = 2^k$ then $a_{\mathsf{rot}[p]}$ denotes array b which is a permuted by rot$[p]$, i.e. $b[i \oplus p] = a[i]$ for $i \in \mathsf{Z}_n$. Finally, if $a \in$ array$[n](\ell)$, $p \in \{0,1\}\ell$, and $t \in \mathsf{Z}_n$, then $b = a^{\mathsf{xor}[p\ @\ t]}$ denotes array b s.t. $b[t] = a[t] \oplus p$ and $b[i] = a[i]$ for $i \neq t$.

Permutations vs. Arrays. For $\sigma \in \mathsf{perm}_n$ we use $[\sigma]$ to denote a representation of σ as an array $b \in$ array$[n](\log n)$ s.t. $b[i] = \sigma(i)$ for $i \in \mathsf{Z}_n$, i.e. $[\sigma] = [\sigma(0), ..., \sigma(n-1)]$. Conversely, if $x \in$ array$[n](\log n)$ satisfies that $x_i \in \mathsf{Z}_n$ and $(x_i = x_j) \Rightarrow (i = j)$ for all $i, j \in \mathsf{Z}_n$ then we can cast x as a permutation denoted $\hat{x} \in \mathsf{perm}_n$ where $\hat{x}(i) = x[i]$ (hence $[\hat{x}] = x$). For $\pi \in \mathsf{perm}_n$ and $x \in$ array$[n]$ we use $\pi(x)$ to denote an array containing elements of x permuted according to π. Note that $\pi(x) = [x_{\pi^{-1}(0)}, ..., x_{\pi^{-1}(n-1)}]$, because π moves the i-th element of x to the $\pi(i)$-th position in $\pi(x)$, i.e. $(\pi(x))_{\pi(i)} = x_i$, or, equivalently, $(\pi(x))_i = x_{\pi^{-1}(i)}$.

Note that if $\sigma \in \text{perm}_n$ and we permute an array representation $[\sigma]$ of σ according to another permutation $\pi \in \text{perm}_n$, then the resulting array $[\rho] = \pi([\sigma])$ encodes permutation $\rho = \sigma \cdot \pi^{-1}$, because $\rho(i) = [\rho]_i = [\sigma]_{\pi^{-1}(i)} = \sigma(\pi^{-1}(i)) = (\sigma \cdot \pi^{-1})(i)$.

Garbled Circuit Wire Keys. Since we use Yao's garbled circuit technique in some of our sub-protocols we will have circuit inputs and outputs represented as garbled wire keys: If variable $x \in \{0,1\}^w$ is either an input or an output in circuit C then we will use $\overline{x} \in \text{array}[w](\kappa)$ to denote the set of circuit wire keys corresponding to value x. For an array $x \in \text{array}[n](w)$, we use $\overline{x}_i$ to denote the subset of w keys associated with element x_i in x.

4 3PC-ORAM PROTOCOL

Our *3PC* Circuit-ORAM protocol follows the same top-level structure as ORAM-Access, Alg. 1, except that all the inputs are shared using the 2-out-of-3 secret-sharing, including the oram datastructure oram = $(\text{tree}_0, ..., \text{tree}_{h-1})$, the address $\text{N} = (\text{N}_1, ..., \text{N}_{h-1})$ (and record n-rec in case of *write* access). The off-line chosen sequence of new random labels $(\text{nL}_1, ...\text{nL}_{h-1})$ will also be shared in the same way. (Note that random 2-out-of-3 sharing can be created non-interactively using the pairwise-shared PRG's, see Section 3.) The 3PC access algorithm will run the same loop as ORAM-Access on these 2-out-3 shared inputs, except that it will replace the client-server main access loop processing algorithm ORAM-MainLoop, Alg. 2, with its 3PC counterpart, protocol 3PC-ORAM-MainLoop, shown in Alg. 6. Protocol 3PC-ORAM-MainLoop, given the current-level leaf label L known to all parties, the sharing of the current-level tree, of the address prefix N, of the next-level address chumk ΔN, and of the new current-level and next-level leaf labels, resp. nL and nL_{nxt}, (and new record payload n-rec if $i = h-1$ and access=*write*) performs the 3PC implementation of the three phases of access: **(1)** In the *Retrieval* phase, it uses protocols CDI and 3ShiftPIR to retrieve from $\langle \text{tree.path(L)} \rangle_{2,3}$ the secret-sharing of payload X of a tuple with the searched-for the secret-shared address N, and it uses protocol 3ShiftXorPIR to publicly reconstruct the next-level label L_{nxt} (unless it is the top-level tree) from X and the secret-shared next-level address chunk ΔN; **(2)** In the *Post-Process* phase, (2a) it uses protocol PPT to form a secret-shared new tuple Tup using the secret-shared retrieved payload X, address N, and new labels nL, nL_{nxt}, (2b) it uses protocol FlipFlag to flip the free bit to 0 in the old version of this tuple found in tree.path(L), via sub-protocol FlipFlag, and (2c) it inserts Tup into the root, which each party does non-interactively on shares; **(3)** In the *Eviction* phase,[20] it uses protocols GC,PermuteTarget,PermuteIndex,XOT-m,Reshare to perform the eviction on path tree.path(L). [21] Below we discuss the protocol in more details, the Retrieval and Post-Process phase in Section 4.1 and Eviction phase in Section 4.2.

[20] pre-processing the root buckets of every tree tree_i using protocol UpdateRoot, to deal with root buckets having larger size

[21] Note that following the overflow analysis of Circuit-ORAM [24] we will do two evictions per each record access. However, because we will not access another tuple on the second path, we do not need to run protocols in **(1)**, and the second eviction will be accomplished by just protocols in **(2)** and **(3)**.

Algorithm 6 3PC-ORAM-MainLoop: 3PC Circuit ORAM

Input: L^{CDE}, $\langle \text{tree, N, ΔN, nL, nL}_{\text{nxt}}, {}^{**}\text{n-rec} \rangle_{2,3}$

Output: (1) $\text{L}^{\text{CDE}}_{\text{nxt}}$ where L_{nxt} = Tup.data[ΔN] for Tup on tree.path(L) s.t. Tup.(fb|adr) = 1|N, (* or $\langle \text{rec} \rangle_{2,3} := \langle \text{Tup.data} \rangle_{2,3}$)
(2) $\langle \text{tree.path(L)} \rangle_{2,3}$ modified by eviction, with Tup.lb := nL and Tup.data[ΔN] := nL_{nxt} (** and Tup.data := n-rec)

Retrieval of Next Label/Record

1: $\langle \text{path} \rangle_{2,3} := \langle \text{tree.path(L)} \rangle_{2,3}$

2: CDI: $\langle \text{path.(fb|adr)} \rangle^{\text{D-E}}_{\text{xor}} \langle 1|\text{N} \rangle^{\text{D-E}}_{\text{xor}} \rightarrow \langle i \rangle_{2,1\text{-shift}}$
where path[i].(fb|adr) = 1|N

3: 3ShiftXorPIR: $\langle \text{path.data} \rangle_{2,3}, \langle i \rangle_{2,1\text{-shift}}, \langle \Delta\text{N} \rangle_{2,1\text{-xor}} \rightarrow \text{L}^{\text{CDE}}_{\text{nxt}}$
where L_{nxt} = path[i].data[ΔN] (*, **: skip)

4: 3ShiftPIR: $\langle \text{path.data} \rangle_{2,3}, \langle i \rangle_{2,1\text{-shift}} \rightarrow \langle X \rangle_{2,3}$
where X = path[i].data (*,**: $\langle \text{rec} \rangle_{2,3} := \langle X \rangle_{2,3}$)

Post-Process

5: PPT: $\langle \text{N, nL}, X, \Delta\text{N, nL}_{\text{nxt}} \rangle_{2,3}, \text{L}^{\text{CDE}}_{\text{nxt}} \rightarrow \langle \text{Tup} \rangle_{2,3}$ (*,**: skip)
Tup := (1, N, nL, X), X[ΔN] := nL_{nxt}, (* X stays; ** X := n-rec)

6: FlipFlag: $\langle \text{path.fb} \rangle_{2,3}, \langle i \rangle^{\text{DE-C}}_{2,1\text{-shift}} \rightarrow \langle \text{path.fb} \rangle_{2,3}$
where path.fb[i] is set to 0

7: $\langle \text{path} \rangle_{2,3} := \langle \text{path.append-to-root(Tup)} \rangle_{2,3}$

Eviction

8: GC(Routing): $(\text{L}, \delta)^{\text{CE}}, \langle \text{path.(fb, lb)} \rangle^{\text{C-E}}_{\text{xor}} \rightarrow (\overline{[\sigma]}, \text{t}')^{\text{D}}$
where $\text{t}' = \text{t} \oplus \delta$, for $\delta^{\text{CE}} \xleftarrow{\$} \text{array}[d](\log(w+1))$
and σ, t are expanded outputs of CircORAM-Routing

9: PermuteTarget: $\overline{[\sigma]}^{\text{D}}, \pi^{\text{CE}} \rightarrow [\sigma^\circ]^{\text{D}}$
where $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$ for $\pi^{\text{CE}} \xleftarrow{\$} \text{perm}_d$

10: PermuteIndex: $\text{t}'^{\text{D}}, (\pi, \rho)^{\text{CE}} \rightarrow \text{t}^{\circ\text{D}}$
where $\text{t}^\circ = \rho \oplus \pi(\text{t}')$ for $\rho^{\text{C,E}} \xleftarrow{\$} \text{array}[d](\log(w+1))$

11: $\text{EM}^{\circ\text{D}} := \text{EM-Comp}(\sigma^\circ, \text{t}^\circ)$

12: XOT-m: $\text{EM}^{\circ\text{D}}, \langle \text{path} \rangle^{\text{C-E}}_{\text{xor}}, (\pi, \delta, \rho)^{\text{CE}} \rightarrow \langle \text{path}' \rangle^{\text{C-E}}_{\text{xor}}$
where path′ = $\Pi_{\text{out}}(\text{EM}^\circ(\Pi_{\text{in}}(\text{path})))$ [= $\text{EM}_{\sigma,\text{t}}(\text{path})$]
for $\Pi_{\text{in}} = \tilde{\rho} \cdot \ddot{\pi} \cdot \tilde{\delta}$ and $\Pi_{\text{out}} = \Pi^{-1}_{\text{in}}$

13: Reshare: $\langle \text{path}' \rangle^{\text{C-E}}_{\text{xor}} \rightarrow \langle \text{path}' \rangle_{2,3}$

14: $\langle \text{tree.path(L)} \rangle_{2,3} := \langle \text{path}' \rangle_{2,3}$

* On *read* for top-level ORAM tree, ** On *write* for top-level ORAM tree

4.1 Retrieval and Post-Process Protocols

The retrieval and post-process steps, from step 1 to 6 in protocol 3PC-ORAM-MainLoop, are handled as follows. We break up the Retrieval phase into subprotocol CDI, which given a 2,1-xor-sharing of path matadata, namely fields fb|N, and a 2,1-xor-sharing of the searched-for address chunk N, returns a 2,1-shift-sharing of the

index i of the tuple containing the matching address. Protocols shift-xor-PIR and shift-PIR used in the next two steps can be run in parallel, and they are implemented similarly: First, the single 2,1-shift-sharing of the index i is re-shared to form the two complementary 2,1-shift-sharings, so we have three copies of this sharing, $\langle i\rangle^{\mathsf{CD-E}}_{2,1\text{-shift}}$, $\langle i\rangle^{\mathsf{DE-C}}_{2,1\text{-shift}}$, and $\langle i\rangle^{\mathsf{EC-D}}_{2,1\text{-shift}}$. Next, both shift-xor-PIR and shift-PIR perform the corresponding variants of the 2-server PIR for each of these three sets of parties. This results in the 3-way xor-sharing of the values we need, which is the next-level label retrieved by shift-xor-PIR, which this protocol publicly reconstructs, and the retried tuple X, output by shift-PIR, which we need to keep secret-shared. The post-process protocols PPT and FlipFlag do the following: PPT injects the new labels nL, $\mathsf{nL}_{\mathsf{nxt}}$ into the secret-shared tuple X, and this modified secret-shared tuple is then appended to the secret-shared stash. Protocol FlipFlag can work in parallel to PPT because it operates on the freebit fields of the existing tuples in path, to flip the bit of the tuple indexed by the secret-shared index i from fresh to used. The bandwidth of these two post-process steps is respectively $O(|X|)$ and $2n$.

4.2 Eviction

The eviction protocol, which we will denote as Eviction, stands for the eviction stage of protocol 3PC-ORAM-MainLoop, i.e. steps 8-14 in Alg. 6. Protocol Eviction runs as follows.

