UNIVERSITY OF CALIFORNIA,
IRVINE

Low Bandwidth and Latency Secure Computation Oblivious RAM with Three Parties

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Boyang Wei

Dissertation Committee:
Professor Stanislaw Jarecki, Chair
Professor Michael B. Dillencourt
Professor Michael T. Goodrich

2018

DEDICATION

To my parents, for their love from the other side of the earth.

To Nan, my love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

First and foremost, I would like to express the deepest appreciation to my Ph.D. advisor, Professor Stanislaw Jarecki. Professor Jarecki has great passion in cryptography research and education. He has taught me not only how to do research effectively but also how to think with open-minds. Without his generous trust, patience, support, and guidance, I would not be able to make achievements in this field and accomplish this degree. I am very grateful of having the opportunity to work with you, and will never forget all you have done for me.

I would also like to thank my defense committee members, Professors Stanislaw Jarecki, Michael Dillencourt, Michael Goodrich. Thank you very much for your time, support, and guidance.

In addition, many thanks to my collaborators and friends, Stanislaw Jarecki, Sky Faber, Sotirios Kentros, Jonathan Katz, Mariana Raykova, Xiao Wang, Jiayu Xu, Di Yang, Wei Wang, Fei Yu. Your efforts, support, and help were critical in my development.

Also, thanks to the Dean's Fellowship from UCI Donald Bren School of Information and Computer Sciences, NSF Secure and Trustworthy Cyberspace funding, which support my study. And thanks to Springer Nature for permission to incorporate my publications into this dissertation.

CURRICULUM VITAE

Boyang Wei

EDUCATION

Doctor of Philosophy in Computer Science — **2018**
University of California, Irvine — *Irvine, California*

Master of Science in Computer Science — **2018**
University of California, Irvine — *Irvine, California*

Bachelor of Arts in Computer Science and Mathematics — **2013**
St. Olaf College — *Northfield, Minnesota*

RESEARCH EXPERIENCE

Graduate Research Assistant — **2013–2018**
University of California, Irvine — *Irvine, California*

TEACHING EXPERIENCE

TA for Programming with Software Libraries (ICS 32) — **Spring 2018**
University of California, Irvine — *Irvine, California*

TA for Programming with Software Libraries (ICS 32) — **Winter 2018**
University of California, Irvine — *Irvine, California*

Reader for Introduction to Cryptography (CS 167) — **Winter 2017**
University of California, Irvine — *Irvine, California*

TA for Programming with Software Libraries (ICS 32) — **Spring 2016**
University of California, Irvine — *Irvine, California*

Reader for Programming in Java (ICS 45J) — **Fall 2016**
University of California, Irvine — *Irvine, California*

TA for Programming with Software Libraries (ICS 32) — **Spring 2015**
University of California, Irvine — *Irvine, California*

TA for Programming with Software Libraries (ICS 32) — **Winter 2015**
University of California, Irvine — *Irvine, California*

TA for Programming in C/C++ (ICS 45C) — **Fall 2015**
University of California, Irvine — *Irvine, California*

TA for Principles in System Design (ICS 53) — **Spring 2014**
University of California, Irvine — *Irvine, California*

INTERNSHIP EXPERIENCE

Software Engineer Intern **Summer 2017**
Google *Mountain View, California*

REFEREED CONFERENCE PUBLICATIONS

3PC ORAM with Low Latency, Low Bandwidth, and Fast Batch Retrieval **2018**
Applied Cryptography and Network Security (ACNS)

Three-Party ORAM for Secure Computation **2015**
Theory and Application of Cryptology and Information Security (Asiacrypt)

SOFTWARE

3pc-dpforam `https://github.com/Boyoung-/3pc-dpforam`
3PC-DPF-ORAM C++ implementation

3pc-dpforam-java `https://github.com/Boyoung-/3pc-dpforam-java`
3PC-DPF-ORAM Java implementation

sqrt-oram-3pc `https://github.com/Boyoung-/sqrt-oram-3pc`
3PC-Sqrt-ORAM Java implementation

circuit-oram-3pc `https://github.com/Boyoung-/circuit-oram-3pc`
3PC-Circuit-ORAM Java implementation

oram3pc `https://github.com/Boyoung-/oram3pc`
3PC-ORAM Java implementation

ABSTRACT OF THE DISSERTATION

Low Bandwidth and Latency Secure Computation Oblivious RAM with Three Parties

By

Boyang Wei

Doctor of Philosophy in Computer Science

University of California, Irvine, 2018

Professor Stanislaw Jarecki, Chair

An Oblivious RAM (ORAM) protocol allows a client to access memory outsourced at the server without leaking the access pattern. A related notion is Multi-Party Computation ORAM (MPC-ORAM), which is a protocol that can implement the RAM functionality for secure computation of any RAM programs on large data, e.g., MPC processing of database queries on a secret-shared database. MPC-ORAM can be constructed from any (client-server) ORAM by implementing the ORAM client algorithm with MPC protocols. However, efficient ORAM does not necessarily translate to efficient MPC-ORAM. Most previous work constructed efficient MPC-ORAM in the two-party secure computation (2PC) setting, but since secure computation of many functions is more efficient in the three-party honest-majority setting than in the two-party setting, it is natural to ask if the cost of an MPC-ORAM scheme can be reduced if one is willing to use three servers instead of two and assumes an honest majority. In this study, we show four constructions of MPC-ORAM in the three-party setting (3PC-ORAM) with honest majority, using customized 3PC protocols for performance optimization, which can make the resulting 3PC-ORAM schemes to be orders of magnitude more efficient than the current best 2PC-ORAM schemes. These improvements help to achieve practical and efficient performance results, and make MPC-ORAM more suitable for real world applications.

Chapter 1

Introduction

Oblivious RAM (ORAM) is a protocol between client and server which allows the client to access outsourced memory at server while hiding the accessed memory locations, i.e., the access pattern to data. The reason of using ORAM is that sensitive information may be leaked if the data access pattern can be monitored by the adversary. Starting from the seminal work of ORAM by Goldreich and Ostrovsky [22], numerous improvements on ORAM techniques have been proposed to achieve more efficient asymptotic and concrete costs on computation, communication bandwidth, rounds, and storage overhead [8, 23, 24, 32, 38, 41, 42, 48].

The above classic client-server ORAM provides secure outsourced memory for a single client. This presents some issue: If there are multiple clients holding the same secrets to access the memory and some of the clients are corrupted, then there is no security guarantee for the rest of the clients. Multi-client ORAM has been studied to solve this problem [33, 34, 9]. However, a common issue for multi-client ORAM is that the number of clients does not scale.

In this study I focus on another solution to the above problem, which is ORAM in the secure computation setting (SC-ORAM), or Multi-Party Computation ORAM (MPC-ORAM).

MPC-ORAM securely performs memory access on shared memory index and array. Because sensitive information are secret-shared in MPC-ORAM, and no secrets like the encryption master key used in client-server ORAM are needed and stored at the client, security is still preserved for non-compromised clients. MPC-ORAM can provide oblivious RAM access for secure computation of any RAM programs, and is especially well suited for information retrieval algorithms which rely heavily on the RAM model.

The efficiency of MPC-ORAM is crucial for the overall efficiency of RAM-based secure computation. As pointed out by Ostrovsky et al. [37], MPC-ORAM can be constructed generically by implementing the client algorithm of the ORAM with MPC protocols. However, because algorithms translated into MPC version in general become more expensive, the generic MPC-ORAM implementations can be practically inefficient. Many work [46, 44, 51, 15] have been done to find and design MPC-friendly ORAM client algorithm in order to improve efficiency of MPC-ORAM in the two-party setting (i.e., 2PC-ORAM) in terms of bandwidth, computation, and rounds. However, we observe that for MPC protocols, often the same protocol can be done more efficiently with three-party than with two-party. Thus, we conduct the study on 3PC-ORAM, with the goal to construct more efficient MPC-ORAM both asymptotically and practically.

1.1 Definitions

Oblivious RAM. Let the RAM data array M have length n, and each data record have B bits. The RAM M is accessed by $\mathsf{op}(i, \mathsf{rec}')$, where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$ is the access operation, $i \in \{1, ..., n\}$ is the memory access location, and rec' is the new record to be inserted to M. Access $\mathsf{read}(i, \mathsf{rec}')$ returns record $\mathsf{rec} = \mathsf{M}[i]$ and keep M unchanged, while access $\mathsf{write}(i, \mathsf{rec}')$ returns $\perp$ but set $\mathsf{M}[i] := \mathsf{rec}'$.

Correctness: For a sequence of ℓ accesses $[\mathsf{op}_1(i_1, \mathsf{rec}'_1), \mathsf{op}_2(i_2, \mathsf{rec}'_2), ..., \mathsf{op}_\ell(i_\ell, \mathsf{rec}'_\ell)]$, and for all $t \in \{1, ..., \ell\}$, if $\mathsf{op}_t = \mathsf{read}$, then the result of $\mathsf{op}_t(i_t, \mathsf{rec}'_t)$ is equal to the last value written to location i_t on M.

Security: For any two access sequences x and y with same length ℓ (each of which has form $[\mathsf{op}_1(i_1, \mathsf{rec}'_1), \mathsf{op}_2(i_2, \mathsf{rec}'_2), ..., \mathsf{op}_\ell(i_\ell, \mathsf{rec}'_\ell)]$), x is computationally indistinguishable from y.

MPC-ORAM. Secure Multi-Party Computation allows m parties, each of which holding some secret input x_i, to jointly compute some function $y = f(x_1, x_2, ..., x_m)$, and guarantees that no party learns information more than what is revealed from the output y and each of their secret input. And Multi-Party Computation Oblivious RAM (MPC-ORAM) is a protocol which lets m parties implement access operation to a secret-shared RAM and hides both the memory access locations and data records at the same time. The correctness of MPC-ORAM follows naturally from the underlying ORAM, and the security guarantee holds as long as no more than a threshold t number of parties are corrupted.

3PC-ORAM. In this study I focus on the design of 3PC-ORAM, i.e., MPC-ORAM in the three-party setting that $m = 3$, and the security assumption is that at most $t = 1$ party will be corrupted. In other words, I consider 3PC-ORAM in the honest majority setting. And the adversary model assumed in this study is honest-but-curious, or semi-honest adversary. In such model, all the m participating parties will follow the executing MPC-ORAM protocol honestly, but up to t parties would collude and be interested in finding out more information other than their given input and output.

1.2 Problem Statement

This study addresses the problem of large performance overhead of MPC-ORAM in practice. MPC-ORAM can be a good solution for many real world applications, including constructing

oblivious data structures like queues and heaps, designing oblivious versions of widely used algorithms like Dijkstra's Algorithm and Stable Matching, and building applications like search-able encryption system and oblivious database [30, 47, 43, 35, 17, 52]. However, often because of the large overhead on bandwidth, computation, or round complexity, MPC-ORAM does not have competitive practical performance. Therefore, the goal of this study is to design efficient 3PC-ORAM schemes (MPC-ORAM with efficiency advantage of the three-party setting) in practice, which can be used to implement real world applications and maintain data privacy with small overhead.

In this study we consider communication bandwidth, round, and computation cost as the efficiency measurements of MPC-ORAM (as observed by Wang et al. [44], circuit size is also a good performance metric especially for 2PC-ORAM, because generic 2PC garbled circuit is often the bottleneck and affects bandwidth and computation, but with 3PC we may be able to avoid the use of 2PC garbled circuit). The theoretical focus of this study is to improve the asymptotic costs of the MPC-ORAM with 3PC. And the practical focus of this study is to validate the theoretical improvements in practice, compare the trade-offs between variables, and try to further improve the concrete performance.

1.3 Road-map

In Chapter 2 I give a summary of relevant ORAM work and state-of-art MPC-ORAM literatures related to this 3PC-ORAM study, which covers well-known ORAM constructions, cryptographic primitives, and generic MPC protocols commonly used for constructing MPC ORAM. Chapter 3 is an overview of methods I used for conducting this 3PC-ORAM study. Chapter 4 describes the first 3PC-ORAM scheme we designed with customized 3PC protocols to achieve better efficiency over 2PC-ORAM at that time. Chapter 5 explains the our 3PC-Circuit-ORAM construction, which has the best asymptotic complexity and improves

concrete efficiency over previous schemes. Chapter 6 includes our 3PC-Sqrt-ORAM scheme, which has the best concrete performance for small memory array size. Chapter 7 talks about our 3PC-DPF-ORAM, which is extremely efficient on bandwidth, and also has better computation cost for a wide range of memory sizes. Chapter 8 reports our results and findings overall, and concludes this study.

Chapter 2

Literature Review

2.1 First ORAM, and Hierarchical Construction

Oblivious RAM was first introduced by Goldreich and Ostrovsky [22] for the software protection issue. Software, as an intellectual property, is often difficult and expensive to produce, but relatively easy to copy and steal. Goldreich et al. [22] argued that the software protection problem could be reduced to the problem of preventing the adversary from inspecting the contents of memory as well as the communication between CPU and memory during software execution. And to solve this problem, they proposed the simulation of arbitrary RAMs by oblivious RAM to hide the memory access pattern.

An naive approach for oblivious RAM is to visit every memory cell once per each access so the access pattern is always the same and thus oblivious. However, the overhead for such approach is not practical. Thus, the question is, can the oblivious RAM be achieved with less cost.

Goldreich et al. [22] showed in their study that they could do a simulation of any RAM program by oblivious RAM with only polylogarithmic overhead $O(\text{polylog}(n))$ of the RAM size n, where the overhead means the additional oblivious RAM cost comparing to the cost of the underlying RAM program. Goldreich et al. illustrated their oblivious RAM by first showing a "Square-Root" scheme, with $O(\sqrt{n})$ overhead, to give some intuition for the later more efficient $O(\text{polylog}(n))$ solution.

2.1.1 "Square-Root" ORAM

The general idea of this "Square-Root" ORAM is as following. Besides the main memory array with size n, additionally there are $\sqrt{n}$ dummy memory blocks appended to the main memory and a data structure called *stash* with size $\sqrt{n}$ which temporarily stores the accessed memory records so far. The main memory will be randomly permuted by some permutation π. For each ORAM access, the stash will first be linearly scanned to check existence of the expected memory record. If not found, the record will be retrieved by lookup on the permuted main memory using permutation π. Otherwise, a fake lookup will be perform on some us-used dummy main memory block. And in any of the above two cases, the block from lookup on the main memory will be appended to the stash. Once after $\sqrt{n}$ accesses that the stash becomes full and with the possibility that dummy memory blocks are all used, a re-initialization of the ORAM will be performed, that new dummy blocks will be re-created, and real data records on stash will be put back to the main memory, which will be randomly permuted again by some new permutation π'.

The "Square-Root" construction is an ORAM because, first, the adversary monitoring the RAM activity cannot tell the real order of memory locations that are being accessed because of the random permutation, and second, the adversary cannot tell if a particular memory location is accessed multiple times, because the access pattern is that each record from the

permuted main memory will only be accessed at most once, and same record that is accessed multiple times will be on stash and retrieved with oblivious linear scan.

"Square-Root" ORAM has an amortized access overhead $O(\sqrt{n})$. This is because, first, each access requires to linearly scan the stash which has size $\sqrt{n}$, and second, the periodic shuffling cost using Batcher's Sorting Network is $O(n \cdot \log^2(n))$. Amortizing this re-initialization cost over $\sqrt{n}$ accesses, the overhead of "Square-Root" ORAM is $O(\sqrt{n} \cdot \log^2(n)) = O(\sqrt{n})$.

2.1.2 Hierarchical ORAM

Goldreich et al. [22] also presented a hierarchical ORAM solution based on the "Square-Root" ORAM idea, and reduced access overhead from $O(\sqrt{n})$ to $O(\text{polylog}(n))$. The idea of the efficiency improvement was to decrease the amortized cost of the periodic permuting of the main memory by storing the memory array as a recursive $\log(n)$-level hierarchy of permuting buffers (implemented by hash tables) with increasing sizes, to which the permuting frequency is inversely proportional. The idea of the access algorithm is still similar to the "Square-Root" scheme, that we either find the memory record on stash or at some unvisited location. The difference in the hierarchical ORAM is that when we lookup record on some level L, all levels before L kind of act like the stash for L.

The general steps of hierarchical ORAM access are as following. The first level of the hierarchy will be scanned, and if record not found, then on the second level using its hash function, we can locate one hash bucket and scan $O(\log(n))$ blocks in the bucket to keep looking; if the record is found in first level scan, then a fake lookup will be performed on the second level (and the following levels). This whole lookup operation on the first and second levels is like a "stash scan" for the third level, that based on whether record has been found or not before the third level, real or fake hash bucket lookup will be performed on the third level. And such lookup will be applied recursively on all levels, which makes a

single ORAM access. After each access, the (new) record will be inserted back to the first level, like the idea of putting the record back onto stash in "Square-Root" scheme. And if at this time any level is full, then that level will be re-initialized by randomly shuffling its data records into the next level, and the hash tables of these levels will be re-built, which is like the "Square-Root" idea of shuffling records from stash back to main memory, rebuilding stash, and re-shuffling main memory.

This hierarchical solution preserves obliviousness as long as the hashing is oblivious so that the lookup location has no correlation to the real memory address, and the access pattern is random for particular memory location being accessed multiple times because accessed records will be on "stash" and "scanned", while the access pattern is supplemented with fake lookup. The cost of overhead for this hierarchical ORAM is $O(\text{polylog}(n))$, as for each access, on every $\log(n)$ level, the hash bucket scan cost is $O(\log(n))$. And the cost of re-hashing (oblivious sorting) of each level amortized over its level size is also $O(\text{polylog}(n))$.

2.1.3 The $\Omega(\log(n))$ Asymptotic Lower-bound

Besides the cornerstone theory on oblivious RAM and two ORAM constructions, Goldreich et al. [22] also showed in their paper that for memory array size n, the lower-bound for ORAM access overhead is $\Omega(\log(n))$. This is a powerful lower-bound, which is for arbitrary memory block sizes and applies to both online and offline phrases of ORAM implementations as well as other additional performance metrics used in MPC setting such as circuit size, which will be explained more in the later chapters.

2.2 Binary-Tree ORAM

Since the hierarchical ORAM proposed by Goldreich et al. [22], many subsequent work [8, 48, 23, 24, 32, 38] had been done to improve the construction. However, since these work inherited the hierarchical design from [22], they relied on expensive primitive operations like oblivious hashing, sorting, and periodic reshuffling. And people started to wonder whether there could be alternative constructions of ORAM that could avoid expensive operations like oblivious sorting and periodic reshuffling, and achieve $O(\text{polylog}(n))$ cost even in the worst case comparing to the amortized $O(\text{polylog}(n))$ cost of [22].

The answer to the above question appeared in 2011, that Shi et al. [41] proposed a novel ORAM construction called binary-tree ORAM. Binary-tree ORAM's construction is fundamentally different from the previous hierarchical ORAM construction, that in general the memory is organized and stored in a binary tree, where each tree node is treated as a bucket which can hold some number of data records. Data records will be moved and relocated obliviously along the tree paths to preserve oblivious access pattern. And, comparing to the hierarchical ORAM, the binary-tree ORAM is conceptually simpler and eliminates the need to perform expensive operations like oblivious sorting and periodic reshuffling, which leads to more practical results.

Since then, many work [14, 19, 42, 46, 40] has been done to continue the study and improve the design of binary-tree ORAM. And Path-ORAM [42], is one of the binary-tree based ORAMs well-known for its simplicity and practicality. In the next section, I will explain the idea of binary-tree ORAM based on Path-ORAM.

2.3 Path-ORAM

Using the same notation in Section 1.1, let the memory array M has length n, and each data record has size B. On the server side, the memory array is stored in a binary tree with height $\log(n)$. Each tree node is a *bucket* that can store w number of *tuples*, and each tuple is in the format of $(\mathsf{fb}, \mathsf{adr}, \mathsf{lbl}, \mathsf{rec})$, where fb is a full/empty bit flag indicating whether this tuple contains real or dummy data record, adr is the memory location address of the data record, lbl is a label indicating which binary tree path this tuple resides on, and rec stores the data record. Notice that this binary-tree structure can store $2nw$ tuples, but up to n tuples in the binary tree will be tuples containing real records, and the rest tuples will be dummy ones.

On the client side, the client stores a position map, which is a two column table that maps memory location address adr to binary tree path label lbl. In addition a stash is also stored at the client, and the stash is an array of tuples with size s, a temporary storage for tuples accessed before and not yet inserted back to the server side binary tree.

2.3.1 Access

The ORAM access in such binary-tree construction can be viewed as three steps, Retrieval, PostProcess, and Eviction:

1. **Retrieval**. Given memory location address adr, Retrieval allows the client to retrieve data record $\mathsf{M}[\mathsf{adr}]$ from the server. This is done as following: the client searches in its position map PM with address adr to get binary tree path label $\mathsf{lbl} = \mathsf{PM}[\mathsf{adr}]$. The client sends lbl to the server, who retrieves every tuples on the tree path corresponding to the path label lbl from root to leaf, and send this array of tuples path back to the client. The client will then decrypt the path, linear scan the stash and path, and the

invariant of the Path-ORAM algorithm guarantees that there is one and only one real tuple among stash and path such that the tuple's full/empty bit fb is 1 and its address is adr. And this tuple contains the data record rec = M[adr] the client wants to access.

2. **PostProcess**. This is the process for the client to prepare writing the (updated) tuple just retrieved back to the ORAM. The rec of the retrieved tuple can be updated directly by the client with the new record rec′. Then the client will pick a new random label lbl′, and update both the label field in the tuple and the position map by setting PM[adr] = lbl′. The purpose of pick the new random label is to obliviously relocate the retrieved tuple in the final Eviction step so the access pattern can be oblivious. The client puts the updated tuple back to stash/path where it was retrieved.

3. **Eviction**. This step evicts tuples from stash and path along the path according to the matching degree of each tuple's label and the path label, so that tuples can be relocated obliviously while maintaining the invariant that tuple with label lbl must reside on the tree path with label lbl. Specifically, for each tuple with label lbl_t, the buckets on path that this tuple can reside in are the buckets from root to the lowest common ancestor bucket of lbl_t and $\mathsf{lbl}_{\mathsf{path}}$. And the client can relocate this tuple to the deepest such bucket that is not yet full (recall that each bucket has a capacity w). If during this process some tuples cannot be relocated to the path from stash due to all satisfied buckets are full, then these tuples will be kept on the stash. Finally, once the Eviction is done, the client will re-encrypt the path and send the path back to the server, who writes this new evicted path back to the binary-tree path where it was retrieved.

2.3.2 Recursive Position Map

For the above access algorithm, the client needs to store a position map for finding the path label associated with the access memory address. If the memory stores n data records, and

each data record's binary-tree path label has size $\log(n)$ (because the tree height is $\log(n)$), then the size of the position map is $n \cdot \log(n)$, which can be a large storage for the client. For example, for a memory storing $n = 2^{30}$ records, the position map size is already $2^{30} \cdot 30$ bits which is about 4GB.

This large client storage problem can be solved by building this position map as a recursive ORAM [41]. The idea is, consider a packing parameter τ, that if we pack 2^τ path labels as a data record, then the position map storing n path labels can be stored using a new ORAM for memory size $n/2^\tau$ with each data record size $2^\tau \cdot \log(n)$. Now this ORAM will have its own position map storing $n/2^\tau$ labels, and if we apply the above packing algorithm again, we can pack and store this position map with another ORAM, which gives an even smaller position map storing only $n/2^{2\tau}$ labels. So if we apply this algorithm recursively, eventually we can have an $O(1)$ size position map stored at the client.

2.3.3 Complexity

Unlike hierarchical ORAM, binary-tree based Path-ORAM does not have expensive periodic re-initialization. As the bucket size w of Path-ORAM can be a small constant [42], the access overhead complexity of Path-ORAM is $O(\log^3(n))$, because each accessed tree path contains $O(\log(n))$ tuples, each tuple stores adr and lbl which are $O(\log(n))$ size, and there are $\log(n)$ recursive levels. One can see from the above Path-ORAM algorithm that it is simpler than the hierarchical ORAM algorithm described in [22]. And because the Path-ORAM construction does not require sophisticated and expensive operations like oblivious sorting and hashing, it has much better practical performance than the previous hierarchical ORAM constructions.

2.4 Yao's Garbled Circuit

With efficient client-server ORAM constructions, people started to think about building efficient 2PC-ORAMs based on efficient client-server ORAMs. In the following sections I will discuss the state-of-art 2PC-ORAM constructions. As pointed out in [37], with a client-server ORAM scheme, one can construct a 2PC-ORAM by implementing the client algorithm of the ORAM with 2PC protocols. So before describing 2PC-ORAM constructions, I will first introduce a well-known generic 2PC secure computation protocol, Yao's garbled circuit [49], which is a common building block for various 2PC-ORAM schemes.

Yao's garbled circuit [49] is a protocol for generic two-party secure computation, meaning that it can be used to compute any functionality based on the boolean circuit representation of the function algorithm. It consumes only constant round of communication, and is secure against semi-honest adversary. Besides two-party secure computation, Yao's garbled circuit also has other practical applications like private function evaluation, zero-knowledge proofs, key-independent-message secure encryption, etc. [7].

A garbled circuit is essentially an encrypted circuit, with the following properties:

1. Each input wire of the circuit has a pair of keys (w_0, w_1) corresponding to the 0 and 1 values of the wire.

2. For each boolean gate of the circuit, given two keys for the two input wires representing the input values, the output of the gate can be computed and also returned as a wire key. In such a way, the whole circuit can be evaluated using the wire keys.

3. Since the circuit is computed using the wire keys instead of the actual values, the evaluator of the circuit is not able to learn anything but only the output expected to be computed.

In order to do Yao's garbled circuit, we need a auxiliary tool called Oblivious Transfer (OT) [39]. Oblivious transfer is a protocol for obliviously transferring some message from the sender party to the receiver party. In such protocol, the sender has two messages m_0 and m_1, and the receiver has a bit b. The goal is for the receiver to only receive message m_b but nothing else, while the sender learns absolutely nothing. With oblivious transfer, one can construct the garbled circuit in the following way:

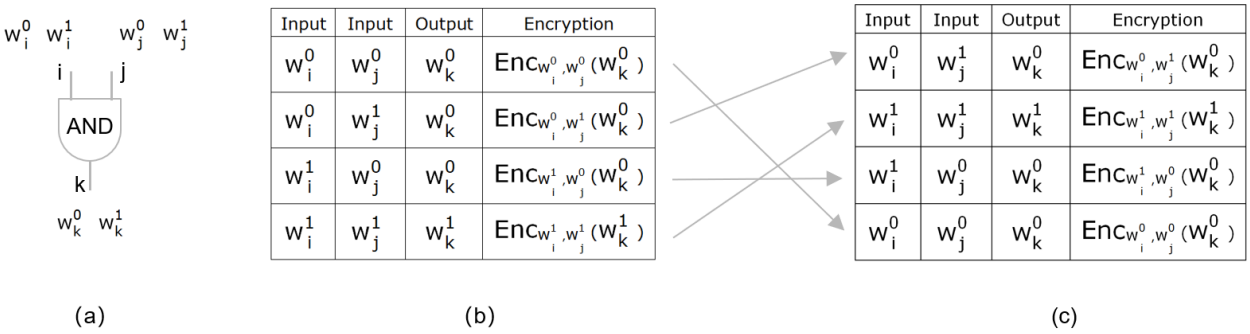
1. Call the two parties of the garbled circuit protocol circuit generator G and circuit evaluator E. Party G first constructs the garbled circuit (explained below), and sends the garbled circuit along with wire keys corresponding to G's input values to E. This reveals nothing to E as E is just getting encrypted gates and randomly looking keys.

2. E executes the oblivious transfer protocol with G to receive wire keys corresponding to E's input values. Now E has all the input value wire keys for the garbled circuit. As the property of OT, this reveals nothing about E's input to G.

3. With the entire garble circuit, and one key per input wire, E can now evaluate the circuit and get the output of the circuit, which is the output of the computation on G and E's input (what E directly gets are output wire keys, but G can provide an output translation table just on output wires for E to acquire the true output values). E can then share the output with G if needed.

It remains to show how garbled circuit is constructed and evaluated, and the demonstration below uses the AND gate as an example, because it is the same method for all other boolean gates, where the same garbled gate construction is used for all gates in the garbled circuit.

First, we will replace the bit values of the input and output of the boolean gate with their corresponding wire keys. As Fig. 2.1 shows, the truth table of the AND gate is replaced with the wire keys of the corresponding values. Then for the output column of the truth



Input	Input	Output	Encryption
w_i^0	w_j^0	w_k^0	$Enc_{w_i^0, w_j^0}(w_k^0)$
w_i^0	w_j^1	w_k^0	$Enc_{w_i^0, w_j^1}(w_k^0)$
w_i^1	w_j^0	w_k^0	$Enc_{w_i^1, w_j^0}(w_k^0)$
w_i^1	w_j^1	w_k^1	$Enc_{w_i^1, w_j^1}(w_k^1)$

Input	Input	Output	Encryption
w_i^0	w_j^1	w_k^0	$Enc_{w_i^0, w_j^1}(w_k^0)$
w_i^1	w_j^1	w_k^1	$Enc_{w_i^1, w_j^1}(w_k^1)$
w_i^1	w_j^0	w_k^0	$Enc_{w_i^1, w_j^0}(w_k^0)$
w_i^0	w_j^0	w_k^0	$Enc_{w_i^0, w_j^0}(w_k^0)$

Figure 2.1: Yao's garbled circuit for the AND gate: (a) AND gate, with input wires i, j and output wire k, and the pairs of wire keys; (b) truth table of the AND gate represented using the corresponding wire keys, and the encryption; (c) garbled version of the encrypted truth table.

table, the output wire key will be encrypted using its two input wire keys. The final step is to garble the order of the four encryptions, and these garbled encryptions are the actual garbled gate for this AND gate.

Now for evaluating this garbled gate, with only one key for each input wire, there is only one of the four encryptions that can be correctly decrypted which returns the correct wire key for the output value. Since the order of the encryption is garbled, the evaluator has no idea which row of the real-order truth table it has decrypted and thus does not know if it obtains a key for 0 or 1 output bit.

With this construction and evaluation of a single garbled gate, it is easy to construct and evaluate an entire garbled circuit, as we just connect garbled gates with wire keys, and when output of some gates are the input for other gates for the underlying circuit, we pass the corresponding output wires evaluated from those garbled gates to the other garbled gates as input wire keys.

2.4.1 Optimizations

Yao's garbled circuit protocol is a fundamental primitive used in many applications. And through out the years people have worked on optimizing this protocol. Below is a list of the major optimizations:

1. **Point and Permute**. Beaver et al. [5] noticed that by annotating each wire key using one bit, the evaluator of the garbled circuit does not need to decrypt all 4 rows of the garbled gate but only one. This is done by always using different bits for the first bit position when randomly picking a pair of keys for a wire. Then instead of randomly garbling the rows of the encrypted truth table, we sort the rows according to the first bit of the two input wire keys. In such a way, the evaluator immediately knows which row of the garbled gate to decrypt using the first bits of the input keys, and thus saves the work of decrypting the rest rows.

2. **Row Reduction**. Naor et al. [36] noticed that when transmitting each garbled gate, the transmission of three rows instead of four are already enough. To achieve this, when generating the output wire for the gate, instead of random generation, we always generate it as a function of the input label such that the encryption of the first row of the truth table is always 0. In such a way, as the first row is public information, there is no need for sending it. Thus the communication bandwidth is reduced by 25%.

3. **Free XOR Gate**. Kolesnikov et al. pointed out in [31] that if a gate is an XOR gate, then no encryption/decryption/garbling is needed for securely evaluating such gate. This is done by picking a random global wire key offset, and randomly pick pairs of wire keys in a way that the xor of each pair of the wire keys is always equal to this global offset. In this way, the output wire key can simply be computed as the xor of the two input wire keys. With this optimization, when designing the circuit for secure

computation, circuits with more XOR gates and less AND gates (all other gates can be constructed using AND gates) will have better efficiency.

4. **Half AND Gate**. Zahur et al. showed in [50] that if the circuit generator and evaluator will know in advance the wire key they will be using on one of the input wire for the AND gate, then the evaluation becomes evaluating "half AND" gate which only requires one encryption/decryption with the row reduction optimization. And then because a full AND can be represented as two such half gates and one free XOR gate, only two cipher-texts (instead of three from row reduction optimization) are needed for this garbled AND gate. Though this optimization has an additional cost that each garbled AND gate now requires two decryption instead of one, the communication bandwidth, which is often the bottle-neck, is further reduced by 33% from the row reduction optimization.

2.5 2PC-Circuit-ORAM

In the following sections I will summarize three state-of-art 2PC-ORAMs. This section talks about Circuit-ORAM by Wang et al. [44], which is an asymptotically and concretely efficient ORAM for MPC secure computation applications.

For ORAM in the client-server setting, most time bandwidth is the main performance bottleneck, and thus also the primary performance metric. And many ORAM work have been focusing on optimizing the communication bandwidth. However, when constructing MPC-ORAM, ORAM optimized on bandwidth does not necessarily translate into optimized MPC-ORAM. This is because the client algorithm of the underlying ORAM scheme may not have an efficient MPC implementation. Because often the MPC implementation of the ORAM's client algorithm is done using generic 2PC protocol Yao's garbled circuit, the circuit complexity of the ORAM's client algorithm becomes a natural performance metric

for MPC-ORAM [44]. Therefore, ORAM with efficient and "MPC-friendly" client algorithm is needed and has been studied.

Wang et al. in 2014 proposed a binary-tree based ORAM called Circuit-ORAM [44] with the goal to achieve optimal circuit complexity for the MPC setting. Want et al. noticed that by naively implementing the state-of-art ORAM schemes at that time like Path-ORAM in the MPC setting, it was hard to get practical performance due to the complex circuit of Path-ORAM's eviction algorithm. Though Want et al. found alternative oblivious sorting solution to perform the same Path-ORAM eviction algorithm, due to the cost of oblivious sorting which is an expensive circuit, the improvement was not significant. Thus a new "MPC-friendly" eviction algorithm would be needed for optimizing the circuit size.