In step 8, the parties securely compute the eviction circuit Routing given the xor-sharing of the metadata fields fb and L of the path array, with party D learning the output in a masked form. Circuit Routing is almost identical to the Circuit-ORAM eviction circuit CircORAM-Routing: It computes the Circuit-ORAM eviction map Φ, t, and then expands the bucket map Φ into σ (see Figure 2 in Section 2) and masks t by xor-ing it with a random array δ. In Alg. 6 this step is implemented using protocol GC, a variant of Garbled Circuit protocol where the inputs are xor-shared between two parties C, E, and instead of using OT's for transferring input wire keys, as is necessary in the 2PC setting, the input wire keys are xor-shared by C and E in such a way that each party can send its key-share to D based on its share of the input bit, and D reconstructs the input wire keys corresponding to the secret-shared input without any party learning the input. In Step 8 this Garbled Circuit algorithm allows D to compute output $\mathsf{t}' = \mathsf{t} \oplus \delta$ to D in the clear, but for output $[\sigma]$ party D computes only $\overline{[\sigma]}$, i.e. the output *wires* corresponding to the bits encoding $[\sigma]$. It is easy to modify a garbled circuit algorithm in this way, by simply omitting the decoding from wire keys to bits on the output wires corresponding to $[\sigma]$. Note that the translation between the wire values $\overline{y}$ and the bits y in the garbled circuit can be known to both C and E.

In steps 9-10, D with the help of C and E transforms $\overline{[\sigma]}$ and t' into the masked transition map $\sigma^\circ = \pi\sigma\pi^{-1}$ and the masked index array $\mathsf{t}^\circ = \rho \oplus \pi(\mathsf{t}')$. The masked transition map $\sigma^\circ = \pi\sigma\pi^{-1}$ is computed in step 9, via protocol PermuteTarget (Alg. 19 in Appendix A), which performs two things, given a random permutation π on the buckets sampled jointly by C and E: First, it translates between the set of wires $\overline{[\sigma]}$ and the array $[\sigma]$ itself, without revealing $[\sigma]$ to any party (recall that C and E hold the translation between wire keys and values they represent), and secondly it converts $[\sigma]$ to $[\sigma^\circ]$ given π known to C and E. The masked index array, $\mathsf{t}^\circ = \rho \oplus \pi(\mathsf{t}')$, is computed in step 10, via protocol PermuteIndex (Alg. 20 in Appendix A), given a random array ρ of the same size as t, sampled jointly by C and E.

Given the bucket map σ° and the index array t° party D locally computes the corresponding eviction map $\mathsf{EM}^\circ = \mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ}$, and in step 12 this map is used to apply the eviction movement defined by map $\mathsf{EM} = \mathsf{EM}_{\sigma,\mathsf{t}}$ to path. Note that, as introduced in the Overview Section, in order to make this masked transition map EM° complete and oblivious, we need to add a fake tuple to each bucket, so this EM° output by Routing is not a map on $d \times w$ tuples but instead $d \times (w+1)$. Then the eviction movement of EM° is done using protocol XOT−m, which is a wrapper over protocol XOT from [9] (see Alg. 22 in Appendix A), and which works as follows: First, C and E append trivial sharing of one fake tuple to each bucket on the path, and extrapolate masks δ, ρ and permutation π to form three permutations $\tilde{\delta}$, $\tilde{\rho}$, and $\ddot{\pi}$, all permutations over Z_n (we explain how this extrapolation below). The parties then run protocol XOT on D's input permutation EM° and C & E's inputs the xor-sharing of $\Pi_{\mathsf{in}}(\mathsf{path})$ where $\Pi_{\mathsf{in}} = \tilde{\rho}\cdot\ddot{\pi}\cdot\tilde{\delta}$. Note that C and E can form a sharing of $\Pi_{\mathsf{in}}(\mathsf{path})$ by locally applying Π_{in} to their shares of path. The functionality of XOT is to permute the xor-shared array held by C and E by a permutation held by D, and so C and E's output from this XOT instance is an xor-sharing of $\mathsf{EM}^\circ(\Pi_{\mathsf{in}}(\mathsf{path}))$. Lastly, C and E locally apply permutation $\Pi_{\mathsf{out}} = \Pi_{\mathsf{in}}^{-1}$ to this sharing, forming an xor-sharing of $\mathsf{path}' = \Pi_{\mathsf{out}}(\mathsf{EM}^\circ(\Pi_{\mathsf{in}}(\mathsf{path})))$, and finally remove the fake tuples added at the beginning of XOT−m from each bucket so path' has the original tree path size. Below we explain that permutation $\Pi_{\mathsf{in}}^{-1}\cdot\mathsf{EM}^\circ\cdot\Pi_{\mathsf{in}}$ which is thus applied to array path is identical to permutation $\mathsf{EM} = \mathsf{EM}_{\sigma,\mathsf{t}}$. It follows that C and E's output in this step is an xor-sharing of $\mathsf{path}' = \mathsf{EM}(\mathsf{path})$.

The last steps of the eviction protocol consist of transforming the xor-sharing of path' held by C, E into a "2 out of 3" sharing, and then parsing it back as a stash and the tree-path proper, so that the latter can be written back as the new version of the path tree.path(L) retrieved in the first step of algorithm 3PC-ORAM-MainLoop.

Eviction Correctness. We claim that the eviction protocol described above implements mapping $\mathsf{EM}_{\sigma,\mathsf{t}}$ applied to path, i.e. that

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

To see that eq. (1) holds, note that the subset S' of tuples on path moved by $\mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ} = \mathsf{EM}_{\pi\sigma\pi^{-1},\rho\oplus\pi(t i\oplus\delta)}$ consists of tuples of the form

$$(k, \mathsf{t}^\circ_k) = (k, \rho_k \oplus \mathsf{t}_{\pi^{-1}(k)} \oplus \delta_{\pi^{-1}(k)}) = (\pi(j), \rho_{\pi(j)} \oplus \mathsf{t}_j \oplus \delta_j)$$

where subset S of path contains tuples of the form (j, t_j) which are the only ones moved by $\mathsf{EM}_{\sigma,\mathsf{t}'}$. Note that $(\tilde{\rho}\cdot\ddot{\pi}\cdot\tilde{\delta})(j,\mathsf{t}_j)$ equals:

$$(\tilde{\rho}\cdot\ddot{\pi})(j, \mathsf{t}_j \oplus \delta_j) = \tilde{\rho}(\pi(j), \mathsf{t}_j \oplus \delta_j) = (\pi(j), \rho_{\pi(j)} \oplus \mathsf{t}_j \oplus \delta_j)$$

which implies that $S' = \Pi_{\mathsf{in}}(S)$, hence if $(j,t) \notin S$ then $\Pi_{\mathsf{in}}(j,t) \notin S'$, in which case $(\mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ}\cdot\Pi_{\mathsf{in}})(j,t) = \Pi_{\mathsf{in}}(j,t)$, and therefore $\Pi_{\mathsf{out}} = \Pi_{\mathsf{in}}^{-1}$ implies that $\Pi_{\mathsf{out}}\cdot\mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ}\cdot\Pi_{\mathsf{in}}$ and $\mathsf{EM}_{\sigma,\mathsf{t}}$ are identical on $(j,t) \notin S$. On the other hand, for tuples (u, t_j) in S we have that

$$
\begin{array}{rcl}
(j, \mathsf{t}_j) & \xrightarrow{(\tilde{\rho}\cdot\ddot{\pi}\cdot\tilde{\delta})} & (\pi(j), \rho_{\pi(j)} \oplus \mathsf{t}_j \oplus \delta_j) \\
& = & (\pi(j), \mathsf{t}^{\circ}_{\pi(j)}) \\
& \xrightarrow{\mathsf{EM}_{\pi\sigma\pi^{-1},\mathsf{t}^{\circ}}} & (\pi\sigma\pi^{-1}(\pi(j)), \mathsf{t}^{\circ}_{\pi\sigma\pi^{-1}(\pi(j))}) \\
& = & (\pi\sigma(j), \mathsf{t}^{\circ}_{\pi\sigma(j)}) \\
& = & (\pi\sigma(j), \rho_{\pi\sigma(j)} \oplus \mathsf{t}_{\sigma(j)} \oplus \delta_{\sigma(j)}) \\
& \xrightarrow{\tilde{\rho}} & (\pi\sigma(j), \mathsf{t}_{\sigma(j)} \oplus \delta_{\sigma(j)}) \\
& \xrightarrow{\ddot{\pi}^{-1}} & (\sigma(j), \mathsf{t}_{\sigma(j)} \oplus \delta_{\sigma(j)}) \\
& \xrightarrow{\tilde{\delta}} & (\sigma(j), \mathsf{t}_{\sigma(j)})
\end{array}
$$

hence $\Pi_{\mathsf{out}} \cdot \mathsf{EM}_{\sigma^{\circ},\mathsf{t}^{\circ}} \cdot \Pi_{\mathsf{in}}$ and $\mathsf{EM}_{\sigma,\mathsf{t}}$ are also identical on $(i, t) \in S$.

Below we explain first how our eviction logic Routing circuit computes the eviction map σ, t, and then we show how we tweak the computation of Routing to let D compute the masked permutation σ° and index pointer table t°.

Eviction Logic Circuit Routing**.** The idea of our eviction logic circuit Routing (Alg. 23 in Appendix B) is based on the eviction algorithm of Circuit ORAM [24], a greedy approach that, when scanning the path from top to bottom, always finds and holds on to the deepest tuple at current level to evict, and drop down the deepest tuple found at some previous level. As mentioned in the overview, this greedy approach has privacy issue in the 3PC model of [9] because its eviction map is "irregular" and may leak access information when revealed to some party in clear. To resolve this issue, we add extra spurious eviction jumps on top of the eviction map generated from Circuit ORAM's eviction algorithm and make it into a "regular" cycle. We have shown in the overview that by applying random permutations/masks on the cycle, the masked extended eviction map looks random to the party who learns it in clear.

Our Routing circuit consists of three sub-circuits PrepareDeepest, PrepareTarget, and MakeCycle. PrepareDeepest and PrepareTarget are modifications of existing algorithms of Circuit ORAM. Like Circuit ORAM's PrepareDeepest algorithm, our PrepareDeepest computes a deepest array describing levels from which tuples that can travel farthest are coming. Based on this deepest array, our PrepareTarget algorithm computes the $[\sigma]$ array containing the source and destination levels of eviction jumps just like Circuit ORAM's PrepareTarget algorithm, except we also add extra spurious jumps between gaps of real eviction jumps. And finally by MakeCycle, we add backward spurious jumps and construct a cyclic eviction map. Because our underlying greedy eviction algorithm is the same as the one in Circuit ORAM, we leave details of sub-circuits PrepareDeepest, PrepareTarget, and MakeCycle in Appendix B, as Alg. 24-29.

Make Eviction Map Oblivious. Finally, we show how we tweak the garbled circuit computation of the eviction logic circuit to let D compute the masked permutation and pointer table

$$\sigma^{\circ} = \pi \cdot \sigma \cdot \pi^{-1} \quad \mathsf{t}^{\circ} = \rho \oplus \pi(\mathsf{t} \oplus \delta)$$

given the Circuit-ORAM circuit which computes (σ, t). We will do this using another 3-party OT variants, similar to XOT of [9] as well as an "oblivious table look-up" sub-protocol. Both protocols utilize the fact that parties C, E know the masking factors π, ρ, δ which define $(\sigma^{\circ}, \mathsf{t}^{\circ})$ as a function of (σ, t). The crucial point is that we do this without implementing permutation π or π^{-1} in the garbled circuit, which would increase the circuit size by a *multiplicative* $O(\log d)$ factor.

To this end, party E who garbles the eviction logic circuit, embeds δ in the circuit so that it computes (σ, t') for $\mathsf{t}' = \mathsf{t} \oplus \delta$. Secondly, we remove the output look-up tables for the σ output wires in the circuit, i.e. D will output not array $[\sigma]$ in the clear but only the set of garbled circuit *wire keys* corresponding to these output bits, which we denote $\overline{[\sigma]}$.

We will then use two 3-party sub-protocols: First, protocol PermuteTarget (Appendix A, Alg. 19), which lets D compute $\sigma^{\circ} = \pi \cdot \sigma \cdot \pi^{-1}$, given D's input $\overline{[\sigma]}$ and C/E's input permutation π and the set of output wire keys of circuit Routing which define all possible values of $\overline{[\sigma]}$. Second, protocol PermuteIndex (Appendix A, Alg. 20), which lets D compute $\mathsf{t}^{\circ} = \rho \oplus \pi(ti')$, given D's input t' and C/E's input ρ, π. Protocol PermuteIndex is an adaptation of the three-party "oblivious permutation" protocol Reshuffle of [9] using an output mask ρ. Protocol PermuteTarget is more involved, and it combines two instances of the "oblivious permutation" protocol, implementing π^{-1} and π, and an "oblivious look-up" table which allows for an oblivious look-up of the corresponding clear text value $\sigma(i)$ (but masked so that not to reveal intermediary values) given the set of wire keys $[\sigma(i)]$ which encode it.

Once party D gets output $(\sigma^{\circ}, \mathsf{t}^{\circ})$, it is trivial to compute the permuted extended eviction map defined by $(\sigma^{\circ}, \mathsf{t}^{\circ})$ by performing the local computation EM-Comp as shown in Appendix A, Alg. 21.

4.3 Optimizations

Root Bucket Pre-Processing before Eviction. In the overview when introducing the regular eviction map formation we assumed for simplicity that the root bucket also has size same as non-root bucket size w. To adjust for the fact that the root bucket has a different size $s > w$ we precede the eviction algorithm with protocol UpdateRoot (see Alg. 7) which finds an empty position in the root, inserts the accessed tuple (modified with a new random leaf) at this position, and finds the "deepest" tuple Tup^* in the root, i.e. the tuple that can go furthest down the accessed path, swaps Tup^* with some tuple among the first w tuples in the root. After this, the only part of the root that goes into the rest of the eviction algorithm is the w-sized prefix of the root, which guaranteed to contain the "deepest" tuple in the root. Then our eviction algorithm only deals with paths with uniform bucket size w. The deepest and empty tuples are found by a circuit RFD (we leave RFD discussion in Appendix B.2 as it is an adoption from [24]), and once the positions (masked by random permutation) are found, the tuple-swapping is implemented by XOT (Alg. 22 in Appendix A). Note that RFD can be integrated into the main eviction Routing circuit (discussed in Section 4.2), but separating it out reduces the overall circuit size because Routing circuit then only needs $\log(w)$ bits to describe each tuple index instead of $\log(s)$ bits.

... to output indexes j_1, j_2, of resp. the deepest and empty tuples, in the clear, we must ensure that as long as the root contains at least one empty tuple (otherwise the ORAAM datastructure encounters a failure), this pair of indexes is distributed as two *different* random integers in Z_s. We ensure this in two ways: First, protocol UpdateRoot computes FDAE on the root bucket permuted by a random permutation (see Alg. 7), which ensures that the subset S_1 of deepest tuples and the subset S_2 of empty tuples in the root are sets of resp. $|S_1|$ and $|S_2|$ randomly chosne distinct indexes in Z_s. That would suffice to ensure security if $|S_1| = |S_2| = 1$, i.e. if there was just one deepest and one empty tuple in the root. However, if either set has more than one element then circuit FDAE, which outputs the first element in both sets,...

Algorithm 7 Protocol UpdateRoot (Optimization before Step 8, Alg 6)

Param: Tree height d, root (= stash) size s, tuple bitsize ℓ.
Input: $\mathsf{L}^{\mathsf{CE}}, \langle X, T\rangle_{\mathsf{xor}}^{\mathsf{C-E}}$, for $\mathsf{L}\in\{0,1\}^d$, $X \in \mathsf{array}[s](\ell)$, $T\in\{0,1\}^\ell$
Output: $\langle Y\rangle_{\mathsf{xor}}^{\mathsf{C-E}}$ s.t. Y contains all elements in X with T inserted and the deepest tuple (rel. to leaf L) moved to first position

1: $p^{\mathsf{CE}} \xleftarrow{\$} \mathsf{perm}_s$; $\langle Z\rangle_{\mathsf{xor}}^{\mathsf{C-E}} := \langle p(X)\rangle_{\mathsf{xor}}^{\mathsf{C-E}}$; $(\Delta_1, \Delta_2)^{\mathsf{CE}} \xleftarrow{\$} \mathsf{Z}_s$
2: GC(RFE): $\left\langle \Delta_1, Z_{\mathsf{shift}[\Delta_1]}.\mathsf{fb}\right\rangle_{\mathsf{xor}}^{\mathsf{C-E}} \rightarrow j_{\mathsf{em}}^{\mathsf{D}}$ (in clear)
s.t. j_{em} points to the first empty tuple after Δ_1 in Z=$p(X)$
3: GC(RFD): $\left\langle \mathsf{L}, \Delta_2, Z_{\mathsf{shift}[\Delta_2]}.(\mathsf{fb}, \mathsf{lb})\right\rangle_{\mathsf{xor}}^{\mathsf{C-E}} \rightarrow j_{\mathsf{dp}}^{\mathsf{D}}$ (in clear)
s.t. j_{dp} points to the first deepest tuple after Δ_2 in Z=$p(X)$
4: D defines $I : \mathsf{Z}_s \rightarrow \mathsf{Z}_{s+1}$ s.t. $I[0] = j_{\mathsf{dp}}$, $I[j_{\mathsf{dp}}] = 0$, $I[j_{\mathsf{em}}] = s$, and $I(i) = i$ at all other points $i \in \mathsf{Z}_s$
5: XOT: $I^{\mathsf{D}}, \langle W\rangle_{\mathsf{xor}}^{\mathsf{C-E}} \rightarrow \langle Y\rangle_{\mathsf{xor}}^{\mathsf{C-E}}$ for $W = Z|\{T\}$
(Y contains elements of $p(X)$, with 0-th and j_{dp}-th tuples swapped and tuple T injected in place of j_{em}-th tuple)

Bandwidth: Online: $\approx s \cdot (2\kappa(d + 2\log s + 1) + 4\ell)$, Offline: $\approx 4\kappa(3d + 2\log s)s$; Rounds: 2;
Security: By correctness of circuit RFD and Security of subprotocol XOT, step ?? generates a fresh two-party xor-sharing of list Y which contains all elements of X with tuple T inserted and the deepest tuple placed at the front of Y. The only additional information protocol UpdateRoot reveals are indexes j_{dp}, j_{em} computed by RFD, but which is list X with tuple T inserted and the only additional C, D, E's views contain only output of MPC building blocks RFD and XOT, and thus are indistinguishable from random.

4.4 Efficiency

Bandwidth.

Round Complexity. The message round for an individual 3PC-ORAM-MainLoop execution is 8, which is the sum of 3 rounds for Retrieval, 1 round for Post-Processing, and 4 rounds for Eviction. However, if we consider a complete ORAM read/write operation which involves h executions of 3PC-ORAM-MainLoop, the total rounds can be $3h + 1 + 4 = 3h + 5$ instead of $8h$. This is because, while the Retrieval of 3PC-ORAM-MainLoop can only be executed sequentially through h ORAM levels, the executions of Post-Processing and Eviction on different levels can run in parallel (Post-Processing and Eviction of on certain level can be triggered as long as the Retrieval of that level is finished). Therefore the total rounds for such parallel ORAM read/write operation are the total rounds for h Retrievals plus the rounds for the last Post-Processing and Eviction, which is $3h + 5$.

When considering B number of ORAM accesses with B batched Retrievals + Post-Processing and postponed Evictions, where Retrievals and Post-Processing are executed in a pipelined fashion s.t. when Retrievals and Post-Processing of i-th access finishes on the current level, we start to run $(i + 1)$-th access, our Retrieval + Post-Processing rounds can be $4B + h$ instead of $4hB$.

Note that when counting the rounds for building blocks of 3PC-ORAM-MainLoop above, we assume the optimal rounds for its sub-protocols, and discuss them in the following:

(1) 3ShiftXorPIR: Protocol 3ShiftXorPIR in Alg 9 consists of 2 Reshare protocols and 3 ShiftXorPIR protocols. The round of first Reshare protocol (step 1) can be eliminated by performing 3 copies of protocol CDI in ORAM-MainLoop, each of which will produce one (2,1)-shift sharing of the index for one of the 3 ShiftXorPIR protocols. Then since the round for ShiftXorPIR protocol is 2 (see below) and we can execute the 3 copies of them in parallel, we add 2 to the rounds. Finally the last Reshare round can also be removed by incorporating this step into the last round of PIR protocol. This is because the input of this Reshare can be reconstructed by the last flow of PIR and then do local xor computation. Thus protocol 3ShiftXorPIR can be optimized with 2 rounds.

(2) 3ShiftPIR: Same reason as in 3ShiftXorPIR, protocol 3ShiftPIR in Alg 10 can be optimized with 2 rounds because the rounds of 2 Reshare protocols can be reduced, and the 3 copies of ShiftPIR protocol, each of which has 2 rounds, can run in parallel.

(3) ShiftXorPIR: Protocol ShiftXorPIR in Alg 11 executes 2 sub-protocols ShiftPIR and XorPIR. While both of the sub-protocols requires 2 rounds, they can execute in parallel, and therefore the round of ShiftXorPIR is 2.

(4) PPT: Protocol PPT in Alg 14 consists of 3 Reshare and 2 InsertLbl protocols. The 2 InsertLbl protocols, each of which takes 1 round, can execute in parallel, so they are still 1 round. The first Reshare protocol in step 1 can be done by local computation with pre-generated sharing during the offline, so the round can be eliminated. The second Reshare in step 3 can also be done locally because: Notice that the InsertLbl will output $c^{\mathsf{P}_2}, e^{\mathsf{P}_3}$ where $c \oplus e = M_1$ and c can be pre-computed. If we switch roles of $\mathsf{P}_1, \mathsf{P}_2$ and do the second InsertLbl, it will output, for example $c_2^{\mathsf{P}_1}, e_2^{\mathsf{P}_3}$ where $c_2 \oplus e_2 = M_2$ and c_2 can be pre-computed ($M_1 \oplus M_2 =$ payload M). Let $c2$ be pre-computed as $a \oplus b$. If P_1 pre-share a with P_2 and b with P_3, then we have $((a\oplus c)^{\mathsf{P}_2}, (e \oplus e_2 \oplus b)^{\mathsf{P}_3}) = \langle M\rangle_{\mathsf{xor}}^{\mathsf{P}_2-\mathsf{P}_3}$, where $(a \oplus c)^{\mathsf{P}_2}$ can be pre-computed. So the second Reshare is achieved with no extra round. Now for the last Reshare in step 5: By doing one copy of 2 InsertLbl protocols, we can have $A_1^{\mathsf{P}_2}, A_2^{\mathsf{P}_3}$, where $A_1 \oplus A_2 = M$ and A_1 is pre-computed. So if we switch roles of P_1 and P_3 and do another copy of 2 InsertLbl, we should get $B_1^{\mathsf{P}_2}, B_2^{\mathsf{P}_1}$ where $B_1 \oplus B_2 = M$ and B_1 is pre-computed. So together we have the following: $(A_1, B_1)^{\mathsf{P}_2}$(pre-computed), $B_2^{\mathsf{P}_1}, A_2^{\mathsf{P}_3}$, where $A_1 \oplus A_2 = B_1 \oplus B_2 = M$. Our goal is that we need $\langle M\rangle_{2,3} = (X_1^{\mathsf{P}_1,\mathsf{P}_2}, X_2^{\mathsf{P}_2,\mathsf{P}_3}, X_3^{\mathsf{P}_1,\mathsf{P}_3})$ where X_1, X_2 can be pre-computed. Since A_1, B_1, X_1, X_2 are all pre-computed, P_2 can sends $B_1 \oplus X_1 \oplus X_2$ to P_1, and sends $A_1 \oplus X_1 \oplus X_2$ to P_3. Now when P_1 gets B_2 and P_3 gets A_2 from the above InsertLbl protocols, P_1 can locally compute $X_3 = (B_1 \oplus X_1 \oplus X_2) \oplus B_2 = M \oplus X_1 \oplus X_2$, and P_3 can locally compute $X_3 = (A_1 \oplus X_1 \oplus X_2) \oplus A_2 = M \oplus X_1 \oplus X_2$.