The solution Want et al. [44] had was to have a less complex eviction algorithm than the Path-ORAM one but still preserve most of the eviction effectiveness. Their key idea was to first determine the eviction pattern, i.e., how tuples will be evicted on the path, by only using linear scans on the metadata, i.e., flag bits and labels of the data blocks, without including the data record fields. Then, for the most expensive eviction movement, i.e., actually evicting the tuples along the path, they accomplished it only with a single scan on the path, and thus reduced the eviction algorithm circuit complexity.

In more detail, 2PC-Circuit-ORAM's eviction algorithm can be spitted into two steps:

1. **Eviction Logic**. This step determines the eviction pattern, that how the tuples on the current retrieved path can be evicted towards the bottom of the path. The idea is as following: For every bucket on the path, scan only the metadata fields of each tuple in the bucket. If the current tuple is a real tuple, compute the deepest bucket on the path that this tuple can be evicted to. Once the tuple metadata scan for a bucket is completed, a tuple from this bucket that can be evicted deepest on the path is found. When the deepest tuple for each bucket is found, the eviction pattern can

be determined by starting from the root bucket and checking with the rest buckets on path in order: if the deepest tuple from previous bucket can be evicted deeper than the deepest tuple in the current bucket, then the deepest tuple from previous bucket should keep comparing with the rest buckets, and the deepest tuple in the current bucket will not be evicted; however, if the previous deepest tuple cannot be evicted deeper than the current deepest tuple, then the previous deepest tuple should be evicted to the current bucket, and the current deepest tuple should continue comparing with the rest buckets. Notice that at all time of this eviction algorithm, at most one tuple is being evicted and compared with the rest deepest tuples from each bucket, which is an oblivious pattern and leads to a simple algorithm circuit.

2. **Eviction Movement**. Once the eviction logic is finished that an eviction pattern is determined, the eviction movement can be performed by a single scan on every tuples on the path. This is because, as pointed out above, at all time at most once tuple is being held for eviction. So at each bucket level during the eviction, if an eviction jump is happening at this level, then the tuple being held will be dropped at the current bucket, and the tuple being kicked out in the current bucket will be on hold to be moved into the following buckets. It is like an oblivious swap operation, that if there is an eviction on current bucket level, we perform the tuple swap; otherwise, we do not perform the swap and continue to the next bucket. Therefore, only a single on the actual data records on the path is needed, which has the optimized circuit size.

In comparison with ORAM schemes at that time, 2PC-Circuit-ORAM achieves about 58x times improvement over straightforward 2PC implementation of the Path-ORAM, and about 5x times improvement over another SCORAM with client algorithm designed for the 2PC setting [44]. And notice that 2PC-Circuit-ORAM still has the $O(\log^3(n))$ complexity, while the second efficient 2PC-SCORAM is a heuristic scheme without theoretical performance bounds.

2.6 2PC-Sqrt-ORAM

As discussed above, for MPC-ORAM, the underlying ORAM client algorithm circuit size is an important performance metric because it can greatly affect the efficiency of MPC-ORAM like communication bandwidth. At the same time there is work like Circuit-ORAM which tried to find MPC-friendly ORAM algorithm with binary-tree structure, people also tried to work on the hierarchical ORAM construction originated from [22] to see if there could be MPC-friendly solution to reduce the expensive cost of oblivious sorting, hashing, and secure computation of pseudo-random functions (PRF) used in [22].

In 2016, Zahur et al. [51] published a new 2PC-Sqrt-ORAM design based on the Square-Root ORAM of [22]. Like in [22] that its "Square-Root" client-server ORAM has $O(\sqrt(n))$ overhead, the new 2PC-Sqrt-ORAM scheme also has the similar overhead complexity which is not asymptotically better than the 2PC-Circuit-ORAM by Want et al. [44]. However, as illustrated by Zahur et al., for smaller memory array length like $n < 2^{16}$, the 2PC-Sqrt-ORAM has very efficient concrete performance due to small constants in the cost formula, and their concrete performance is actually better than the 2PC-Circuit-ORAM. Moreover, for initializing the ORAM, unlike binary-tree based ORAM which may require inserting memory data records one by one, 2PC-Sqrt-ORAM can initialize the whole memory array at once and the initialization time significantly outperforms the previous schemes.

Zahur et al. [51] pointed out that the original Square-Root ORAM construction was not designed and well-suited for the MPC setting and there were two main problems that made it very expensive if using generic MPC implementation: first, the access of the original Square-Root ORAM requires evaluation of PRF, which is very inefficient because doing PRF in MPC requires thousands of circuit gates. Second, the shuffling of data blocks require many oblivious sorting operations, which is also a noticeable cost in the MPC setting.

The new 2PC-Sqrt-ORAM [51] solved the above problems mainly with the three modifications below:

1. The expensive PRF is replaced with random permutation secret-shared among the two parties. This eliminates the needs for expensive oblivious sorting on PRF output during re-shuffling of initialization, which can be done with more efficient Waksman shuffling network.

2. The 2PC-Sqrt-ORAM does not require the use of extra dummy blocks in the main memory array. Whenever a dummy access is needed, it will access a random unaccessed real block, and append it to the stash just like other real accessed block. As stash is always scanned at first for each access, it still guarantees to find the real target record that should be accessed.

3. Each accessed location is now a public information. This eliminates the need for moving blocks from stash back to memory array using oblivious sorting during re-initialization. The reason this is still secure is that each main memory location will be accessed at most once. And if some location is being requested for access multiple times, then fake accesses on unvisited locations will be performed after the first access, and the real record lookup is done by scanning the stash, so the access pattern is still the same.

The asymptotic complexity of the 2PC-Sqrt-ORAM is $O(\sqrt{n \log^3(n)})$, which is not as good as the $O(\log^3(n))$ complexity of 2PC-Circuit-ORAM mentioned in the above section. However, according to the performance measurement and comparisons in [51], for small memory array size, the 2PC-Sqrt-ORAM concrete cost can even be very close to the linear scan cost (so much better than 2PC-Circuit-ORAM), and for moderate memory array size, the 2PC-Sqrt-ORAM performs better than 2PC-Circuit-ORAM for all memory block counts up to 2^{16}. And for concrete initialization cost, the 2PC-Sqrt-ORAM can be about two orders of magnitude better than 2PC-Circuit-ORAM [51].

2.7 2PC-FLORAM

Recently in 2017, Doerner et al. proposed a new 2PC-ORAM scheme called Function secret-sharing Linear ORAM, or FLORAM [15]. Unlike previous 2PC-ORAM schemes like 2PC-Circuit-ORAM and 2PC-Sqrt-ORAM, 2PC-FLORAM considers a variant of the base ORAM model called distributed ORAM, where the ORAM memory is replicated among two servers instead of stored at a single server. With this slightly weaker model of ORAM and the use of techniques of DPF based two-server PIR of Boyle et al. [10, 11], the 2PC-FLORAM achieves constant round ORAM access, efficient access time and smaller bandwidth, and small initialization time. Though due to the nature of PIR protocol that the computation complexity of FLORAM is $O(n)$, linear to the memory size, the local computation can be fastened with multi-threading, and Doerner et al. showed up to 10x times runtime improvement with parallelism.

Before showing the construction of the FLORAM, Distributed Point Function (DPF) and Private Information Retrieval (PIR) will be introduced which serve as the building blocks of the 2PC-FLORAM.

2.7.1 Private Information Retrieval

Private information retrieval (PIR) [13] is the protocol that allows the client to retrieve some item from a database on server without revealing to the server which item was retrieved. It is a weaker notion of the Oblivious Transfer, that it does not require the client to not learn information about other items in the database. For the construction of FLORAM, the two-server PIR is used, that for such protocol, there will be two servers holding replications of the entire database, while the client is retrieving some database item with some secret index.

A very primitive way to implement the two-server PIR is as following [13]. Let the client be C and two servers be S_1 and S_2. The memory storage at the servers is denoted as M, with size n. C would like to retrieve item $\mathsf{M}[i]$ at index i from S_1 and S_2:

1. C generates an array v_1 of n random bits, and another array v_2 of n bits such that $v_2[j] = v_1[j]$ for all $j \neq i$ and $v_2[i] = 1 \oplus v_1[i]$. C sends v_1 to S_1 and v_2 to S_2.
2. S_1 computes $b_1 = \bigoplus_{j=1}^{n}(v_1[j] \cdot \mathsf{M}[j])$. S_2 computes $b_2 = \bigoplus_{j=1}^{n}(v_2[j] \cdot \mathsf{M}[j])$. S_1 and S_2 sends back b_1, b_2 to C.
3. C computes $b = b_1 \oplus b_2$, which is $\mathsf{M}[i]$.

It is trivial to verify that $b = \mathsf{M}[i]$ because for $j \neq i$, $(v_1[j] \cdot \mathsf{M}[j]) \oplus (v_2[j] \cdot \mathsf{M}[j]) = (v_1[j] \oplus v_2[j]) \cdot \mathsf{M}[j] = 0$, and $(v_1[i] \cdot \mathsf{M}[i]) \oplus (v_2[i] \cdot \mathsf{M}[i]) = (v_1[i] \oplus v_2[i]) \cdot \mathsf{M}[i] = \mathsf{M}[i]$. And C's index i remains secret because v_1, v_2 looks random to S_1, S_2.

Notice that the bandwidth cost for such two-server PIR protocol is $O(n + B)$, where B is the size of each memory item. And the servers are doing $O(n)$ local computation. For large n, the computation can be parallelized to reduce runtime, however, the $O(n)$ bandwidth can be a bottle-neck for such protocol.

2.7.2 Distributed Point Function

Boyle et al. [10, 11] pointed out that two-server PIR can be constructed using the technique called Distributed Point Function (DPF) with bandwidth complexity $O(\log(n)\kappa + B)$ where κ is the security parameter. In the following I will first introduce distributed point function, and then describe the two-server PIR construction using DPF.

A point function is a function $f_{\alpha,\beta}$ such that $f_{\alpha,\beta}(x) = \beta$ if $x = \alpha$, and $f_{\alpha,\beta}(x) = 0$ for all $x \neq \alpha$. In other words, at at most one domain point, the function has some non-zero

value, while at all other points the function output is zero. And a distributed point function is a two-party function secret-sharing scheme of such point function, which is a pair of probabilistic polynomial time algorithms Gen and Eval:

1. $\mathsf{Gen}(\alpha, \beta)$ is a key generation algorithm, that with the description of the point function $f_{\alpha,\beta}$ as the input, it outputs a pair of keys k_1, k_2 as the secret-sharing of the function $f_{\alpha,\beta}$.

2. $\mathsf{Eval}(k_b, x)$ is an evaluation algorithm, which takes a point function secret-sharing key k_b (for $b \in \{1, 2\}$) and a function domain point x, and outputs y such that y is one of the xor secret-sharing of $f_{\alpha,\beta}(x)$. This means the Eval algorithm satisfies that $\mathsf{Eval}(k_1, x) \oplus \mathsf{Eval}(k_2, x) = f_{\alpha,\beta}(x)$ for all x.

With this distributed point function technique, Boyle et al. [10, 11] gave a new two-server PIR construction as following:

1. C runs $\mathsf{Gen}(i, 1)$, and send the DPF keys k_1, k_2 to corresponding servers S_1, S_2. This is like C secret-shares a vector of bits among S_1, S_2, where the i-bit on vector is 1, and all rest bits are 0.

2. S_1 computes $b_1 = \bigoplus_{j=1}^{n}(\mathsf{Eval}(k_1, j) \cdot \mathsf{M}[j])$. S_2 computes $b_2 = \bigoplus_{j=1}^{n}(\mathsf{Eval}(k_2, j) \cdot \mathsf{M}[j])$. S_1 and S_2 sends back b_1, b_2 to C.

3. C computes $b = b_1 \oplus b_2$, which is $\mathsf{M}[i]$.

With the property of the DPF, $(\mathsf{Eval}(k_1, j) \cdot \mathsf{M}[j]) \oplus (\mathsf{Eval}(k_2, j) \cdot \mathsf{M}[j]) = (\mathsf{Eval}(k_1, j) \oplus \mathsf{Eval}(k_2, j)) \cdot \mathsf{M}[j] = 0$, and $(\mathsf{Eval}(k_1, i) \cdot \mathsf{M}[i]) \oplus (\mathsf{Eval}(k_1, i) \cdot \mathsf{M}[i]) = (\mathsf{Eval}(k_1, i) \oplus \mathsf{Eval}(k_1, i)) \cdot \mathsf{M}[i] = \mathsf{M}[i]$. Therefore such two-server PIR construction is still correct. And the security of this new construction follows directly from the security of DPF that each party's function share does not leak information about the domain point i.

In this new construction, the servers' local computation complexity remains the same as $O(n)$. What is change is the bandwidth complexity that the client is only sending DPF secret-sharing keys instead of bit vectors, so the bandwidth complexity is changed to the DPF key length plus database item size which is $O(\log(n)\kappa + B)$. For large n, this new construction will be more efficient on bandwidth.

2.7.3 FLORAM Construction

2PC-FLORAM [15] has three main data structures, read-only memory rom, write-only memory wom, and a stash. For the memory array M stored in this FLORAM scheme, rom is the encrypted M replicated at every parties, wom is encrypted M secret-shared among the two parties, and stash is the temporary storage for accessed and updated data records so far. For ORAM access with address adr, the stash will be linearly scanned to search record with adr, similar to other ORAM schemes. What is different is that after the stash scan, the two parties will run DPF key Gen algorithm on adr with secure computation. This will output one DPF secret-sharing key to each of the party, who acts like servers in the two-server PIR to retrieve secret-sharing of $\mathsf{rom}[\mathsf{adr}]$. Then through secure computation, $\mathsf{rom}[\mathsf{adr}]$ can be updated to $\mathsf{rom}[\mathsf{adr}]'$ and appended to stash for later scan, and $\triangle(\mathsf{rom}[\mathsf{adr}]) = \mathsf{rom}[\mathsf{adr}] \oplus \mathsf{rom}[\mathsf{adr}]'$ will be output to each party. With this $\triangle(\mathsf{rom}[\mathsf{adr}])$, the two parties can perform a private information writing (PIW) protocol (similar to the PIR protocol, but applied on secret-sharing data structure wom instead of replication data structure rom) to update the wom data structure by using the same DPF secret-sharing keys so that $\mathsf{wom}[\mathsf{adr}]' = \mathsf{wom}[\mathsf{adr}] \oplus \triangle(\mathsf{rom}[\mathsf{adr}])$. So wom always stores the most recent copies of the memory records, which allows us to re-initialize and update the rom when stash is full. This rom re-initialization is done by simple re-share protocol, to reconstruct encrypted replication rom of most recent M using secret-sharing of M which is wom.

There is no recursive ORAM structure in the above FLORAM like in Circuit-ORAM and Sqrt-ORAM, so the access has constant round of communication. However, this requires the DPF key Gen algorithm to be done entirely with generic 2PC secure computation, which is expensive even with AES circuits to implement the required PRG calls of Gen. Doerner et al. [15] proposed an optimization to actually move the PRG computation out of the secure computation, which greatly reduced the secure computation circuit size. However, this induces $O(\log(n))$ more communication rounds for computing DPF key Gen. But for small values of n, as secure computation tends to be the bottle-neck instead of the linear complexity local computation, this significantly improves FLORAM's performance.

Despite the server local computation cost, like the 2PC-Sqrt-ORAM construction, the 2PC FLORAM also has $O(\sqrt{n})$ complexity due to amortizing periodic reinitialization over accesses. And the $O(n)$ local computation cost seems to be the most obvious disadvantage of 2PC FLORAM. In fact for very large n, though multi-threading can be used and the local computation is highly parallelizable, the computation is indeed the bottle-neck and more asymptotically efficient scheme like Circuit-ORAM is more efficient on access run-time according to the trend of the runtime graph in [15]. But still, [15] has shown that for a wide range of n, 2PC-FLORAM has the most efficient concrete cost on both bandwidth and runtime.

2.8 Summary of 2PC-ORAM

Table 2.1 compares the asymptotic complexity of bandwidth of the three recent 2PC-ORAM schemes. The round complexity for these three schemes with best concrete performance construction are all $O(\log(n))$ and thus not listed redundantly in the table. The secure computation complexity of each of these schemes is the same as their bandwidth and

thus omitted as well. But note that 2PC-FLORAM also has an $O(nB)$ complexity local computation.

MPC-ORAM schemes	Bandwidth
2PC-Circuit-ORAM [44]	$O(\kappa \log^3(n) + \kappa B \log(n))$
2PC-Sqrt-ORAM [51]	$O(\kappa B \sqrt{n \log^3(n)})$
2PC-FLORAM [15]	$O(\kappa B \sqrt{n})$

Table 2.1: Bandwidth per memory access for recent 2PC-ORAM schemes as a function of security parameter κ, array length n, and record size B.

According to Table 2.1, 2PC-Circuit-ORAM has the best asymptotic complexity, which indicates that for large n, 2PC-Circuit-ORAM will have the smallest cost. 2PC-Sqrt-ORAM has the worst asymptotic cost, but in practice it is very efficient and has the smallest cost for small n values. 2PC-FLORAM's asymptotic complexity is worse than 2PC-Circuit-ORAM, but in practice, its performance is better than 2PC-Circuit-ORAM and 2PC-Sqrt-ORAM for a wide range of n's. However, due to $O(nB)$ local computation cost, 2PC-FLORAM it not well-suited for very large n or B.

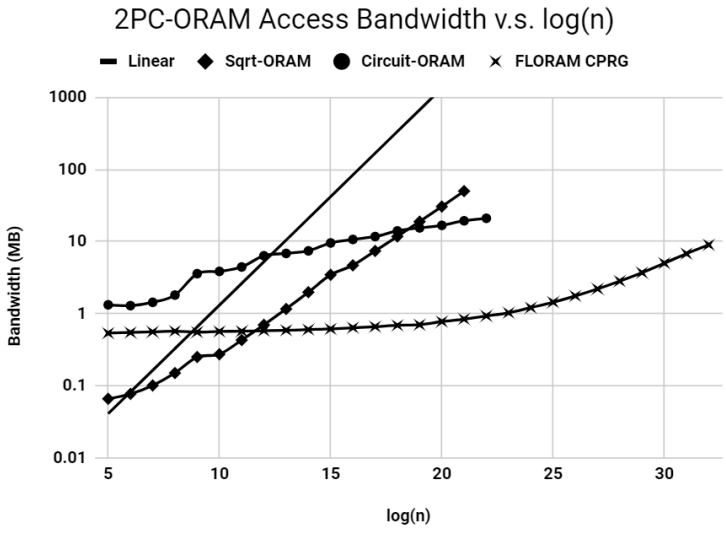


Figure 2.2: 2PC-ORAM access bandwidth comparison, for record size = 4 bytes, from [15]

Doerner et al. reported concrete bandwidth comparisons between these three 2PC-ORAM schemes in [15], which I include here in Figure 2.2. From the figure we see that for

$B = 4$ bytes, roughly for $n < 2^6$, linear scan is most efficient; then for $2^6 < n < 2^{12}$, 2PC-Sqrt-ORAM is most efficient; then for the tested range $2^{12} < n < 2^{32}$, 2PC-FLORAM is most efficient. But the line trends show that eventually at some point 2PC-Circuit-ORAM will have the most efficient bandwidth. And for larger B, this break-even point should be even earlier.

Chapter 3

Method Overview

Starting this chapter, I will present my 3PC-ORAM study aiming to achieve more efficient asymptotic and concrete performance for real world applications. And in this chapter I will give an overview of the methods and approaches I used to study, design, construct, and analyze the 3PC-ORAM schemes.

3.1 Study, Design, and Customize 3PC Protocols

Learn existing 3PC protocols. As briefly mentioned in Chapter 1, for multi-party secure computation, often the 3PC implementation can be much more efficient than the 2PC implementation of the same protocol. An example is the Oblivious Transfer protocol. Though nowadays there are great techniques and optimizations like OT-extensions [4, 27, 2] (running a small number of OTs with public key crypto, which can be used as a base to obtain many OTs via only symmetric key crypto) to reduce the cost, doing oblivious transfer in 2PC still requires some work equivalent to public key cryptography [26]. However, in the 3PC setting, the same OT protocol can be done much simpler with symmetric key encryption,

specifically, only four xor operations are needed [3] (see Figure 3.1). So there exists secure computation building blocks like OT that has been proved that 3PC can be more efficient than 2PC. Learning these existing work would be beneficial since they may be used for the construction of the 3PC-ORAM.

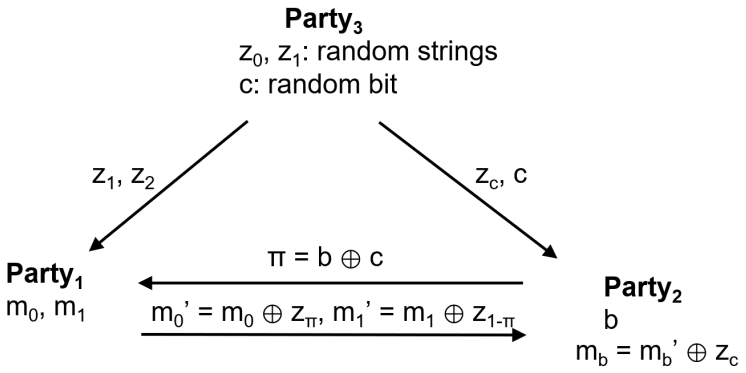


Figure 3.1: 3PC Oblivious Transfer: with Party_1's input two messages m_0, m_1 and Party_2's input bit b, obliviously transfer message m_b from Party_1 to Party_2

Build 3PC-ORAM from 2PC scheme. Many work has been done in the 2PC setting to find and design efficient ORAM solution for secure computation [46, 44, 51, 15]. And existing 2PC-ORAM can be great starting point for designing more efficient 3PC-ORAM. One possible approach for constructing 3PC scheme is that, the 2PC-ORAM may be done using generic MPC protocol, which means it already has a generic 3PC implementation. However, the 3PC generic implementation may not have the optimal cost comparing to protocol implementations customized just for 3PC. Another approach is that, take an efficient 2PC scheme, and explore whether some of its components can be done with more efficient 3PC implementation. Though this may require more work to design customized 3PC protocol which is not generic MPC solution, often this gives better performance with the advantage of 3PC.

Modify 2PC algorithm to meet 3PC requirements. 2PC protocol can still be executed in the 3PC setting as long as two of the three parties still provide the same input required by the 2PC protocol. For example, the 2PC Yao's garbled circuit protocol [49] is used

in some schemes of this 3PC-ORAM study. However, often the better efficiency of 3PC protocol comes from the simpler and more efficient 3PC data structures and secret-sharing, and sometimes there is no direct way of converting 2PC protocol input/output to the 3PC ones. In such cases, to still adopt the MPC-friendly ORAM algorithm designed for 2PC and try to improve efficiency in the 3PC setting, some modification of the algorithm may be required for the conversion from 2PC to 3PC, which may also lead to a new customized 3PC design of the protocol.

3.2 Analysis of 3PC-ORAM

Once we have some ideas about a possible 3PC-ORAM design, it is important to analyze it so the correctness can be verified, security can be argued, efficiency can be estimated, and performance can be compared with previous schemes for improvements. Below is a list of aspects that should be considered when analyzing a 3PC-ORAM design:

1. **Correctness**. An ORAM construction often involves many sub-protocols. To prove the correctness of an ORAM scheme, besides having a good intuition and pseudo-code for the idea of the ORAM algorithm, it is also important to prove the correctness of each building block/sub-protocol. Then the correctness of the ORAM algorithm can be argued.

2. **Failure probability**. Some ORAM constructions are correct with failure probability. For example, the binary-tree based ORAM like Circuit-ORAM has the possibility of overflowing the tree nodes (buckets) which causes the ORAM algorithm to fail. However, we still consider this ORAM construction correct if the failure probability is negligible. Thus, in such case, we would need to prove the possible design of a

3PC-ORAM scheme also has negligible failure probability. Often this follows the same failure probability argument of the underlying ORAM.

3. **Security**. The 3PC-ORAM scheme should be a 3PC secure emulation of the underlying client-server ORAM algorithm, therefore the security proof of the 3PC-ORAM scheme can be done by proving the 3PC-ORAM protocol securely realizes the underlying ORAM functionality in the 3PC setting as long as the underlying client-server ORAM algorithm is secure (by the argument of [44]). To argue the 3PC-ORAM is indeed a secure MPC protocol, we can first verify each building block of the 3PC-ORAM protocol is a MPC protocol. Then as long as the output of these MPC protocols do not leak secret ORAM information when entered in other MPC protocols as input, the whole ORAM protocol is also an MPC protocol.

4. **Asymptotic Complexity**. The asymptotic complexity should be considered when designing the 3PC-ORAM scheme as the goal is to build more efficient 3PC-ORAM construction. When the correctness and security are both verified, the asymptotic cost of the ORAM scheme can then be claimed and compared with other schemes for improvements.

5. **Concrete Performance**. As MPC-ORAMs are more expensive than the underlying client-server ORAM, it is often desirable to verify the concrete cost of a MPC-ORAM design and its practicality. Bandwidth and runtime are the performance metrics mostly used to compare between different schemes. While the runtime may depend on many factors like machine specs, actual code implementations, network, and more, the bandwidth is a more reliable indicator of the actual performance.

6. **Code implementation**. Many MPC protocols and recent MPC-ORAM constructions have implementations to support their claimed improvements. So for a 3PC-ORAM design, it is better to also have an actual code implementation to verify the theory and estimated cost of the 3PC-ORAM design and see the concrete improvements. Like

recent 2PC-ORAM implementations, in this study, my 3PC-ORAM implementations are tested using Amazon Web Service (AWS) to ensure fair comparison with standard machine configuration and testing environment.

Chapter 4

First Customized 3PC-ORAM

This chapter describes a customized 3PC-ORAM scheme to achieve efficient performance. It is based on my publication *Three-Party ORAM for Secure Computation*, a joint work with Sky Faber, Stanislaw Jarecki, and Sotirios Kentros, published on Asiacrypt 2015 [18].

We proposed our first customized 3PC-ORAM which uses a variant of the binary-tree ORAM as the underlying data-structure. Recall that the access of a binary-tree ORAM can be viewed as three steps: Retrieval, PostProcess, and Eviction. The *retrieval* part of the proposed 3PC-ORAM is based on the following observation: If P_1 and P_2 secret-share an array of (keyword,value) pairs (k, v) (this will be a path in the binary-tree ORAM) and a searched-for keyword k^* (this will be the searched-for address prefix), then a variant of the Conditional Disclosure of Secret protocol of [21] which we call *Secret-Shared Conditional OT* (SS-COT) allows P_3 to receive value v associated with keyword k^* at the cost roughly equal to the symmetric encryption and transmission of the array. Moreover, while SS-COT reveals the location of pair (k^*, v) in the array to P_3, this leakage can be easily masked if P_1, P_2 first shift the secret-shared array by a random offset. The *eviction* part of the 3PC-ORAM springs from an observation that instead of performing the eviction computation on all the

data in the path from retrieval, one can use garbled circuit to encode only the procedure determining eviction *movement logic*, i.e., determining which entries in each bucket should be evicted down the path. Then, if P_1, P_2 secret-share the retrieved path, and hence the bits which enter this computation, we can let P_3 evaluate this garbled circuit and learn the positions of the entries to be moved if (1) the eviction moves a constant number of entries in each bucket in a predictable way, e.g., one step down the path, and (2) P_1, P_2 randomly permute the entries in each bucket, so that P_3 always computes a fixed number of randomly distributed distinct indexes for each bucket. Computation of this movement logic uses only two input bits (appropriate direction bit and a full/empty flag) and 17 non-xor gates per bucket entry, so the garbled circuit is much smaller than if it coded the whole eviction procedure. Finally, the secret-shared data held by P_1, P_2 can be moved according to the movement logic matrix held by P_3 in another OT/CDS variant we call *Secret-Shared Shuffle OT* which uses only xor's and whose bandwidth is roughly four times the size of the secret-shared path.

Assuming constant record sizes, the bandwidth of the resulting 3PC-ORAM protocol is $O(w(\log^3(n) + \kappa \log^2(n)))$ where w is the bucket size in the underlying binary-tree ORAM. Since the best exact bound on overflow probability we can give requires $w = \Omega(\lambda + \log(n))$ where λ is a statistical security parameter, and since $\log(n) < \kappa$, this asymptotic bound is essentially the same as the one of the 2PC-ORAM of [46]. However, the exact numbers for bandwidth and computation cost (measured in the number of block cipher or hash operations) are much lower, and this is because of two factors: First, even though we still use garbled circuits, the circuits involved have dramatically smaller complexity than in the 2PC implementations (see Table 4.1). Secondly, the cost of all operations *outside* the garbled circuits is a small factor away from the cost of transmission and decryption of server data in the underlying binary-tree ORAM algorithm. Concretely, the non-GC bandwidth is under $9|\mathsf{path}|$ where $|\mathsf{path}|$ is the (total) size of tree *path* retrieved by the binary-tree ORAM, and computation is bounded by symmetric encryption of roughly $20|\mathsf{path}|$ bits,

whereas for the underlying binary-tree ORAM both quantities are $2|\mathsf{path}|$. Finally, stochastic evidence suggests that it suffices that $w = \Omega(\sqrt{\lambda + \log(n)})$, which for concrete parameters of $\log(n) = 36$ and $\lambda = 40$ reduces the required w, and hence all our protocol costs, by a further factor between 3 and 4.

ORAM	Circuit Size (Asymptotic Bounds)	Circuit Size (gates)		Number of Inputs	
		$\log(n) = 20$	$\log(n) = 29$	$\log(n) = 20$	$\log(n) = 29$
2PC-Path-ORAM	$\tilde{O}\left(\log^3(n) + B\log(n)\right)\omega(1)$	37.2 M	111.7 M	0.2 M	0.3 M
2PC-SCORAM	N/A (heuristic)	4.6 M	13.0 M	0.3 M	0.9 M
3PC-ORAM ($w = 128$)	$O\left(\log^3(n)\right)\omega(1)$	96.9K	213.9K	11.5K	25.3K
3PC-ORAM ($w = 32$)	$O\left(\log^3(n)\right)\omega(1)$	28.5K	62.6K	3.4K	7.4K

Table 4.1: Comparison of Circuit Size between this proposal (without optimizations) and the 2PC-SCORAM scheme [46]. All numbers are reported as function of array size $|\mathsf{M}| = n$ for statistical security parameter $\lambda = 80$. The first 3PC-ORAM estimation uses bucket size $w = 128$ mandated by the strict bound implied by Lemma 4.2, while the second one uses bucket size $w = 32$ derived from the Markov Chain approximation.

4.1 Technical Overview

Our implementation is based on the Shi et al. [41], and uses a combination of three-party OT's and secure computation (using Yao's garbled circuits [49]) in order to ensure privacy in the three party setting. Our protocol follows the same technical approach of 2PC-SCORAM schemes, i.e., of providing a secure computation protocol for retrieval, post-process, and eviction algorithms in a client-server ORAM. However, the existence of a third party allows us to greatly reduce the cost of this secure computation. Our main observation is that in binary-tree based retrieval and eviction algorithms, like that of Gordon et al. [25], there is a separation in the role played by the input bits of the access or eviction circuit. Part of the bits are used to implement the logic of the circuit, but the majority are data that do not participate in the output of the logic and are, at best, just being moved between some locations based on the output of the logic. We exploit this separation in the three-party setting, by isolating the bits necessary for the logic, using Yao's garbled circuit to securely

compute the logic only on those bits, and then use several variants of the (three-party) Oblivious Transfer (OT) protocol to move data to the locations pointed out by the output of the circuits. Since all these variants of OT can be implemented at a cost similar to just the *secure transmission* of the data the OT operates on, this leads to dramatic reductions in the cost of the resulting secure computation protocol. In addition, in the *retrieval* protocol, as opposed to the *eviction*, we avoid using garbled circuits entirely, as the entire logic comes down to finding an index where two lists of m bit-strings contain a matching entry, which we implement using a three-party variant of *Conditional OT* which takes a single interaction round and costs roughly as much as encryption and transmission of these m bit-strings.

We make several modifications in the binary-tree ORAM of Shi et al. [41] to make it more efficient for the type of operations we are interested in. We use ideas from Gentry et al. [20] and Stefanov et al. [42]. In particular, we make the ORAM trees more shallow, as in Gentry et al. [20] by increasing how many entries in the ORAM will be mapped to each leaf in expectation and increasing the total capacity (in terms of entries) of the leaf nodes. To be more precise, for a tree that has a total capacity of n entries and a capacity in each node of w, instead of having n leafs in the ORAM tree, we have $\frac{n}{w}$ leafs instead. In order to ensure that overflow does not occur in the leafs of the tree we increase capacity of leaf nodes to $4w$. With this change we achieve linear overhead in terms of storage needed for the ORAM, meaning that now the total entries that can be stored in the ORAM are $O(n)$, in contrast with the $O(w \cdot n)$ entries that the ORAM of Shi et al. [41] had (note that for most settings $w = O(\log(n))$). In addition, we observe that for internal nodes it is not necessary to increase their capacity, since the overflow of internal nodes is mandated by a difference probabilistic process that the one of leaf nodes. In contrast with the approach used in Gentry et al. [20], by only increasing the capacity of leaf nodes, we avoid doubling the bandwidth needed by the ORAM protocol (which is what happened in Gentry et al. [20], since they increase the capacity of all nodes, whether they are leafs or internal nodes).

We adopt the idea of eviction through a single path introduced by Gentry et al. [20]. The main problem we identified in the single path eviction, is that both Gentry et al. [20] and Stefanov et al. [42] evict all entries in all nodes of the path, as far down in the path as they can go. Although this is easy to do in a client-server ORAM where the client retrieves the whole path and performs all operations in the clear text data, in the setting of secure computation on the secret shared data, such eviction is very costly. For this reason, we modify the eviction to only evict at most two items from each node to the next node in the path, provided such items exist. This operation is limited enough to allow for simple garbled circuits. Moreover, it is an oblivious operation in the sense that always two entries are evicted to the next level (we evict empty entries if appropriate entries do not exist), which allows for its simple 3-party implementation. We choose not to increase the fan-out of nodes as Gentry et al. [20] do, since this would complicate both our circuits and the rest of the protocols. We also choose to avoid using the overflow cache used in Path-ORAM of Stefanov et al. [42] in order to decrease the total space requirements for their ORAM, deciding instead to experiment at first with a design which maximally simplifies the eviction logic and the associated garbled circuits.

4.2 Three-Party Protocol Building Blocks

Our 3PC-ORAM protocols retrieval, post-process, and eviction of Section 4.3 rely on several variants of Oblivious Transfer (OT) or Conditional Disclosure of Secrets (CDS) protocols which we detail here. The efficiency of our 3PC-ORAM protocol relies on the fact that all these OT variants, including the OT variant employed in Yao's Garbled Circuit (GC) protocol, have significantly cheaper realizations in the 3-party setting. All presented protocols assume secure channels, although in many instances encryption overhead can be eliminated with simple protocol changes, e.g., using pairwise-shared keys in PRG's and PRF's.

Notation. We refer to the three parties involved in our 3PC-ORAM protocol as C, D, and E. Let κ denote the cryptographic security parameter, which is both the key length and the block length of a symmetric cipher. Let G^ℓ be a PRG which outputs ℓ-bit strings given a seed of length κ. Let $\mathsf{F}^\ell_\mathsf{k}$ be a PRF which maps domain $\{0,1\}^\kappa$ onto $\{0,1\}^\ell$, for k randomly chosen in $\{0,1\}^\kappa$. We will write G and F_k when $\ell = \kappa$. In our implementation both F and G are implemented using counter-mode AES. If party A holds value a and party B holds value b s.t. $a \oplus b = v$ then we call pair (a, b) an "A/B secret-sharing" of v and denote it as $(\mathrm{s_A}[v], \mathrm{s_B}[v])$. Whenever we describe an intended output of some protocol as A/B secret sharing of value v, we mean this to be a *random* xor-sharing of v, e.g., pair $(r, r \oplus v)$ for r random in $\{0,1\}^{|v|}$. Let $x[j]$ denote the j-th bit of bit-string x, and let $[m]$ denote integer range $\{1, ..., m\}$.

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

Secret-Shared Conditional OT, $\mathsf{SS\text{-}COT}^{[N]}(\mathrm{S}, \mathrm{R}, \mathrm{H})$, is a protocol where S inputs two lists, $(m_1, \ldots, m_N)$ and $(a_1, \ldots, a_N)$, H inputs a single list $(b_1, \ldots, b_N)$, and the protocol's goal is for R to output all pairs (t, m_t) s.t. $a_t = b_t$. This is a very close variant of the Conditional

Disclosure of Secrets protocol of [21], and it can be implemented e.g., using modular arithmetic in a prime field. In Alg. 4.1 we provide an alternative design which uses fewer (pseudo)random bits, and hence requires fewer PRG ops in pre-computation, but uses block ciphers in the on-line phase (the algorithm proposed here was faster in our implementation even in the on-line stage). S and H share two PRF keys k, k', and for each t helper H sends to R a pair $(p_t, w_t) = (\mathsf{F}_\mathsf{k}(b_t), \mathsf{F}_{\mathsf{k}'}(b_t))$, while S sends (e_t, v_t) where e_t is an xor of message m_t and $\mathsf{G}(\mathsf{F}_\mathsf{k}(a_t))$ while $v_t = \mathsf{F}_{\mathsf{k}'}(a_t)$. For each t receiver R checks if $v_t = w_t$, and if so then it concludes that $a_t = b_t$ and outputs $m_t = e_t \oplus \mathsf{G}(p_t)$. To protect against collisions in (short) a_t, b_t values both within each protocol instance and across protocol instances each a_t and b_t is xor-ed by respectively S and H by a pre-shared one-time κ-bit random nonce r_t, with all nonces derived via a PRG on a seed shared by S and H.

Algorithm 4.1 Protocol $\mathsf{SS\text{-}COT}^{[N]}(\mathrm{S}, \mathrm{R}, \mathrm{H})$ - Secret-Shared Conditional OT

Input: S's input $(m_1, ..., m_N)$ and $(a_1, ..., a_N)$, H's input $(b_1, ..., b_N)$.
Output: R outputs pairs (t, m_t) s.t. $a_t = b_t$.
Parameters: ℓ and ℓ' s.t. $|m_t| = \ell$ and $|a_t| = |b_t| = \ell' \leq \kappa$ for all t.
Pre-computation phase: S, H share PRF F keys k, k' and κ-bit random nonces $r_1, ..., r_N$.

1: S sends $\{(e_t, v_t) = (\mathsf{G}^\ell(\mathsf{F}^\kappa_\mathsf{k}(x_t)) \oplus m_t, \mathsf{F}^\kappa_{\mathsf{k}'}(x_t))\}_{t=1}^N$ to R where $x_t = r_t \oplus [a_t | 0^{\kappa-\ell'}]$.
2: H sends $\{(p_t, w_t) = (\mathsf{F}^\kappa_\mathsf{k}(y_t), \mathsf{F}^\kappa_{\mathsf{k}'}(y_t))\}_{t=1}^N$ to R where $y_t = r_t \oplus [b_t | 0^{\kappa-\ell'}]$.
3: R outputs (t, m_t) where $m'_t = e_t \oplus \mathsf{G}^\ell(p_t)$ for each t s.t. $v_t = w_t$.

Secret-Shared Index OT, $\mathsf{SS\text{-}IOT}^{[2^\tau]}(\mathrm{S}, \mathrm{R}, \mathrm{H})$, is a close variant of Secret-Shared Conditional OT, where S holds a list of messages $(m_1, \ldots, m_N)$ for $N = 2^\tau$ and an index share $j_\mathrm{S} \in \{0,1\}^\tau$ while H holds the other share $j_\mathrm{H} \in \{0,1\}^\tau$, and the aim of the protocol is for R to output (j, m_j) s.t. $j = j_\mathrm{S} \oplus j_\mathrm{H}$. Our protocol for $\mathsf{SS\text{-}IOT}^{[2^\tau]}$ executes similarly to $\mathsf{SS\text{-}COT}^{[N]}$ except H sends only two values, $(p, v) = (\mathsf{F}_\mathsf{k}(j_\mathrm{H}), \mathsf{F}_{\mathsf{k}'}(j_\mathrm{H}))$ and S's messages are computed as $e_t = \mathsf{G}(\mathsf{F}_\mathsf{k}(j_\mathrm{S} \oplus t))$ and $v_t = \mathsf{F}_{\mathsf{k}'}(j_\mathrm{S} \oplus t)$. Finally, to avoid correlations across protocol instances, H and S xor their PRF inputs with a single pre-shared random κ-bit nonce r.

Shuffle OT, $\mathsf{XOT}\left[\binom{N}{k}\right](\mathrm{S}, \mathrm{R}, \mathrm{I})$, is a protocol between sender S, receiver R, and *indicator* I, where S inputs a sequence of messages $m_1, ..., m_N$, I inputs a sequence of indexes $i_1, ..., i_k$

and a sequence of masks $\delta_1, ..., \delta_k$, and the protocol lets R output a sequence of messages $m_{i_1} \oplus \delta_1, ..., m_{i_k} \oplus \delta_k$, without leaking anything else about S's and I's inputs. See Alg. 4.2 for an implementation of this protocol.

Algorithm 4.2 Protocol $\mathsf{XOT}\binom{N}{k}(\mathrm{S}, \mathrm{R}, \mathrm{I})$ - Shuffle OT

Input: S's input $(m_1, ..., m_N)$ and I's input $(i_1, ..., i_k)$ and $(\delta_1, ..., \delta_k)$.
Output: R's output $(z_1, ..., z_k)$ s.t. $z_\sigma = m_{i_\sigma} \oplus \delta_\sigma$ for all σ.
Parameters: Let $|m_t| = \ell$ for all t.
Pre-computation phase: I and S pick a random permutation π on $(1, ..., N)$ and a sequence of ℓ-bit random pads $r_1, ..., r_N$.

1: S sends $(a_1, ..., a_N) = (m_{\pi(1)} \oplus r_1, ..., m_{\pi(N)} \oplus r_N)$ to R.
2: I sends $(j_1, ..., j_k) = (\pi^{-1}(i_1), ..., \pi^{-1}(i_k))$ and $(p_1, ..., p_k) = (r_{j_1} \oplus \delta_1, ..., r_{j_k} \oplus \delta_k)$ to R.
3: R outputs $(z_1, ..., z_k) = (a_{j_1} \oplus p_1, ..., a_{j_k} \oplus p_k)$.
(Note that $z_\sigma = (m_{\pi(j_\sigma)} \oplus r_{j_\sigma}) \oplus (r_{j_\sigma} \oplus \delta_\sigma) = m_{i_\sigma} \oplus \delta_\sigma$ because $\pi(j_\sigma) = i_\sigma$.)

Secret-Shared Shuffle OT, $\mathsf{SS\text{-}XOT}\binom{N}{k}(\mathrm{A}, \mathrm{B}, \mathrm{I})$, involves indicator I and two parties A and B. It is a close variant of the Shuffle OT above, where I holds indexes $i = (i_1, ..., i_k)$, the pads $\delta_1, ..., \delta_k$ are all set to zero, and both inputs $m_1, ..., m_k$ and outputs $m_{i_1}, ..., m_{i_k}$ are secret shared by A and B. We implement this protocol with two instances of $\mathsf{XOT}\binom{N}{k}$, shown in Alg. 4.3. The indicator I first chooses a sequence of random masks $\delta = (\delta_1, ..., \delta_k)$, and inputs i, δ into both instances, where the first instance runs on A's input $(\mathrm{s_A}[m_1], ..., \mathrm{s_A}[m_N])$, and lets B output $(\mathrm{s_A}[m_{i_1}] \oplus \delta_1, ..., \mathrm{s_A}[m_{i_k}] \oplus \delta_k)$, while the second instance runs on B's inputs $(\mathrm{s_B}[m_1]$, and lets A output $(\mathrm{s_B}[m_{i_1}] \oplus \delta_1, ..., \mathrm{s_B}[m_{i_k}] \oplus \delta_k)$. It's easy to see that these outputs form a randomized A/B secret-sharing of $(m_{i_1}, ..., m_{i_k})$.

Shifting a Secret-Shared Sequence, $\mathsf{Shift}(\mathrm{A}, \mathrm{B}, \mathrm{H})$. As the access protocol traverses the forest of $(h+1)$ ORAM trees $\mathsf{forest} = (\mathsf{tree}_0, \mathsf{tree}_1, ..., \mathsf{tree}_h)$, parties D and E recover the secret sharing of path $\mathrm{P}_{\mathrm{L}^i}$ for label L^i, for $i = 1, ..., h$, and make several modifications to it. In particular, the buckets in the path are rotated by a random shift σ_i known to D and E. In the eviction protocol on this retrieved path we need a sub-protocol Shift to reverse this shift by transforming the secret-sharing of this path, which is a sequence of buckets, to a

Algorithm 4.3 Protocol $\mathsf{SS\text{-}XOT}\binom{N}{k}(\mathrm{A},\mathrm{B},\mathrm{I})$ - Secret-Shared Shuffle OT

Input: A/B secret-sharing of $(m_1,...,m_N)$, index vector $(i_1,...,i_k)$ held by I.
Output: A/B secret-sharing of $(m'_1,...,m'_k)$ s.t. $y_\sigma = m_{i_\sigma}$ for all σ.
Parameters: Let $|m_t| = \ell$ for all t.
Pre-computation phase: I generates k random bit-strings $(\delta_1,...,\delta_k)$ of length ℓ each.

1: Parties run $\mathsf{XOT}\binom{N}{k}(\mathrm{B},\mathrm{A},\mathrm{I})$ (see Alg. 4.2) on inputs $(\mathrm{s_B}[m_1],...,\mathrm{s_B}[m_N])$ for B and $(i_1,...,i_k)$ and $(\delta_1,...,\delta_k)$ for I. Protocol outputs $(a_1,...,a_k)$ for A s.t. $a_j = \mathrm{s_B}[m_{i_j}] \oplus \delta_j$. A outputs $(\mathrm{s_A}[y_1],...,\mathrm{s_A}[y_k]) = (a_1,...,a_k)$.
2: Parties run $\mathsf{XOT}\binom{N}{k}(\mathrm{A},\mathrm{B},\mathrm{I})$ (see Alg. 4.2) on inputs $(\mathrm{s_A}[m_1],...,\mathrm{s_A}[m_N])$ for A and $(i_1,...,i_k)$ and $(\delta_1,...,\delta_k)$ for I. Protocol outputs $(b_1,...,b_k)$ for B s.t. $b_j = \mathrm{s_A}[m_{i_j}] \oplus \delta_j$. B outputs $(\mathrm{s_B}[y_1],...,\mathrm{s_B}[y_k]) = (b_1,...,b_k)$.

(fresh) secret-sharing of the same buckets but rotated back by σ_i positions. An inexpensive implementation of this task relies on the fact that in our three-party setting player D can act as a "helper" party and create, in pre-computation, *correlated* random inputs for E and C, which allows for an on-line protocol which consists of a few xor operations and a transmission of a single $|\mathrm{P_{L^i}}|$-bit message from C to E.

Yao's Garbled Circuit on Secret-Shared Inputs. The last component used in our ORAM construction is protocol $\mathsf{GC}[\mathsf{F}](\mathrm{A},\mathrm{B},\mathrm{R})$, a Yao's garbled circuit solution for secure computation of an arbitrary function [49], executing on public inputs a circuit of function F, where the inputs X to this circuit are secret-shared between A and B, i.e., A's inputs $\mathrm{s_A}[X]$ and B's inputs $\mathrm{s_B}[X]$, and the protocol lets R compute $\mathsf{F}(X)$. We stress that even though we do use Yao's garbled circuit evaluation as a sub-protocol in our 3PC-ORAM scheme, we use it sparingly, and the computation involved is comparable, for realistic $\log(n)$ values, to the necessary cost of decryption of paths $\mathrm{P_{L^i}}$ retrieved by the underlying binary-tree client-server ORAM scheme. The protocol is a simple modification of the delivery of the input-wire keys in Yao's protocol, adopted to the setting where the input X is secret-shared by parties A and B, while the third party R will compute the garbled circuit and get the $\mathsf{F}(X)$. Let $m = |X|$ and let κ be the bit-length of the keys used in Yao's garbled circuit. In the off-line stage either A or B, say party A, prepares the garbled circuit for function F

and sends it to R, and then for each input wire key pair (K_i^0, K_i^1) created by Yao's circuit garbling procedure, A picks random Δ_i in $\{0,1\}^\kappa$, computes $(A_i^0, A_i^1) = (\Delta_i, K_i^0 \oplus K_i^1 \oplus \Delta_i)$ and $(B_i^0, B_i^1) = (K_i^0 \oplus \Delta_i, K_i^1 \oplus \Delta_i)$, and sends (B_i^0, B_i^1) to B (to optimize pre-computation A can send to B a random seed from which $\{K_i^0, K_i^1, \Delta_i\}_{i=1}^m$ can be derived via a PRG). In the on-line phase, for each $i = 1, ..., m$, party A on input bit $a = \mathrm{s_A}[X_i]$ sends A_i^a to R, while party B on input bit $b = \mathrm{s_B}[X_i]$ sends B_i^b. For each $i = 1, ..., m$, party R computes $K_i = A_i \oplus B_i$ for A_i, B_i received respectively from A and B, and then runs Yao's evaluation procedure inputting keys $K_1, ..., K_m$ into the garbled circuit received for F. Observe that $A_i \oplus B_i = K_i^v$ for $v = a \oplus b$, and hence if a, b is the XOR secret-sharing of the i-th input bit, i.e., if $a \oplus b = X_i$, then $K_i = K_i^0$ if $X_i = 0$ and $K_i = K_i^1$ if $X_i = 1$. The protocol is secure because of the random pad Δ_i, since for every X_i and every possible sharing (a, b) of X_i, values (A_i, B_i) sent to R are distributed as two random bit-strings s.t. $A_i \oplus B_i = K_i^v$ for $v = X_i$.

4.3 3PC-ORAM Protocol

In this section we describe our 3PC-ORAM access protocol. The basic idea for the protocol is to secret-share the data structure forest between two parties D and E, and have these two parties implement the server's algorithm of the underlying binary-tree client-server ORAM scheme, while the corresponding client's algorithm will be implemented with a three-party secure computation involving parties C, D, E. In the description below we combine these two conceptually separate parts into a single protocol, but almost all of the protocol implements the three-party computation of the ORAM client's algorithm, as the server's side consists only of retrieving (the shares of) path $\mathrm{P}_{\mathrm{L}^i}$ from (the shares of) the i-th tree tree_i at the beginning of i-th iteration of the access procedure, and then writing (the shares of) a new path $\mathrm{P}^\diamond_{\mathrm{L}^i}$ in place of (the shares of) $\mathrm{P}_{\mathrm{L}^i}$ at the end.

Given this secret-sharing scenario, the task of the 3PC-ORAM protocol is to securely compute the following two functionalities:

1. The *retrieval* functionality computes the next-tree label $\mathrm{L}^{i+1} = \mathrm{F}_i(\mathrm{N}^{i+1})$ given the D/E secret-sharing of path $\mathrm{P}_{\mathrm{L}^i}$, for $\mathrm{L}^i = \mathrm{F}_{i-1}(\mathrm{N}^i)$ and the D/E secret-sharing of address prefix N^{i+1};

2. The *eviction* functionality computes the D/E secret-sharing of path $\mathrm{P}^{\diamond}_{\mathrm{L}^i}$ output by the eviction algorithm applied to the D/E secret-shared path $\mathrm{P}_{\mathrm{L}^i}$, after the tuple containing the label identified by the access functionality is moved to the root node.

Both tasks can be computed using standard secure computation techniques but the protocol we show beats a generic one by a few orders of magnitude, and comes close to the computation cost of the underlying client-server ORAM itself. Note that the i-th iteration of client-server ORAM needs a server-to-client transmission and decryption of path $\mathrm{P}_{\mathrm{L}^i}$ and then encryption and client-to-server transmission of path $\mathrm{P}^{\diamond}_{\mathrm{L}^i}$ (see Section 2.3). Therefore the base-line cost we want the 3PC-ORAM to come close to are $h+1$ rounds of client-server interaction with $2 \cdot |\mathrm{P}_{\mathrm{L}^i}|$ bandwidth and $(2/\kappa) \cdot |\mathrm{P}_{\mathrm{L}^i}|$ block cipher operations for $i = 0, ..., h$. The main idea which allows us to come close to these parameters is that if the inputs to either access or eviction functionalities, secret-shared by two parties, e.g., D and E, are permuted and masked in an appropriate way, then the correspondingly permuted and masked outputs of these functionalities can be revealed to the third party, e.g., C.

In the 3-party setting we separate the ORAM access protocol into Retrieval, PostProcess, and Eviction. Protocol Retrieval contains all parts of the client-server access which have to be executed *sequentially*, i.e., for memory address N, the retrieval of sequence $\mathrm{F}_{\mathsf{forest}}(\mathrm{N}) = (\mathrm{L}^1, \mathrm{L}^2, ..., \mathrm{L}^h, \mathsf{M}[\mathrm{N}])$ done by sequential identification (and removal from the tree_i trees) of the tuple sequence $(\mathrm{T}^1, \mathrm{T}^2, ..., \mathrm{T}^h)$ where T^i is defined as path $\mathrm{P}_{\mathrm{L}^i}$ of tree tree_i whose address field is equal to N's prefix N^i and whose rec field contains label L^{i+1} at position $\mathrm{N}^{(i+1)}$. Protocol

PostProcess performs cleaning-up operations on each tuple T^i in this tuple sequence, by modifying its label field from L^i to $(\mathrm{L}^i)'$ and modifying the label held at $\mathrm{N}^{(i+1)}$-th position in the rec^i array of this tuple from L^{i+1} to $(\mathrm{L}^{i+1})'$. Importantly, the PostProcess and Eviction protocols can be done *in parallel* for all trees tree_i, which allows for a better CPU utilization in the protocol execution.

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

Protocol Retrieval proceeds by executing loop Retrieval$[i]$ *sequentially*, see Alg. 4.4, for $i = 0, \ldots, h$. The inputs to Retrieval$[i]$ are: (1) D/E secret-sharing of tree_i; (2) D/E secret sharing of address prefix $\mathrm{N}^{i+1} = [\mathrm{N}^i | \mathrm{N}^{(i+1)}]$; (3) Leaf label L^i as the input of D and E (with N^0, $\mathrm{N}^{(h+1)}$, and L^0 all empty strings). Its outputs are: (1) C's output the next leaf label $\mathrm{L}^{i+1} = \mathrm{F}_i(\mathrm{N}^{i+1})$, for $i \neq h$, or the C/E secret-sharing of record $\mathsf{rec} = \mathsf{M}[\mathrm{N}]$, for $i = h$; (2) C/E secret-sharing of tuple T^i defined above; and (3) C/E secret-sharing of path $\mathsf{Rot}^{[\sigma_i, \delta_i, \rho_i]}(\mathrm{P}'_{\mathrm{L}^i})$ which results from rotating the data in $\mathrm{P}_{\mathrm{L}^i}$ by three random shifts $(\sigma_i, \delta_i, \rho_i)$ known to E and D (and of removing T^i from $\mathrm{P}_{\mathrm{L}^i}$).

Data-Rotations and Conditional OT's. We first explain how E and D perform the three data-rotations on the secret-shared path $\mathrm{P}_{\mathrm{L}^i}$ retrieved from the (shares of) the i-th level ORAM tree tree_i (and randomized by D and E xor-ing the shares of $\mathrm{P}_{\mathrm{L}^i}$ retrieved from tree_i by a pre-agreed random pad). E and D pick three values during preprocessing, $\sigma_i, \delta_i, \rho_i$, at random in ranges resp. $\{1, ..., \mathrm{d}_i + 4\}$, $\{1, ..., w\}$, and $\{0,1\}^\tau$, for d_i the depth of tree_i. The

Algorithm 4.4 Protocol $\mathsf{Retrieval}[i]$ - Oblivious Retrieval of Next Label

Input: D, E's inputs: label L^i and secret-sharing of tree_i and $\mathrm{N}^{i+1} = [\mathrm{N}^i|\mathrm{N}^{(i+1)}]$;
Output: (1) C outputs $\mathrm{L}^{i+1} = \mathsf{rec}^i[\mathrm{N}^{(i+1)}]$ where rec^i is the rec field of tuple T^i in $\mathrm{P}_{\mathrm{L}^i}$ whose address field matches N^i; (2) C and E output a secret-sharing of T^i and $\mathrm{P}^*_{\mathrm{L}^i} = \mathsf{Rot}^{[\sigma,\delta,\rho]}(\mathrm{P}'_{\mathrm{L}^i})$, where $\mathrm{P}^*_{\mathrm{L}^i}$ is $\mathrm{P}_{\mathrm{L}^i}$ without tuple T^i; (3) D & E output σ, ρ;
Pre-computation phase: D & E's input: $(\sigma, \delta, \rho, p) \leftarrow [\mathrm{d}_i{+}4] \times [w] \times \{0,1\}^\tau \times \{0,1\}^{|\mathrm{P}_{\mathrm{L}^i}|}$;
Parameters: $n = w(\mathrm{d}_i{+}4)$.

1: D retrieves share $\mathrm{s_D}[\mathrm{P}_{\mathrm{L}^i}]$ from $\mathrm{s_D}[\mathsf{tree}_i]$ and sets $\mathrm{s_D}[\mathsf{Rot}^{[\sigma,\delta,\rho]}(\mathrm{P}_{\mathrm{L}^i})]$ as the result of the three data-rotations using shifts (σ, δ, ρ) applied to $(\mathrm{s_D}[\mathrm{P}_{\mathrm{L}^i}] \oplus p)$. E computes $\mathrm{s_E}[\mathsf{Rot}^{[\sigma,\delta,\rho]}(\mathrm{P}_{\mathrm{L}^i})]$ in the corresponding way.
2: D sends $\mathrm{s_D}[\mathsf{Rot}^{[\sigma,\delta,\rho]}(\mathrm{P}_{\mathrm{L}^i})]$ and $\mathrm{s_D}[\mathrm{N}^{i+1}] = (\mathrm{s_D}[\mathrm{N}^i]|\mathrm{s_D}[\mathrm{N}^{(i+1)}])$ to C.
3: D and E isolate in their shares of $\mathsf{Rot}^{[\sigma,\delta,\rho]}(\mathrm{P}_{\mathrm{L}^i})$ a vector of shares of pairs $(\mathrm{fb}_j, \mathrm{N}_j)$ for $j = 1, ..., n$ of fb and adr fields of all tuples in this (rotated) path. E also isolates in $\mathrm{s_E}[\mathsf{Rot}^{[\sigma,\delta,\rho]}(\mathrm{P}_{\mathrm{L}^i})]$ shares $(\mathrm{s_E}[\mathsf{Rot}^{[\rho]}(\mathsf{rec}_1)], ..., \mathrm{s_E}[\mathsf{Rot}^{[\rho]}(\mathsf{rec}_n)])$ of the rec field of all tuples. The parties then run $\mathsf{SS\text{-}COT}^{[n]}(\mathrm{E}, \mathrm{C}, \mathrm{D})$ on E's input $(m_1, ..., m_n)$ and $(a_1, ..., a_n)$ and D's input $(b_1, ..., b_n)$ where $m_t = \mathrm{s_E}[\mathsf{Rot}^{[\rho]}(\mathsf{rec}_t) \oplus y]$, $a_t = \mathrm{s_E}[\mathrm{fb}_t|\mathrm{N}_t] \oplus [0|\mathrm{s_E}[\mathrm{N}^i]]$, and $b_t = \mathrm{s_D}[\mathrm{fb}_t|\mathrm{N}_t] \oplus [1|\mathrm{s_D}[\mathrm{N}^i]]$. This sub-protocol outputs $(j_1, \bar{e})$ for C s.t. $[\mathrm{fb}_{j_1}|\mathrm{N}_{j_1}] = [1|\mathrm{N}^i]$ and $\bar{e} = y \oplus \mathrm{s_E}[\mathsf{Rot}^{[\rho]}(\mathsf{rec}_{j_1})]$. The client computes $z = \bar{e} \oplus \bar{d}$ where $\bar{d}$ is the rec field in the j_1-th tuple in $\mathrm{s_D}[\mathsf{Rot}^{[\sigma,\delta,\rho]}(\mathrm{P}_{\mathrm{L}^i})]$. (Note that j_1-th tuple in $\mathsf{Rot}^{[\sigma,\delta,\rho]}(\mathrm{P}_{\mathrm{L}^i})$ is equal to T^i, hence $\mathsf{rec}_{j_1} = \mathsf{rec}^i$ and $z \oplus y = \mathsf{Rot}^{[\rho]}(\mathsf{rec}^i)$.)
4: Parties run $\mathsf{SS\text{-}IOT}^{[2^\tau]}(\mathrm{E}, \mathrm{C}, \mathrm{D})$ on E's input $(y_1, \ldots, y_{2^\tau})$ and $\mathrm{s_E}[\mathrm{N}^{(i+1)}]$ and D's input $\mathrm{s_D}[\mathrm{N}^{(i+1)}] \oplus \rho_i$, which outputs pair (j_2, y_{j_2}) for C.
5: Each party computes its output as follows:

- C outputs $\mathrm{L}^{i+1} = y_{j_2} \oplus z_{j_2}$ where z_{j_2} is j_2-th d_{i+1}-bit segment in z;
- C and E form $(\mathrm{s_C}[\mathrm{T}^i], \mathrm{s_E}[\mathrm{T}^i])$ as $((1, \mathrm{s_D}[\mathrm{N}^i], 0^{\mathrm{d}_i}, z), (0, \mathrm{s_E}[\mathrm{N}^i], \mathrm{L}^i, y))$;
- C and E form secret-sharing of $\mathrm{P}^*_{\mathrm{L}^i}$ by C setting its share to $\mathrm{s_D}[\mathsf{Rot}^{[\sigma,\delta,\rho]}(\mathrm{P}_{\mathrm{L}^i})]$ but with the j_1-th tuple modified by flipping bit fb and setting its other bits at random, and E setting its share to $\mathrm{s_E}[\mathsf{Rot}^{[\sigma,\delta,\rho]}(\mathrm{P}_{\mathrm{L}^i})]$;
- D and E output (σ, ρ).

data-rotation defined by σ_i is performed on the bucket level, i.e., the $\mathrm{d}_i + 4$ buckets in path $\mathrm{P}_{\mathrm{L}^i}$ (recall that there are d_i internal nodes containing a bucket each and that the leaf node contains 4 buckets) are rotated clock-wise by σ_i positions. The data-rotation defined by δ_i is performed on the level of tuples within each bucket, i.e., in each of the $\mathrm{d}_i + 4$ buckets in $\mathrm{P}_{\mathrm{L}^i}$ the sequence of w tuples held in that bucket is rotated clock-wise by δ_i positions. Finally, the bit-vector ρ_i defines τ flips which will be applied to the array rec in each of the $(\mathrm{d}_i + 4) \cdot w$ tuples in the path. Namely, the rec field in each tuple in the path is treated

as a τ-dimensional cube whose content is flipped along the j-th dimension if the j-th bit in ρ_i is 1. Such τ flips define a permutation on elements of rec where an element at position t moves to position $t \oplus \rho_i$, for each $t \in \{0,1\}^\tau$. Note that E and D can perform all these data-rotations locally on their shares of the path $\mathrm{P}_{\mathrm{L}^i}$. We use $\mathsf{Rot}^{[\sigma_i,\delta_i,\rho_i]}(\mathrm{P}_{\mathrm{L}^i})$ to denote the resulting tree, and we use $\mathsf{Rot}^{[\rho_i]}(\mathsf{rec})$ to denote the result of the permutation defined by $\rho_i \in \{0,1\}^\tau$ on field rec as explained above. After applying these data-rotations to $\mathrm{P}_{\mathrm{L}^i}$ the parties run protocols $\mathsf{SS\text{-}COT}^{[n]}$ and $\mathsf{SS\text{-}IOT}^{[2^\tau]}$ described in Section 4.2, with E as the sender, D as the helper, and C as the receiver in both protocols. The goal of protocol $\mathsf{SS\text{-}COT}^{[n]}$, for $m = (\mathrm{d}_i + 4) \cdot w$, is two-fold: (1) to let C compute the index $j_1 \in \{1, ..., m\}$ where path $\mathsf{Rot}^{[\sigma_i,\delta_i,\rho_i]}(\mathrm{P}_{\mathrm{L}^i})$ contains the unique tuple T^i defined above (i.e., the tuple that contains the searched-for address prefix N^i); and (2) to create a C/E secret-sharing of this tuple. The goal of $\mathsf{SS\text{-}IOT}^{[2^\tau]}$ is to let C compute the $\mathrm{N}^{(i+1)}$-th entry in the rec field of this secret-shared tuple T^i, because that field contains the next-tree label $\mathrm{L}^{i+1} = \mathrm{F}_i(\mathrm{N}^{i+1})$.

Note that D and E hold the secret-sharing of N^i and for each $t = 1, ..., m$ they also hold the shares of the address N_t in the t-th tuple in $\mathsf{Rot}^{[\sigma_i,\delta_i,\rho_i]}(\mathrm{P}_{\mathrm{L}^i})$. If D and E form values a_t and b_t as an xor of these two sharings, i.e., $a_t = \mathrm{s}_\mathrm{E}[\mathrm{N}^i \oplus \mathrm{s}_\mathrm{E}[\mathrm{N}_t]]$ and $b_t = \mathrm{s}_\mathrm{D}[\mathrm{N}^i \oplus \mathrm{s}_\mathrm{D}[\mathrm{N}_t]]$ then $a_t = b_t$ if and only if $\mathrm{N}_t = \mathrm{N}^i$, i.e., if and only if t points to a unique tuple T^i in (rotated) path $\mathsf{Rot}^{[\sigma_i,\delta_i,\rho_i]}(\mathrm{P}_{\mathrm{L}^i})$ whose address field N equals the searched-for address N^i. Therefore if D and E run the Secret-Shared Conditional OT $\mathsf{SS\text{-}COT}^{[m]}$ on $(a_1, ..., a_m)$ and $(b_1, ..., b_m)$ defined above as their condition-share vectors, then C will compute the index j_1 to the searched-for tuple T^i contained in this path. Moreover, $\mathsf{SS\text{-}COT}^{[m]}$ will also compute the secret-sharing of T^i if E picks a random pad y of length $2^\tau \cdot \mathrm{d}_{i+1}$, and defines the message vector it inputs to $\mathsf{SS\text{-}COT}^{[m]}$ as $(v_1, ..., v_n)$ where v_t is an xor of y with E's share of the rec field in the t-th tuple in $\mathsf{Rot}^{[\sigma_i,\delta_i,\rho_i]}(\mathrm{P}_{\mathrm{L}^i})$. Note that the rec field in any entry in the rotated path corresponds to array $\mathsf{Rot}^{[\rho_i]}(\mathsf{rec})$ where rec was the field of the corresponding entry in the original path. Therefore C's output in this $\mathsf{SS\text{-}COT}^{[m]}$ instance will be j_2 together with $\bar{e} = y \oplus \mathrm{s}_\mathrm{E}[\mathsf{Rot}^{[\rho_i]}(\mathsf{rec}^i)]$ where the searched-for tuple T^i is defined as $(1, \mathrm{L}^i, \mathrm{N}^i, \mathsf{rec}^i)$.

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

It remains for us to explain how $\mathsf{SS\text{-}IOT}^{[2^\tau]}$ computes an entry in this secret-shared field that corresponds to the next-level address chunk $\mathrm{N}^{(i+1)}$, because that's the entry which contains $\mathrm{L}^{i+1} = \mathrm{F}_i(\mathrm{N}^{i+1})$. Note that E and D hold the secret-sharing of $\mathrm{N}^{(i+1)}$ and that they also hold the bit-vector ρ_i s.t. the entry at t-th position in rec^i is located at position $t \oplus \rho_i$ in $\mathsf{Rot}^{[\rho_i]}(\mathsf{rec}^i)$. Since L^{i+1} sits at the t-th position in rec^i for $t = \mathrm{N}^{(i+1)}$, we will find if we retrieve the j_2-th entry of $\mathsf{Rot}^{[\rho_i]}(\mathsf{rec}^i)$ for $j_2 = \mathrm{N}^{(i+1)} \oplus \rho_i$. Note, however, that e.g., $\mathrm{s}_{\mathrm{D}}[\mathrm{N}^{(i+1)}] \oplus \rho_i$ and $\mathrm{s}_{\mathrm{E}}[\mathrm{N}^{(i+1)}]$ form a secret-sharing of j_2, and therefore the Secret-Shared Index OT protocol $\mathsf{SS\text{-}IOT}^{[2^\tau]}$ executed on sharing $(\mathrm{s}_{\mathrm{D}}[\mathrm{N}^{(i+1)}] \oplus \rho_i, \mathrm{s}_{\mathrm{E}}[\mathrm{N}^{(i+1)}])$ and E's data vector $y = (y_1, ..., y_{2^\tau})$, will let C output j_2 together with the j_2-th fragment y_{j_2} of y. Since (z, y) form the secret-sharing of $\mathsf{Rot}^{[\rho_i]}(\mathsf{rec}^i)$, C can compute the j_2-th entry of $\mathsf{Rot}^{[\rho_i]}(\mathsf{rec}^i)$, i.e., the next-level tree label L^{i+1}, by xor-ing y_{j_2} with j_2-th fragment of $z = (z_1, ..., z_{2^\tau})$.