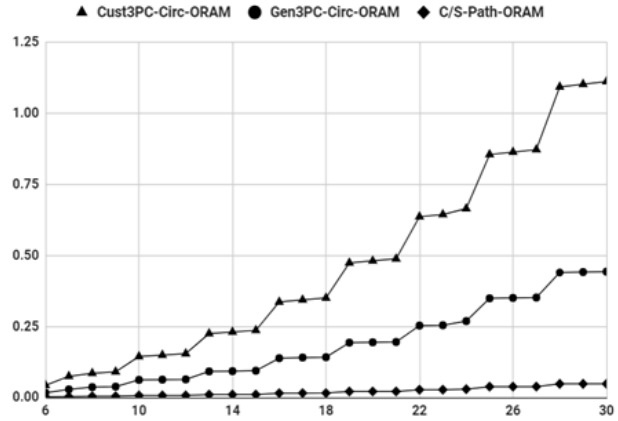

Figure 3: Scheme Comparison: Online Bandwidth (MB) vs m, for D=4B

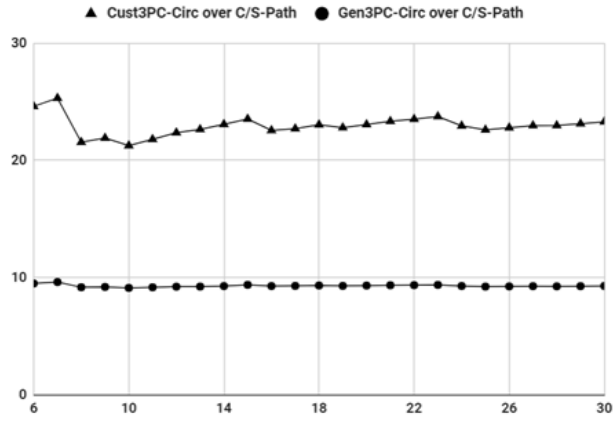

Figure 4: Online Bandwidth of Circ/AFLNO ORAM over Path ORAM vs m, for D=4B

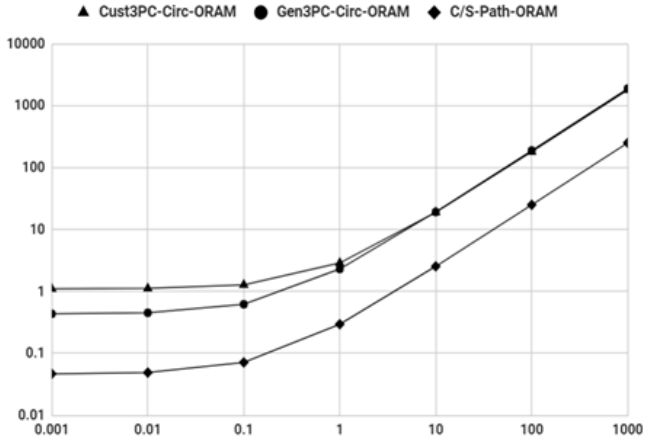

Figure 5: Scheme Comparison: Online Bandwidth (MB) vs D (KB), for m=30

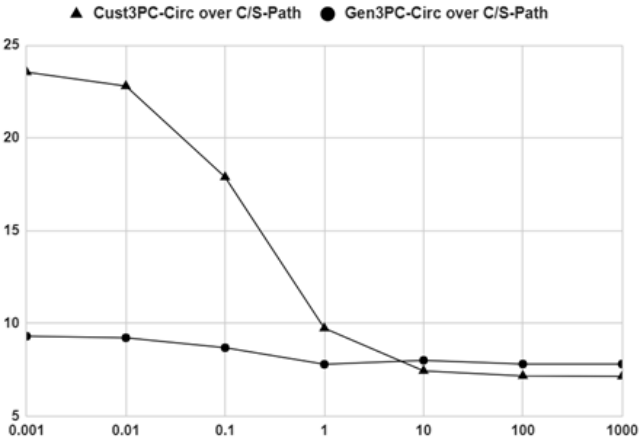

Figure 6: Online Bandwidth of Circ/AFLNO ORAM over Path ORAM vs D (KB), for m=30

Therefore we can get the $\langle M \rangle_{2,3}$ with no extra round to $2*2 = 4$ parallel InsertLbl protocols, which is just 1 round. Thus we have the conclusion that protocol PPT can be optimized to 1 round.

(5) FlipFlag: Protocol FlipFlag actually takes 2 rounds as the way written in Alg 16. However, FlipFlag can execute immediately when protocol CDI is finished in 3PC-ORAM-MainLoop, which means FlipFlag can run in parallel with 3ShiftXorPIR and 3ShiftPIR. Therefore the 2 rounds of FlipFlag don't need to be added to the rounds of Post-Processing, which can be just 1, from protocol PPT.

5 PERFORMANCE EVALUATION

We first explain how our 3PC Circuit ORAM compares against generic 3PC ORAM schemes using AFLNO-style MPC, and we explain why it makes sense to (1) implement ORAM Retrieval and Data Movement steps with our protocols instead of using generic MPC approaches, and (2) implement the Eviction Logic circuit using Yao's garbled circuits rather than AFLNO-style MPC. We then report the performance data we collected about our prototype.

5.1 Our Scheme vs Generic 3PC ORAM

Consider three 3PC ORAM variants, an ORAM client implemented using generic AFLNO MPC [1] (Ψ_1), our 3PC Circuit ORAM with the Eviction Logic circuit implemented using AFLNO MPC (Ψ_2), and our 3PC Circuit ORAM (Ψ_3). The base-line for all these schemes is the underlying Client-Server Path ORAM scheme (Ψ_0). Because AFLNO MPC has constant bandwidth per circuit gate and the Eviction Logic circuit (which is the most costly part of the Client computation) is aymptotically size-optimal, i.e. its size is $O(1)$ times the size of its inputs, it follows that Ψ_1 and Ψ_2 have same $O(m^3 + mD)$ bandwidth as Ψ_0. However, the Eviction Logic circuit has the non-xor gate depth of $\Omega(m(\log m))$, which implies $\Omega(m^2(\log m))$ lower-bound on round complexity of both Ψ_1 and Ψ_2. By contrast, protocol Ψ_3 has $O(m^3\kappa + mD)$ bandwidth, which is asymptotically higher than Ψ_0 for small records, namely $m = \omega(m^2)$, but its round complexity is optimal $O(m)$, just like Ψ_0. We note that protocol Ψ_2 is close to Ψ_1 on bandwidth but much better on round complexity: Even generic AFLNO MPC's bandwidth is $\approx 6 \cdot |path|$ bits in Ψ_1, and Ψ_2's bandwidth is $\approx 7 \cdot |path|$, but the round complexity is $\Omega(m^2(\log(m)))$

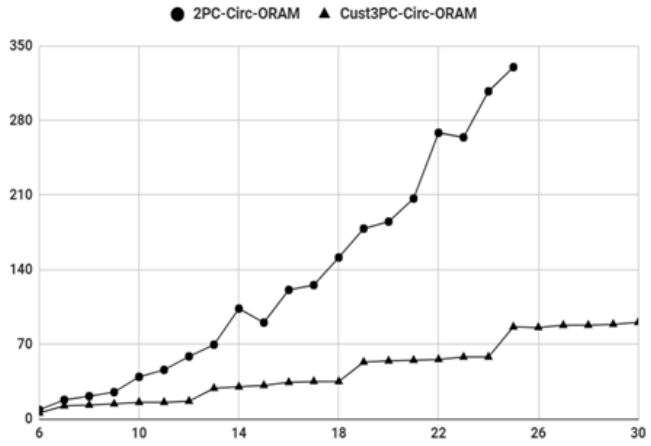

Figure 7: Online CPU (ms) vs m, for $D = 4$B

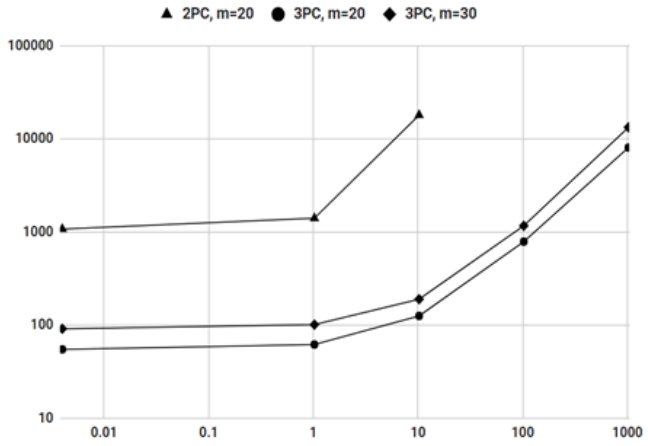

Figure 8: Online CPU (ms) vs D (KB)

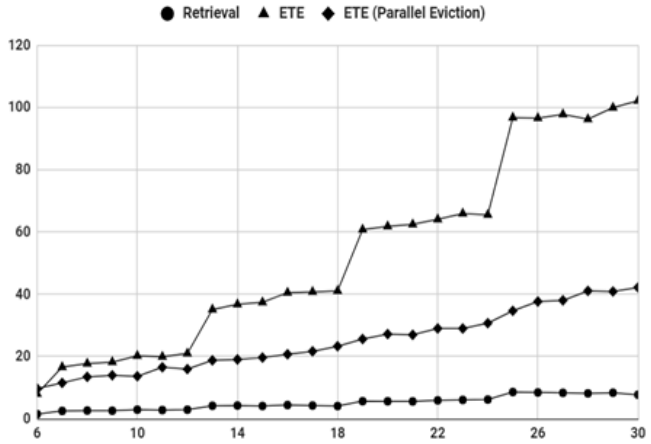

Figure 9: 3PC Circuit ORAM: Online WC Time(ms) vs m(bits), for $D = 4$ bytes

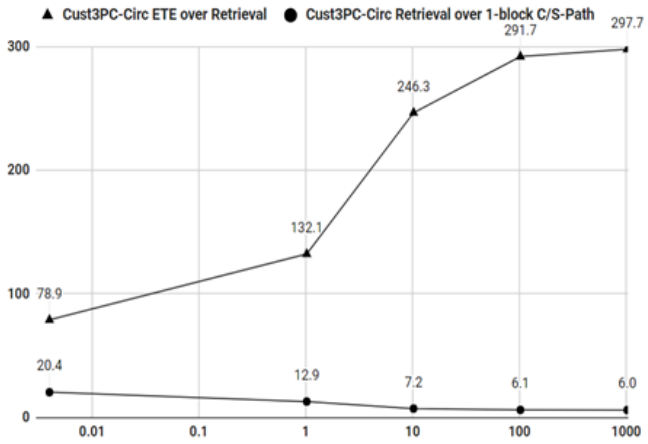

Figure 10: Bandwidth Ratio vs D(bytes), for $\tau = 6, m = 30$

in Ψ_1 and $O(m)$ in Ψ_2. We are thus left with comparing only Ψ_2 and Ψ_3.

The round complexity of $\Omega(m^2(\log m))$, which in our estimates would be > 1000 for $m = 30$, makes it doubtful whether Ψ_2 would be practical in spite of its lower asymptotic bandwidth. By contrast, the exact round complexity of Ψ_3 is 8 for address size $m = 30$. Nevertheless, we want to compare the bandwidth of schemes Ψ_2 and Ψ_3 in exact terms. Assuming that the AFLNO computation uses just 3 bits to compute a non-xor gate, we show estimates for bandwidth Ψ_2 and Ψ_3 in Figures 3 and 5, together with the base-line bandwidth Ψ_0. (In these Figures "C/S Path ORAM" is Bndw(Ψ_0), "3PC AFLNO ORAM" is Bndw(Ψ_2), and "3PC Circ ORAM" is Bndw(Ψ_3).) Figure 3 shows the bandwidth as a function of the address size m (for $D = 4$B), and Figure 5 as a function of record size D (for $m = 30$). To make it easier to compare how far Ψ_2 and Ψ_3 are from the base-line scheme Ψ_0, in Figures 6 and 4 we graph Bndw(Ψ_2)/Bndw(Ψ_0) and Bndw(Ψ_3)/Bndw(Ψ_0). Figure 4 confirms that for fixed D and κ, bandwidth Bndw(Ψ_2) and Bndw(Ψ_3) are related by a constant factor, and that for for small records of 4B (and $\kappa = 80$), Bndw(Ψ_2) is roughly 12 times Bndw(Ψ_0) and Bndw(Ψ_3) is roughly 2 times higher than Bndw(Ψ_2). However, from Figure 6 we see that the difference between Bndw(Ψ_2)/Bndw(Ψ_0) and Bndw(Ψ_3)/Bndw(Ψ_0) diminishes rapidly as record size D grows, and they become almost the same already for $D >$ 1KB. Thus using garbled circuits instead of OT-based computation for the Eviction Circuit dramatically reduces the round complexity at the cost of increasing bandwidth by a factor of 2 for small records, or not at all for records larger than 1KB.

The Eviction Logic circuit could also be implemented using BGW-style MPC [2], using an arithmetic representation of this circuit. However, the arithmetic BGW-style protocol would also increase round complexity, and the bandwidth of the minimal-round approach which uses Yao's garbled circuits, i.e. scheme Ψ_3, is only a factor of about 8 over the underlying Client-Server Path-ORAM scheme, Ψ_0, already for records of size $D = 1$KB (see Figure 6).

5.2 Implementation and Performance

We performed tests of our protocol 3PC-ORAM on three AWS EC2 c4.2xlarge servers, and our garbled circuit implementation uses the

ObliVM library by Wang [24]. In our tests the RAM address bits m ranges from 6 to 30, and rec size D ranges from 4 Bytes to 1 MB. We stress that $m = 30$ and $D = 1MB$ are not the limits of either our prototype or our implementation that we only used these two ranges to show our prototype performance and compare with 2PC Circuit ORAM (denote as Circ-ORAM).

Sequential and Parallel Eviction Time. In Fig. 9 we show the retrieval wall clock (WC) time, end-to-end (retrieval + eviction) WC, and end-to-end WC with parallel evictions. Parallel execution of evictions means that when ever the Retrieval protocol on some tree tree_i is finished, we will execute Eviction on tree_i immediately while the Retrieval on tree_{i+1} is also started, so evictions on different levels, i.e. i and $i + 1$, may run at the same time to shorten overall end-to-end time.

As Fig. 9 shows, for small rec size $D = 4B$, each access takes about 12 milliseconds (online) when the record address m is 30 bits long. When looking at the end-to-end WC time, we observer that most of the time is spent on Eviction protocols, and the Eviction/Retrieval ratio grows as m increases. However, if executed in our parallel mode, the WC time reduction can be more than 50% for large m. Thus, our 3PC-ORAM allows fast online accesses, and even without support of parallel accesses, the post-access update time can be shortened using the parallel eviction to reduce waiting time between accesses in practice.

Retrieval Bandwidth. In Fig. 10, we show how our retrieval bandwidth compare to our end-to-end bandwidth, and also client/server path oram bandwidth with 1-block size retrieval, over data size D. As Fig. 10 indicates, the ratio of our end-to-end bandwidth over retrieval converges to a large constant, as D grows, because the end-to-end bandwidth has the extra $\log(m)$ factor in it. And if we compare our retrieval bandwidth with the underlying client/server Path ORAM scheme which only sends 1 block of payload per each level's retrieval, we see the ratio converges to 6 because our PIR protocols sends only 6 blocks of payload size data for retrieval. So Fig. 10 supports the fact that our retrieval bandwidth only contains constant number of payload size data, where the constant is also small and close to the underlying client/server scheme.

Online CPU Time. We compare online CPU time between 2PC and 3PC Circuit ORAM with respect to both rec size D and address bit length m. 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 m increases (shown in Fig. 7), and 3PC's online CPU time can be about 6 times smaller than 2PC for $m = 25$ mainly because 2PC has much larger circuit size. We also concluded that if fixing m 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. 8 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 $\kappa = 80$.

Additional Performance Considerations. In Appendix C we include discussion of comparisons between the 3PC-ORAM and Circ-ORAM in terms of their bandwidth, as a function of array size m and record size D. We also show the break-down of different CPU cost components, namely garbled circuit computation and transmission encryption, as fractions of the overall CPU cost, as function of the record size D. Furthermore, we explain how we pick the parameter τ which determines the nummber of oram trees in the Path-ORAM datastructure, and how this choice differs between 3PC-ORAM and Circ-ORAM.

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] 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.

[3] 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.

[4] E. Boyle, K.-M. Chung, and R. Pass. *Oblivious Parallel RAM and Applications*, pages 175–204. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[5] 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.

[6] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, Nov. 1998.

[7] I. Damgard, S. Meldgaard, and J. B. Nielsen. Perfectly secure oblivious ram without random oracles. In *Theory of Cryptography*, pages 144–163, 2011.

[8] 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.

[9] S. Faber, S. Jarecki, S. Kentros, and B. Wei. *Three-Party ORAM for Secure Computation*, pages 360–385. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[10] 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.

[11] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs. Optimizing oram and using it efficiently for secure computation. In *Privacy Enhancing Technologies*, PETS'13, pages 1–18, 2013.

[12] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 151–160, 1998.

[13] 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.

[14] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[15] M. T. Goodrich and M. Mitzenmacher. *Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation*, pages 576–587. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[16] 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.

[17] 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.

[18] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 143–156, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.

[19] S. Lu and R. Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, pages 377–396, 2013.

[20] 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.

[21] B. Pinkas and T. Reinman. *Oblivious RAM Revisited*, pages 502–519. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[22] 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, Berlin, Heidelberg, 2011.

[23] 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.

[24] 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.

[25] 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.

[26] P. Williams and R. Sion. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 293–304, New York, NY, USA, 2012. ACM.

A AUXILIARY THREE-PARTY PROTOCOLS

Here we specify all sub-protocols used in the 3PC-ORAM algorithm 3PC-ORAM-MainLoop, Alg 6, in Section 4.

Algorithms 8-**??** are used in the Retrieval phase of 3PC-ORAM-MainLoop. All of them are secret-sharing variants of information-theoretic 2-server PIR of Chor et al. [6], where all the inputs and outputs, the array x, the index i, and the retrieved record $x[i]$, are secret-shared in a way that makes both the Retrieval stage and the subsequent phases have low bandwidth *and* low round complexity. The 2-server PIR of [6] we use does not have the best known asymptotic performance, using $O((n \cdot \ell)^{1/2})$ bits if x has n records of ℓ bits each, but we use it because (1) in our application the record size ℓ and the number of records n are either about the same or $\ell > n$ for last ORAM tree level and large records, in which case this simple 2-server PIR has optimal $O(\ell)$ bandwidth; and (2) we can convert this PIR to 3PC variants where inputs and outputs are secret-shared while preserving the $O(\ell)$ bandwidth.

Algorithms 14-17 are used in the PostProcess phase of 3PC-ORAM-MainLoop. These algorithms are variants of protocols Reshuffle and XOT of Faber et al. [9] which we overview in Section 4.

Algorithm 7 is an optimization step

Algorithms 18-22 are used in the Eviction phase of 3PC-ORAM-MainLoop. Protocol GC in Alg 18 is a (slightly modified) three-party Garbled Circuit computation variant of Faber et al. [9], where the input values are xor-shared between parties C and E, and party D receives the circuit output. The modification needed in GC is that the circuit returns two outputs, y and z, where value y is reconstructed by D in the clear while value z is reconstructed only as $\overline{z}$, i.e. the output keys corresponding to the bits encoding z. The last protocol we include, protocol XOT in Alg 22, is from Faber et al [9]. It creates C/E xor-sharing of array $\pi(x)$ given a C/E xor-sharing of array x and a permutation π held by D. It does so with one flow and $2n(\ell + \log n)$ bandwidth. Protocol PermuteIndex, Alg 20, is one of the two output-masking protocols used in the Eviction, and it is a variant of protocol Reshuffle, Alg 17 because it applies permutation π to a list (and applies an outer mask to the result) where π is held by two protcocol parties. Protocol PermuteTarget, Alg 19 can also be seen as a version of protocol Reshuffle, but it is more involved. Its goal is to translate from D holding wire-keys representing $\overline{[\sigma]}$ to D holding $\sigma^\circ = \pi\sigma\pi^{-1}$ where C and E hold π. It does it in the following steps: Given that E holds both π and a translation table from wire-keys to output values that these keys encode, $\overline{[\sigma]}$ is transformed to a secret-sharing of $[\pi \cdot \sigma]$. Secondly, since applying permutation π to an array representation of permutation δ, i.e. to array $[\delta]$, creates an array representation of $[\delta \cdot \pi^{-1}]$, we use a variant of Reshuffle to apply π to (the sharing of) $[\pi \cdot \sigma]$, which creates $[\sigma^\circ]$ for $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$.

A.1 Protocols for Retrieval

Algorithm 8 Protocol CDI (from [9]) (Step 2 in Alg 6)

Param: Security parameter κ, soundess parameter λ, PRF $\mathsf{F} : \{0,1\}^\kappa \to \{0,1\}^\lambda$, array size n, record size $\ell \le \kappa$.
Input: $\langle u, v\rangle^{\mathsf{D-E}}_{\mathsf{xor}}$, where $u \in \mathsf{array}[n](\ell)$, $|v| = \ell$
Output: $\langle i\rangle_{2,\,1\text{-shift}}$ for *unique* index i in Z_n s.t. $u[i] = v$
Offline: key k^{DE} of PRF F, $r^{\mathsf{DE}} \xleftarrow{\$} \mathsf{array}[n](\kappa)$, $s^{\mathsf{DE}} \xleftarrow{\$} \mathsf{Z}_n$

1: $(a^{\mathsf{D}}, b^{\mathsf{E}}) := \langle z\rangle^{\mathsf{D-E}}_{\mathsf{xor}}$ for z s.t. $z[j] = u[j-s \bmod n] \oplus v$ for all j
2: D sends array x to C s.t. $x[j] = \mathsf{F}_{\mathsf{k}}\left(r[j] \oplus (a[j]|0^{\kappa-\ell})\right)$ for all j
3: E sends array y to C s.t. $y[j] = \mathsf{F}_{\mathsf{k}}\left(r[j] \oplus (b[j]|0^{\kappa-\ell})\right)$ for all j
4: Let $\langle i\rangle^{\mathsf{DE-C}}_{2,\,1\text{-shift}} := (s^{\mathsf{DE}}, t^{\mathsf{C}})$ for unique t s.t. $x[t] = y[t]$
(if $x[t] = y[t]$ for two t's, go back to step 2 with fresh r^{DE})
5: Reshare: $\langle i\rangle^{\mathsf{DE-C}}_{2,\,1\text{-shift}} \;\to\; \langle i\rangle_{2,\,1\text{-shift}}$

Bandwidth: $2n\lambda$; Rounds: 2; (*with* $\le 2^{-\lambda+\ln n}$ *probability of re-run*);
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_n s.t. $a[i] = b[i]$, where $x[i] = y[i]$ is distributed as single random λ-bit value.
Note: This holds over multiple executions using same PRF key k*, due to freshess of* r.

Algorithm 9 Protocol 3ShiftXorPIR (Step 3 in Alg 6)

Input: $\langle x\rangle_{2,3} = (x_1^{\mathsf{DE}}, x_2^{\mathsf{CE}}, x_3^{\mathsf{CD}})$, $\langle i\rangle_{2,\,1\text{-shift}}$, $\langle j\rangle_{2,\,1\text{-xor}}$, for $x \in \mathsf{array}[n, m](\ell)$, $i \in \mathsf{Z}_n$, $j \in \mathsf{Z}_m$
Output: $X[i][j]^{\mathsf{CDE}}$
Offline: $(\delta_c^{\mathsf{C}}, \delta_d^{\mathsf{D}}, \delta_e^{\mathsf{E}}) \xleftarrow{\$} \left\langle 0^\ell \right\rangle_{\mathsf{xor}}$

1: ShiftXorPIR: $x_1^{\mathsf{DE}}, \langle i\rangle^{\mathsf{DE-C}}_{2,\,1\text{-shift}}, \langle j\rangle^{\mathsf{DE-C}}_{2,\,1\text{-xor}} \;\to\; \langle x_1[i][j]\rangle^{\mathsf{D-E}}_{\mathsf{xor}}$
ShiftXorPIR: $x_2^{\mathsf{EC}}, \langle i\rangle^{\mathsf{EC-D}}_{2,\,1\text{-shift}}, \langle j\rangle^{\mathsf{EC-D}}_{2,\,1\text{-xor}} \;\to\; \langle x_2[i][j]\rangle^{\mathsf{E-C}}_{\mathsf{xor}}$
ShiftXorPIR: $x_3^{\mathsf{CD}}, \langle i\rangle^{\mathsf{CD-E}}_{2,\,1\text{-shift}}, \langle j\rangle^{\mathsf{CD-E}}_{2,\,1\text{-xor}} \;\to\; \langle x_3[i][j]\rangle^{\mathsf{C-D}}_{\mathsf{xor}}$
for $\langle x_1[i][j]\rangle^{\mathsf{D-E}}_{\mathsf{xor}} = (\alpha_d^{\mathsf{D}}, \alpha_e^{\mathsf{E}})$, $\langle x_2[i][j]\rangle^{\mathsf{E-C}}_{\mathsf{xor}} = (\beta_e^{\mathsf{E}}, \alpha_c^{\mathsf{C}})$, $\langle x_3[i][j]\rangle^{\mathsf{C-D}}_{\mathsf{xor}} = (\beta_c^{\mathsf{C}}, \beta_d^{\mathsf{D}})$
2: Each party broadcasts its share $z_t = \alpha_t \oplus \beta_t \oplus \delta_t$, for $t = c, d, e$.
Output $x[i][j]^{\mathsf{CDE}} := z_c \oplus z_d \oplus z_e$.

Bandwidth: $3nm + 6\ell$; Rounds: 2;
Security: By security of zero-sharing (δ_c, δ_d, δ_e), the broadcast values (z_c, z_d, z_e) are distributed as random xor-sharing of output $x[i][j]$. The rest is secure computation ShiftXorPIR which outputs random shares to each participant.