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

Boundary Cases. Alg. 4.4 shows protocol $\mathsf{Retrieval}[i]$ for $0 < i < h$. For $i = 0$ tree_0 contains a single node, shifts σ_0, δ_0 are not used, sub-protocol $\mathsf{SS\text{-}COT}^{[m]}$ is skipped, index j_1 is not used, and the outputs include only j_2 for C, ρ_0 for D and E, and the C/E secret-sharing of T^0 (with L^0 and N^0 set to empty strings). For $i = h$ the $\mathsf{SS\text{-}IOT}^{[2^\tau]}$ sub-protocol is skipped, shift ρ_h and index j_2 are not used, and (z, y) held by C and E form a secret-sharing of record $\mathsf{M}[\mathrm{N}]$.

PostProcess. The post-process protocol PostProcess transforms the C/E secret-shared tuples $\mathrm{T}^0, ..., \mathrm{T}^h$ output by Retrieval to prepare the inputs for protocol Eviction. It does so by executing a loop $\mathsf{PostProcess}[\mathrm{i}]$ in Alg. 4.5 *in parallel* for $i = 0, ..., h-1$ (tuple T^h is not part of this step). The goal of post-processing is to replace the L^{i+1} value which sits at the j_2-th position in the rec field of the *secret-shared* tuple T^i (where j_2 is an index C learns in $\mathsf{Retrieval}[i]$), with the *secret-shared* value $(\mathrm{L}^{i+1})'$. In other words, we need to inject a secret-shared value into a secret-shared array at a secret position known only to one party. However, we can utilize the fact that this secret-shared value to be injected can be chosen in preprocessing and that E's share of it can be revealed to D. Let $(c, e) = (\mathrm{s_C}[(\mathrm{L}^{i+1})'], \mathrm{s_E}[(\mathrm{L}^{i+1})'])$ and let E sends its share e to D in preprocessing. If D precomputes two $|\mathsf{rec}|$-long correlated random pads, one for C and one for E, with the known difference e between them at random location α known to C, then $e \oplus c$ can be injected at position j_2 into the C/E secret-sharing of T^i if (1) C sends $\delta = \alpha - j_2 \bmod 2^\tau$ to E, (2) both parties rotate the pads they receives from D counter-clockwise by δ positions, in this way placing the unique pad cells that differ by e at position j_2, (3) both parties xor their shares of T^i with these pads, with C injecting an xor with c at position j_2 into her share (in addition C will also erase the previous leaf value at position j_2 in rec field of T^i by adding L^{i+1} to that xor).

Eviction. Protocol Eviction executes sub-protocol $\mathsf{Eviction}[i]$ in Alg. 4.6 in *parallel* for each $i = 0, ..., h$ (for $i = 0$ protocol $\mathsf{Eviction}[i]$ skips all the steps in Alg. 4.6 except the last

Algorithm 4.5 Protocol $\mathsf{PostProcess}[i]$ - Inserting New Labels into T^i

Input: C's input $\mathrm{s_C}[\mathrm{T}^i], \mathrm{L}^i, \mathrm{L}^{i+1}, j_2$; E's input $\mathrm{s_E}[\mathrm{T}^i]$;
Input known in pre-computation: C/D secret-sharing of labels $(\mathrm{L}^i)'$ and $(\mathrm{L}^{i+1})'$, where E forwards its shares to D;
Output: E/C secret-sharing of tuple $(\mathrm{T}^i)' = (1, \mathrm{N}^i, (\mathrm{L}^i)', \mathsf{rec}')$ where $\mathsf{rec}'[j_2] = (\mathrm{L}^{i+1})'$ and $\mathsf{rec}'[t] = \mathsf{rec}[t]$ for all $t \neq j_2$ where $\mathrm{T}^i = (1, \mathrm{N}^i, \mathrm{L}^i, \mathsf{rec})$;
Pre-computation phase: D picks $r_1, ..., r_{2^\tau}$ in $\{0,1\}^{\mathrm{d}_{i+1}}$ and α in $\{0,1\}^\tau$, and sends $\alpha, r_1, ..., r_{2^\tau}$ to C and $s_1, ..., s_{2^\tau}$ to E s.t. $s_\alpha = r_\alpha \oplus \mathrm{s_E}[(\mathrm{L}^{i+1})']$ and $s_t = r_t$ for $t \neq \alpha$.

1: C sends $\delta = \alpha - j_2 \pmod{2^\tau}$ to E.
2: C outputs $\mathrm{s_C}[(\mathrm{T}^i)'] = \mathrm{s_C}[\mathrm{T}^i] \oplus (0, 0^{i\tau}, \mathrm{L}^i \oplus \mathrm{s_C}[(\mathrm{L}^i)'], (c_1|...|c_{2^\tau}))$ where $c_t = r_{t+\delta \pmod{2^\tau}}$ for all $t \neq j_2$ and $c_t = r_{t+\delta \pmod{2^\tau}} \oplus \mathrm{L}^{i+1} \oplus \mathrm{s_C}[(\mathrm{L}^{i+1})']$ for $t = j_2$.
3: E outputs $\mathrm{s_E}[(\mathrm{T}^i)'] = \mathrm{s_E}[\mathrm{T}^i] \oplus (0, 0^{i\tau}, \mathrm{s_E}[(\mathrm{L}^i)'], (e_1|...|e_{2^\tau}))$ where $e_t = s_{t+\delta \pmod{2^\tau}}$.

one). Sub-protocol $\mathsf{Eviction}[i]$ performs an ORAM eviction procedure on path $\mathrm{P}^*_{\mathrm{L}^i}$, whose C/E secret-sharing is output by protocol $\mathsf{Retrieval}$. The protocol has two parts: First, using Yao's garbled circuit protocol GC (see Section 4.2) it allows D to identify two tuples in each internal bucket of $\mathrm{P}^*_{\mathrm{L}^i}$ which are either movable one notch down this path or they are empty. Another instance of GC will similarly find two empty tuples in the four buckets corresponding to the leaf in $\mathrm{P}^*_{\mathrm{L}^i}$. The reason these pairs of indexes j_0, j_1 can be leaked to D is that (1) C and E randomly permute the tuples in each bucket in $\mathrm{P}^*_{\mathrm{L}^i}$ before using them in this protocol, and (2) index j_b computed for $b = 0, 1$ for each bucket in $\mathrm{P}^*_{\mathrm{L}^i}$ is defined as the first movable tuple in that bucket after a random offset λ_b (counting the tuples cyclically), where shifts λ_0, λ_1 are chosen by E independently for each bucket at random in $\{1, ..., w\}$. The circuit computed for every internal bucket takes only $2w$ bits of input (one for bit fb and one for an agreement in the i-th bit of a leaf label in the tuple and the i-th bit of label L^i defining path $\mathrm{P}^*_{\mathrm{L}^i}$), and has only about $16w$ non-xor gates. Once D gets two indexes per each bucket in the path, it uses the Secret-Shared Shuffle OT protocol $\mathsf{SS\text{-}XOT}\binom{k+2}{k}$ (see Section 4.2) to randomizes the secret-sharing of all tuples in $\mathrm{P}_{\mathrm{L}^i}$ while (1) moving the secret-shared tuple $(\mathrm{T}^i)'$ prepared by $\mathsf{PostProcess}$ into the root bucket, and (2) moving the two chosen tuples in each bucket to the space vacated by the two tuples chosen in the bucket below. Finally, C and E randomize their secret-sharing of the resulting path $\mathrm{P}^{**}_{\mathrm{L}^i}$ by xor-ing their shares with a pre-agreed random pad, C sends its share of $\mathrm{P}^{**}_{\mathrm{L}^i}$ to D, and D and E insert their respective

Algorithm 4.6 Protocol $\mathsf{Eviction}[i]$ - Eviction in Path $\mathrm{P}_{\mathrm{L}^i}$ of tree_i

Input: C/E secret-sharing of path $\mathrm{P}^*_{\mathrm{L}^i}$ and tuple $(\mathrm{T}^i)'$; σ, ρ held by E, D;
Output: D/E secret-sharing of path $\mathrm{P}^\diamond_{\mathrm{L}^i}$ to be inserted into tree tree_i in place of $\mathrm{P}_{\mathrm{L}^i}$.
Notation: Let $W = \{1, ..., w\}$, $\mathrm{IB} = \{0, ..., \mathrm{d}_i - 1\}$, and $\mathrm{EB} = \{\mathrm{d}_i, \mathrm{d}_i + 1, \mathrm{d}_i + 2, \mathrm{d}_i + 3\}$.
Pre-computation phase: C and E share random permutations $\pi_1, ..., \pi_{\mathrm{d}_i}$ on set $[w]$, a random permutation π_{d_i+1} on set $[4 \cdot w]$, and a random pad ξ of length $|\mathrm{P}_{\mathrm{L}^i}|$;

1: Parties run protocol $\mathsf{Shift}(\mathrm{C}, \mathrm{E}, \mathrm{D})$ on inputs C/E-secret-sharing of $\mathrm{P}^*_{\mathrm{L}^i}$ and on D, E input a shift σ. The protocol outputs a C/E-secret-sharing of path identical to $\mathrm{P}^*_{\mathrm{L}^i}$ but with buckets shifted back by σ positions. In addition, for each $j \in \mathrm{IB}$, C and E use π_j to permute (their shares of) the tuples in the j-th bucket in the resulting path, and they use π_{d_i+1} to permute (their shares of) the tuples in the four buckets corresponding to the leaf node. The resulting path, shared by C and E, is denoted $\mathrm{P}^{**}_{\mathrm{L}^i}$.

2: Let fb^j_ℓ and L^j_ℓ be the fb and L fields of the ℓ-th tuple in the j-th bucket in $\mathrm{P}^{**}_{\mathrm{L}^i}$. For each $j \in \mathrm{IB}$, parties run protocol $\mathsf{GC}[\mathsf{F2FT}](\mathrm{E}, \mathrm{C}, \mathrm{D})$, see Sec. 4.2, on C's inputs $\{\mathrm{s_C}[\mathrm{fb}^j_\ell, \mathrm{L}^j_\ell[j+1]]\}_{\ell \in W}$ and on E's inputs $\{\mathrm{s_E}[\mathrm{fb}^j_\ell], B_j\}_{\ell \in W}$ where $B_j = \mathrm{s_E}[\mathrm{L}^j_\ell[j+1]] \oplus 1 \oplus \mathrm{L}^i[j+1]$ (note that $\mathrm{s_C}[\mathrm{L}^j_\ell[j+1]] \oplus B_j = 1$ iff the secret-shared value L^j_ℓ and the public value L^i agree on $(j+1)$-st bit). For each $j \in \mathrm{IB}$, D defines $\alpha^1_j, \alpha^2_j \in [1, ..., w]$ as the indexes of the two output wires of $\mathsf{F2FT}$ on which D received output bit 1 in the j-th instance of $\mathsf{GC}[\mathsf{F2FT}]$.

3: The parties run protocol $\mathsf{GC}[\mathsf{F2ET}](\mathrm{E}, \mathrm{C}, \mathrm{D})$ on E's inputs $\{\mathrm{s_E}[\mathrm{fb}^j_\ell]\}_{\ell \in W, j \in \mathrm{EB}}$ and C's inputs $\{\mathrm{s_C}[\mathrm{fb}^j_\ell]\}_{\ell \in W, j \in \mathrm{EB}}$. D defines $\alpha^1_{\mathrm{d}_i}, \alpha^2_{\mathrm{d}_i} \in [1, ..., 4 \cdot w]$ as the indexes of the two output wires of $\mathsf{F2ET}$ on which D received output bit 1 in this instance of $\mathsf{GC}[\mathsf{F2ET}]$.

4: D prepares a sequence of $k = w \cdot (\mathrm{d}_i + 4)$ indexes $I = (\beta_1, ..., \beta_k)$ s.t.

$$\beta_{w \cdot j + \ell} = \begin{cases} k+1, & \text{if } j = 0 \text{ and } \ell = \alpha^1_0 \\ k+2, & \text{if } j = 0 \text{ and } \ell = \alpha^2_0 \\ w \cdot (j-1) + \alpha^1_{j-1}, & \text{if } 1 \le j \le \mathrm{d}_i - 1 \text{ and } \ell = \alpha^1_j \\ w \cdot (j-1) + \alpha^2_{j-1}, & \text{if } 1 \le j \le \mathrm{d}_i - 1 \text{ and } \ell = \alpha^2_j \\ w \cdot (\mathrm{d}_i - 1) + \alpha^1_{\mathrm{d}_i - 1}, & \text{if } j \ge \mathrm{d}_i \text{ and } w \cdot (j - \mathrm{d}_i) + \ell = \alpha^1_{\mathrm{d}_i} \\ w \cdot (\mathrm{d}_i - 1) + \alpha^2_{\mathrm{d}_i - 1}, & \text{if } j \ge \mathrm{d}_i \text{ and } w \cdot (j - \mathrm{d}_i) + \ell = \alpha^2_{\mathrm{d}_i} \\ w \cdot j + \ell & \text{otherwise} \end{cases}$$

and then divides I into $\mathrm{d}_i + 4$ chunks, each of which has w indexes, and permutes each chunk with the corresponding ϱ_r.

5: C prepares a sequence of $k+2$ shares $(\mathrm{s_C}[a_1], ..., \mathrm{s_C}[a_{k+2}])$ by setting $\mathrm{s_C}[a_{w \cdot j + \ell}] = \mathrm{s_C}[\mathrm{T}^j_\ell]$ where T^j_ℓ is ℓ-th tuple in j-th bucket B^j in $\mathrm{P}^{**}_{\mathrm{L}^i}$, for $\ell \in W$ and $j \in \mathrm{IB} \cup \mathrm{EB}$, $\mathrm{s_C}[a_{k+1}]$ as $\mathrm{s_C}[(\mathrm{T}^i)']$, and $\mathrm{s_C}[a_{k+2}]$ as 0 concatenated with a random string of $i \cdot \tau + \mathrm{d}_i + 2^\tau \cdot \mathrm{d}_{i+1}$ bits. E prepares a sequence of $k+2$ shares $(\mathrm{s_E}[a_1], ..., \mathrm{s_E}[a_{k+2}])$ in the corresponding way.

6: The parties run protocol $\mathsf{SS\text{-}XOT}\binom{k+2}{k}$ on C's input $(\mathrm{s_C}[a_1], ..., \mathrm{s_C}[a_{k+2}])$, E's input $(\mathrm{s_E}[a_1], ..., \mathrm{s_E}[a_{k+2}])$, and D's input I. C and E set their shares of path $\mathrm{P}^\diamond_{\mathrm{L}^i}$ to their output in this $\mathsf{SS\text{-}XOT}\binom{k+2}{k}$ protocol xor'ed with string ξ.

7: C sends $\mathrm{s_C}[\mathrm{P}^\diamond_{\mathrm{L}^i}]$ to D; D and E insert their shares of $\mathrm{P}^\diamond_{\mathrm{L}^i}$ into their shares of tree_i.

shares of $\mathrm{P}_{\mathrm{L}^i}^{**}$ into their shares of tree_i, in place of the shares of the original path $\mathrm{P}_{\mathrm{L}^i}$ retrieved in the first step of $\mathsf{Retrieval}[i]$.

4.4 Protocol Analysis

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

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

The performance of the scheme is linear in the bucket size parameter w, and the size of this parameter should be set so that the probability of overflow of any bucket throughout the execution of the scheme is bounded by $2^{-\lambda}$ for the desired statistical security parameter λ. The probability that an internal node overflows and the probability that a leaf node overflows are independent stochastic processes and for this reason we examine them separately. The

analytical bounds we give for both cases are not optimal. For the leaf node overflow probability the bound we give in Lemma 4.1 could be made tight if the number of ORAM accesses N is equal to the number of memory locations n, but for the general case of $\mathsf{N} > n$ we use a simple union bound which adds a N factor. If a tighter analysis could be made, it could potentially reduce the required w by up to $\log(\mathsf{N})$ bits. The bound we give for the internal node overflow probability in Lemma 4.2 is simplistic and clearly far the optimal. We amend this bound by a discussion of a stochastic model which we used to approximate the eviction process. If this approximation is close to the real stochastic process then the scheme can be instantiated with much smaller bucket sizes than those implied by Lemma 4.2.

Lemma 4.1. (Leaf Nodes) *If we have* N *accesses in an ORAM forest with the total capacity for* n *records and with leaf nodes which hold* $4w$ *entries, then the probability that some leaf node overflows at some access is bounded by:*

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

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

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

Lemma 4.2. (Internal Nodes) *If we have* N *accesses and subsequent evictions in an ORAM forest with internal buckets of size* w*, then the probability that some internal bucket overflows at some access is bounded by:*

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

We can prove Lemma 4.2 by assuming that there exists an internal node that during all accesses and subsequent evictions is on the verge of overflowing (has w or $w-1$ entries in it). We also assume the worst case of each node always receiving exactly two new entries, and we compute the probability that a node is not able to evict two entries, thus causing an overflow.

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

Stochastic Approximation. The above analysis is pessimistic, since it assumes that there exists a critical bucket that is always full, having w or $w-1$ entries and bounds the probability of such a bucket having a "bad event". It does not explore how difficult it is for a bucket to reach such a state, or how a congested bucket is emptying over time. In order to better understand such behaviors we observe that each internal node can be modeled as a Markov Chain, where the state of the chain counts how many entries are currently in the node. The node is initially empty. Whenever a node is selected in an eviction path it may receive up to two entries depending on whether the parent node was able to evict one or two entries. Moreover the node could evict up to two entries to its child that participates in the eviction path. The root always receives 1 entry and may evict up to two entries. Intuitively since the eviction path is picked at random and each entry is assigned to a random leaf node, each entry in a node in the eviction path can be evicted to the selected child node with probability

$\frac{1}{2}$. So for this model we make the following relaxation: Instead of mapping an entry to a leaf node, when it is inserted for the first time in the root, we just let the leaf node be "defined" as the entry is pushed down the tree during eviction. In that sense we abstract entries and the only think we need to care for, is how many entries there exist in a given internal node at a given moment, which is expressed by the state of the Markov Chain.

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

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

We use this last model in order to bound the probability of overflow for internal nodes in our implementation and in order to set bucket sizes. In particular we generate a Markov

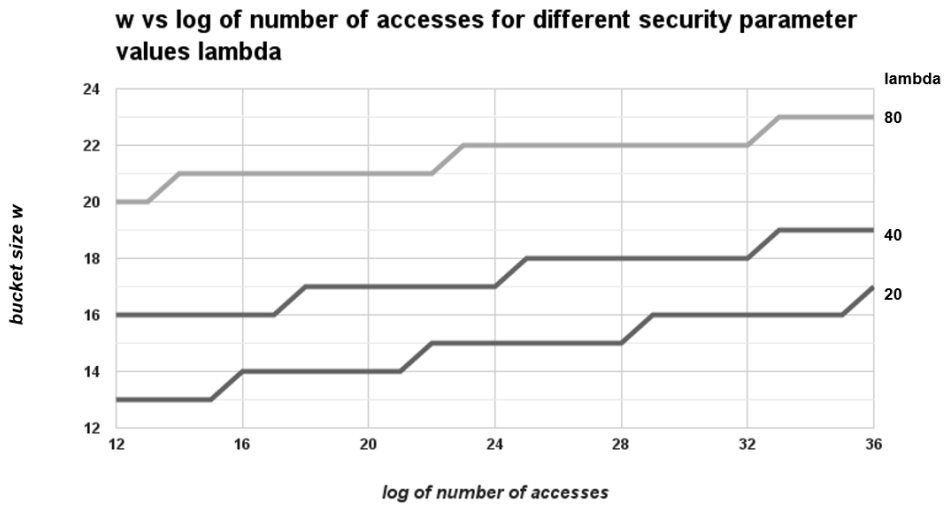


Figure 4.1: w for different $\log \mathsf{N}$

Chain the has $w + 2$ states, w for the bucket size, one empty state and one overflow state. The overflow state is a sink. We compute the probability of being in the overflow state after N accesses assuming the node was initially empty and perform a union bound on the number of nodes in all paths of the ORAM forest forest. In Figure 4.1 for different statistical security parameters λ equal with $20, 40$ and 80, we show the minimum bucket sizes w for $\log \mathsf{N}$ in the range $12, \ldots 36$ and $\log(n) = \mathrm{O}(\log \mathsf{N})$. Generally, we observe that using the Markov Chain based approximation can lead to tighter bounds on the internal node sizes, from which we conjecture that the size of *internal nodes* can be reduced to $\mathrm{O}\left(\sqrt{\lambda + \log \mathsf{N}}\right)$ $(w > 2\sqrt{\lambda + \log \mathsf{N} + 2 \log \log(n)})$.

4.5 Implementation and Testing

We built and benchmarked a prototype Java implementation of the proposed 3PC-ORAM protocol. We tested this implementation on the entry-level AWS EC2 t2.micro virtual servers, which have one hyper-thread on a 2.5GHz CPU and 1GB RAM. Each of the three

protocol participants C, D, E where co-located in the same availability zone and connected via a local area network.

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

Figure 4.2 shows the wall clock time of the online stage as a function of the bit-length $\log(n)$ of the RAM address space, for the two cases $(w, B) = (16, 4)$ and $(w, B) = (32, 4)$. We found that the CPU utilization in the online phase of our protocol is pretty stable, growing from about 25% for smaller $\log(n)$'s to 35% for $\log(n) \geq 30$, hence the graph of the CPU costs as function of $\log(n)$ has a very similar shape. Our testing showed that the influence of the record size B on the overall performance is very small for B less than 100 bytes, but higher payload sizes start influencing the running time. Our testing confirms that the running time has clear linear relationship to the bucket size w: The wall clock for $w = 64$ grows by a factor close to 1.8 compared to $w = 32$, and for $w = 128$ by a factor close to 3.5 (for large $\log(n)$ and small B). The offline wall clock time grows from 400 msec for $\log(n) = 12$ to 1300 msec for $\log(n) = 36$ for $w = 32$, but these numbers should be taken only as loose upper bounds on the pre-computation overhead of our 3PC-ORAM. Finally, we profiled the code to measure the percentage of CPU time spent on different protocol components. We

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

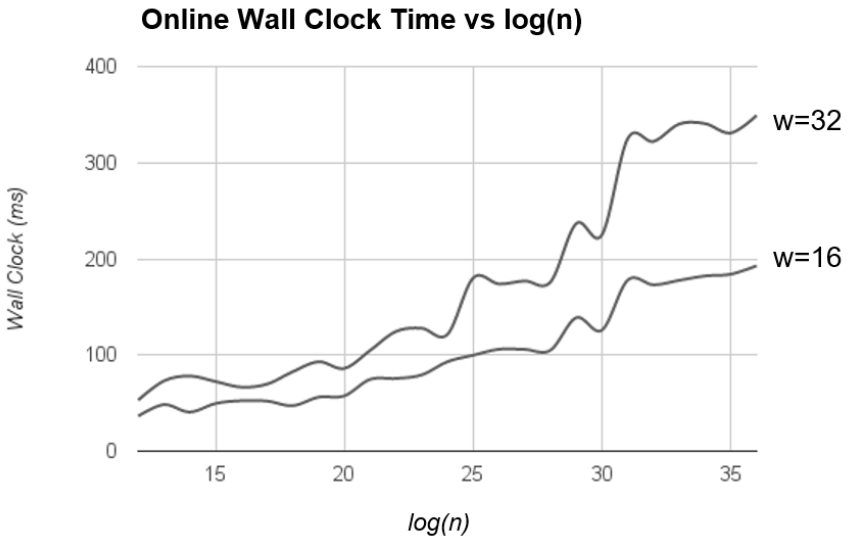


Figure 4.2: Online Wall Clock vs RAM address size $\log(n)$

Chapter 5

3PC-Circuit-ORAM

This chapter describes our 3PC-Circuit-ORAM scheme, which adopts MPC-friendly client eviction algorithm from 2PC-Circuit-ORAM [44] and improves performance efficiency over 2PC-Circuit-ORAM and our previous 3PC-ORAM construction [18]. It is based on my publication *3PC ORAM with Low Latency, Low Bandwidth, and Fast Batch Retrieval*, a joint work with Stanislaw Jarecki, published on Applied Cryptography and Network Security 2018 [29].

5.1 Technical Overview

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

1. **Retrieval**, which given $\mathsf{path} = \mathsf{tree.path}(\mathrm{L})$ and address prefix N, locates tuple $\mathrm{T} = (1, \mathrm{N}, \mathrm{L}, \mathsf{rec})$ in path and retrieves next-level label in rec;

2. **PostProcess**, which removes T from path, injects new labels into T, and re-inserts it in the root (= stash);

3. **Eviction**, which can be divided into two sub-steps:

   (a) *Eviction Logic*: An eviction map EM is computed, by function denoted Route, on input label L and the *metadata* fields (fb, lb) of tuples in path,

   (b) *Data Movement*: Permute tuples in path according to map EM.

Our 3PC-ORAM is a secure emulation of the above procedure, with the Eviction Logic function Route instantiated as in Circuit-ORAM [44], and it performs all the above steps on the *sharings* of inputs tree and N, given label L as a public input known to all parties. With the exception of the next-level label recovered in Retrieval, all other variables remain secret-shared. Our implementation of the above steps resembles the 3PC-ORAM emulation of binary-tree ORAM by [18] such that we use garbled circuit for Eviction Logic, and specialized 3PC protocols for Retrieval, PostProcess, and Data Movement. However, our implementations are different from [18]: First, to enable low-bandwidth batch processing of retrieval we use different sharings and protocols in Retrieval and PostProcess. Second, to securely "glue" Eviction Logic and Data Movement we need to mask mapping EM computed by Eviction Logic and implement Data Movement given this masked mapping. We explain both points in more detail below.

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

The 3PC implementation of Retrieval in [18] has $O(\ell B)$ bandwidth where $\ell = O(\log(n))$ is the number of tuples in path, and here we reduce it to $3B + O(\ell \log \ell)$ as follows: First, we re-use the *Keyword Search* protocol $\mathsf{KSearch}$ of [18] to create a secret-sharing of index j of a location of the keyword-matching tuple in path. This sub-protocol reduces the problem to finding an index where a secret-shared array of length ℓ contains an all-zero string, which has $\Theta(\ell \log \ell)$ communication complexity. Our $\mathsf{KSearch}$ implementation has $2\ell(c + \log \ell)$ bandwidth where 2^{-c} is the probability of having to re-run $\mathsf{KSearch}$ because of collisions in ℓ pairs of $(c + \log \ell)$-bit hash values. The overall bandwidth is optimal for $c \approx \log \log \ell$, but we report performance numbers for $c = 20$.

Secondly, we use a *Secret-Shared PIR* (SS-PIR) protocol, which creates a fresh sharing of the j-th record given the shared array and the shared index j. We implement SS-PIR in two rounds from any 2-server PIR [16] whose servers' PIR responses form an xor-sharing of the retrieved record. Many 2-server PIR's have this property, e.g., [13, 6, 28], but we exemplify this generic construction with the simplest form of 2-server PIR of Chor et al. [13] which has $3\ell + 3B$ bandwidth. This is not optimal in ℓ, but in our case $\ell \leq 150 + b$ where b is the number of accesses with postponed eviction, the optimized version of SS-PIR sends only $\approx \ell + 3B$ bits *on-line*, and $\mathsf{KSearch}$ already sends $O(\ell \log \ell)$ bits. Our generic 2-PIR to 3PC-SS-PIR compiler is simple (a variant of it appeared in [28]) but the 3-round 3PC Kw-SS-PIR protocol is to the best of our knowledge novel.

Efficient 3PC-Circuit-ORAM Eviction. In Eviction we use a simple Data Movement protocol, with 2 round and $\approx 2|\mathsf{path}|$ bandwidth. With three parties denoted as $(\mathrm{C}, \mathrm{D}, \mathrm{E})$, our protocol creates a two-party (C, E)-sharing of $\mathsf{path}' = \mathsf{EM}(\mathsf{path})$ from a (C, E)-sharing of path *if* party D holds eviction map EM in the clear. Naively outputting $\mathsf{EM} = \mathsf{Route}(\mathsf{path})$ to party D is insecure, as eviction map is correlated with the ORAM access pattern, so the question is whether EM can be *masked* by some randomizing permutation known by C and E. [18] had an easy solution for its binary-tree ORAM variant because its algorithm Route

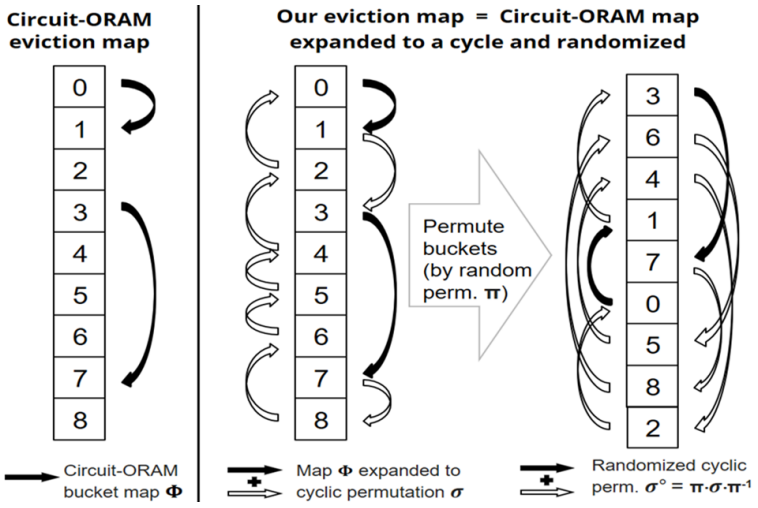


Figure 5.1: Randomization of Circuit ORAM's Bucket Map

outputs a *regular* EM, that buckets on every except the last level of the retrieved path always move two tuples down to the next level, so all [18] needed to do is to randomly permute tuples on each bucket level of path, and the resulting new EM′ on the permuted path leaks no information on EM. By contrast, Circuit-ORAM eviction map is *non-regular* (see Fig. 5.1): Its bucket level map Φ of EM can move a tuple by variable distance and can leave some buckets untouched, both of which are correlated with the density of tuples in path, and thus with ORAM access pattern.

Thus our goal is to transform the underlying Circuit-ORAM eviction map $\mathsf{EM} = (\Phi, \mathsf{t})$ into a map whose distribution does not depend on the data (Φ describes the bucket-level movement, while t is an array containing one tuple index from each bucket that will be moved). We do so in two steps. First, we add an extra empty tuple to each bucket and we modify Circuit ORAM algorithm Route to *expand* function $\Phi : \mathsf{Z}_\mathrm{d} \to \mathsf{Z}_\mathrm{d} \cup \{\perp\}$ into a cyclic permutation σ on Z_d (d is the depth of path, Z_d is the set $\{0, ..., \mathrm{d}-1\}$), by adding spurious edges to Φ in the deterministic way illustrated in Fig. 5.1. Second, we apply two types of masks to the resulting output (σ, t) of Route, namely a random permutation π on Z_d and two arrays (δ, ρ), each of which contains a random tuple index in each bucket. Our Eviction Logic protocol will use (π, δ, ρ) to mask (σ, t) by computing $(\sigma^\circ, \mathsf{t}^\circ)$ s.t. $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$ (permutation

composition) and $\mathsf{t}^\circ = \rho \oplus \pi(\mathsf{t} \oplus \delta)$. And now we have a masked eviction map $\mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ}$ that can be revealed to party D but does not leak information on $\mathsf{EM}_{\sigma,\mathsf{t}}$ or $\mathsf{EM}_{\Phi,\mathsf{t}}$.

5.2 3PC-Circuit-ORAM Protocol

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

Shared Variables, Bitstrings, Secret-Sharing. Each pair of parties $\mathsf{P}_1, \mathsf{P}_2$ in our protocol is initialized with a shared seed to a Pseudorandom Generator (PRG), which allows them to generate any number of shared (pseudo)random objects. We write $x^{\mathsf{P}_1\mathsf{P}_2} \xleftarrow{\$} \mathsf{S}$ if P_1 and P_2 both sample x uniformly from set S using the PRG on a jointly held seed. We use several forms of secret-sharing, and here introduce four of them which are used in our high level protocols 3PC-ORAM.Access and 3PC-ORAM.ML (Alg. 5.1 & 5.2):

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

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

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

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

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

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

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

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

3PC-ORAM Protocol. Our 3PC-ORAM protocol, 3PC-ORAM.Access, Alg. 5.1, performs the same recursive scan through data-structure $\mathsf{tree}_0, ..., \mathsf{tree}_{h-1}$ as the client-server Path ORAM (and Circuit-ORAM), included for reference as Alg. A.1 in Appendix A, except it runs on inputs in $\langle \cdot \rangle$ secret-sharing format, i.e., sharings of ORAM trees, $\langle \mathsf{tree}_0 \rangle, ..., \langle \mathsf{tree}_{h-1} \rangle$, sharing of address $\langle \mathrm{N} \rangle$, and sharing of a new record $\langle \mathsf{rec}' \rangle$ if $\mathsf{instr} = \mathsf{write}$. The main loop of 3PC-ORAM.Access, i.e., protocol 3PC-ORAM.ML, Alg. 5.2, also follows the corresponding client-server algorithm ORAM.ML, included for reference as Alg. A.2 in Appendix A, except that apart of the current-level leaf label L which is known to all parties, all its other inputs are secret-shared as well.