Algorithm 10 Protocol 3ShiftPIR (Step 4 in Alg 6)

Input: $\langle x\rangle_{2,3} = (x_1^{\mathsf{DE}}, x_2^{\mathsf{EC}}, x_3^{\mathsf{CD}})$, $\langle i\rangle_{2,1\text{-shift}}$, for $x \in \mathsf{array}[n](\ell)$, $i \in \mathsf{Z}_n$

Output: $\langle x[i]\rangle_{2,3}$

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

2: $\mathsf{Reshare}: \left((c_1 \oplus c_2)^{\mathsf{C}}, (d_1 \oplus d_2)^{\mathsf{D}}, (e_1 \oplus e_2)^{\mathsf{E}}\right) = \langle x[i]\rangle_{\mathsf{xor}} \longrightarrow \langle x[i]\rangle_{2,3}$

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

Algorithm 11 Protocol ShiftXorPIR (Used in Alg. 9)

Input: $x^{\mathsf{P_1P_2}}$, $\langle i\rangle_{2,1\text{-shift}}^{\mathsf{P_1P_2-P_3}} = (i_{12}^{\mathsf{P_1P_2}}, i_3^{\mathsf{P_3}})$, $\langle j\rangle_{2,1\text{-xor}}^{\mathsf{P_1P_2-P_3}} = (j_{12}^{\mathsf{P_1P_2}}, j_3^{\mathsf{P_3}})$, for $x \in \mathsf{array}[n,m](\ell)$, $i \in \mathsf{Z}_n$, $j \in \mathsf{Z}_m$

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

1: P_1 and P_2 compute x' s.t $x'[s][t] = x[s + i_{12}][t \oplus j_{12}]$, for all $(s,t) \in \mathsf{Z}_n \times \mathsf{Z}_m$. P_3 computes $k = i_3 \cdot m + j_3$.
2: $\mathsf{SSPIR}: x'^{\mathsf{P_1P_2}}, k^{\mathsf{P_3}} \rightarrow \langle x'[i_3][j_3]\rangle_{\mathsf{xor}}^{\mathsf{P_1-P_2}} (= \langle x[i][j]\rangle_{\mathsf{xor}}^{\mathsf{P_1-P_2}})$

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

Algorithm 12 Protocol ShiftPIR (Used in Alg. 10)

Input: $x^{\mathsf{P_1P_2}} \in \mathsf{array}[n](\ell)$, $\langle i\rangle_{2,1\text{-shift}}^{\mathsf{P_1P_2-P_3}} = (i_{12}^{\mathsf{P_1P_2}}, i_3^{\mathsf{P_3}})$, for $i \in \mathsf{Z}_n$

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

1: P_1 and P_2 compute x' s.t. $x'[t] = x[t + i_{12}]$, for all $t \in \mathsf{Z}_n$.
2: $\mathsf{SSPIR}: (x')^{\mathsf{P_1P_2}}, i_3^{\mathsf{P_3}} \rightarrow \langle x'[i_3]\rangle_{\mathsf{xor}}^{\mathsf{P_1-P_2}} (= \langle x[i]\rangle_{\mathsf{xor}}^{\mathsf{P_1-P_2}})$

Bandwidth: n; Rounds: 1;
Security: No message is sent besides secure computation SSPIR.

Algorithm 13 Protocol SSPIR (from [6]) (Used in Alg. 11 and 12)

Input: $x^{\mathsf{P_1P_2}} \in \mathsf{array}[n](\ell)$, $i^{\mathsf{P_3}} \in \mathsf{Z}_n$

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

Offline: $a_1^{\mathsf{P_1P_3}} \xleftarrow{\$} \mathsf{array}[n](1)$, $c_1^{\mathsf{P_1}} := c_2^{\mathsf{P_2}} := 0^{\ell}$

1: P_3 sends a_2 to P_2 where $a_2 = a_1$ except $a_2[i] = a_1[i] \oplus 1$.
2: P_1: for all $t \in \mathsf{Z}_n$, if $a_1[t] = 1$, then $c_1 := c_1 \oplus x[t]$.
P_2: for all $t \in \mathsf{Z}_n$, if $a_2[t] = 1$, then $c_2 := c_2 \oplus x[t]$.
3: Output $\langle x[i]\rangle_{\mathsf{xor}}^{\mathsf{P_1-P_2}} = (c_1^{\mathsf{P_1}}, c_2^{\mathsf{P_2}})$.

Bandwidth: n; Rounds: 1;
Security: This is the basis of security of PIR of Chor et al. [6]: P_2's view a_2 is indistinguishable from a random string because a_1 is random.

A.2 Protocols for PostProcess

Algorithm 14 Protocol PPT - Update Labels in Tuple (Step 5, Alg 6)

Input: $\langle \mathsf{N}, \mathsf{nL}, X, \Delta\mathsf{N}, \mathsf{nL_{nxt}}\rangle_{2,3}$, $\mathsf{L}_{\mathsf{nxt}}^{\mathsf{CDE}} = X[\Delta\mathsf{N}]$ where $X \in \mathsf{array}[2^{\tau}](\ell)$, $|\Delta\mathsf{N}| = \tau$, $|\mathsf{L_{nxt}}| = |\mathsf{nL_{nxt}}| = \ell$

Output: $\langle \mathsf{Tup}\rangle_{2,3}$ for $\mathsf{Tup} = (1|\mathsf{N}|\mathsf{nL}|X)$ with $X[\Delta\mathsf{N}] := \mathsf{nL_{nxt}}$

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: $\langle \mathsf{xor\text{-}L_{nxt}}\rangle_{2,3} := \langle \mathsf{nL_{nxt}} \oplus \mathsf{L_{nxt}}\rangle_{2,3}$
Local-Transform: $\langle \Delta\mathsf{N}, \mathsf{xor\text{-}L_{nxt}}\rangle_{2,3} \rightarrow \langle \Delta\mathsf{N}, \mathsf{xor\text{-}L_{nxt}}\rangle_{\mathsf{xor}}^{\mathsf{C-D}}$
Local-Transform: $\langle \Delta\mathsf{N}, \mathsf{xor\text{-}L_{nxt}}\rangle_{2,3} \rightarrow \langle \Delta\mathsf{N}, \mathsf{xor\text{-}L_{nxt}}\rangle_{\mathsf{xor}}^{\mathsf{E-D}}$
2: $\mathsf{InsertLbl}: \langle \Delta\mathsf{N}\rangle_{\mathsf{xor}}^{\mathsf{C-D}}, \langle \mathsf{xor\text{-}L_{nxt}}\rangle_{\mathsf{xor}}^{\mathsf{C-D}} \rightarrow \langle M\rangle_{\mathsf{xor}}^{\mathsf{D-E}} = (a_1^{\mathsf{D}}, b_1^{\mathsf{E}})$
$\mathsf{InsertLbl}: \langle \Delta\mathsf{N}\rangle_{\mathsf{xor}}^{\mathsf{E-D}}, \langle \mathsf{xor\text{-}L_{nxt}}\rangle_{\mathsf{xor}}^{\mathsf{E-D}} \rightarrow \langle M\rangle_{\mathsf{xor}}^{\mathsf{D-C}} = (a_2^{\mathsf{D}}, b_2^{\mathsf{C}})$
for M which is an all-zero array except $M[\Delta\mathsf{N}] = \mathsf{xor\text{-}L_{nxt}}$
3: $\langle M\rangle_{2,3} := (x_1^{\mathsf{CD}}, x_2^{\mathsf{DE}}, x_3^{\mathsf{CE}})$ for $x_3^{\mathsf{C}} := m_c \oplus b_2$, $x_3^{\mathsf{E}} := m_e \oplus b_1^{\mathsf{E}}$
Output $\langle \mathsf{Tup}\rangle_{2,3} := \langle 1|\mathsf{N}|\mathsf{nL}|(X \oplus M)\rangle_{2,3}$

Bandwidth: Online: $\approx 4|X|$, Offline: $\approx 4|X|$;
Rounds: 2 (the first round requires only input $\langle \Delta\mathsf{N}\rangle_{2,3}$, 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 InsertLbl - Inserting Label (Used in Alg 14)

Input: $\langle \Delta N \rangle_{xor}^{P_1-P_2}$, $\langle L \rangle_{xor}^{P_1-P_2} = (L_1^{P_1}, L_2^{P_2})$, for $|\Delta N| = \tau$, $|L| = \ell$

Output: $\langle M \rangle_{xor}^{P_2-P_3} = (z_2^{P_2}, z_3^{P_3})$, for $M \in \text{array}[2^\tau](\ell)$ s.t. $M[\Delta N] = L$ and $M[t] = 0^\ell$ for $t \neq \Delta N$

Offline:

1: $(p, a, b)^{P_1P_2} \xleftarrow{\$} \text{array}[2^\tau](\ell)$, $(v, w)^{P_1P_2} \xleftarrow{\$} \{0,1\}^\tau$
2: $P_1 : \alpha_1 \xleftarrow{\$} \{0,1\}^\tau$, set $u_1 := \alpha_1 \oplus v$, $p^* := p \oplus a_{\text{rot}[u_1]}$
3: $P_2 : \beta_2 \xleftarrow{\$} \{0,1\}^\tau$, set $u_2 := \beta_2 \oplus w$, $z_2 := p \oplus b_{\text{rot}[u_2]}$
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: Reshare: $\langle \Delta N \rangle_{xor}^{P_1-P_2}, \alpha_1^{P_1} \rightarrow \alpha_2^{P_2}$ s.t. $(\alpha_1, \alpha_2) = \langle \Delta N \rangle_{xor}^{P_1-P_2}$
Reshare: $\langle \Delta N \rangle_{xor}^{P_1-P_2}, \beta_2^{P_2} \rightarrow \beta_1^{P_1}$ s.t. $(\beta_1, \beta_2) = \langle \Delta N \rangle_{xor}^{P_1-P_2}$
2: P_1 sends $s_1 = b^{\text{xor}[L_1 \,@\, \beta_1 \oplus w]}$ to P_3
i.e. $s_1 = b$ except $s_1[\beta_1 \oplus w] = b[\beta_1 \oplus w] \oplus L_1$
P_2 sends $s_2 = a^{\text{xor}[L_2 \,@\, \alpha_2 \oplus v]}$ to P_3
i.e. $s_2 = a$ except $s_2[\alpha_2 \oplus v] = a[\alpha_2 \oplus v] \oplus L_2$
3: P_3 outputs $z_3 := p^* \oplus (s_2)_{\text{rot}[u_1]} \oplus (s_1)_{\text{rot}[u_2]}$

Correctness:
Observe that $z_2 = p \oplus b_{\text{rot}[\beta_2 \oplus w]}$ and $p^* = p \oplus a_{\text{rot}[\alpha_1 \oplus v]}$.
Note that $(s_2)_{\text{rot}[u_1]} = (a^{\text{xor}[L_2 \,@\, \alpha_2 \oplus v]})_{\text{rot}[\alpha_1 \oplus v]}$
$= (a_{\text{rot}[\alpha_1 \oplus v]})^{\text{xor}[L_2 \,@\, (\alpha_2 \oplus v) \,\oplus\, (\alpha_1 \oplus v)]}$
$= (a_{\text{rot}[\alpha_1 \oplus v]})^{\text{xor}[L_2 \,@\, \Delta N]}$.
Likewise $(s_1)_{\text{rot}[u_2]} = (b_{\text{rot}[\beta_2 \oplus w]})^{\text{xor}[L_1 \,@\, \Delta N]}$.
It follows that
$z_3 = p^* \oplus (s_2)_{\text{rot}[u_1]} \oplus (s_1)_{\text{rot}[u_2]}$
$= p \oplus a_{\text{rot}[\alpha_1 \oplus v]} \oplus (a_{\text{rot}[\alpha_1 \oplus v]})^{\text{xor}[L_2 \,@\, \Delta N]} \oplus (b_{\text{rot}[\beta_2 \oplus w]})^{\text{xor}[L_1 \,@\, \Delta N]}$
By xor-ing z_2 and z_3 observer that pad p and rotated pads a and b cancel out and we get $M = z_2 \oplus z_3 = [0, \ldots, 0]^{\text{xor}[L \,@\, \Delta N]}$ where $[0, \ldots, 0]$ 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 $\langle \Delta N \rangle_{xor}^{P_1-P_2}$);
Security:
(1) For P_1, P_2: Let $(\Delta N_1^{P_1}, \Delta N_2^{P_2})$ denote input $\langle \Delta N \rangle_{xor}^{P_1-P_2}$. P_2 receives $\Delta N_1 \oplus \alpha_1$ and P_1 receives $\Delta N_2 \oplus \beta_2$ in in 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 indpendently 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 FlipFlag (Step 6, Alg 6)

Input: $\langle \text{fb} \rangle_{2,3}$, $\langle i \rangle_{2,1\text{-shift}}^{DE-C} = (i_1^C, i_2^{DE})$, for $\text{fb} \in \text{array}[n](1)$, $i \in Z_n$

Output: $\langle \text{fb}' \rangle_{2,3}$ s.t fb′ is the same as fb except $\text{fb}'[i] = \text{fb}[i] \oplus 1$

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

Bandwidth: $4n$; Rounds: 2;
Security: Protocol FlipFlag is secure if protocol Reshuffle is a secure computation of $\langle m \rangle_{xor}^{C-E}$ and Reshare produces fresh secret-sharing $\langle m \rangle_{2,3}$ from $\langle m \rangle_{xor}^{C-E}$.