Protocol 3PC-ORAM.ML calls sub-protocols whose round/bandwidth specifications are stated in Fig. 5.2. (We omit computation costs because they are all comparable to link-encryption of communicated data). The low costs of these sub-protocols are enabled by different forms of secret-sharings, e.g., xor versus additive, or 2-party versus 3-party, and by low-cost (or

Algorithm 5.1 Protocol 3PC-ORAM.Access - 3PC-Circuit-ORAM Access

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

1: $\{\langle \mathrm{L}'_i \rangle \xleftarrow{\$} \{0,1\}^{i \cdot \tau}\}_{i=1}^{h-1}$; $\langle \mathrm{N}_0, \mathrm{N}_h, \mathrm{L}'_0, \mathrm{L}'_h \rangle := \perp$; $\mathrm{L}_0 := \perp$
2: **for** $i = 0$ **to** $h-1$ **do**

3PC-ORAM.ML: L_i, $\left\langle \mathsf{tree}_i, (\mathrm{N}_0|...|\mathrm{N}_i), \mathrm{N}_{i+1}, \mathrm{L}'_i, \mathrm{L}'_{i+1}, {}^* \mathsf{rec}' \right\rangle$
$\longrightarrow$ L_{i+1} (* $\langle \mathsf{rec} \rangle$ instead of L_{i+1}), $\langle \mathsf{tree}_i \rangle$

3: **end for**

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

no cost) conversions between them. For implementations of these protocols we refer to Appendix A.

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

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

Algorithm 5.2 Protocol 3PC-ORAM.ML - Main Loop of 3PC-Circuit-ORAM

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

Retrieval of Next Label/Record

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

1: KSearch: $\langle \mathsf{path}.(\mathsf{fb}|\mathsf{adr}), 1|\mathrm{N} \rangle \rightarrow \langle j \rangle_{\mathsf{shift}}$ ▷ $\mathsf{path}[j].(\mathsf{fb}|\mathsf{adr}) = 1|\mathrm{N}$

2: 3ShiftPIR: $\langle \mathsf{path.rec} \rangle, \langle j \rangle_{\mathsf{shift}} \rightarrow \langle X \rangle$ (* $\langle \mathsf{rec} \rangle := \langle X \rangle$) ▷ $X = \mathsf{path}[j].\mathsf{rec}$

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

Post-Process of Found Tuple

4: ULiT: $\langle X, \mathrm{N}, \Delta\mathrm{N}, \mathrm{L}'_i, \mathrm{L}'_{i+1}$ (* rec')$\rangle, \mathrm{L}_{i+1} \rightarrow \langle \mathrm{T} \rangle$
▷ $X[\Delta\mathrm{N}] := \mathrm{L}'_{i+1}$ (* $X := \mathsf{rec}'$), $\mathrm{T} = (1, \mathrm{N}, \mathrm{L}'_i, X)$

5: FlipFlag: $\langle \mathsf{path.fb} \rangle, \langle j \rangle_{\mathsf{shift}} \rightarrow \langle \mathsf{path.fb} \rangle$ ▷ $\mathsf{path}[j].\mathsf{fb} := 0$

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

Eviction

6: GC(Route): $\mathrm{L}_i, \delta^{\mathrm{CE}}, \langle \mathsf{path}.(\mathsf{fb}, \mathsf{lb}) \rangle \rightarrow (\overline{\sigma}, \mathsf{t}')^{\mathrm{D}}, \mathsf{wk}^{\mathrm{E}}$
▷ $\overline{\sigma} = \{\mathsf{wk} : \sigma\}$ and $\mathsf{t}' = \mathsf{t} \oplus \delta$ for expanded Circuit-ORAM eviction map (σ, t)

7: PermBuckets: $\overline{\sigma}^{\mathrm{D}}, \pi^{\mathrm{CE}}, \mathsf{wk}^{\mathrm{E}} \rightarrow \sigma^{\circ \mathrm{D}}$ ▷ $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$

8: PermTuples: $\mathsf{t}'^{\mathrm{D}}, (\pi, \rho)^{\mathrm{CE}} \rightarrow \mathsf{t}^{\circ \mathrm{D}}$ ▷ $\mathsf{t}^\circ = \rho \oplus \pi(\mathsf{t}')$

9: SSXOT: $\langle \mathsf{path} \rangle, (\pi, \delta, \rho)^{\mathrm{CE}}, (\sigma^\circ, \mathsf{t}^\circ)^{\mathrm{D}} \rightarrow \langle \mathsf{path}' \rangle$ ▷ $\mathsf{path}' = \mathsf{EM}_{\sigma,\mathsf{t}}(\mathsf{path})$

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

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

(2) PostProcess runs the Update-Label-in-Tuple protocol ULiT to form sharing $\langle \mathrm{T} \rangle$ of a new tuple using sharing $\langle X \rangle$ of the retrieved tuple's payload, sharings $\langle \mathrm{N} \rangle$ and $\langle \Delta\mathrm{N} \rangle$ of the address prefix and the next address chunk, and sharings $\langle \mathrm{L}'_i \rangle, \langle \mathrm{L}'_{i+1} \rangle$ of new labels; In parallel to ULiT it also runs protocol FlipFlag to flip the full/empty flag to 0 in the old version of this tuple in path, which executes on inputs the sharings $\langle \mathsf{path.fb} \rangle$ of field fb of tuples in path and on the "shift" sharing $\langle j \rangle_{\mathsf{shift}}$; Once ULiT terminates the parties can insert $\langle \mathrm{T} \rangle$ into

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

Figure 5.2: Round and bandwidth for sub-protocols of Alg. 5.2, for ℓ the number of tuples on path and x the circuit input size ($\approx \ell(\mathrm{d} + \log(n)) + \mathrm{d}\log(w+1)$)

sharing of the root bucket in path. At this point the root bucket has size $s{+}1$ (or $s{+}b$ if we postpone eviction for a batch of b accesses).

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

Eviction Procedure. As we explain in Section 5.1, we split Eviction into Eviction Logic, which uses garbled circuit sub-protocol to compute the eviction map EM, and Eviction Movement, which uses special-purpose protocols to apply EM to the shared path, which in protocol 3PC-ORAM.ML will be $\langle\mathsf{path}\rangle$. However, recall that revealing the eviction map to any party would leak information about path density, and consequently the access pattern. We avoid this leakage in two steps: First, we modify the Circuit-ORAM eviction logic computation Route, so that when it computes bucket-level map Φ and the tuple pointers array t, which define an eviction map $\mathsf{EM}_{\Phi,\mathsf{t}}$, the algorithm scans through the buckets once more to *expand* the partial map Φ into a complete cycle σ over the d buckets (see Fig. 5.1,

and we include the modified Circuit-ORAM algorithm Route in Appendix A.3). Second, the garbled circuit computation $\mathsf{GC}(\mathsf{Route})$, see Step 6, Alg. 5.2, does not output (σ, t) to D in the clear: Instead, it outputs $\mathsf{t}' = \mathsf{t} \oplus \delta$ where δ is a random array, used here as a one-time pad, and the *garbled wire encoding* of the bits of $\sigma = [\sigma(1), ..., \sigma(\mathrm{d})]$, i.e., the output wire keys $\{\mathsf{wk} : \sigma\} = \mathsf{wk}[i][\sigma[i]]\}_{i=1}^{\mathrm{d}\log\mathrm{d}}$.

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

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

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

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

$$\mathsf{EM}_{\sigma,\mathsf{t}} = \Pi^{-1} \cdot \mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ} \cdot \Pi = (\tilde{\delta} \cdot \ddot{\pi}^{-1} \cdot \tilde{\rho}) \cdot (\mathsf{EM}_{\pi\sigma\pi^{-1},\rho\oplus\pi(\mathsf{t}\oplus\delta)}) \cdot (\tilde{\rho} \cdot \ddot{\pi} \cdot \tilde{\delta}) \tag{5.1}$$

Consider the set of points $S = \{(j, \mathsf{t}[j]) \mid j \in \mathsf{Z}_\mathrm{d}\}$ which are moved by the left hand side (LHS) permutation $\mathsf{EM}_{\sigma,\mathsf{t}}$. To argue that eq. (5.1) holds we first show that the RHS permutation maps any point $(j, \mathsf{t}[j])$ of S in the same way as the LHS permutation:

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

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

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

Also note that:

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

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

Optimizations and Efficiency. As mentioned above, we can improve on both bandwidth and rounds in the Retrieval phase of 3PC-ORAM.ML shown in Alg. 5.2. The optimization comes from an observation that our protocol KSearch (see Alg. A.6, App. A.2) takes just one round to compute shift-sharing $\langle j \rangle^{\mathrm{DE-C}}_{\mathsf{shift}}$ of index j, and its second round is a resharing which transforms $\langle j \rangle^{\mathrm{DE-C}}_{\mathsf{shift}}$ into $\langle j \rangle_{\mathsf{shift}}$. This round of resharing can be saved, and we can re-arrange protocols 3ShiftPIR and 3ShiftXorPIR (shown as Alg. A.9 and A.11 in App. A.2) so they use only $\langle j \rangle^{\mathrm{DE-C}}_{\mathsf{shift}}$ as input and effectively piggyback creating the rest of $\langle j \rangle_{\mathsf{shift}}$ in such a way that the modified protocols, denoted resp. 3ShiftPIR-Mod and 3ShiftXorPIR-Mod (shown as Alg. A.12 and A.13 in App. A.2) take 2 rounds, which makes the whole Retrieval take only 3 rounds, hence access protocol 3PC-ORAM.Access takes $3h$ rounds in Retrieval, and, surprisingly, the same is true for Retrieval *with PostProcess* (protocols 3ShiftPIR-Mod and 3ShiftXorPIR-Mod also use resp. 2ℓ and $2{\cdot}2^{\tau}\ell$ less bandwidth than 3ShiftPIR and 3ShiftXorPIR).

Surprisingly, the modified Retrieval *and PostProcess* phases together take only 3 rounds, amortized over the tree traversal, which enables pipelined processing of b accesses in $3b + 3h$ rounds (with postponed eviction). Very briefly, this is because (1) the 2-round protocol FlipFlag can start after the first round of Retrieval (and thus terminates in round 3) because KSearch produces FlipFlag's input $\langle j \rangle^{\mathrm{DE-C}}_{\mathsf{shift}}$ in round 1; and (2) protocol ULiT has 2 rounds, but its first round can be computed in parallel to 1st round of KSearch because it needs only ΔN as an input, and while its 2nd round requires L_{i+1} which is output only in round 3 by 3ShiftXorPIR-Mod (other inputs of ULiT are available before), this 2nd round of ULiT can execute *in parallel* with the 1st round of Retrieval instance for the next access request on the same tree, and this is because the 1st round of retrieval consists of KSearch which takes only fb, adr fields of the tuples in path as inputs while the 2nd round of ULiT works only on the rec field of tuple T.

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

5.3 Security

Protocol 3PC-ORAM of Section 5.2 is a three-party secure computation of an Oblivious RAM functionality, i.e., it can implement RAM for any 3PC protocol in the RAM model. We define a Universally Composable (UC) Oblivious RAM functionality $\mathsf{F}_{\mathsf{ORAM}}$ for 3-party computation (3PC) in the framework of Canetti [12], and we argue that our 3PC ORAM realizes $\mathsf{F}_{\mathsf{ORAM}}$ in the setting of $m = 3$ parties with *honest majority*, i.e., only $t = 1$ party is (statically) corrupted, assuming *honest-but-curious* (HbC) adversary, i.e., corrupted party follows the protocol. We assume secure pairwise links between the three parties. Since we have static corruptions, HbC adversary, and non-rewinding simulators, security holds even if communication is asynchronous.

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

Security of our 3PC-ORAM. Our protocol securely realizes $\mathsf{F}_{\mathsf{ORAM}}$ in the $(t, m) = (1, 3)$ setting *if* Circuit-ORAM defines a secure client-server ORAM, which implies security of 3PC-ORAM by the argument for Circuit-ORAM security given in [44]. We note that protocol 3PC-ORAM.Access of Section 5.2 implements only procedure Access. Procedure Init can be implemented by running 3PC-ORAM.Access with instr = write in a loop for N from 0 to $n-1$ (and arbitrary rec′'s). And our main security claim, stated as Corollary 5.1 below, assumes that Init is executed by a trusted-party.

Corollary 5.1. *Assuming secure initialization,* 3PC-ORAM.Access *is a UC-secure realization of 3PC ORAM functionality* $\mathsf{F}_{\mathsf{ORAM}}$.

Briefly, the proof uses UC framework, arguing that each protocol securely realizes its intended input/output functionality *if* each sub-protocol it invokes realizes its idealized input/output functionality. All sub-protocols executed by protocol 3PC-ORAM.ML of Section 5.2 are accompanied with brief security arguments which argue precisely this statement. As for 3PC-ORAM.ML, its security proof is centered around two facts argued in Section 5.2, namely that our way of implementing Circuit-ORAM eviction map, with D holding $\sigma^\circ = \pi \cdot \sigma \cdot \pi^{-1}$ and $\mathsf{t}^\circ = \rho \oplus \pi(\mathsf{t} \oplus \delta)$ and E, C holding π, ρ, δ is (1) correct, because $\Pi^{-1} \cdot \mathsf{EM}_{\sigma^\circ, \mathsf{t}^\circ} \cdot \Pi = \mathsf{EM}_{\sigma, \mathsf{t}}$ for $\Pi = \tilde{\rho} \cdot \ddot{\pi} \cdot \tilde{\delta}$, and (2) it leaks no information to either party, because random π, ρ, δ induce random $\sigma^\circ, \mathsf{t}^\circ$ in D's view.

5.4 Performance Evaluation

We tested a Java prototype of our 3PC-Circuit-ORAM, with garbled circuits implemented by ObliVM library of Wang [44], on three AWS EC2 c4.2xlarge servers, with communication links encrypted using AES-128. Each c4.2xlarge instance is equipped with eight Intel Xeon E5-2666 v3 CPU's (2.9 GHz), 15 GB memory, and has 1 Gbps bandwidth. (However, our tested prototype utilizes multi-threading only in parallel Eviction, see below.)

In the discussion below we use the following acronyms:

- cust-3PC: our 3PC-Circuit-ORAM protocol;
- gen-3PC: generic 3PC-Circuit-ORAM using 3PC of Araki et al. [1];
- 2PC: 2PC-Circuit-ORAM [44];
- C/S: the client-server Path-ORAM [42].

Wall Clock Time. Fig. 5.3 shows online timing of cust-3PC for small record sizes ($B = 4$ bytes) as a function of address size $\log(n)$. It includes Retrieval wall clock time (WC), End-to-End (Retrieval+PostProcess+Eviction) WC, and End-to-End WC with *parallelized* Eviction for all trees, which shows 60% reduction in WC due to better CPU utilization. Note that Retrieval takes about 8 milliseconds for $\log(n) = 30$ (i.e., 2^{30} records), and that Eviction takes only about 4-5 times longer. Recall that Retrieval phase has $3h$ rounds while Eviction has 6, which accounts for much smaller CPU utilization in Retrieval.

CPU Time. We compare total and online CPU time of cust-3PC and 2PC in Fig. 5.4 with respect to memory size n, for $B = 4$ bytes[1]. Since 2PC implementation [44] does not provide online/offline separation, we approximate 2PC online CPU time by its garbled circuit evaluation time, because 2PC costs due to OT's can be pushed to pre-computation. As Fig. 5.4 shows, the cust-3PC CPU costs are between 6x and 10x lower than in 2PC, resp. online and total, already for $\log(n) = 25$, and the gap widens for higher n.

Bandwidth Comparison with Generic 3PC. Timing results depend on many factors (language, network, CPU, etc.), and bandwidth is a more reliable predictor of performance for protocols using only light symmetric crypto. In Fig. 5.5 we compare online bandwidth of cust-3PC, gen-3PC, and C/S, as a function of the address size $\log(n)$, for $B = 4$ bytes. We

[1]We include CPU comparisons only with 2PC-Circuit-ORAM [44], and not 2PC-Sqrt-ORAM [51] and 2PC-FLORAM [15], because the former uses the same Java ObliVM GC library while the latter two use the C library Obliv-C. Still, note that for $n = 30$, the on-line computation due to FSS evaluation and linear memory scans contributes over 1 sec to wall-clock in [15], while our on-line wall-clock comes to 40 msec.

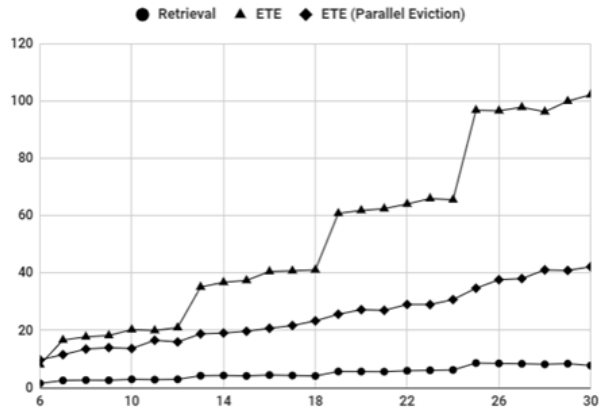


Figure 5.3: Our 3PC-ORAM Online Wall-Clock Time(ms) vs $\log(n)$ for $B = 4$ bytes

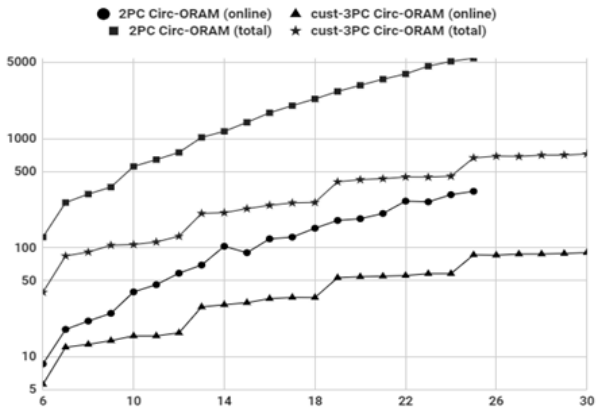


Figure 5.4: CPU Time (ms) vs $\log(n)$, for $B = 4$ bytes

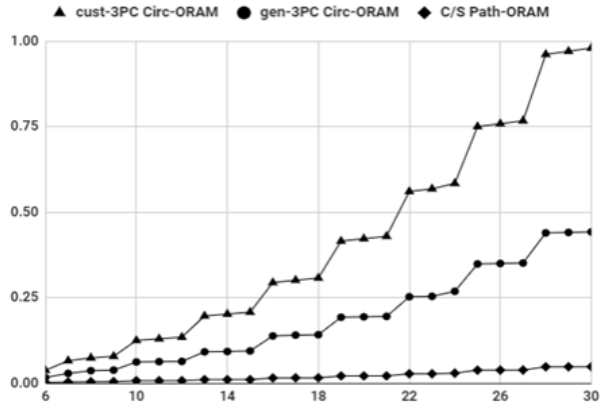


Figure 5.5: Online bndw.$_{(\text{MB})}$ vs $\log(n)$ for B=4 bytes

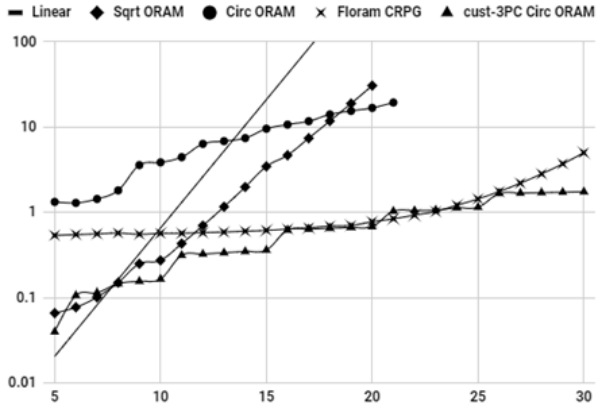


Figure 5.6: Comparison with 2PC-ORAM's in online+offline bndw.$_{(\text{MB})}$ vs $\log(n)$ for B=4 bytes

see for small records our cust-3PC is only a factor of 2x worse than the optimal-bandwidth gen-3PC (which, recall, has completely impractical round complexity).

Bandwidth Comparison with 2PC ORAMs. In Fig. 5.6 we compare total bandwidth of cust-3PC and several 2PC-ORAM schemes, including 2PC-FLORAM scheme of [15], the 2PC Sqrt-ORAM of [51], and a trivial linear-scan scheme. Our cust-3PC bandwidth is competitive to FLORAM for all n's, but for $n \geq 24$ the $O(\sqrt{n})$ asymptotics of FLORAM takes over. Note also that FLORAM uses $O(n)$ local computation vs. our $O(\log^3 n)$, so in the FLORAM case

bandwidth comparison does not suffice. Indeed, for $n = 2^{30}$ and $B = 4$ bytes, [15] report > 1 sec overall processing time on LAN vs. 40 msec for us.

Chapter 6

3PC-Sqrt-ORAM

This chapter describes our 3PC-Sqrt-ORAM scheme, which is based on the 2PC-Sqrt-ORAM of [51] but improves efficiency by simplifying the access procedure and replacing expensive 2PC protocols with 3PC protocols, especially by completely eliminating the use of Yao's garbled circuit and replacing $O(n \log(n))$ 2PC permutation network with $O(n)$ 3PC-OT variant. This a joint work with Stanislaw Jarecki.

We describe our 3PC-Sqrt-ORAM by first recalling the Sqrt-ORAM protocol of Goldreich and Ostrovksky [22] (see Alg. 6.1), using the notations suitable for our 3PC-Sqrt-ORAM construction. The goal of an ORAM scheme is for the client to outsource an array M of n records to the server in such a way that the client can read (and write to) any record $\mathsf{M}[\mathrm{N}]$ in M without leaking any information about the accessed address N. The first idea of Sqrt ORAM is to let the server store $\mathsf{fresh} = \pi(\mathsf{M})$, a random permutation of array M, while the client stores permutation π, which it can do succinctly if it is a pseudorandom permutation (PRP) and the client stores its key. In addition, each record in fresh is encrypted under the client's key, so the server always sees only the ciphertexts. To retrieve a record at address N, the client sends $\mathrm{N}' = \pi(\mathrm{N})$ to the server, who sends back $\mathsf{fresh}[\mathrm{N}'] = (\pi(\mathsf{M}))[\pi(\mathrm{N})] = \mathsf{M}[\mathrm{N}]$.

Algorithm 6.1 Original Sqrt-ORAM Scheme from [22]

```
function Initialize(Data, T)
    n ← |Data|
    π ← pseudorandom function
    append √n dummy blocks to Data
    Shuffle ← ObliviousSort(Data, π)
    Oram ← (n, ctr ← 0, T, π, Shuffle, Stash ← ∅)
    return Oram
end function

function SqrtOram(Oram, ⟨N⟩)
    // Access
    ⟨found⟩ ← LinearScan(Oram.Stash, ⟨N⟩)
    if ⟨found⟩ then
        read/write found block
        ⟨k⟩ ← Oram.n + Oram.ctr
    else
        ⟨k⟩ ← ⟨N⟩
    end if
    p ← π(⟨k⟩)
    if not ⟨found⟩ then
        read/write Oram.Shuffle[p]
    end if
    append Oram.Shuffle[p] to Oram.Stash
    Oram.Shuffle[p] ← dummy block
    Oram.ctr ← Oram.ctr + 1 (mod Oram.T)
    // (Re)initialization
    if Oram.ctr = 0 then
        Data ← real blocks in Oram.Shuffle ∪ Oram.Stash
        Oram ← Initialize(Data, Oram.T)
    end if
end function
```

Note that $\pi(\mathrm{N})$ is uncorrelated to N because π is random, except that the server learns if two accessed addresses N_1 and N_2 are equal, because $\mathrm{N}_1 = \mathrm{N}_2$ if and only if $\pi(\mathrm{N}_1) = \pi(\mathrm{N}_2)$. To avoid this leakage the server stores an additional record list stash, initially empty, and (1) whenever the server retrieves a record in fresh, it adds this record to stash, and (2) before the client sends N' to the server, the server sends the entire list stash to the client, who checks if $\mathsf{fresh}[\pi(\mathrm{N})]$ is already in stash: If it is then the client picks N' at random in Z_n, and if it not

then the client sets $\mathrm{N}' = \pi(\mathrm{N})$. Either way the server retrieves $\mathsf{fresh}[\mathrm{N}']$, sends it to the client, and adds it to stash. Note that this way the client always finds record $\mathsf{M}[\mathrm{N}] = \mathsf{fresh}[\pi(\mathrm{N})]$, but the values N' the server sees are now random in Z_n regardless of the client's access pattern, in particular regardless whether some addresses are accesed more than once. To keep list stash short, after T of accesses the client retrieves the entire fresh, puts the elements in stash back to their positions in fresh, permutes fresh by π^{-1} to get array M in the standard form, picks a new permutation π', and re-initializes the scheme by sending $\mathsf{fresh}' = \pi'(\mathsf{M})$ to the server and using a fresh empty list stash. Note that the amortized bandwidth per access is $\mathrm{T} + n/\mathrm{T}$ records for re-initialization period T, which is minimized to $O(\sqrt{n})$ if $\mathrm{T} = \sqrt{n}$, hence the algorithm's name.

2PC-ORAM based on Sqrt-ORAM. A standard approach to converting Sqrt-ORAM to 2PC-ORAM, i.e., the two-party secure computation of ORAM functionality, would be to secret-share $\mathsf{fresh}, \mathsf{stash}$ between the two parties (which allows us to avoid encrypting the records) and to implement the Sqrt-ORAM client's code using two-party secure computation. However, since this would involve computing a PRP π on (secret-shared) address N and a key, rather than resorting to rather expensive two-party secure PRP computation, the 2PC Sqrt-ORAM due to [51] proposed to store permutation π using a recursive data structure, somewhat similar to the recursive way the Path-ORAM keeps the position map [42]. Let $h = (\log(n)/\tau)+1$ for some constant τ, and let $\mathrm{N}^{(0)}|...|\mathrm{N}^{(h-2)}$ be the $\log(n)$-bit address N divided into τ-bit segments (we will denote $\mathrm{N}^{(0)}|...|\mathrm{N}^{(i)}$ as N^i). The recursive data structure contains h lists, from fresh_0 to fresh_{h-1}, where the top-level list fresh_{h-1} is the array $\mathsf{fresh} = \pi(\mathsf{M})$ described above, with the top-level permutation π denoted π^{h-1}, while fresh_i, for $i<h-1$, is a list containing $2^{i\cdot\tau}$ records, each of which is an array of size 2^τ, permuted by a random permutation π^i. These lists maintain an invariant that for every $(i-1)\tau$-bit address prefix N^*, the record array at position $\pi^i(\mathrm{N}^*)$ in fresh_i, stores 2^τ pointers to the locations in fresh_{i+1} which correspond to the 2^τ extensions of N^* by the next chunk of τ bits. For example, if

$\tau = 2$ then fresh_i stores at location $\pi^i(\mathrm{N}^*)$ a record containing the following $2^\tau = 4$ pointers to fresh_{i+1}:

$$(\pi^{i+1}(\mathrm{N}^*|00)\,,\, \pi^{i+1}(\mathrm{N}^*|01)\,,\, \pi^{i+1}(\mathrm{N}^*|10)\,,\, \pi^{i+1}(\mathrm{N}^*|11))$$

The first list fresh_0 is a single record with 2^τ pointers to fresh_1 corresponding to the τ-bit address prefixes $\mathrm{N}^{(0)}$, e.g., for $\tau = 2$, $\mathsf{fresh}_0 = (\pi^1(00), \pi^1(01), \pi^1(10), \pi^1(11))$.

In this way position $\pi(\mathrm{N})$ in the top-level list $\mathsf{fresh}_{h-1} = \pi(\mathsf{M})$ can be retrieved by the following walk through $\mathsf{fresh}_0, ...\mathsf{fresh}_{h-2}$: Given the (sharing of) address $\mathrm{N} = \mathrm{N}^{h-2}$, use $\mathrm{N}^{(0)}$ to reconstruct $\pi^1(\mathrm{N}^{(0)})$ from position $\mathrm{N}^{(0)}$ in fresh_0, and then for $i = 1, 2, ...$ use pointer $\pi^i(\mathrm{N}^{i-1})$ to retrieve (the sharing of) record $\mathsf{fresh}_i[\pi^i(\mathrm{N}^{i-1})]$ from (the sharing of) fresh_i, and use (the sharing of) $\mathrm{N}^{(i)}$ to publicly reconstruct $\pi^{i+1}(\mathrm{N}^i)$ contained at position $\mathrm{N}^{(i)}$ in this record. In other words, given the sharing of N and $\mathsf{fresh}_0, ..., \mathsf{fresh}_{h-1}$, the walk through this sequence of lists reconstructs pointer $\pi^1(\mathrm{N}^0)$ from fresh_0, then uses $\pi^1(\mathrm{N}^0)$ to retrieve pointer $\pi^2(\mathrm{N}^1)$ from fresh_1, then uses $\pi^2(\mathrm{N}^1)$ to retrieve pointer $\pi^3(\mathrm{N}^2)$ from fresh_2, and so forth, until it uses pointer $\pi^{h-1}(\mathrm{N}^{h-2}) = \pi^{h-1}(\mathrm{N})$ to retrieve record $\mathsf{M}[\mathrm{N}] = \mathsf{fresh}_{h-1}[\pi^{h-1}(\mathrm{N})]$ from the top-level list $\mathsf{fresh}_{h-1} = \pi^{h-1}(\mathsf{M})$.

In the 2PC-Sqrt-ORAM construction of [51], each level list fresh_i maintains its own stash_i, and if some position on fresh_i was already visited, then this record will be retrieved from the stash by linear scan, but a random-looking pointer is publicly generated for a fake access to maintain obliviousness. And, just like other 2PC-ORAMs, [51] used circuits to facilitate oblivious operations, like creation of these random-looking entries but letting the search algorithm pursue the real access path from the stashes until some level and then use the new entries when position pointers to these levels are fresh and never accessed/revealed before.

In our 3PC-Sqrt-ORAM construction, we make use of the similar data structure used in 2PC-Sqrt-ORAM [51], but make modifications so customized 3PC protocols can be used to

achieve the similar access operations without the use of any 2PC generic secure computation like Yao's garbled circuit, and thus efficiency can be improved.

6.1 Technical Overview

As shown in Fig. 6.1, similar to [51], our 3PC-Sqrt-ORAM can be constructed by first arranging the last level of record array and recursive levels of position map with identity alignment. Then, random permutation will be picked at every level to shuffle the fresh blocks. During this process, because of the physical locations of the blocks are changed, the pointers stored in corresponding previous levels will also be updated accordingly to the shuffling.

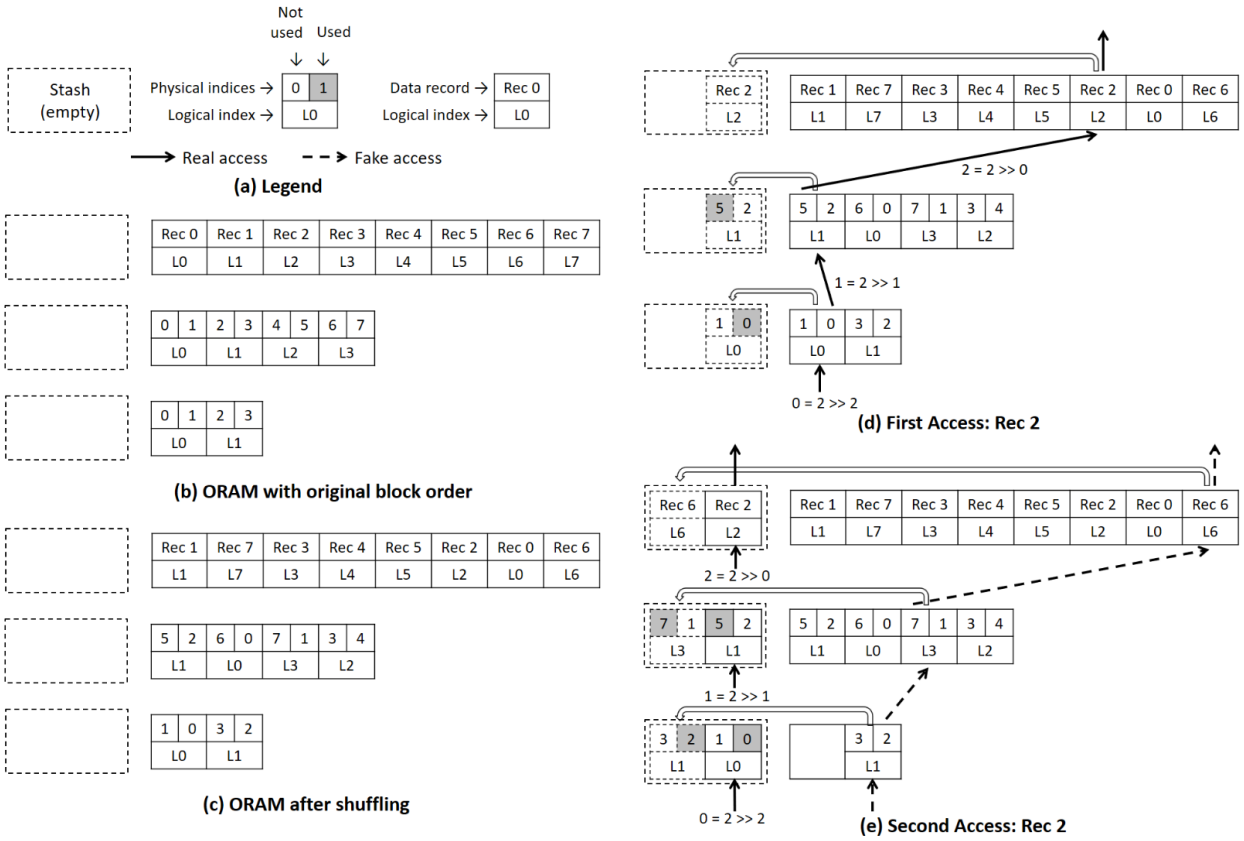


Figure 6.1: 3PC-Sqrt-ORAM Access Pattern (graph style based on [51])

Our 3PC-Sqrt-ORAM protocol is presented in Alg. 6.2. To perform the ORAM access, at each level, two blocks A and B from stash and fresh will be accessed. Block B is accessed using the pointer output from the previous level (on the first level block B will always be the current first block on fresh_0, as fresh_0 size shrinks on the first level). As on fresh, this block B is guaranteed to be a block never accessed before with fresh pointers to the next level. We append block B to stash, and then perform stash linear scan using the 3PC-OT variant protocol KSearch from [29] to find the block A that we want to access. The reason to perform stash linear scan after appending block B is that the KSearch protocol requires exactly one block to be found with the search key, and this requirement is met only if we include B on the stash (the most updated version of all the blocks either reside on stash or fresh, so for each access, either we can find the block on stash meaning it is accessed before, or we are accessing it right now using the pointer from previous level). So appending block B to stash guarantees the correctness of access using the 3PC KSearch protocol. Notice that while B is guaranteed to be a fresh block, A can be either a fresh block (meaning A = B), or a used block.

Once we identify blocks A and B, we can find the pointer for the next level access. If the pointer associated with the access address suffix in A is used, meaning we have done such access before, then we must output the pointer in B, which is fresh, to later perform a fake access on the next level. Otherwise, A's pointer can be used for output. This pointer search step is skipped on the last level of the ORAM, because the record we want to access is stored in block A, which can be output directly. After the update of the record for the access operation = write, or the update of the pointer flag bit in block A or B, which ever is used for pointer output, this updated block will be inserted back onto the stash using the 3PC-OT variant protocol SSXOT from [44].

When the access counter indicates the stash is full on all levels, a re-initialization is needed for putting the accessed blocks on stash_{h-1} on last level back to fresh_{h-1} to be re-shuffled again,

Algorithm 6.2 Protocol 3PC-Sqrt-ORAM

Parameters: Pack parameter τ. Shuffle period T. Number of levels h.
Input: Public used. C/E's secret-sharing of address $\mathrm{N} \in \{0,1\}^{\log(n)}$ and new record $\mathsf{rec}' \in \{0,1\}^B$, D/E's secret-sharing of $\mathsf{stash} = [\mathsf{stash}_0, ..., \mathsf{stash}_{h-1}], \mathsf{fresh} = [\mathsf{fresh}_0, ..., \mathsf{fresh}_{h-1}]$.
Output: Updated used'. C/E's secret-sharing of record $\mathsf{rec} \in \{0,1\}^B$ at address N. New D/E's secret-sharing of $\mathsf{stash}', \mathsf{fresh}'$ with rec replaced by rec'.
Pre-computation: Complete pre-computation of all sub-protocols for all levels.
Online-stage:

// Access
1: Run AccFirst with D/E's secret-sharing of $\mathrm{N}^{(0,1)}, \mathsf{fresh}_0$ and stash_0, which outputs pointer p^1 and secret-sharing of $\mathsf{fresh}'_0, \mathsf{stash}'_0$ to D/E.
2: **for** $i := 1$ **to** $h - 2$ **do**
3: Run AccMid with index i, pointer p^i, and D/E's secret-sharing of N^{i+1}, fresh_i, stash_i, which outputs pointer p^{i+1}, and secret-sharing of stash'_i to D/E.
4: **end for**
5: Run AccLast with p^{h-1}, used, C/E's secret-sharing of rec', and D/E's secret-sharing of $\mathrm{N}, \mathsf{fresh}_{h-1}, \mathsf{stash}_{h-1}$, which outputs used', C/E's secret-sharing of rec and D/E's secret-sharing of stash'_{h-1} with rec' inserted.
6: Access counter ctr := ctr + 1 (mod T).
7: Output used', C/E's secret-sharing of rec, D/E's secret-sharing of $\mathsf{stash}' = [\mathsf{stash}'_0, \mathsf{stash}'_1, ..., \mathsf{stash}'_{h-1}]$ and $\mathsf{fresh}' = [\mathsf{fresh}'_0, \mathsf{fresh}_1, ..., \mathsf{fresh}_{h-1}]$ (note that only fresh_0 is changed).
// (Re)initialization
8: **if** ctr = 0 **then**
9: D/E move blocks from stash'_{h-1} to used' positions on fresh_{h-1}.
10: Run INIT with D/E's secret-sharing of fresh_{h-1}, which outputs new D/E secret-sharing of $\mathsf{fresh}' = [\mathsf{fresh}'_0, ..., \mathsf{fresh}'_{h-1}]$.
11: D/E empty/reset used' and secret-sharing of stash'.
12: **end if**

and also re-building the position map. Moving the used block on stash_{h-1} back to fresh_{h-1} is only local operation, as the accessed positions are public information stored in used, an array data structure. Then the re-shuffling is done by first reverting the previous permutation used for shuffling, and then shuffling the fresh_{h-1} again with new randomly picked permutation. And all previous levels, which is the position map, is completely re-built as in the first initialization, using the newly picked last level permutation.

6.2 Access Protocols

6.2.1 AccFirst, AccMid, and AccLast

In this section we explain how access protocol is done in 3PC-Sqrt-ORAM in detail. Because there are differences for access on different ORAM levels, we will explain all these access protocols, AccFirst - access on the first level, AccMid - access on the middle levels, and AccLast - access on the last level, in the following.

AccFirst. AccFirst protocol in Alg. 6.3 takes stash_0, fresh_0, address prefix N^0, and address suffix $\Delta\mathrm{N}^0$ as input, and outputs the pointer to the next level. At the same time, block of the output pointer will have that pointer flag set to used, and the first block of fresh will be *moved* to stash (not copied as in all other levels), with stash_0 size increased by 1 and fresh_0 size decreased by 1. Detailed access steps are described as following.

First, we retrieve the first block B of fresh_0 which is guaranteed to be a fresh block. We also append stash_0 and fresh_0 together into list, which contains all the blocks on the first level. Then the 3PC protocol KSearch is performed with input N^0 and list to identify the index i of the block with address N^0. With this index i, 3PC protocol 3ShiftPIR can be performed to retrieve the block A with address N^0, and 3ShiftXorPIR can be performed on input A, B, $\Delta\mathrm{N}^0$ to retrieve the pointers a (from A) and b (from b) at pointer position $\Delta\mathrm{N}^0$ in A and B. We now can execute 3PC protocol SSOT to obliviously select which pointer of a, b is the fresh one we want to output for the next level. The decision is made based on the pointer flag $a.\mathsf{fb}$ of a, that if $a.\mathsf{fb} = 0$ (not used), then we should output pointer $a.\mathsf{ptr}$; otherwise, we should output $b.\mathsf{ptr}$. Once we have decided which pointer to use, we should now update the pointer flag accordingly. Since B is always a fresh block that $b.\mathsf{fb} = 0$, there are only two different cases, that $a.\mathsf{fb}$ is either 0 or 1. So after the flag update, $a.\mathsf{fb}$ becomes 1 if it was 0, or $b.\mathsf{fb}$ becomes 1 from 0 if $a.\mathsf{fb}$ is already 1. Thus we have such observation: $a.\mathsf{fb}$ after

Algorithm 6.3 Protocol AccFirst - Access on the First Level

Input: $\langle \mathrm{N}, \Delta\mathrm{N}, \mathsf{stash}, \mathsf{fresh} \rangle$ (super/subscript omitted)

Output: (1) $p_{\mathsf{nxt}} = X.\mathsf{rec}[\Delta\mathrm{N}].\mathsf{ptr}$ for block X s.t.

$$X = \begin{cases} \mathsf{list}[i] & \text{if } (\mathsf{list}[i].\mathsf{adr} = \mathrm{N}) \wedge (\mathsf{list}[i].\mathsf{rec}[\Delta\mathrm{N}].\mathsf{fb} = 0) \\ \mathsf{fresh}[0] & \text{if no such } i \text{ exists} \end{cases}$$

(2) $\langle \mathsf{stash}' \rangle := \langle \mathsf{stash} | X \rangle$, with block X s.t. $X.\mathsf{rec}[\Delta\mathrm{N}].\mathsf{fb}$ is set to 1

(3) $\langle \mathsf{fresh}' \rangle := \langle \mathsf{fresh}[1:-1] \rangle$
(if $\exists\, i > 0$ s.t $\mathsf{fresh}[i].\mathsf{adr} = \mathrm{N}$, then $\mathsf{fresh}'[i-1]$ is set to $\mathsf{fresh}[0]$)

Online:

1: $\langle B \rangle := \langle \mathsf{fresh}[0] \rangle$, $\langle \mathsf{list} \rangle := \langle \mathsf{stash} | \mathsf{fresh} \rangle$ ▷ $|\mathsf{list}| = |\mathsf{stash}| + |\mathsf{fresh}|$

2: KSearch: $\langle \mathsf{list}, \mathrm{N} \rangle \rightarrow \langle i \rangle_{\mathsf{shift}}$ ▷ i s.t. $\mathsf{list}[i].\mathsf{adr} = \mathrm{N}$

3: 3ShiftPIR: $\langle \mathsf{list} \rangle, \langle i \rangle_{\mathsf{shift}} \rightarrow \langle A \rangle$ ▷ $A = \mathsf{list}[i]$

4: 3XorPIR: $\langle A, B, \Delta\mathrm{N} \rangle \rightarrow \langle a, b \rangle$ ▷ $a, b = A.\mathsf{rec}[\Delta\mathrm{N}], B.\mathsf{rec}[\Delta\mathrm{N}]$

5: SSOT: $\langle a.\mathsf{fb}, a.\mathsf{ptr}, b.\mathsf{ptr} \rangle \rightarrow p_{\mathsf{nxt}}$ ▷ if $a.\mathsf{fb} = 0, p_{\mathsf{nxt}} = a.\mathsf{ptr}$; else $p_{\mathsf{nxt}} = b.\mathsf{ptr}$

6: SSXOT $\langle \Delta\mathrm{N}, A \rangle \rightarrow \langle A' \rangle$ ▷ set $A.\mathsf{rec}[\Delta\mathrm{N}].\mathsf{fb} := 1$

7: SSXOT: $\langle \Delta\mathrm{N}, B, a.\mathsf{fb} \rangle \rightarrow \langle B' \rangle$ ▷ set $B.\mathsf{rec}[\Delta\mathrm{N}].\mathsf{fb} := a.\mathsf{fb}$

8: SSXOT: $\langle i \rangle_{\mathsf{shift}}, \langle \mathsf{list}, B' \rangle \rightarrow \langle \mathsf{list}' \rangle$ ▷ set $\mathsf{list}[i] := B'$

9: $\langle \mathsf{list}'[|\mathsf{stash}|] \rangle := \langle A' \rangle$

10: $\langle \mathsf{stash}' | \mathsf{fresh}' \rangle := \langle \mathsf{list}' \rangle$ ▷ $|\mathsf{stash}'| = |\mathsf{stash}| + 1, |\mathsf{fresh}'| = |\mathsf{fresh}| - 1$

the update is always 1 no matter what it was before, and $b.\mathsf{fb}$ after the update is equal to the same value of $a.\mathsf{fb}$ before the update. With this observation, using SSXOT protocol from [18], we can first set $a.\mathsf{fb}$ to 1, and also set $b.\mathsf{fb}$ to the old $a.\mathsf{fb}$ value. And now blocks A and B are obliviously updated to A′, B′ based on whose pointer is used for output.

The final step of AccFirst is to move the updated block on to stash_0. We do this by first inserting B′ back to list at position i using protocol SSXOT, and then resetting the $|\mathsf{stash}_0|$-th item on list to A′. And then the new list' is split into the new stash'_0 and fresh'_0, where stash'_0 now has 0-th to $|\mathsf{stash}_0|$-th items from list', and fresh'_0 has the rest blocks (so the total blocks on the first level doesn't change, but every time we *move* the first block of fresh_0 on to stash_0). The order here for inserting A′ and B′ back to list is critical to the correctness of the access. The reason is as following: there are three possibilities for the locations of A and B on list,

(1) A is before B, meaning A is on stash_0; (2) A is at the same location of B, meaning A = B is a fresh block; (3) A is after B, meaning A is a different fresh block on fresh_0. For case (1), no matter whether A or B is updated on pointer flag, after inserting back, the new versions of A′ and B′ are still on the new stash'_0. For case (2), since A is a fresh block, A's pointer flag must be updated while B's flag is not, and because A and B are at the same location, inserting B back first and then inserting A ensures that the correct version of the block, which is A′, can overwrite the old version of the block B′. For case (3), A is also the block to be updated while B is not, but A's position i is after B's position, meaning this position i is still on the new fresh'_0 at the end of this access, and we cannot put an non-fresh block at this position. Thus, we first insert B′, which is unchanged and fresh, back to the position i, and then insert A′ to the old position of B. In such a way, the updated block A' will be on the new stash'_0, while the fresh block B′ is still on fresh'_0.

AccMid. AccMid protocol in Alg. 6.4 takes $\mathsf{stash}_i, \mathsf{fresh}_i, \mathrm{N}^i, \Delta\mathrm{N}^i$, and pointer p from previous level as input, outputs next level pointer, and updates block pointer flag accordingly, similar to AccFirst. What's different is that now block B is not the first block of fresh_i but the p-th block, and when we do KSearch to search for index i of block A, we are only linear scanning the stash_i appended with block B (after each access the stash_i size is increased by 1 and fresh_i remains unchanged). In AccMid, the same procedure of AccFirst is followed to retrieve block A and the next level pointer. However, unlike AccFirst, blocks A and B are both on stash now, so we don't need to update blocks A and B and insert whole blocks of them back to stash. Instead, to make pointer flag update, we may just set the i-th block's (A's) pointer flag to 1, and set the pointer flag of last block on stash (B) to A's previous pointer flag (same reason as in AccFirst). Both of these updates can be done with SSXOT as the way in AccFirst.

AccLast. Protocol AccLast in Alg. 6.5 is very similar to AccMid, except for access = write, we have the extra input new record rec', and we don't output pointer any more but the accessed record rec. We still perform the same procedure for finding index i of block A. Then 3ShiftPIR

Algorithm 6.4 Protocol AccMid - Access on Each Middle Level

Input: p, $\langle \mathrm{N}, \Delta\mathrm{N}, \mathsf{stash}, \mathsf{fresh} \rangle$ (super/subscript omitted)

Output: (1) $p_{\mathsf{nxt}} = X.\mathsf{rec}[\Delta\mathrm{N}].\mathsf{ptr}$ for block X s.t.

$$X = \begin{cases} \mathsf{stash}[i] & \text{if } (\mathsf{stash}[i].\mathsf{adr} = \mathrm{N}) \wedge (\mathsf{stash}[i].\mathsf{rec}[\Delta\mathrm{N}].\mathsf{fb} = 0) \\ \mathsf{fresh}[p] & \text{if no such } i \text{ exists} \end{cases}$$

(2) $\langle \mathsf{stash}' \rangle := \langle \mathsf{stash} | \mathsf{fresh}[p] \rangle$,
with block X on stash' s.t. $X.\mathsf{rec}[\Delta\mathrm{N}].\mathsf{fb}$ is set to 1

Online:

1: $\langle B \rangle := \langle \mathsf{fresh}[p] \rangle$, $\langle \mathsf{list} \rangle := \langle \mathsf{stash} | B \rangle$ ▷ $|\mathsf{list}| = |\mathsf{stash}| + 1$

2: KSearch: $\langle \mathsf{list}, \mathrm{N} \rangle \rightarrow \langle i \rangle_{\mathsf{shift}}$ ▷ i s.t. $\mathsf{list}[i].\mathsf{adr} = \mathrm{N}$

3: 3ShiftPIR: $\langle \mathsf{list} \rangle, \langle i \rangle_{\mathsf{shift}} \rightarrow \langle A \rangle$ ▷ $A = \mathsf{list}[i]$

4: 3XorPIR: $\langle A, B, \Delta\mathrm{N} \rangle \rightarrow \langle a, b \rangle$ ▷ $a, b = A.\mathsf{rec}[\Delta\mathrm{N}], B.\mathsf{rec}[\Delta\mathrm{N}]$

5: SSOT: $\langle a.\mathsf{fb}, a.\mathsf{ptr}, b.\mathsf{ptr} \rangle \rightarrow p_{\mathsf{nxt}}$ ▷ if $a.\mathsf{fb} = 0, p_{\mathsf{nxt}} = a.\mathsf{ptr}$; else $p_{\mathsf{nxt}} = b.\mathsf{ptr}$

6: SSXOT: $\langle i \rangle_{\mathsf{shift}}, \langle \Delta\mathrm{N}, \mathsf{list} \rangle \rightarrow \langle \mathsf{list}' \rangle$ ▷ set $\mathsf{list}[i].\mathsf{rec}[\Delta\mathrm{N}].\mathsf{fb} := 1$

7: SSXOT: $\langle \Delta\mathrm{N}, \mathsf{list}', a.\mathsf{fb} \rangle \rightarrow \langle \mathsf{stash}' \rangle$ ▷ set $\mathsf{list}'[-1].\mathsf{rec}[\Delta\mathrm{N}].\mathsf{fb} := a.\mathsf{fb}$

Algorithm 6.5 Protocol AccLast - Access on the Last Level

Input: p, used, $\langle \mathrm{N}, \mathsf{stash}, \mathsf{fresh}, \mathsf{rec}' \rangle$

Output: (1) $\langle \mathsf{rec} \rangle = \langle X.\mathsf{rec} \rangle$ for block X s.t.

$$X = \begin{cases} \mathsf{stash}[i] & \text{if } \mathsf{stash}[i].\mathsf{adr} = \mathrm{N} \\ \mathsf{fresh}[p] & \text{if no such } i \text{ exists} \end{cases}$$

(2) $\langle \mathsf{stash}' \rangle := \langle \mathsf{stash} | \mathsf{fresh}[p] \rangle$
with block X on stash' s.t. $X.\mathsf{rec}$ is set to rec'
(3) $\mathsf{used}' := \mathsf{Append}(\mathsf{used}, p)$

Online:

1: $\langle B \rangle := \langle \mathsf{fresh}[p] \rangle$, $\langle \mathsf{list} \rangle := \langle \mathsf{stash} | B \rangle$ ▷ $|\mathsf{list}| = |\mathsf{stash}| + 1$

2: KSearch: $\langle \mathsf{list}, \mathrm{N} \rangle \rightarrow \langle i \rangle_{\mathsf{shift}}$ ▷ i s.t. $\mathsf{list}[i].\mathsf{adr} = \mathrm{N}$

3: 3ShiftPIR: $\langle \mathsf{list} \rangle, \langle i \rangle_{\mathsf{shift}} \rightarrow \langle \mathsf{rec} \rangle$ ▷ $\mathsf{rec} = \mathsf{list}[i].\mathsf{rec}$

4: SSXOT: $\langle i \rangle_{\mathsf{shift}}, \langle \mathsf{list}, \mathsf{rec}' \rangle \rightarrow \langle \mathsf{stash}' \rangle$ ▷ set $\mathsf{list}[i].\mathsf{rec} := \mathsf{rec}'$

5: $\mathsf{used}' := \mathsf{Append}(\mathsf{used}, p)$

can be used to directly output the record rec. And the record update operation can also be done directly using SSXOT, to overwrite rec with rec′ at position i. Finally the accessed position is recorded in used. And a 3PC-Sqrt-ORAM record retrieval is completed at this point.

Sub-protocol SSOT. Most of the 3PC sub-protocols like KSearch, 3ShiftPIR, 3ShiftXorPIR, SSXOT, are adoption or modification from the 3PC-Circuit-ORAM of [18]. Sub-protocol SSOT in Alg. 6.6 is a new protocol used in our 3PC-Sqrt-ORAM construction, but it is essentially another variant of the 3PC-OT protocol.

Algorithm 6.6 Protocol SSOT - Secret-Shared OT

Input: $\langle b\rangle_{\mathsf{xor}}^{\mathrm{C}-E} = (b_0^{\mathrm{C}}, b_1^{\mathrm{E}})$, $\langle m_0\rangle_{\mathsf{xor}}^{\mathrm{C}-E} = (u_0^{\mathrm{C}}, v_0^{\mathrm{E}})$, $\langle m_1\rangle_{\mathsf{xor}}^{\mathrm{C}-E} = (u_1^{\mathrm{C}}, v_1^{\mathrm{E}})$.
Output: $p = m_b$.
Offline: Let $l = |m_0|$. D picks $x_0, x_1, y_0, y_1, \delta \xleftarrow{\$} \{0,1\}^l$, $c, e \xleftarrow{\$} \{0,1\}^1$, computes $x = x_e \oplus \delta, y = y_c \oplus \delta$, and sends x_0, x_1, y, c to C and y_0, y_1, x, e to E.
Online:

1: C sends $s = b_0 \oplus c$ to E. E sends $t = b_1 \oplus e$ to C.
2: C sends $u'_0 = u_{b_0} \oplus x_t, u'_1 = u_{\overline{b_0}} \oplus x_{\bar{t}}$ to E.
E sends $v'_0 = v_{b_1} \oplus y_s, v'_1 = v_{\overline{b_1}} \oplus y_{\bar{s}}$ to C.
C computes $p_0 = v'_{b_0} \oplus y$, E computes $p_1 = u'_{b_1} \oplus x$,
s.t. $p_0 \oplus p_1 = m_b$ (so we have $\langle p\rangle_{\mathsf{xor}}^{\mathrm{C-E}} = (p_0^{\mathrm{C}}, p_1^{\mathrm{E}}) = \langle m_b\rangle_{\mathsf{xor}}^{\mathrm{C-E}}$).
3: Reshare: $\langle p\rangle_{\mathsf{xor}}^{\mathrm{C-E}} \rightarrow p$.

Correctness: When $(b_0, b_1) = (0, 0)$, meaning $b = 0$, $p = p_1 \oplus p_0 = (u'_0 \oplus x) \oplus (v'_0 \oplus y) = (u_0 \oplus x_t \oplus x_e \oplus \delta) \oplus (v_0 \oplus y_s \oplus y_c \oplus \delta) = (u_0 \oplus x_e \oplus x_e) \oplus (v_0 \oplus y_c \oplus y_c) = u_0 \oplus v_0 = m_0 = m_b$.
The other 3 cases with different (b_0, b_1) values can be verified in the same way.

6.2.2 Initialization

To randomize the data block locations for oblivious access, at the beginning when initializing the ORAM, or after each T accesses when re-initializing the ORAM, we need to shuffle the data blocks on the last level as well as (re-)building the position map as the previous levels. In this section we describe how the initialization protocol works.

By the observation of [51], the logical indexes stored in data blocks and the physical indexes of these blocks together define the inverse of the permutation used to shuffle the data blocks from the identity positions. Therefore the permutation for the data block shuffling can be efficiently stored as secret-sharing among parties. And in our 3PC-Sqrt-ORAM, we still apply this trick to secret-share the permutation used for shuffling, and in addition, we design efficient 3PC protocol to convert secret-sharing of permutation between permutation composition format and standard xor sharing format, which allows us to efficiently shuffle data blocks with complexity $O(n)$ using 3PC-OT variants instead of the $O(n\log(n))$ shuffling network in [51].

The Initialization protocol INIT is presented in Alg. 6.7 and works as following. Before any access, or after T accesses, we may isolate the logical index field of data blocks on last level fresh_{h-1} into an array L, and L defines either the identity permutation if this is the first time we plan to shuffle the data blocks, or the permutation used to shuffle the data blocks for the past T accesses. In order to update this L and re-shuffle the data blocks, party D will pick new random permutation π_{D}, and parties C/E will pick random permutation π_{E}. Then through the 3PC protocol $\mathsf{OblivPermute}$ of Alg. 6.9, secret-sharing of the new logic index array $L' = \pi_{\mathrm{D}} \cdot \pi_{\mathrm{E}}(L)$ can be generated (and its inverse $(L')^{-1}$ is equivalent to the permutation for shuffling blocks if started with the identity positions). The above logic index array shuffling can be done offline, as once initialized, logic index array of blocks do not change during the ORAM accesses. However, as records may be updated during the access, the (re-)shuffling of block data part has to be done online. This is also done by using the 3PC protocol $\mathsf{OblivPermute}$ with the permutations π_{D} and π_{E}.

When the permutation concatenation sharing $\pi_{\mathrm{D}}, \pi_{\mathrm{E}}$ for (re-)shuffling data blocks on the last level is determined, the previous levels position map can be re-built. If this is the first time we build the position map, then obliviously permutation $\pi = \pi_{\mathrm{D}} \cdot \pi_{\mathrm{E}}$ is the permutation we should use for building the position map. However, if we are re-building the position

Algorithm 6.7 Protocol INIT - Initialize ORAM

Parameter: Number of levels h. Number of data records $n = 2^{\log(n)} = |\mathsf{fresh}_{h-1}|$.
Input: D/E's secret-sharing of fresh_{h-1}.
Output: New D/E's secret-sharing of $\mathsf{fresh} = [\mathsf{fresh}_0, ..., \mathsf{fresh}_{h-1}]$.

Pre-computation:

1: D picks $\pi_{\mathrm{D}} \xleftarrow{\$} \mathsf{perm}_n$. E picks $\pi_{\mathrm{E}} \xleftarrow{\$} \mathsf{perm}_n$ and sends it to C.
2: D/E extract secret-sharing of l field of every block in fresh_{h-1} into $L \in \mathsf{perm}_n$.
3: D sends $\mathrm{s}_\alpha[L]$ to C. Run OblivPermute with input D's π_{D}, C and E's π_{E}, C and D's $\mathrm{s}_\alpha[L]$, and E's $\mathrm{s}_\beta[L]$, which outputs D/E's secret-sharing of $L' = \pi_{\mathrm{D}} \cdot \pi_{\mathrm{E}}(L)$.
4: **if** first time initializing **then**
5: D sets $\pi_{\mathrm{D}}^{h-1} := \pi_{\mathrm{D}}$. C and E sets $\pi_{\mathrm{E}}^{h-1} := \pi_{\mathrm{E}}$.
6: **else**
7: Run GenPermConcat with D/E's secret-sharing of L', which outputs π_{D}^{h-1} to D and π_{E}^{h-1} to E where $\pi_{\mathrm{D}}^{h-1} \cdot \pi_{\mathrm{E}}^{h-1} = (L')^{-1}$. E sends π_{E}^{h-1} to C.
8: **end if**
9: Run InitPosMap with D's π_{D}^{h-1} and C and E's π_{E}^{h-1}, which outputs D/E's secret-sharing of $\mathsf{fresh}_0, ..., \mathsf{fresh}_{h-2}$.

Online-stage:

1: D/E extract secret-sharing of rec field of every block in fresh_{h-1} into $Y \in \mathsf{array}^B[n]$.
2: D sends $\mathrm{s}_\alpha[Y]$ to C. Run OblivPermute with input D's π_{D}, C and E's π_{E}, C and D's $\mathrm{s}_\alpha[Y]$, and E's $\mathrm{s}_\beta[Y]$, which outputs D/E's secret-sharing of $Y' = \pi_{\mathrm{D}} \cdot \pi_{\mathrm{E}}(Y)$.
3: D/E rewrite secret-sharing of fresh_{h-1} with L' and Y', and output $\mathsf{fresh} = [\mathsf{fresh}_0, ..., \mathsf{fresh}_{h-1}]$.

map, then the current permutation defined by the logic index array is $(L')^{-1}$, and we should use concatenation sharing of $(L')^{-1}$ to re-build the position map. With the 3PC protocol GenPermConcat of Alg. 6.10, given the xor sharing of L', we can compute $\pi_{\mathrm{D}}^{h-1}, \pi_{\mathrm{E}}^{h-1}$ such that $\pi_{\mathrm{D}}^{h-1} \cdot \pi_{\mathrm{E}}^{h-1} = (L')^{-1}$. And this allows us to re-build the position map with concatenation sharing $\pi_{\mathrm{D}}^{h-1}, \pi_{\mathrm{E}}^{h-1}$ of $(L')^{-1}$.

Initialize Position Map. Each level of the position map is built and shuffled similarly to the initialization of the last data block level. Given the concatenation sharing of permutation used to shuffle the next level, xor sharing of such permutation can be generated with 3PC protocol GenPermShare of Alg. 6.11, and pointers to the next level, which are stored in xor sharing format, can be arranged accordingly. Then permutation concatenation sharing for the current level will be randomly picked to shuffle blocks on the current level using

Algorithm 6.8 Protocol InitPosMap - Initialize Position Map

Input: D's π_{D}^{h-1} and C and E's π_{E}^{h-1} where $\pi_{\mathrm{D}}^{h-1}, \pi_{\mathrm{E}}^{h-1} \in \mathsf{perm}_{|\mathsf{fresh}_{h-1}|}$.
Output: D/E's secret-sharing of $\mathsf{fresh}_0, ..., \mathsf{fresh}_{h-2}$.
Pre-computation: D picks random $\pi_{\mathrm{D}}^0, ..., \pi_{\mathrm{D}}^{h-2}$ and E picks random $\pi_{\mathrm{E}}^0, ..., \pi_{\mathrm{E}}^{h-2}$ where $\pi_{\mathrm{D}}^i, \pi_{\mathrm{E}}^i \in \mathsf{perm}_{|\mathsf{fresh}_i|}$. E sends $\pi_{\mathrm{E}}^0, ..., \pi_{\mathrm{E}}^{h-2}$ to C.

1: **for** $i := h-2$ **to** 0 **do**
2: Run GenPermShare with D's π_{D}^{i+1} and C and E's π_{E}^{i+1}, which outputs D/E's secret-sharing of $[\pi^{i+1}]$ where $\pi^{i+1} = \pi_{\mathrm{D}}^{i+1} \cdot \pi_{\mathrm{E}}^{i+1}$.
3: **for** $j := 0$ **to** $n-1$ **do**
4: D/E set secret-sharing of $\mathsf{fresh}_i[j].l := j$.
5: **for** $k := 0$ **to** $2^\tau - 1$ **do**
6: D/E set secret-sharing of $\mathsf{fresh}_i[j].\{F[k], P[k]\} := \{0, [\pi^{i+1}][j \cdot 2^\tau + k]\}$.
7: **end for**
8: **end for**
9: D sends $\mathrm{s}_\alpha[\mathsf{fresh}_i]$ to C. Run OblivPermute with D's π_{D}^i, C and E's π_{E}^i, C and D's $\mathrm{s}_\alpha[\mathsf{fresh}_i]$, and E's $\mathrm{s}_\beta[\mathsf{fresh}_i]$, which outputs new D/E's secret-sharing of $\mathsf{fresh}_i = \pi_{\mathrm{D}}^i \cdot \pi_{\mathrm{E}}^i(\mathsf{fresh}_i)$.
10: **end for**
11: D/E output secret-sharing of $\mathsf{fresh}_0, ..., \mathsf{fresh}_{h-2}$.

OblivPermute, and this permutation concatenation sharing will also be passed on to build the previous level position map. The implementations of OblivPermute, GenPermConcat, and GenPermShare will be included in the following for completeness.

Oblivious Permutation. Given input permutations $\pi_{\mathrm{D}}, \pi_{\mathrm{E}}$ and secret-sharing of an array x, protocol OblivPermute outputs secret-sharing of array y where $y = \pi_{\mathrm{D}} \cdot \pi_{\mathrm{E}}(x)$. OblivPermute is a secure (1,3)-MPC against semi-honest adversaries because: (1) E's view is its output. (2) D's view $\mathrm{s}_\alpha[y]$ can be simulated as $\mathrm{s}_\alpha[y]' \xleftarrow{\$} \mathsf{array}^t[v]$ because it is an output of a secure (1,3)-MPC SSXOT. (3) Same reason as above for a semi-honest C.

Generate Permutation Concatenation Shares Using Xor Shares. Given secret sharing $\mathrm{s}_\alpha[\pi], \mathrm{s}_\beta[\pi]$ of a permutation π, GenPermConcat outputs two permutations π_{D} and π_{E} s.t the permutation concatenation $\pi_{\mathrm{D}} \cdot \pi_{\mathrm{E}} = \pi^{-1}$. GenPermConcat is a secure (1,3)-MPC against semi-honest adversaries because: (1) E's view includes σ_1, r_1, z_1, z_2. σ_1, r_1 can be simulated as $\sigma_1' \xleftarrow{\$} \mathsf{perm}_v, r_1' \xleftarrow{\$} \mathsf{array}^{\log(v)}[v]$. $z_1 = \gamma_1(a_1) \oplus t_1 = \gamma_1(a_1) \oplus \gamma_1(r_1) \oplus s$ can be

Algorithm 6.9 Protocol $\mathsf{OblivPermute}$ - Obliviously Permute

Parameter: Positive integers v, t.

Input: D's $\pi_{\mathrm{D}} \in \mathsf{perm}_v$, C and E's $\pi_{\mathrm{E}} \in \mathsf{perm}_v$. C and D's $\mathrm{s}_\alpha[x]$ and E's $\mathrm{s}_\beta[x]$, where $\mathrm{s}_\alpha[x], \mathrm{s}_\beta[x]$ are secret-sharing of array $x \in \mathsf{array}^t[v]$.

Output: D/E's secret-sharing of y where $y = \pi_{\mathrm{D}} \cdot \pi_{\mathrm{E}}(x)$.

Pre-Computation: C/E pick $r \xleftarrow{\$} \mathsf{array}^t[v]$.

Online-stage:

1: Run SSXOT with D's π_{D}^{-1}, and C/E's secret-sharing of $\pi_{\mathrm{E}}(x)$, which outputs secret-sharing $\mathrm{s}_\alpha[y], \mathrm{s}_\beta[y]$ of $y \in \mathsf{array}^t[v]$ to C/E, where each $y[i] = \pi_{\mathrm{E}}(x)[\pi_{\mathrm{D}}^{-1}[i]]$, which means $y[\pi_{\mathrm{D}}[i]] = \pi_{\mathrm{E}}(x)[i]$, so $y = \pi_{\mathrm{D}}(\pi_{\mathrm{E}}(x)) = \pi_{\mathrm{D}} \cdot \pi_{\mathrm{E}}(x)$.

2: C sends $\mathrm{s}_\alpha[y] = \mathrm{s}_\alpha[y] \oplus r$ to D. D outputs $\mathrm{s}_\alpha[y]$. E outputs $\mathrm{s}_\beta[y] = \mathrm{s}_\beta[y] \oplus r$.

Algorithm 6.10 Protocol $\mathsf{GenPermConcat}$ - Generate Permutation Concatenation

Input: $\langle \pi \rangle_{\mathsf{xor}}^{\mathrm{D-E}} = (\pi_a^{\mathrm{D}}, \pi_b^{\mathrm{E}})$, for $\pi \in \mathsf{perm}_v$ and some positive integer v.