Algorithm 17 Protocol Reshuffle (based on [9]) (Used in Alg 16)

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

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

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

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 $\langle y \rangle_{xor}^{C-E}$ 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.3 Protocols for Eviction

Algorithm 18 Protocol GC(circ) (Step 8, Alg 6, and Step 2, Alg 7)

Input: $\langle x \rangle_{xor}^{C-E} = (x_1^C, x_2^E)$, where x is the input of function computed by the circuit

Output: $(y, \overline{z})^D$, where $(y, z) = \text{circ}(x)$ for $x = x_1 \oplus x_2$. Note that output z is in the form of output wire keys $\overline{z}$

Offline: E sends to D garbling of circ′ for $\text{circ}'(x_1, x_2) = \text{circ}(x_1 \oplus x_2)$, and to C the wire key pairs corresponding to input x_1

1: C and E select input wire keys according to their respective input x_1^C, x_2^E and send resp. $\overline{x_1}^C$ and $\overline{x_2}^E$ to D
2: D computes $(y, \overline{z})$ by evaluating garbled circ′ on $(\overline{x_1}, \overline{x_2})$

Bandwidth: Online: $2\kappa|x|$, for security parameter κ, Offline: $4\kappa|\text{circ}|$;
Rounds: 1;
Security: Standard garbled circuit secure computation.

Algorithm 19 Protocol PermuteTarget (Step 9, Alg 6)

Param: Tree height d, security parameter κ.

Input: $\overline{[\sigma]}^{\mathsf{D}} \in \mathsf{array}[d, \log(d)](\kappa)$, $\pi^{\mathsf{CE}} \in \mathsf{perm}_d$, and E's Routing circuit output key pairs on $[\sigma]$ wires (which allow E to compute $\overline{[\sigma][i]}$ for each of d possible values of $[\sigma][i]$).

Output: $\sigma^{\circ\mathsf{D}} \in \mathsf{perm}_d$ for $[\sigma^\circ] = [\pi \cdot \sigma \cdot \pi^{-1}]$.

Offline:

1: D, E have access to hash function $H : \{0,1\}^{\log(d)\cdot\kappa} \to \{0,1\}^{\kappa}$.
2: E sets $keys \in \mathsf{array}[d, d, \log(d)](\kappa)$ s.t. $keys[i][j] = \overline{[\sigma][i]}$ for $[\sigma][i] = j$, for each $(i, j) \in \mathsf{Z}_d \times \mathsf{Z}_d$.
3: E picks $MK \xleftarrow{\$} \mathsf{array}[d, d](\log(d))$.
4: E sets $TB \in \mathsf{array}[d, d](\kappa + \log(d))$ s.t. each $TB[i] \in \mathsf{array}[d](\kappa + \log(d))$ is a sequence of d key,value pairs which bind keys $H(keys[i][j])$ to values $\pi(j) \oplus MK[i][j]$, i.e. $TB[i][j] = (H(keys[i][j]), \pi(j) \oplus MK[i][j])$. In another words, $TB[i]$ implements a look-up table which maps $H(keys[i][j])$ to $\pi(j) \oplus MK[i][j]$ for each j.
5: For every $i \in \mathsf{Z}_d$, E picks a random permutation in perm_d and uses it to permute the entries of both $TB[i]$ *and* $MK[i]$.
6: E picks $p, r \xleftarrow{\$} \mathsf{array}[d](\log(d))$, and computes $a = \pi(p \oplus r)$.
7: E sends TB, p, a to D and MK, r to C.

Online:

1: D initializes $I \in \mathsf{array}[d](\log(d))$. For every $i \in \mathsf{Z}_d$, D sets $[\sigma'][i] = TB[i](H(\overline{[\sigma][i]}))$, and sets $I[i] = j$ s.t. $H(\overline{[\sigma][i]})$ is the prefix of $TB[i][j]$. D then sends $z = [\sigma'] \oplus p$ and I to C.
2: C sets $m \in \mathsf{array}[d](\log(d))$ s.t. $m[i] = MK[i][I[i]]$ for every $i \in \mathsf{Z}_d$. C sends $g = \pi(z \oplus r \oplus m)$ to D
3: D outputs σ° for $[\sigma^\circ] = a \oplus g$.

$$[\sigma^\circ] = a \oplus g = \pi(p \oplus r) \oplus \pi([\sigma'] \oplus p \oplus r \oplus m) = \pi([\sigma'] \oplus m)$$
$$= \pi([(TB_{0,I_0} \oplus MK_{0,I_0}), \ldots, (TB_{d-1,I_{d-1}} \oplus MK_{d-1,I_{d-1}})])$$
$$= \pi([\pi([\sigma]_0), \ldots, \pi([\sigma]_{d-1})]) = \pi([\pi \cdot \sigma]) = [\pi \cdot \sigma \cdot \pi^{-1}]$$

Bandwidth: $3d\log(d)$; 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 $i \in \mathsf{Z}_d$, $TB[i]$ and $MK[i]$ are permuted by a random permutation in Z_d. Given D's input a (from pre-computation) and output $[\sigma^\circ]$, D's view g can be simulated as $a \oplus [\sigma^\circ]$. E receives nothing online.

Algorithm 20 Protocol PermuteIndex (Step 10, Alg 6)

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

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

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

Offline: $p^{\mathsf{ED}}, r^{\mathsf{EC}} \xleftarrow{\$} \mathsf{array}[d](w+1)$.
E computes and 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 $\mathsf{t}^\circ = a \oplus g$.

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

Bandwidth: $2d(w+1)$; Rounds: 2;
Security: C's view z is indistinguishable from random string because p is random and unknown to C. D's view g is indistinguishable from random string because ρ is random and unknown to D. E receives nothing online.

Algorithm 21 EM-Comp (Step 11, Alg 6)

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

Input: $[\sigma] \in \mathsf{array}[d](\log(d)), \mathsf{t} \in \mathsf{array}[d](w+1)$.

Output: $\mathsf{EM}^\circ \in \mathsf{perm}_{d\cdot(w+1)}$.

```
Set [EM°] ∈ array[d·(w+1)](log(d·(w+1)))
for r := 0 to d − 1 do
    tupleIndex := r·(w+1) + t[r]
    for c := 0 to w do
        currIndex := r·(w+1) + c
        if currIndex = tupleIndex then
            targetIndex := [σ][r]·(w+1) + t[[σ][r]]
            EM°[targetIndex] := currIndex
        else
            EM°[currIndex] := currIndex
        end if
    end for
end for
```

This is local computation.

Algorithm 22 Protocol XOT (Step 12, Alg 6, and Step 2, Alg 7)

Param: Positive integers n, k, l where $k \le n$.

Input: $\langle X \rangle_{\mathsf{xor}}^{\mathsf{C-E}} = (X_1^{\mathsf{C}}, X_2^{\mathsf{E}})$ and I^{D},
for $X \in \mathsf{array}[n](l)$, and $I \in \mathsf{array}[k](\log(n))$.

Output: $\langle Y \rangle_{\mathsf{xor}}^{\mathsf{C-E}} = (Y_1^{\mathsf{C}}, Y_2^{\mathsf{E}})$,
for $Y \in \mathsf{array}[k](l)$ s.t. $Y[t] = X[I[t]]$ for all $t \in \mathsf{Z}_k$
(Y contains k elements of X selected by indexes in array I)

Offline: $\pi_1^{\mathsf{CD}}, \pi_2^{\mathsf{ED}} \xleftarrow{\$} \mathsf{perm}_n$; $r^{\mathsf{CD}}, s^{\mathsf{ED}} \xleftarrow{\$} \mathsf{array}[n](l)$;
$\delta^{\mathsf{D}} \xleftarrow{\$} \mathsf{array}[k](l)$.

1: C sends array a to E, where $a[t] = X_1[\pi_1(t)] \oplus r[t]$ for $t \in \mathsf{Z}_n$.
E sends array b to C, where $b[t] = X_2[\pi_2(t)] \oplus s[t]$ for $t \in \mathsf{Z}_n$.
2: D sends arrays g, p to E, where $g[u] = \pi_1^{-1}(I[u]), p[u] = r[g[u]] \oplus \delta[u]$ for $u \in \mathsf{Z}_k$.
D sends arrays h, q to C, where $h[u] = \pi_2^{-1}(I[u]), q[u] = s[h[u]] \oplus \delta[u]$ for $u \in \mathsf{Z}_k$.
3: C sets Y_1 s.t. $Y_1[u] = b[h[u]] \oplus q[u]$ for $u \in \mathsf{Z}_k$.
E sets Y_2 s.t. $Y_2[u] = a[g[u]] \oplus p[u]$ for $u \in \mathsf{Z}_k$.

$$Y_2[u] = a[g[u]] \oplus p[u] = X_1[\pi_1(g[u])] \oplus r[g[u]] \oplus r[g[u]] \oplus \delta[u]$$
$$= X_1[\pi_1(g[u])] \oplus \delta[u] = X_1[I[u]] \oplus \delta[u],$$
$$Y_1[u] = X_2[I[u]] \oplus \delta[u],$$
$$Y[u] = Y_2[u] \oplus Y_1[u] = X_1[I[u]] \oplus X_2[I[u]] = X[I[u]]$$

Bandwidth: $2l(n + k + \log(n))$; Rounds: 1;
Security: C's view includes b, h, q, which are indistinguishable from random strings because s, π_2, δ are random and unknown to C. E's view is symmetric to C's, so it is indistinguishable for the same reason. D receives nothing online.

B ROUTING CIRCUIT COMPUTATION

In this section we explain the construction of the routing circuit Routing used in the eviction phase of the 3PC ORAM algorithm 3PC-ORAM-MainLoop (see Step 8 in Alg. 6, Section 4).

B.1 Main Routing Circuit

As we explain in Section 4.2, circuit Routing determines the eviction map by generating a deepest array (PrepareDeepest), computing the eviction $[\sigma]$ array from deepest (PrepareTarget), and making

the $[\sigma]$ array into a cycle (MakeCycle). Circuits PrepareDeepest and PrepareTarget are, with minor variations, the same circuits which implemented the original Circuit-ORAM eviction computation circuit CircORAM-Routing [24], but circuit MakeCycle is new. Because of some small differences in the implementation of PrepareDeepest and PrepareTarget, the combined size of circuit Routing is virtually identical to the size of CircORAM-Routing reported in [24]. We discuss these three circuits in the following subsections.

Algorithm 23 Circuit Routing (Used in Step 8, Alg 6)

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

Input: Wire keys for secret-sharing of tuples' full/empty bits fb $\in$ array$[d, w](1)$ and labels lbl $\in$ array$[d, w](d)$, path label L $\in \{0,1\}^d$, and masks $\delta \in$ array$[d](\log(w+1))$.

Output: $[\sigma] \in$ array$[d, \log(d)](\kappa)$ and $\mathrm{t}' \in$ array$[d](\log(w+1))$ for $\mathrm{t}' = \delta \oplus \mathrm{t}$, where σ and t are the (extended) Eviction Map permutation and the Tuple Index array computed by the Circuit-ORAM Eviction Logic.

1: (deepest, $j_{\mathsf{dp}}, j_{\mathsf{em}}, e$) :=PrepareDeepest(L, fb, lbl)
2: ($[\sigma]$, t′, $nTop, nBot, eTop, eBot$) :=
PrepareTarget(deepest, $j_{\mathsf{dp}}, j_{\mathsf{em}}, e, \delta$)
3: $[\sigma]$:=MakeCycle($[\sigma], nTop, nBot, eTop, eBot$)

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

B.2 Prepare Array deepest

Algorithm PrepareDeepest in Alg. 24, based on the same name algorithm in [24], outputs an array deepest where deepest$[i] < i$ 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 deepest$[i] = \perp$.) In addition, PrepareDeepest outputs three other arrays $j_{\mathsf{dp}}, j_{\mathsf{em}}, e$, where $j_{\mathsf{dp}}[i]$ 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 $j_{\mathsf{em}}[i]$ is the index of that empty tuple. (If $e[i] = 0$ then $j_{\mathsf{em}}[i]$ is meaningless.)