Output: $\pi_1^{\mathrm{D}}, \pi_2^{\mathrm{E}} \in \mathsf{perm}_v$ s.t. $\pi_{\mathrm{D}} \cdot \pi_{\mathrm{E}} = \pi^{-1}$.

Offline: D picks $\pi_1, \sigma_1, \sigma_2 \xleftarrow{\$} \mathsf{perm}_v$ and $s, r_1, r_2 \xleftarrow{\$} \mathsf{array}^{\log(v)}[v]$, and computes $\gamma_1 = \pi_1^{-1} \cdot \sigma_1^{-1}, \gamma_2 = \pi_1^{-1} \cdot \sigma_2^{-1}, t_1 = \gamma_1(r_1) \oplus s, t_2 = \gamma_2(r_2) \oplus s$. D sends γ_1, t_1 to C and σ_1, r_1 to E.

Online:

1: E sends $a_1 = \sigma_1(\pi_b) \oplus r_1$ to C.

2: C sends $z_1 = \gamma_1(a_1) \oplus t_1$ to E.
D computes $a_2 = \sigma_2(\pi_a) \oplus r_2,\ z_2 = \gamma_2(a_2) \oplus t_2$ and sends z_2 to E.
D outputs π_1. E outputs $\pi_2 = (z_1 \oplus z_2)^{-1}$.

$$
\begin{aligned}
\textit{Correctness}: z_1 \oplus z_2 &= \gamma_1(a_1) \oplus t_1 \oplus \gamma_2(a_2) \oplus t_2 \\
&= \gamma_1(a_1) \oplus \gamma_1(r_1) \oplus \gamma_2(a_2) \oplus \gamma_2(r_2) \\
&= \gamma_1(a_1 \oplus r_1) \oplus \gamma_2(a_2 \oplus r_2) \\
&= \gamma_1(\sigma_1(\pi_b)) \oplus \gamma_2(\sigma_2(\pi_a)) \\
&= \gamma_1 \cdot \sigma_1(\pi_b) \oplus \gamma_2 \cdot \sigma_2(\pi_a) \\
&= \pi_1^{-1}(\pi_b) \oplus \pi_1^{-1}(\pi_a) \\
&= \pi_1^{-1}(\pi_b \oplus \pi_a) \\
&= \pi_1^{-1}([\pi]) \\
&= [\pi \cdot \pi_1] \\
\pi_1 \cdot \pi_2 = \pi_1 \cdot (z_1 \oplus z_2)^{-1} &= \pi_1 \cdot (\pi \cdot \pi_1)^{-1} = \pi_1 \cdot \pi_1^{-1} \cdot \pi^{-1} = \pi^{-1}
\end{aligned}
$$

simulated as $z_1' \xleftarrow{\$} \mathsf{array}^{\log(v)}[v]$ because s is unknown to E. Then given E's output π_{E}, z_2 can be simulated as $z_2' = [\pi_{\mathrm{E}}^{-1}] \oplus z_1$. (2) D receives nothing. (3) C's view includes γ_1, t_1, a_1. γ_1 can be simulated as $\gamma_1' \xleftarrow{\$} \mathsf{perm}_v$ because π_{D} is unknown to C. t_1 can be simulated as $t_1' \xleftarrow{\$} \mathsf{array}^{\log(v)}[v]$ because s is unknown to C. And a_1 can be simulated as $a_1' \xleftarrow{\$} \mathsf{array}^{\log(v)}[v]$ because r_1 is unknown to C.

Generate Permutation Xor Shares from Concatenation Shares. Given permutations π_{D} and π_{E}, GenPermShare outputs secret-sharing $\mathrm{s}_\alpha[\pi], \mathrm{s}_\beta[\pi]$ of permutation π s.t. π is the permutation concatenation $\pi_{\mathrm{D}} \cdot \pi_{\mathrm{E}}$. GenPermShare is a secure (1,3)-MPC against semi-honest adversaries because: (1) E's view includes r and z. z can be simulated as $z' \xleftarrow{\$} \mathsf{array}^{\log(v)}[v]$ because p is unknown to E. Then given E's input π_{E} and output $\mathrm{s}_\beta[\pi]$, r can be simulated as $r' = \pi_{\mathrm{E}}(\mathrm{s}_\beta[\pi]) \oplus z'$. (2) D's view $\mathrm{s}_\alpha[\pi]$ is its output. (3) C receives nothing.

Algorithm 6.11 Protocol GenPermShare - Generate Permutation Secret-Sharing

Input: $\pi_1^{\mathrm{D}}, \pi_2^{\mathrm{CE}} \in \mathsf{perm}_v$, for some positive integer v.
Output: $\langle \pi \rangle_{\mathsf{xor}}^{\mathrm{D-E}}$, where $\pi = \pi_1 \cdot \pi_2$.
Offline: Pick $p^{\mathrm{CD}}, r^{\mathrm{CE}} \xleftarrow{\$} \mathsf{array}^{\log(v)}[v]$.
Online:

1: C sends $a = \pi_2^{-1}(p \oplus r)$ to D. D sends $z = [\pi_1] \oplus p$ to E.
E computes $b = \pi_2^{-1}(z \oplus r)$. $\langle \pi \rangle_{\mathsf{xor}}^{\mathrm{D-E}} = (a^{\mathrm{D}}, b^{\mathrm{E}})$.

$Correctness: a \oplus b = \pi_2^{-1}(p \oplus r) \oplus \pi_2^{-1}(z \oplus r) = \pi_2^{-1}(p \oplus z) = \pi_2^{-1}([\pi_1]) = [\pi_1 \cdot \pi_2]$

6.3 Analysis

Asymptotic Complexity. At each level of the access, the main cost comes from linearly scanning and updating the stash with size T using 3PC-OT variant protocols from [29], therefore the asymptotic cost would be $O(\mathrm{T}B)$ for the last level and $O(\mathrm{T}\log(n))$ for each of the rest levels. For every T accesses, the ORAM is re-initialized with 3PC oblivious shuffling, which has complexity $O(nB)$ for the last level and $O(n\log(n))$ for each of the

rest levels. Amortizing the re-initialization cost over each access, and taking the $O(\log(n))$ recursive levels into the account, we get the the total 3PC-Sqrt-ORAM access complexity is $O(\sqrt{n} \cdot (\log^2(n) + B))$ for T = $O(\sqrt{n})$. As access on each level takes constant rounds, the total round complexity is $O(\log(n))$.

Security. Our Sqrt-ORAM algorithm meets the ORAM requirement that no information is revealed about the memory access pattern. At each level of the access, a new and fresh block is visited and added to the stash regardless of the sequence and address of the access, and the stash is always linearly scanned, which guarantees to find a block and a new fresh pointer for the next level. The entire ORAM structure is reshuffled every fixed number of accesses, which is independent of the access pattern. The only public information are the visited block positions and the current stash size counter, which also contains no information about the access locations and pattern. Therefore, as the 3PC protocols used in this scheme are secure (1,3)-MPC protocols and leaks no information about the oblivious variables of the ORAM scheme, our 3PC-Sqrt-ORAM is a secure 3PC-ORAM, by the argument of [44].

6.4 Implementation and Concrete Performance

We tested a Java prototype of our 3PC-Sqrt-ORAM, on three AWS EC2 c4.2xlarge servers, with communication links encrypted using AES-128. Each c4.2xlarge instance is equipped with eight Intel Xeon E5-2666 v3 CPU's (2.9 GHz), 15 GB memory, and has 1 Gbps bandwidth. As online complexity is more important in the real world, we paid more attention towards the optimization of the online phrase of our algorithm, and compared with the state-of-art 3PC-Circuit-ORAM scheme on both wall clock time and bandwidth.

Online Wall Clock Time. As shown in Fig. 6.2, our 3PC-Sqrt-ORAM is very efficient for small $\log(n)$, and takes only 5ms for each access for small $\log(n) = 6$. For the range of $\log(n)$

from 5 to 22, our 3PC-Sqrt-ORAM requires less runtime than 3PC-Circuit-ORAM, and can be about 4x times better for some $\log(n)$ in that range. Due to the asymptotic complexity that $O(\sqrt{n})$ cost of 3PC-Sqrt-ORAM is worse than $O(\log(n))$ of 3PC-Circuit-ORAM, as $\log(n)$ grows, eventually the runtime of 3PC-Sqrt-ORAM is worse than the runtime of 3PC Circuit-ORAM, but for a wide range of small $\log(n)$, 3PC-Sqrt-ORAM is extremely efficient in practice.

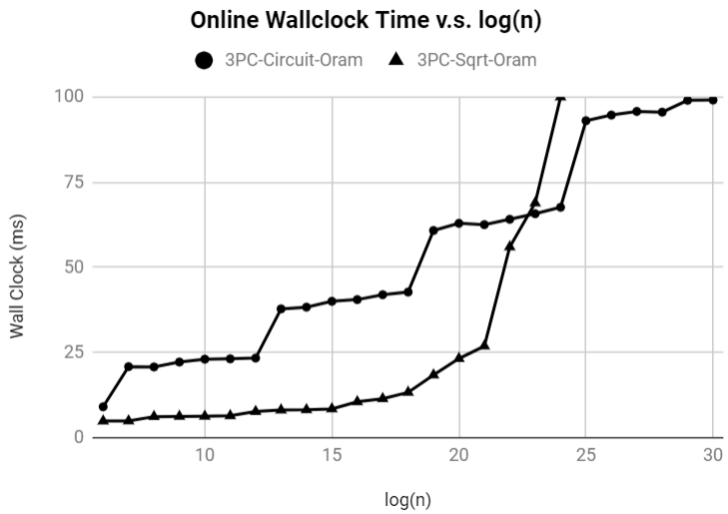


Figure 6.2: Runtime comparison between 3PC-Sqrt-ORAM and 3PC-Circuit-ORAM

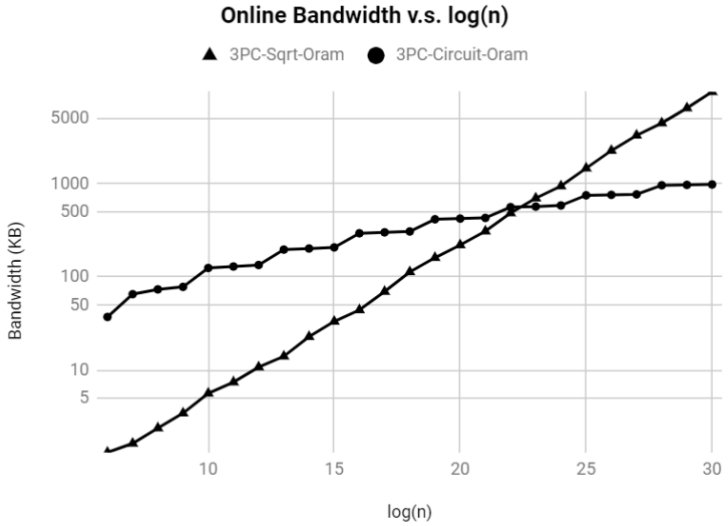


Figure 6.3: Online Bandwidth comparison between 3PC-Sqrt-ORAM and 3PC-Circuit-ORAM

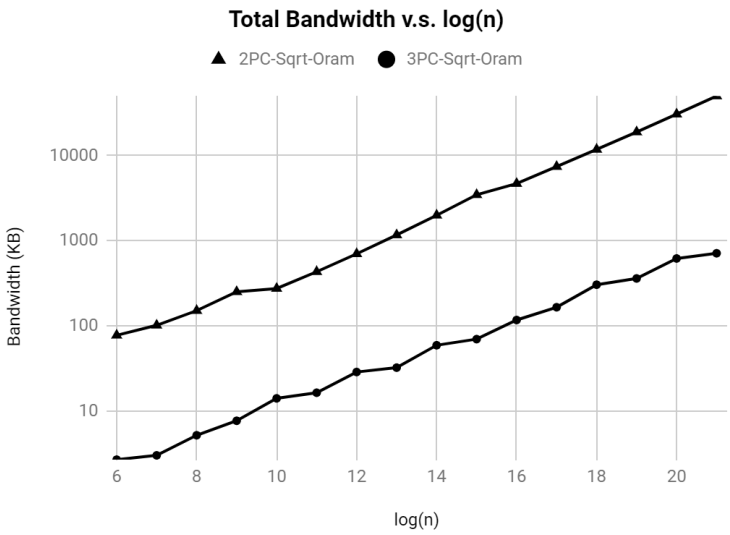


Figure 6.4: Total Bandwidth comparison between 2PC-Sqrt-ORAM and 3PC-Sqrt-ORAM

Online Bandwidth. The online bandwidth comparison graph with the 3PC-Circuit-ORAM has very similar looking to the wall clock time comparison graph. To view it with larger bandwidth range, in Fig. 6.3 we compare the bandwidth v.s. $\log(n)$ on a log scale. The break even point is still around $\log(n) = 22$, that for $\log(n) > 22$, due to the asymptotic complexity, the bandwidth of 3PC-Sqrt-ORAM is worse than the bandwidth of 3PC-Circuit-ORAM. However, for small $\log(n) < 22$, the bandwidth of 3PC-Sqrt-ORAM outperforms bandwidth of 3PC-Circuit-ORAM, and can be around 40x times better for $\log(n) = 7$ due to much better constants in the concrete cost.

Total Bandwidth. The total bandwidth comparison with 2PC-Sqrt-ORAM [51] is shown in Fig. 6.4. As expected, because of the use of cheap 3PC protocols, and the careful design which eliminates the needs of using expensive generic 2PC computation Yao's garbled circuit, our 3PC-Sqrt-ORAM does not have the extra security parameter factor in the cost formula, and thus our bandwidth can outperform the bandwidth of 2PC-Sqrt-ORAM as $\log(n)$ grows, and is 70x better at $\log(n) = 21$, which is getting closer and closer to the garbled circuit security parameter 80 used in 2PC-Sqrt-ORAM. Due to different implementation of 2PC Sqrt-ORAM, which uses C implementation different from our Java implementation, there is

no direct comparison on access runtime between the two schemes. However, we speculate the runtime graph would be very similar to the bandwidth graph if implementation is done in the same setting.

Chapter 7

3PC-DPF-ORAM

This chapter describes our 3PC-DPF-ORAM scheme, which is based on the 2PC-FLORAM of [15] but improves efficiency by eliminating the expensive PRF evaluation, removing the use of Yao's garbled circuit, and replacing the primitive 2PC stash linear scan with more efficient 3PC recursive ORAM access. This a joint work with Stanislaw Jarecki, Jonathan Katz, Mariana Raykova, and Xiao Wang.

7.1 Definitions

We make new definitions or recall from previous chapters to make notations most suitable for the 3PC-DPF-ORAM scheme of this chapter.

7.1.1 Random Access Machines (RAMs)

A RAM program is defined by its next-instruction function Π, which takes as input the current state st as well as the last data item d retrieved from memory. The function

computes an updated state st' together with a new memory access instruction (op, i, v). The computation of the RAM program with external memory M proceeds by repeated execution of the Π function, followed by memory access, until an end state is reached as shown below.

$$
\begin{aligned}
&\mathit{while}(\mathsf{st} \neq \mathsf{stop})\ \{\\
&\quad (\mathsf{st}', (\mathsf{op}, i, v)) \leftarrow \Pi(\mathsf{st}, d)\\
&\quad d \leftarrow \mathsf{Retrieval}^{\mathsf{M}}(\mathsf{op}, i, v)\\
&\quad \mathsf{st} \leftarrow \mathsf{st}'\ \}
\end{aligned}
$$

7.1.2 Oblivious RAM (ORAM)

We use the standard definitions of correctness and security for (two-server) ORAM, adapted from [45]; they are included here for convenience and to establish notation.

Let M be an n-element array containing B-bit entries. For fixed n, B, a *memory access* is a tuple (op, i, v) where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $i \in \{1, \ldots, n\}$, and $v \in \{0,1\}^B$. The result of applying (read, i, v) to M is $\mathsf{M}[i]$, and the array M is unchanged. The result of applying (write, i, v) is $\perp$, and M is updated to a new array M' that is identical to M except that $\mathsf{M}'[i] = v$. Given an initial array M and a sequence of memory accesses $(\mathsf{op}_1, i_1, v_1)$, $\ldots$, $(\mathsf{op}_m, i_m, v_m)$, we define correctness for the sequence of results $o_1, \ldots, o_m$ in the natural way; namely, the sequence of results is correct iff, for all t, if $\mathsf{op}_t = \mathsf{read}$ then the result o_t is equal to the last value written to i_t (or $\mathsf{M}[i_t]$ if there were no previous writes to i_t).

A two-server ORAM scheme is defined by algorithms $\mathsf{ORAM.INIT}$, $\mathsf{ORAM.C}$, $\mathsf{ORAM.S}_1$, and $\mathsf{ORAM.S}_2$ with the following syntax:

- $\mathsf{ORAM.INIT}$ takes as input 1^κ and elements $\mathsf{M}[1], \ldots, \mathsf{M}[n] \in \{0,1\}^B$. It outputs state st and data $\tilde{\mathsf{M}}_1, \tilde{\mathsf{M}}_2$ to be stored at the servers.

- $\mathsf{ORAM.C}$ is an interactive algorithm that takes as input st and a memory access (op, i, v), and outputs updated state st' and a value o.
- $\mathsf{ORAM.S}_1$ and $\mathsf{ORAM.S}_2$ are interactive algorithms that take as input data $\tilde{\mathsf{M}}$ and outputs updated data $\tilde{\mathsf{M}}'$.

We define correctness and security via an experiment Expt. Given an array M (which defines the parameters n and B) and a sequence of memory accesses $\mathsf{seq} = ((\mathsf{op}_1, i_1, v_1), \ldots, (\mathsf{op}_m, i_m, v_m))$, experiment $\mathsf{Expt}(1^\kappa, \mathsf{M}, \mathsf{seq})$ first runs $(\mathsf{st}_0, \tilde{\mathsf{M}}_{1,0}, \tilde{\mathsf{M}}_{2,0}) \leftarrow \mathsf{ORAM.INIT}(1^\kappa, \mathsf{M})$. Then, for $t = 1$ to m:

- Run $\mathsf{ORAM.C}(\mathsf{st}_{t-1}, (\mathsf{op}_t, i_t, v_t))$, $\mathsf{ORAM.S}_1(\tilde{\mathsf{M}}_{1,t-1})$, and $\mathsf{ORAM.S}_2(\tilde{\mathsf{M}}_{2,t-1})$, allowing them to interact until they all terminate. Let (st_t, o_t) denote the output of $\mathsf{ORAM.C}$, and let $\tilde{\mathsf{M}}_{1,t}$ (resp., $\tilde{\mathsf{M}}_{2,t}$) denote the output of $\mathsf{ORAM.S}_1$ (resp., $\mathsf{ORAM.S}_2$).

We let view_1 (resp., view_2) denote the entire view of $\mathsf{ORAM.S}_1$ (resp., $\mathsf{ORAM.S}_2$) throughout the above experiment. We define the output of the experiment to be $(\mathsf{view}_1, \mathsf{view}_2, o_1, \ldots, o_m)$.

Correctness requires that for any κ, M, and any sequence of m memory accesses $\mathsf{seq} = ((\mathsf{op}_1, i_1, v_1), \ldots, (\mathsf{op}_m, i_m, v_m))$, if we compute

$$(\mathsf{view}_1, \mathsf{view}_2, o_1, \ldots, o_m) \leftarrow \mathsf{Expt}(1^\kappa, \mathsf{M}, \mathsf{seq})$$

then the sequence of results $o_1, \ldots, o_m$ is correct (relative to M and seq).

An ORAM scheme is secure if for any PPT adversary A the following is negligible in κ:

$$\left| \Pr \left[\begin{array}{c} (\mathsf{M}_0, \mathsf{seq}_0, \mathsf{M}_1, \mathsf{seq}_1) \leftarrow A(1^\kappa); b \leftarrow \{0,1\}; \\ (\mathsf{view}_1, \mathsf{view}_2, o_1, \ldots, o_m) \leftarrow \mathsf{Expt}(1^\kappa, \mathsf{M}_b, \mathsf{seq}_b) \end{array} : A(\mathsf{view}_1) = b \right] - \frac{1}{2} \right|$$

(and analogously for view_2), where $\mathsf{M}_0, \mathsf{M}_1$ have identical parameters n, B, and where $\mathsf{seq}_0, \mathsf{seq}_1$ have the same length. As usual, this notion of security assumes the servers are semi-honest.

7.1.3 Oblivious Reading/Writing

We define schemes that support (only) oblivious reads or (only) oblivious writes. Definitions can be obtained by appropriately specializing the ORAM definition given above, but because the schemes we construct for oblivious read/write use only one round of interaction, we can simplify the definitions somewhat. We refer to a scheme supporting oblivious reads as a *private information retrieval* (PIR) scheme, and a scheme supporting oblivious writes as a *private information writing* (PIW) scheme (since we do not require sub-linear computation for the memory accesses).

Oblivious Reads. A two-server, one-round PIR scheme is defined by algorithms $\mathsf{PIR.INIT}$, $\mathsf{PIR.C}$, $\mathsf{PIR.C'}$, and $\mathsf{PIR.S}$ with the following syntax:

- $\mathsf{PIR.INIT}$ takes as input 1^κ and elements $\mathsf{M}[1], \ldots, \mathsf{M}[n] \in \{0,1\}^B$. It outputs state st and data M^r to be stored at each server.
- $\mathsf{PIR.C}$ takes as input st and an index i. It outputs a pair of queries q_1, q_2.
- $\mathsf{PIR.S}$ takes as input data M^r and a query q. It outputs a response r.
- $\mathsf{PIR.C'}$ takes as input state st, i, and responses r_1, r_2. It outputs a value o.

Correctness requires that for any κ, any M, any values $\mathsf{st}, \mathsf{M}^\mathsf{r}$ output by $\mathsf{PIR.INIT}(1^\kappa, \mathsf{M})$, any i, any $\tilde{\mathsf{st}}, q_1, q_2$ output by $\mathsf{PIR.C}(\mathsf{st}, i)$, and any r_1 output by $\mathsf{PIR.S}(\mathsf{M}^\mathsf{r}, q_1)$ and r_2 output by $\mathsf{PIR.S}(\mathsf{M}^\mathsf{r}, q_2)$, it holds that

$$\mathsf{PIR.C'}(\mathsf{st}, i, r_1, r_2) = \mathsf{M}[i].$$

A scheme is secure if for any PPT adversary A the following is negligible in κ:

$$\left| \Pr \left[\begin{array}{l} (\mathsf{M}_0, i_0, \mathsf{M}_1, i_1) \leftarrow A(1^\kappa); b \leftarrow \{0,1\}; \\ \quad (\mathsf{st}, \mathsf{M}^\mathsf{r}) \leftarrow \mathsf{PIR.INIT}(1^\kappa, \mathsf{M}_b); \quad : A(\mathsf{M}^\mathsf{r}, q_1) = b \\ \quad (\tilde{\mathsf{st}}, q_1, q_2) \leftarrow \mathsf{PIR.C}(\mathsf{st}, i_b) \end{array} \right] - \frac{1}{2} \right|$$

(and analogously for $\mathsf{M}^\mathsf{r}, q_2$), where $\mathsf{M}_0, \mathsf{M}_1$ have identical parameters n, B. This definition ensures privacy of the data as well as obliviousness of the indexes being accessed.

Oblivious Writes. Similarly, a two-server, one-round PIW scheme consists of algorithms $\mathsf{PIW.INIT}, \mathsf{PIW.C}$, and $\mathsf{PIW.S}$ with the following syntax:

- $\mathsf{PIW.INIT}$ takes as input 1^κ and elements $\mathsf{M}[1], \ldots, \mathsf{M}[n] \in \{0,1\}^B$. It outputs data $\mathsf{M}_1^\mathsf{w}, \mathsf{M}_2^\mathsf{w}$ to be stored at the servers.
- $\mathsf{PIW.C}$ takes as input (i, v_old, v) with $i \in \{1, \ldots, n\}$ and $v_\mathsf{old}, v \in \{0,1\}^B$ (intuitively, v_old is the current value stored at address i, and v is the new value to be written to that address). It outputs a pair of queries q_1, q_2.
- $\mathsf{PIW.S}$ takes as input data M^w and a query q. It outputs new data $\tilde{\mathsf{M}}^\mathsf{w}$.

We also require an algorithm $\mathsf{PIW.Read}$ that is not part of the scheme, but is used to define correctness. This algorithm takes as input $\mathsf{M}_1^\mathsf{w}, \mathsf{M}_2^\mathsf{w}$ and an index i, and outputs a value $o \in \{0,1\}^B$.

We define correctness and security via an experiment $\mathsf{Expt\text{-}W}$. Given an array M and $\mathsf{seq} = ((i_1, v_1), \ldots, (i_m, v_m))$, experiment $\mathsf{Expt\text{-}W}(1^\kappa, \mathsf{M}, \mathsf{seq})$ first runs $(\mathsf{M}_{1,0}^\mathsf{w}, \mathsf{M}_{2,0}^\mathsf{w}) \leftarrow \mathsf{PIW.INIT}(1^\kappa, \mathsf{M})$ and then, for $t = 1$ to m, does:

- Let $v_{\mathsf{old},t}$ be the logical value currently stored at address i_t, i.e., it is equal to the last value written to i_t, or $\mathsf{M}[i_t]$ if $i_t \notin \{i_1, \ldots, i_{t-1}\}$.

- Compute $(q_{1,t}, q_{2,t}) \leftarrow \mathsf{PIW.C}(\mathsf{st}, (i_t, v_{\mathsf{old},t}, v_t))$.
- Compute $\mathsf{M}^{\mathsf{w}}_{1,t} \leftarrow \mathsf{PIW.S}(\mathsf{M}^{\mathsf{w}}_{1,t-1}, q_{1,t})$ and $\mathsf{M}^{\mathsf{w}}_{2,t} \leftarrow \mathsf{PIW.S}(\mathsf{M}^{\mathsf{w}}_{2,t-1}, q_{2,t})$.

We let view_1 (resp., view_2) denote $\mathsf{M}^{\mathsf{w}}_{1,0}, q_{1,1}, \ldots, q_{1,t}$ (resp., $\mathsf{M}^{\mathsf{w}}_{2,0}, q_{2,1}, \ldots, q_{2,t}$), and define the output of the experiment to be $(\mathsf{view}_1, \mathsf{view}_2, \mathsf{M}^{\mathsf{w}}_{1,m}, \mathsf{M}^{\mathsf{w}}_{2,m})$.

Correctness requires that for any κ, M, any sequence $\mathsf{seq} = ((i_1, v_1), \ldots, (i_m, v_m))$, and any i, if we compute

$$(\mathsf{view}_1, \mathsf{view}_2, \mathsf{M}^{\mathsf{w}}_{1,m}, \mathsf{M}^{\mathsf{w}}_{2,m}) \leftarrow \mathsf{Expt\text{-}W}(1^\kappa, \mathsf{M}, \mathsf{seq}),$$

then $\mathsf{PIW.Read}(\mathsf{M}^{\mathsf{w}}_{1,m}, \mathsf{M}^{\mathsf{w}}_{2,m}, i)$ is equal to the last value written to address i (or $\mathsf{M}[i]$ if $i \notin \{i_1, \ldots, i_m\}$).

A scheme is secure if for any PPT adversary A the following is negligible in κ:

$$\left| \Pr \left[\begin{array}{c} (\mathsf{M}_0, \mathsf{seq}_0, \mathsf{M}_1, \mathsf{seq}_1) \leftarrow A(1^\kappa); b \leftarrow \{0,1\}; \\ (\mathsf{view}_1, \mathsf{view}_2, \mathsf{M}^{\mathsf{w}}_{1,m}, \mathsf{M}^{\mathsf{w}}_{2,m}) \leftarrow \mathsf{Expt\text{-}W}(1^\kappa, \mathsf{M}_b, \mathsf{seq}_b) \end{array} : A(\mathsf{view}_1) = b \right] - \frac{1}{2} \right|$$

(and analogously for view_2), where $\mathsf{M}_0, \mathsf{M}_1$ have identical parameters n, B and $\mathsf{seq}_0, \mathsf{seq}_1$ have the same length. This definition ensures privacy of the data and the updates, in addition to obliviousness of the indexes being updated.

7.1.4 Distributed Point Functions

We recall the notion of a *distributed point function* (DPF) as introduced by [10]. Fix some parameters n and ℓ. For $i \in \{1, \ldots, n\}$ and $v \in \{0,1\}^\ell$, we define the *point function*

$\mathsf{PF}_{i,v} : \{1,\ldots,n\} \to \{0,1\}^\ell$ as follows:

$$\mathsf{PF}_{i,v}(x) = \begin{cases} v & \text{if } x = i \\ 0^\ell & \text{otherwise.} \end{cases}$$

A distributed point function can be viewed as providing a secret sharing of a point function.

Definition 7.1. *A distributed point function consists of algorithms* $\mathsf{KG}, \mathsf{Eval}$ *with following functionality:*

- KG *takes as input parameters* 1^κ *and* n*, along with* $i \in \{1,\ldots,n\}$ *and* $v \in \{0,1\}^\ell$*, and outputs a pair of keys* k_1, k_2*.*
- Eval *is a deterministic algorithm that takes as input a key* k *and an index* $i \in \{1,\ldots,n\}$*, and outputs a string* $\tilde{v} \in \{0,1\}^\ell$*.*

Correctness requires that for any κ, n, ℓ*, any* $(i,v) \in \{1,\ldots,n\} \times \{0,1\}^\ell$*, any* k_1, k_2 *output by* $\mathsf{KG}(1^\kappa, n, i, v)$*, and any* $x \in \{1,\ldots,n\}$*:*

$$\mathsf{Eval}(k_1, x) \oplus \mathsf{Eval}(k_2, x) = \mathsf{PF}_{i,v}(x).$$

Security requires that neither k_1 nor k_2 leak information about i or v. Formally, for any PPT adversary A the following is negligible in κ (for k_1 below, and analogously for k_2):

$$\left| \Pr\left[\begin{array}{c} (i_0, v_0, i_1, v_1) \leftarrow A(1^\kappa); b \leftarrow \{0,1\}; \\ (k_1, k_2) \leftarrow \mathsf{KG}(1^\kappa, n, i_b, v_b) \end{array} : A(k_1) = b \right] - \frac{1}{2} \right|$$

Securely Computing a DPF. In several of our protocols we will use the algorithms of the distributed point function in the context of secure computation. While our constructions

assume general distributed point function schemes, for our efficiency estimates we will use the DPF scheme of Boyle, Ishai and Gilboa [10] as well as the optimizations in the context of two party computation introduced by Doerner and Shelat [15]. This DPF construction uses an extension of the GGM pseudorandom function, which in addition to the PRF key which is the initial seed for the first PRG, provides also a set of masking values for each of the $\log(n)$ evaluation levels. These key parts used for masking provide the capability to modify the PRF value at one specific input and fix it to be a desired output. The computation of the masking values depends on the PRG evaluations related to nodes on the evaluation path to the specific input point. Thus, a generic two party computation for the DPF key generation will require secure evaluation of the PRGs. However, the CPRG optimization of [15] shows a two party computation protocol that requires secure computation only for an xor operation per level while moving all PRG evaluations as local computation to each party. As a result this protocol requires only $O(\kappa \log(n))$ communication. Since this construction increases the local computation that each party does, the authors further observe that this computation can be reused during the evaluation of the DPF.

7.1.5 Labeled Private-Key Encryption

In our construction we use a *labeled* private-key encryption scheme and require it to satisfy a weaker notion of security than usual. For completeness, we include the definitions here.

Definition 7.2. *Fix some B. A labeled private-key encryption scheme consists of two algorithms* $(\mathsf{Enc}, \mathsf{Dec})$ *with the following syntax:*

- Enc *takes as input a key K, a label i, and a message $m \in \{0,1\}^B$. It outputs a ciphertext c. We denote this by $c \leftarrow \mathsf{Enc}_K(i, m)$.*
- Dec *takes as input a key K, a label i, and a ciphertext c. It outputs a message m. We denote this by $m := \mathsf{Dec}_K(i, c)$.*

For all K, i, m, if c is output by $\mathsf{Enc}_K(i, m)$ then $\mathsf{Dec}_K(i, c) = m$.

For our application, we require security only as long as the same label is never used to encrypt more than one message. Let $\mathsf{Enc}_K^b(i, m_0, m_1) \stackrel{\text{def}}{=} \mathsf{Enc}_K(i, m_b)$.

Definition 7.3. *A labeled private-key encryption scheme is secure if for all* PPT *adversaries A the following is negligible in κ:*

$$\left| \Pr[K \leftarrow \{0,1\}^\kappa; b \leftarrow \{0,1\} : A^{\mathsf{Enc}_K^b(\cdot,\cdot,\cdot)}(1^\kappa) = b] - \frac{1}{2} \right|,$$

where A may never re-use a label in a query to Enc_K^b.

Note that a scheme satisfying the above definition may be deterministic. In particular, we may take the simple scheme based on a pseudorandom function F in which $\mathsf{Enc}_K(i, m) = F_K(i) \oplus m$.

7.2 Two- and Three-Server DPF-ORAM

In this section we describe a construction of a two-server ORAM scheme that we call DPF-ORAM. Our construction is based on specific constructions of PIR and PIW schemes that are, in turn, based on a distributed point function. Although those schemes have already appeared in prior work, we include them here because we rely on specific features of those schemes.

7.2.1 PIR/PIW Schemes

In what follows, let $(\mathsf{KG}, \mathsf{Eval})$ be a distributed point function.

Oblivious Reads. In the PIR scheme we use, both servers store identical copies of an encrypted version M^{r} of the array M. When the client wants to retrieve the ith element of

the array, it computes $(k_1, k_2) \leftarrow \mathsf{KG}(1^\kappa, n, i, 1)$ and sends k_1 and k_2 as queries to the first and the second server, respectively. The servers compute their responses as

$$r_1 := \bigoplus_{j=1}^{n} \mathsf{Eval}(k_1, j) \cdot \mathsf{M}^{\mathsf{r}}[j] \quad \text{and} \quad r_2 := \bigoplus_{j=1}^{n} \mathsf{Eval}(k_2, j) \cdot \mathsf{M}^{\mathsf{r}}[j].$$

Upon receiving the responses r_1, r_2, the client computes $\mathsf{M}^{\mathsf{r}}[i] := r_1 \oplus r_2$, which it can then decrypt to recover $\mathsf{M}[i]$.