Find the Deepest and Empty Tuples. Algorithm FDAE in Alg. 25 (which stands for FindDeepestAndEmpty) is adopted from [24], and it is a sub-procedure of Alg. 24 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_{dp} 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_{em} 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 $(j_{\mathsf{dp}}, l') = (0, i)$, and if there is no empty tuple then $(j_{\mathsf{em}}, e) = (0, 0)$.

Line 1: j_{dp} stores the index and l the target-depth of the first "deepest tuple", i.e. the tuple which can go deepest along the path to L. j_{dp} 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 zeroes. j_{em}, 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 zeroes 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 if and only if lz' has more leading 0's.

Line 6-8: If the new tuple is non-empty, i.e. 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_{dp} and target-depth l.

Algorithm 24 Circuit PrepareDeepest [24] (Used in Alg. 23)

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

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

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

1: deepest := $[\perp, \perp, ..., \perp]$, $src := \perp$, $goal := -1$
2: **for** $i := 0$ **to** $d-1$ **do** ▷ cycle: d
3: **if** $goal \geq i$ **then** ▷ cost: $log(d)$
4: deepest$[i] := src$ ▷ cost: $log(d)$
5: **end if**
6: $(l, j_{\mathsf{dp}}[i], j_{\mathsf{em}}[i], e[i]) :=$ FDAE$(i, \mathrm{L}, \mathrm{fb}[i], \mathrm{lbl}[i])$ ▷ cost: Alg 25
7: **if** $l > goal$ **then** ▷ cost: $log(d)$
8: $goal := l$ ▷ cost: $log(d)$
9: $src := i$ ▷ cost: $log(d)$
10: **end if**
11: **end for**

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

Algorithm 25 Circuit FDAE [24] (Used in Alg. 24)

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

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

Output: $l' \in \mathsf{Z}_d, j_{\mathsf{dp}}, j_{\mathsf{em}} \in \mathsf{Z}_{w+1}, e \in \{0,1\}$, where l' is the deepest level index, $j_{\mathsf{dp}}, j_{\mathsf{em}}$ 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 or not.

1: $l := 0^i|1^{d-i-1}$; $j_{\mathsf{dp}}, j_{\mathsf{em}}, et := 0$
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** fb$[j] = 1$ **and** $lz' < l$ **then** ▷ cost: d
6: $j_{\mathsf{dp}} := j$ ▷ cost: log w
7: $l := lz'$ ▷ cost: d
8: **else if** fb$[j] = 0$ **and** $e = 0$ **then**
9: $j_{\mathsf{em}} := j$ ▷ cost: log w
10: $e := 1$
11: **end if**
12: **end for**
13: $l' :=$ number of leading 0s in l ▷ cost: $d \cdot \log(d)$

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

Line 8-11: If the new tuple is empty, update j_{em} and e.
Line 13: Compute the deepest level number in an integer format by counting the number of leading zeros in l.

Protocol UpdateRoot, Alg. 7, needs circuits RFE and RFD, shown in Alg. 27 and 26, which stand for resp. RootFindEmpty and RootFindDeepest. Circuits RFE and RFD are variants of circuit FDAE: They also find the deepest tuple and the empty tuple in a list of tuples, but they find each tuple type separately, and with the following changes: (1) They work only on the root bucket (= stash), of size s, typically significantly larger than non-root bucket size w; (2) Circuit RFD does not output index l' of the deepest level; (3) Circuit RFE does not output the empty-tuple-found flag e: If there is no empty tuple in the root then there is a bucket overflow and the 3PC ORAM protocol will fail by overwriting some entry (or by throwing an "overflow" flag); (4) Finally, RFE and RFD perform a search after an offset Δ, i.e. they do not scan bucket entries from position 0 to position $s-1$, but from position Δ to $s-1$ and then from 0 to $\Delta-1$. Since both algorithms output the position of the searched-for type (either empty or deepest), this way we ensure that they find the first position after offset Δ.

Algorithm 26 Circuit RFD (Used in Alg. 7)

Param: Tree height d, list size (=root bucket size) s
Input: Path label $\mathsf{L} \in \{0,1\}^d$, offset $\Delta \in \mathsf{Z}_s$, full/empty flags $\mathsf{fb} \in \mathsf{array}[s](1)$, labels $\mathsf{lbl} \in \mathsf{array}[s](d)$
Output: $j \in \mathsf{Z}_s$, index of the first deepest tuple after Δ in the list

1: $l := 1^d$; $j := 0$
2: **for** $i := \Delta$ **to** $s-1$ **and then** $i := 0$ **to** $\Delta-1$ **do** ▷ cycle: s
3: $lz := \mathsf{lbl}[i] \oplus \mathsf{L}$
4: $lz' :=$ set all bits after the first bit 1 in lz to 1 ▷ cost: d
5: **if** $\mathsf{fb}[i] = 1$ **and** $lz' < l$ **then** ▷ cost: d
6: $j := i$ ▷ cost: $\log s$
7: $l := lz'$ ▷ cost: d
8: **end if**
9: **end for**

Circuit cost: $s(3d + \log s)$

Algorithm 27 Circuit RFE (Used in Alg. 7)

Param: Tree height d, list size (=root bucket size) s
Input: Offset $\Delta \in \mathsf{Z}_s$, full/empty flags $\mathsf{fb} \in \mathsf{array}[s](1)$
Output: $j \in \mathsf{Z}_s$, index of the first empty tuple after Δ (0 if none)

1: $j := 0$, $e := 0$
2: **for** $i := \Delta$ **to** $s-1$ **and then** $i := 0$ **to** $\Delta-1$ **do** ▷ cycle: s
3: **if** $\mathsf{fb}[i] = 0$ **and** $e = 0$ **then**
4: $j := i$ ▷ cost: $\log s$
5: $e := 1$
6: **end if**
7: **end for**

Circuit cost: $s \log s$

B.3 Prepare Arrays $[\sigma]$ and t

Algorithm PrepareTarget in Alg. 28 is an extended version of the corresponding algorithm in [24], which determines the final eviction pattern. PrepareTarget outputs a $[\sigma]$ array which contains the same eviction movement as in [24], 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 $j_{\mathsf{dp}}[i], j_{\mathsf{em}}[i]$, or w (fake/empty tuple index) depending on what kind of eviction

Algorithm 28 Circuit PrepareTarget (following [24]) (Used in Alg. 23)

Param: Tree height d, non-root bucket size w
Input: deepest $\in \mathsf{array}[d](\log(d))$,
$j_{\mathsf{dp}}, j_{\mathsf{em}} \in \mathsf{array}[d](\log(w+1))$,
$e \in \mathsf{array}[d](1)$,
$\delta \in \mathsf{array}[d](\log(w+1))$
Output: $[\sigma] \in \mathsf{array}[d](\log(d))$,
$\mathsf{t}' \in \mathsf{array}[d](\log(w+1))$,
$nTop, nBot, eTop, eBot \in \mathsf{Z}_d$

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

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

movement level i is doing. And this t will be finally masked by δ so the real indices will be hidden to D.

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.
Line 2: 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 3: $[\sigma][i]$ 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 5-6: If we have reached 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 7: We are evicting a full tuple in this case, so we should update $t[i]$ as the full tuple index $j_{\mathsf{dp}}[i]$.
Line 9-12: If deepest$[i] = \perp$ 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 15: 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 16-18: Because of Line 9, we have a gap tail level on record, so we update $[\sigma][i]$ to add this gap jump, and thus should use an empty tuple index $j_{\mathsf{em}}[i]$ as $t[i]$.
Line 20-22: 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 6 section and update $[\sigma]$.
Line 23-25: We update $eTop$ every time we find a new eviction jump; we update $eBot$ only for the first time.
Line 26: At the bottom level where a tuple will be evicted to, we must choose an empty tuple spot to accept the incoming tuple.
Line 30-34: If the current level i does not have any movement during the eviction, then we will set the fake empty tuple index w as $t[i]$. Also we will update $nTop$ and $nBot$ if necessary.
Line 37: Mask each $t[i]$ with $\delta[i]$ so the real value is hidden.

B.4 Making the Eviction Map into A Cycle

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

Line 1: $nPrev$ keeps track of the one previous level during the linear scan that does not have tuple movement.
Line 3-6: 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 need to 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 7-8: 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 9: This just 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 10: We are at the top level where there is no real eviction.
Line 11-12: 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 13-14: 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 16-17: 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 19: 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.

Algorithm 29 Circuit MakeCycle (Used in Alg. 23)

Param: Tree height d.
Input: $[\sigma] \in \text{array}[d](\log(d))$,
$nTop, nBot, eTop, eBot \in \mathsf{Z}_d$
Output: $[\sigma] \in \text{array}[d](\log(d))$

```
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:          end if
7:      else if i = eBot then
8:          [σ][i] := nBot                        ▷ cost: log(d)
9:      else if [σ][i] = ⊥ then                   ▷ cost: log(d)
10:         if i = nTop then                      ▷ cost: log(d)
11:             if eTop = ⊥ then                  ▷ cost: log(d)
12:                 [σ][i] := nBot                ▷ cost: log(d)
13:             else
14:                 [σ][i] := eTop                ▷ cost: log(d)
15:             end if
16:         else
17:             [σ][i] := nPrev                   ▷ cost: log(d)
18:         end if
19:         nPrev := i                            ▷ cost: log(d)
20:     end if
21: end for
```

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

C ADDITIONAL PERFORMANCE DISCUSSION

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 m 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 m 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 dominating turning point should be around/before D = 1KB according to Fig. 12.

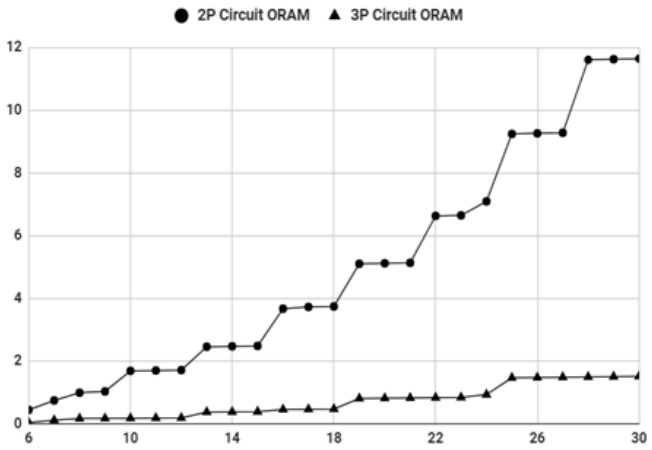

Figure 11: Online Bandwidth (MB) vs m (bits), for D = 4 bytes

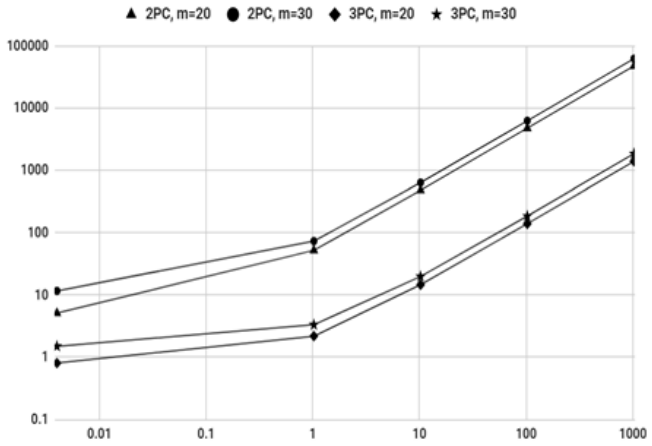

Figure 12: Online Bandwidth (MB) vs D (KB)

CPU Cost Components. In Fig. 13 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

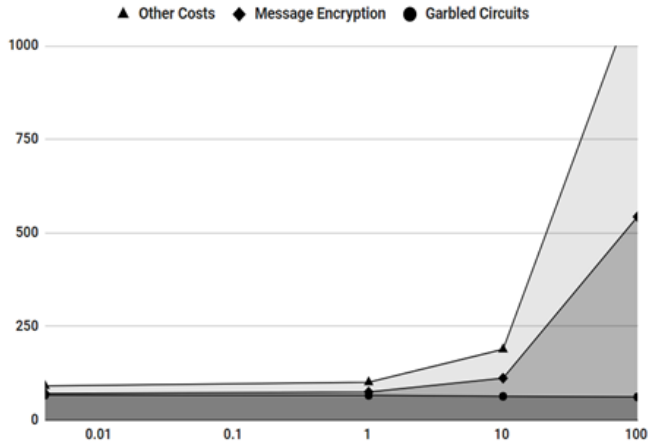

Figure 13: Online CPU Time Components(ms) vs D(KB)

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 τ = 5 and 6 can be good choices in practice, where τ = 6 actually has slight advantages on circuit size, CPU time, and Access WC time. Therefore, we picked τ = 6 as our optimal parameter choice, and produced the performance data accordingly. 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.