If we let $(\mathsf{Enc}, \mathsf{Dec})$ denote a labeled private-key encryption scheme, then we can formally describe the PIR scheme as follows:

- $\mathsf{PIR.INIT}(1^\kappa, \mathsf{M})$ chooses uniform K, and then sets $\mathsf{M}^{\mathsf{r}}[i] := \mathsf{Enc}_K(i, \mathsf{M}[i])$ for all i. It outputs $\mathsf{M}^{\mathsf{r}}_1 := \mathsf{M}^{\mathsf{r}}_2 := \mathsf{M}^{\mathsf{r}}$ and $\mathsf{st} := (K, n)$.
- $\mathsf{PIR.C}((K, n), i)$ computes $(k_1, k_2) \leftarrow \mathsf{KG}(1^\kappa, n, i, 1)$ and outputs k_1 and k_2 as the queries for the two servers.
- $\mathsf{PIR.S}(\mathsf{M}^{\mathsf{r}}, k)$ computes $r := \bigoplus_{j=1}^{n} \mathsf{Eval}(k, j) \cdot \mathsf{M}^{\mathsf{r}}[j]$.
- $\mathsf{PIR.C'}((K, n), i, r_1, r_2)$ outputs $\mathsf{Dec}_K(i, r_1 \oplus r_2)$.

Correctness and security follow immediately from the properties of the distributed point function and the encryption scheme.

We remark that the above scheme has the useful property that it can support *appends*. Specifically, say the client holds state (k, n) and the servers each hold an encrypted array $\mathsf{M}^{\mathsf{r}} = (\mathsf{M}^{\mathsf{r}}[1], ..., \mathsf{M}^{\mathsf{r}}[n])$ corresponding to the plaintext array M. Then the client can append a value v to M (i.e., set $\mathsf{M}[n+1] := v$) by sending $c := \mathsf{Enc}_K(n+1, v)$ to both servers who then set $\mathsf{M}^{\mathsf{r}}[n+1] := c$.

Oblivious Writes. In the PIW scheme we use, the servers store secret shares of the client's array M. That is, the servers hold arrays $\mathsf{M}_1^{\mathsf{w}}, \mathsf{M}_2^{\mathsf{w}}$, respectively, such that for all i we have $\mathsf{M}_1^{\mathsf{w}}[i] \oplus \mathsf{M}_2^{\mathsf{w}}[i] = \mathsf{M}[i]$. To write a new value v to address i, where v_{old} is the value currently stored at that address, the client begins by computing $(k_1, k_2) \leftarrow \mathsf{KG}(1^\kappa, n, i, v \oplus v_{\mathsf{old}})$ and sending k_1 and k_2 as its queries to the first and second servers, respectively. The two servers then update their arrays by computing

$$\tilde{\mathsf{M}}_1^{\mathsf{w}}[j] := \mathsf{M}_1^{\mathsf{w}}[j] \oplus \mathsf{Eval}(k_1, j) \quad \text{and} \quad \tilde{\mathsf{M}}_2^{\mathsf{w}}[j] := \mathsf{M}_2^{\mathsf{w}}[j] \oplus \mathsf{Eval}(k_2, j)$$

for all j. Note that for $j \neq i$ we have

$$\begin{aligned} \tilde{\mathsf{M}}_1^{\mathsf{w}}[j] \oplus \tilde{\mathsf{M}}_2^{\mathsf{w}}[j] &= \mathsf{M}_1^{\mathsf{w}}[j] \oplus \mathsf{Eval}(k_1, j) \oplus \mathsf{M}_2^{\mathsf{w}}[j] \oplus \mathsf{Eval}(k_2, j) \\ &= \mathsf{M}_1^{\mathsf{w}}[j] \oplus \mathsf{M}_2^{\mathsf{w}}[j] \end{aligned}$$

whereas

$$\begin{aligned} \tilde{\mathsf{M}}_1^{\mathsf{w}}[i] \oplus \tilde{\mathsf{M}}_2^{\mathsf{w}}[i] &= \mathsf{M}_1^{\mathsf{w}}[i] \oplus \mathsf{Eval}(k_1, i) \oplus \mathsf{M}_1^{\mathsf{w}}[i] \oplus \mathsf{Eval}(k_2, i) \\ &= v_{\mathsf{old}} \oplus (v \oplus v_{\mathsf{old}}) \;=\; v, \end{aligned}$$

so correctness holds. Security follows immediately from the security of the distributed point function.

The formal definition of the PIW scheme (including $\mathsf{PIW.Read}$) is as follows:

- $\mathsf{PIW.INIT}(1^\kappa, \mathsf{M})$ does: For $i = 1, \ldots, n$, choose uniform $\mathsf{M}_1^{\mathsf{w}}[i] \in \{0,1\}^B$ and set $\mathsf{M}_2^{\mathsf{w}}[i] := \mathsf{M}_1^{\mathsf{w}}[i] \oplus \mathsf{M}[i]$. Output $\mathsf{M}_1^{\mathsf{w}}, \mathsf{M}_2^{\mathsf{w}}$.

- $\mathsf{PIW.C}(i, v_{\mathsf{old}}, v)$ computes $(k_1, k_2) \leftarrow \mathsf{KG}(1^\kappa, n, i, v \oplus v_{\mathsf{old}})$ and outputs k_1 and k_2 as the queries for the two servers.
- $\mathsf{PIW.S}(\mathsf{M}^{\mathsf{w}}, k)$ sets $\tilde{\mathsf{M}}^{\mathsf{w}}[j] := \mathsf{M}^{\mathsf{w}}[j] \oplus \mathsf{Eval}(k, j)$ for all j.
- $\mathsf{PIW.Read}(\mathsf{M}_1^{\mathsf{w}}, \mathsf{M}_2^{\mathsf{w}}, i)$ outputs $\mathsf{M}_1^{\mathsf{w}}[i] \oplus \mathsf{M}_2^{\mathsf{w}}[i]$.

7.2.2 Two-Server DPF-ORAM

We now describe our construction of a two-server ORAM scheme based on the PIR and PIW schemes just introduced. We first describe a base scheme that introduces all the main ideas, but in which the storage of the client is $n \log(n)$ bits. We can then use logarithmically many levels of recursion (in the usual way) to obtain a scheme in which the client's storage is constant.

At a high level, in the base scheme the servers maintain three arrays: a *main array* supporting oblivious reads, a *stash* supporting oblivious reads and appends, and an *auxiliary array* supporting oblivious writes. At any point in time, there is an entry in the auxiliary array that corresponds to the current value $\mathsf{M}[i]$ for each address i of the client's array M. In addition, there are entries for $\mathsf{M}[i]$ in the main array and possibly the stash such that

- If there is an entry for $\mathsf{M}[i]$ in the stash, then the rightmost (i.e., newest) entry is the current value of $\mathsf{M}[i]$.
- If there is no entry for $\mathsf{M}[i]$ in the stash, then the entry for $\mathsf{M}[i]$ in the main array is the current value of $\mathsf{M}[i]$.

Roughly, a read of address i is done by reading the entry for $\mathsf{M}[i]$ from the main array and, if present, the stash. (In addition, dummy operations are performed on the auxiliary array and the stash to obscure the fact that a read is being done.) A write of a value v to address i

is done by first reading the value v_{old} currently stored in $\mathsf{M}[i]$, and then writing the new value v to the auxiliary array as well as appending it to the stash. When appending a value to the stash, the client also maintains local state keeping track of that fact (this will be described in further detail below).

As described, the stash and the client's local state can grow without bound. To prevent this, we have the two servers perform a *conversion* step after every n accesses. In this step, the servers delete the stash and the main array, and then create a new main array that corresponds to a copy of the plaintext data currently stored in the auxiliary array. This is easy to do with the PIR and PIW schemes defined in the previous section. Recall that for the PIW scheme, the servers store $\mathsf{M}_1^{\mathsf{w}}, \mathsf{M}_2^{\mathsf{w}}$ with $\mathsf{M}_1^{\mathsf{w}}[i] \oplus \mathsf{M}_2^{\mathsf{w}}[i] = \mathsf{M}[i]$ for all i, whereas for the PIR scheme the servers both store M^{r} with $\mathsf{M}^{\mathsf{r}}[i] = \mathsf{Enc}_K(i, \mathsf{M}[i])$ for an encryption key K held by the client. If we implement encryption as

$$\mathsf{Enc}_{(K_1,K_2)}(i, x) = x \oplus F_{K_1}(i) \oplus F_{K_2}(i) \tag{7.1}$$

where F is a pseudorandom function, then conversion can be done as follows:

- Servers 1 and 2 choose K_1, K_2, respectively, and send them to the client.
- For all i, server 1 sends $U_1[i] := \mathsf{M}_1^{\mathsf{w}}[i] \oplus F_{K_1}(i)$ to server 2.
- For all i, server 2 sends $U_2[i] := \mathsf{M}_2^{\mathsf{w}}[i] \oplus F_{K_2}(i)$ to server 1.
- For all i, each server locally computes $\mathsf{M}^{\mathsf{r}}[i] := U_1[i] \oplus U_2[i]$.

This conversion step requires $O(n \cdot B)$ bits of communication, but since it is run only every n steps the amortized cost it introduces is only $O(B)$ bits.

Let $\mathsf{PIR}, \mathsf{PIW}$ refer to the schemes described in the previous section, where PIR instantiates encryption Enc with the two-key scheme described above. Then the base ORAM scheme is defined by the following algorithms:

Initialization. $\mathsf{ORAM.INIT}(1^\kappa, \mathsf{M})$ initializes a PIR array M^{r} for the main memory, a PIR array S^{r} for the stash, and PIW arrays $\mathsf{M}_1^{\mathsf{w}}, \mathsf{M}_2^{\mathsf{w}}$ for the main memory, as follows:

1. $((K, n), \mathsf{M}^{\mathsf{r}}) \leftarrow \mathsf{PIR.INIT}(1^\kappa, \mathsf{M})$, where $K = (K_1, K_1)$.
2. $((K', 1), S^{\mathsf{r}}) \leftarrow \mathsf{PIR.INIT}(1^\kappa, 0^B)$, where $K' = (K'_1, K'_2)$.
3. $(\mathsf{M}_1^{\mathsf{w}}, \mathsf{M}_2^{\mathsf{w}}) \leftarrow \mathsf{PIW.INIT}(1^\kappa, \mathsf{M})$.

The client also initializes an array P with $P[1] = \cdots = P[n] := 0^{\log(n)}$, and sets $n' := 1$. It outputs state $\mathsf{st} := (K, n, K', n', P)$, and data $(\mathsf{M}^{\mathsf{r}}, S^{\mathsf{r}}, \mathsf{M}_1^{\mathsf{w}})$ for server 1, and $(\mathsf{M}^{\mathsf{r}}, S^{\mathsf{r}}, \mathsf{M}_2^{\mathsf{w}})$ for server 2.

By way of intuition, P is a *position map* where $P[i]$ contains the location in the stash of the current value of $\mathsf{M}[i]$ if address i has been accessed before. (If address i has not been accessed before, then $P[i] = 0^{\log(n)}$.) Counter n' will keep track of the length of the stash.

Access. Let the state of the client be (K, n, K', n', P) and the data stored at the servers be $(\mathsf{M}^{\mathsf{r}}, S^{\mathsf{r}}, \mathsf{M}_1^{\mathsf{w}})$ and $(\mathsf{M}^{\mathsf{r}}, S^{\mathsf{r}}, \mathsf{M}_2^{\mathsf{w}})$, respectively. When the client wants to perform memory access (op, i, v), the parties interact as follows:

1. The client computes PIR queries to retrieve $\mathsf{M}^{\mathsf{r}}[i]$ and $S^{\mathsf{r}}[p]$ for $p = P[i]$:
   (a) Compute $(q_1, q_2) \leftarrow \mathsf{PIR.C}((K, n), i)$.
   (b) Compute $(q'_1, q'_2) \leftarrow \mathsf{PIR.C}((K', n'), P[i])$.
   (c) Send q_1, q'_1 to server 1, and q_2, q'_2 to server 2.

2. Servers respond to both PIR queries:

   (a) Server 1 sends $r_1 := \mathsf{PIR.S}(\mathsf{M}^{\mathsf{r}}, q_1)$ and $r'_1 := \mathsf{PIR.S}(S^{\mathsf{r}}, q'_1)$ to the client.

   (b) Server 2 sends $r_2 := \mathsf{PIR.S}(\mathsf{M}^{\mathsf{r}}, q_2)$ and $r'_2 := \mathsf{PIR.S}(S^{\mathsf{r}}, q'_2)$ to the client.

3. The client reconstructs $\mathsf{M}[i]$ and updates P, S^{r}, and M^{r}, as follows:

   (a) Compute $o := \mathsf{PIR.C'}((K, n), i, r_1, r_2)$.

   (o *is the original value of* $\mathsf{M}[i]$ *held in read-only memory.*)

   (b) Compute $o' := \mathsf{PIR.C'}((K', n'), P[i], r'_1, r'_2)$.

   (*If* $P[i] \neq 0^{\log(n)}$ *then* o' *is the most recent copy of* $\mathsf{M}[i]$ *held in stash* S.)

   (c) If $P[i] = 0^{\log(n)}$ then set $v_{\mathsf{old}} := o$, else set $v_{\mathsf{old}} := o'$.

   (d) **Output** v_{old}. If $\mathsf{op} = \mathsf{read}$ then set $v := v_{\mathsf{old}}$.

   (e) Compute $c := \mathsf{Enc}_{K'}(n', v)$ and set $P[i] := n'$. Increment n'.

   (f) Compute $(q_1, q_2) \leftarrow \mathsf{PIW.C}(i, v_{\mathsf{old}}, v)$.

   (g) Send q_1, c to server 1 and q_2, c to server 2.

4. The servers append c to S^{r} and update their M^{w} shares:

   (a) Server 1 computes the updated array $\mathsf{M}^{\mathsf{w}}_1 := \mathsf{PIW.S}(\mathsf{M}^{\mathsf{w}}_1, q_1)$.

   (b) Server 2 computes the updated array $\mathsf{M}^{\mathsf{w}}_2 := \mathsf{PIW.S}(\mathsf{M}^{\mathsf{w}}_2, q_2)$.

Periodic refresh. After every n memory accesses, the servers carry out a conversion step as described previously. In detail, let the state of the client be (K, n, K', n', P) and the data stored at the servers be $(\mathsf{M}^{\mathsf{r}}, S^{\mathsf{r}}, \mathsf{M}^{\mathsf{w}}_1)$ and $(\mathsf{M}^{\mathsf{r}}, S^{\mathsf{r}}, \mathsf{M}^{\mathsf{w}}_2)$, respectively. Then:

- Server 1 chooses uniform $K^*_1 \in \{0,1\}^\kappa$ and sends it to the client.
- Server 2 chooses uniform $K^*_2 \in \{0,1\}^\kappa$ and sends it to the client.

- For all i, server 1 sends $U_1[i] := \mathsf{M}_1^{\mathsf{w}}[i] \oplus F_{K_1^*}(i)$ to server 2.
- For all i, server 2 sends $U_2[i] := \mathsf{M}_2^{\mathsf{w}}[i] \oplus F_{K_2^*}(i)$ to server 1.
- For all i, each server locally computes $\mathsf{M}^{\mathsf{r}}[i] := U_1[i] \oplus U_2[i]$.
- Servers set $S^{\mathsf{r}} := 0^B$. The client sets $K := (K_1^*, K_2^*)$, chooses fresh key $K' = (K_1', K_2')$, re-initializes the array P to 0, and sets $n' := 1$.

Parameters. Each keys in the DPF of [15] have length $B+\kappa\log(n)$. Thus, an oblivious read using the PIR construction from the previous section requires a total of $4B+2\kappa\log(n)$ bits of communication (with $B+\kappa\log(n)$ bits sent to each server, and B bits sent from each server back to the client), and an oblivious write using the PIW construction requires $2B$ bits of communication for client to send $\triangle v := v \oplus v_{\mathsf{old}}$ to servers. Consider a sequence of n memory accesses, that for each access, our ORAM scheme performs one oblivious read on an array of length n, one oblivious read on an array of length $n' \leq n$, one oblivious write to an array of length n, and an append. In addition, the refresh step uses $2\kappa+2nB$ bits of communication. The total, amortized communication complexity is thus $12B + 4\kappa\log(n) + 2\kappa/n$ bits.

The storage per server is $3nB$ bits, and the client storage is dominated by the array P that requires $n\log(n)$ bits.

A recursive ORAM scheme. As in prior work [41, 42, 44], we can reduce the client's storage by recursively using our base ORAM scheme to store P, viewing P as an array of $n/2^\tau$ elements each of length $2^\tau\log(n)$. Doing so reduces the client storage by a factor of τ, and thus recursing $O(\log(n))$ times with any constant τ results in $O(1)$ client storage. Since the recursion is independent of B (because the client's storage in the base scheme is $n\log(n)$ bits, regardless of B), the resulting scheme has total (amortized) communication complexity $O(B) + O(\kappa\log^2 n)$.

7.2.3 Three-Server DPF-ORAM

Our two-server ORAM construction above makes a black-box use of the PIR and PIW primitives, so if these functionalities are implemented in a multi-server setting the same protocol becomes a multi-server ORAM. Note that the DPF-based implementation of PIR uses encryption because, unlike PIW, it requires that both servers store identical databases and thus we need to use encryption to provide privacy. This encryption layer creates a significant overhead if this two-server ORAM is converted to 2PC-ORAM, i.e., a two party secure computation of the ORAM functionality. However, both PIR and PIW can be implemented without the encryption overhead in the *three-server* setting, which leads to reduced cost in the 3PC-ORAM protocol based on the same ORAM construction, described in Section 7.3.

If v_1, v_2, v_3 are 3-out-of-3 shares of a value v (i.e., $v = v_1 \oplus v_2 \oplus v_3$), then we let $v_{2,3} = (v_2, v_3)$, $v_{1,3} = (v_1, v_3)$, and $v_{1,2} = (v_1, v_2)$. Note that $v_{2,3}, v_{1,3}$, and $v_{1,2}$ are 2-out-of-3 shares of v: that is, v remains secret given any one of those shares, but can be reconstructed given any two of them.

The idea for a three-server PIR is to share the database in a 2-out-of-3 way, i.e., share M three-way and then give each of the three shares to a different pair of servers. This way each server pair holds one identical share, allowing them to service a DPF-based two-server PIR on that share. Xor-ing the values retrieved by these three two-server PIR instances reconstructs the retrieved record $\mathsf{M}[i]$. This 3-server implementation of PIR increases the DPF overhead because the client runs *three* DPF key generation instances, one for each share of M, and each server executes *two* DPF-based reads, one for each share of M which it holds. However, DPF key generation can be precomputed off-line, and increasing local computation from DPF evaluation by a factor of 2 is a good trade-off for removing garbled circuit over AES, except for very large n.

Below we show the resulting three-server PIR without using key K as input:

- $\mathsf{PIR.INIT}(1^\kappa, \mathsf{M})$: For $i = 1, \ldots, n$, choose two uniform shares $\mathsf{M}_1[i], \mathsf{M}_2[i] \in \{0,1\}^B$ and set $\mathsf{M}_3[i] := \mathsf{M}_1[i] \oplus \mathsf{M}_2[i] \oplus \mathsf{M}[i]$.

  Output $(\mathsf{M}^{\mathsf{r}}_1, \mathsf{M}^{\mathsf{r}}_2, \mathsf{M}^{\mathsf{r}}_3)$ s.t. $\mathsf{M}^{\mathsf{r}}_1 = \mathsf{M}_{2,3} = (\mathsf{M}_2, \mathsf{M}_3)$, $\mathsf{M}^{\mathsf{r}}_2 = \mathsf{M}_{1,3} = (\mathsf{M}_1, \mathsf{M}_3)$, $\mathsf{M}^{\mathsf{r}}_3 = \mathsf{M}_{1,2} = (\mathsf{M}_1, \mathsf{M}_2)$.

- $\mathsf{PIR.C}(n, i)$: For $j = 1, 2, 3$, do $(k_{j,(j-1 \bmod 3)}, k_{j,(j+1 \bmod 3)}) \leftarrow \mathsf{KG}(1^\kappa, n, i, 1)$. Output $q_1 = (k_{2,1}, k_{3,1})$, $q_2 = (k_{1,2}, k_{3,2})$ and $q_3 = (k_{1,3}, k_{2,3})$ as the queries for the three servers.

- $\mathsf{PIR.S}(\mathsf{M}^{\mathsf{r}}, q)$: Parse $q = (k_a, k_b)$ and $\mathsf{M}^{\mathsf{r}} = (\mathsf{M}_a, \mathsf{M}_b)$, and return $r_a \oplus r_b$ for $r_a := \bigoplus_{j=1}^n \mathsf{Eval}(k_a, j) \cdot \mathsf{M}_a[j]$ and $r_b := \bigoplus_{j=1}^n \mathsf{Eval}(k_b, j) \cdot \mathsf{M}_b[j]$.

- $\mathsf{PIR.C'}(n, i, (r_1, r_2, r_3))$: Output $r_1 \oplus r_2 \oplus r_3$.

For completeness we include also three-server PIW. Procedures INIT and Read generalize the two-server PIW in a straightforward way, using three-way xor-sharing instead, while procedures C and S are actually identical because in the three-server PIW only two servers need to update their shares.

- $\mathsf{PIW.INIT}(1^\kappa, \mathsf{M})$: For $i = 1, \ldots, n$, choose uniform $\mathsf{M}^{\mathsf{w}}_1[i], \mathsf{M}^{\mathsf{w}}_2[i] \in \{0,1\}^B$ and set $\mathsf{M}^{\mathsf{w}}_3[i] := \mathsf{M}^{\mathsf{w}}_1[i] \oplus \mathsf{M}^{\mathsf{w}}_2[i] \oplus \mathsf{M}[i]$. Output $(\mathsf{M}^{\mathsf{w}}_1, \mathsf{M}^{\mathsf{w}}_2, \mathsf{M}^{\mathsf{w}}_3)$.

- $\mathsf{PIW.C}(i, v_{\mathsf{old}}, v)$ computes $(k_1, k_2) \leftarrow \mathsf{KG}(1^\kappa, n, i, v \oplus v_{\mathsf{old}})$ and outputs k_1 and k_2 as the queries for any chosen *two* servers, e.g., server 1 and server 2.

- $\mathsf{PIW.S}(\mathsf{M}^{\mathsf{w}}, k)$ modifies M^{w} by setting $\mathsf{M}^{\mathsf{w}}[j] := \mathsf{M}^{\mathsf{w}}[j] \oplus \mathsf{Eval}(k, j)$ for all j.

- $\mathsf{PIW.Read}(\mathsf{M}^{\mathsf{w}}_1, \mathsf{M}^{\mathsf{w}}_2, \mathsf{M}^{\mathsf{w}}_3, i)$ outputs $\mathsf{M}^{\mathsf{w}}_1[i] \oplus \mathsf{M}^{\mathsf{w}}_2[i] \oplus \mathsf{M}^{\mathsf{w}}_3[i]$.

7.3 3PC-DPF-ORAM

Our client-server ORAM of Section 7.2 can be converted to a Three Party Secure Computation ORAM (3PC-ORAM), using a generic conversion. Namely, we implement the client's code with a three-party computation between parties $\mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3$ while secret-sharing all client-held variables between them. We note that by the three-party secure computation (3PC) setting we mean the case of three parties with only *one* fault, i.e., the *honest majority* setting.

In the 3PC setting it is convenient to think of the *three-server* version of our client-server ORAM, discussed in Section 7.2.3, because in addition to secret-sharing the client each of the three parties in the 3PC-ORAM protocol will play a role of one of the servers of the three server ORAM. The point of starting from the three-server ORAM is that the PIR and PIW building blocks can be implemented in the three-server setting using only secret-sharing, without the layer of encryption (see Section 7.2.3). This allows 3PC-ORAM to use secret sharing reconstruction in the MPC(Enc) and MPC(Dec) steps which in the 2PC setting were implemented with GC(AES) sub-protocols, i.e., garbled-circuit over AES. Asymptotically, this reduces the $O(B \cdot \kappa)$ off-line and $O(\kappa(\kappa + B))$ on-line bandwidth terms to $O(B)$.

The second significant cost component of 2PC-ORAM are the 2PC(KG) sub-protocols used in PIR.C and PIW.C, which we implemented re-using the secure computation protocol for DPF key-generation of [15]. Below we show that in the 3PC setting PIR.C and PIW.C can be implemented using *plain* DPF key-generation, i.e., without the secure computation layer, which reduces the $O(\kappa^2 \log(n))$ term in off-line bandwidth to $O(\kappa \log(n))$.

After these modifications the remaining significant costs are in protocol steps 1c and 3d, where $p(i)$ is identified as δ-th record in block $P[i']$ retrieved from the lower-level ORAM, and a new value n' is written to the same position in $P[i']$. Using garbled circuits these steps take $O(\kappa * |P[i']|) = O(\kappa 2^\tau \log(n))$ bandwidth. Alternatively, we could reduce this to $O(2^\tau \log(n))$ by implementing these secret-shared reads and writes with $(1, m)$-OT's, for

$m = 2^\tau$. However, in our context it is easiest to implement these operations by re-using the same DPF-based techniques we use for PIR and PIW, which takes $O(\kappa\tau \log(n))$ offline and $O(2^\tau \log(n))$ online bandwidth.

Optimizations. We describe the last two of these cost-saving ideas in more detail below (the first cost-saving component follows immediately from the three-server ORAM description in Section 7.2.3).

Generating PIR/PIW Queries without Garbled Circuits. Consider the 2-out-of-3 sharing of the PIR database M^{r} and the lookup index i, introduced in Section 7.2.3: The database is shared as $\mathsf{M}^{\mathsf{r}} = \mathsf{M}^{\mathsf{r}}_1 \oplus \mathsf{M}^{\mathsf{r}}_2 \oplus \mathsf{M}^{\mathsf{r}}_3$ and the index as $i = i_1 \oplus i_2 \oplus i_3$. If we use $x_{i,j}$ as a short-cut for pair (x_i, x_j) then in the 2-out-of-3 sharing the shares the three parties of M^{r} and i are respectively $\mathsf{M}^{\mathsf{r}}_{2,3}, \mathsf{M}^{\mathsf{r}}_{1,3}, \mathsf{M}^{\mathsf{r}}_{1,2}$ and $i_{2,3}, i_{1,3}, i_{1,2}$.

We first show how P_1 and P_2 obtain a two-party xor-sharing of $\mathsf{M}^{\mathsf{r}}_3[i]$. Note that if P_1 and P_3 set $i^{1,2} := i_3$ and P_3 sets $i^3 := i_1 \oplus i_2$ then $(i^{1,2}, i^3)$ forms an xor-sharing of $i = i^{1,2} \oplus i^3$. Next, if P_3 runs $(k_1, k_2) \leftarrow \mathsf{KG}(1^\kappa, i^3, 1)$ and sends k_1 to P_1 and k_2 to P_2, then P_1 and P_2 can compute their shares of $\mathsf{M}^{\mathsf{r}}_3[i]$ as follows:

$$\mathsf{M}^{\mathsf{r}}_3[i]^1 = \oplus_{j=1}^n F(k_1, j \oplus i^{1,2}) \cdot \mathsf{M}^{\mathsf{r}}_3[j] \quad ; \quad \mathsf{M}^{\mathsf{r}}_3[i]^2 = \oplus_{j=1}^n F(k_2, j \oplus i^{1,2}) \cdot \mathsf{M}^{\mathsf{r}}_3[j]$$

Note that $\mathsf{PF}_{i^3,1}(j \oplus i^{1,2}) = \mathsf{PF}_{i^3 \oplus i^{1,2},1}(j) = 1$ iff $j = i^3 \oplus i^{1,2} = i$, which implies that $\mathsf{M}^{\mathsf{r}}_3[i]^1 \oplus \mathsf{M}^{\mathsf{r}}_3[i]^2 = \mathsf{M}^{\mathsf{r}}_3[i]$. We run three copies of this protocol in parallel, to form sharing of $\mathsf{M}^{\mathsf{r}}_3[i]$ held by $\mathsf{P}_1, \mathsf{P}_2$, sharing of $\mathsf{M}^{\mathsf{r}}_2[i]$ held by $\mathsf{P}_1, \mathsf{P}_3$, and sharing of $\mathsf{M}^{\mathsf{r}}_1[i]$ held by $\mathsf{P}_2, \mathsf{P}_3$. If the parties xor their shares of these sharings, the result is a three-party xor-sharing of $\mathsf{M}^{\mathsf{r}}[i]$.

The same idea works for generating the PIW queries. Recall that in 3-server PIW of Section 7.2.3 the PIW database M^{w} is secret-shared using a standard xor-sharing, $\mathsf{M}^{\mathsf{r}} = \mathsf{M}^{\mathsf{w}}_1 \oplus \mathsf{M}^{\mathsf{w}}_2 \oplus \mathsf{M}^{\mathsf{w}}_3$, where each P_t holds $\mathsf{M}^{\mathsf{w}}_t$. Assume that $\mathsf{PIW.C}$ inputs (i, v_{old}, v) are secret-shared s.t. i is shared

in the 2-out-of-3 sharing and $\Delta = v_{old} \oplus v$ is secret-shared with standard three-party xor sharing, i.e., each P_t holds Δ_t where $\Delta = \Delta_1 \oplus \Delta_2 \oplus \Delta_3$. The following protocol update shares $\mathsf{M}^{\mathsf{w}}_1, \mathsf{M}^{\mathsf{w}}_2$ held by $\mathsf{P}_1, \mathsf{P}_2$ s.t. $\mathsf{M}^{\mathsf{w}}[i]$ is xor'ed by Δ_3: Let $(i^{1,2}, i^3)$ be as above, and let P_3 run $(k_1, k_2) \leftarrow \mathsf{KG}(1^\kappa, i^3, \Delta_3)$ and sends k_1 to P_1 and k_2 to P_2. Let P_1 and P_2 then update $\mathsf{M}^{\mathsf{w}}_1[j]$ and $\mathsf{M}^{\mathsf{w}}_1[j]$ for all j as follows:

$$\mathsf{M}^{\mathsf{w}}_1[j] := \mathsf{M}^{\mathsf{w}}_1[j] \oplus F(k_1, j \oplus i^{1,2}) \quad ; \quad \mathsf{M}^{\mathsf{w}}_2[j] := \mathsf{M}^{\mathsf{w}}_2[j] \oplus F(k_2, j \oplus i^{1,2})$$

Since $\mathsf{PF}_{i^3,\Delta_3}(j \oplus i^{1,2}) = \mathsf{PF}_{i^3 \oplus i^{1,2},\Delta_3}(j) = \mathsf{PF}_{i,\Delta_3}(j)$, it follows that this transformation updates each $\mathsf{M}^{\mathsf{w}}[j]$ by xor-ing it with $\mathsf{PF}_{i,\Delta_3}(j)$, i.e., with Δ_3 if and only if $j = i$. If three instances of this protocol are run in parallel they update the sharing of M^{w} by xor-ing its entry at the i-th position by $\Delta_1 \oplus \Delta_2 \oplus \Delta_3 = \Delta$.

Implementing Array Read/Write without Garbled Circuits. As mentioned above, there are several ways we can implement reading $p(i)$ from the block of $m = 2^\tau$ pointers in $P[i']$, and writing a new value of $p(i)$ back into the same position in $P[i']$. Note that the general form of the functionality we are dealing here is a read and write operation to a secret-shared data, i.e., the read takes a secret-shared array $\langle \mathsf{M} \rangle$ of n records and a secret-shared index $\langle i \rangle$, and outputs a sharing $\langle \mathsf{M}[i] \rangle$ of $\mathsf{M}[i]$. The write operation takes additional input a sharing $\langle v \rangle$ of a new value to be written into $\mathsf{M}[i]$. In the case of writing $p(i)$ and and from $P[i']$ we have $\mathsf{M} = P[i']$, $m = 2^\tau$, $i = \delta$, $\mathsf{M}[i] = (P[i'])[\delta] = p(i)$, and $v = n'$. However, note that the above PIR and PIW constructions already implement exactly these secret-shared read and write functionalities, except that they assume different forms of secret-sharing of M: PIR assumes a 2-out-of-3 secret-sharing, while PIW assumes a standard 3-party xor-sharing. Transferring a 2-out-of-3 sharing to an xor-sharing is non-interactive, but the opposite direction takes $O(|\mathsf{M}|)$ bandwidth. However, here $|\mathsf{M}| = |P[i']| = 2^\tau \log(n)$, so regardless of the sharing used for P, a sequence of PIR read and PIW write on $P[i']$ will implement the read and write we

need, using $O(\log m \log(n)) = O(\tau \log(n))$ bandwidth offline for DPF key generation and $O(2^\tau \log(n))$ bandwidth online for re-sharing.

Efficiency. We summarize the communication costs for the different steps in the 3PC ORAM protocol, differentiating between online and offline communication, where offline is all which can be precomputed without the inputs.

- Three-party computation of DPF key generation for PIR and PIW on M^{r}, S^{r}, and M^{w}. As described above, in 3PC we do it with three standard DPF key generation procedures. The resulting bandwidth is $O(\kappa \log(n))$ *offline* and $O(\log(n) + B)$ *online*.
- Retrieving $p(i)$ from block $P[i']$ and writing back a new value for $p(i)$, done using same PIR and PIW techniques as explained above. The resulting bandwidth is $O(\tau \log(n))$ *offline* and $O(2^\tau \log(n))$ *online*.
- A mux selecting $\mathsf{M}^{\mathsf{r}}[i]$ or $S^{\mathsf{r}}[p(i)]$ takes $O(B + \log(n))$ *online*.
- Deamortized cost in refresh performed after n accesses is $O(B)$ *offline*.

Assuming $\tau = O(1)$, adding up the above costs, we obtain the following communication cost for a single level:

online: $O(B + \log(n))$ offline: $O(\kappa \log(n))$.

When we add the recursive cost over $\log(n)$ levels have:

online: $O(B + \log^2 n)$ offline: $O(\kappa \log^2 n)$.

7.4 Implementation and Performance

We tested a C++ prototype of our 3PC-DPF-ORAM, on three AWS EC2 r4.4xlarge servers. Each r4.4xlarge instance is equipped with eight Intel Xeon E5-2686 v4 CPU's (2.3 GHz) and 122 GB memory.

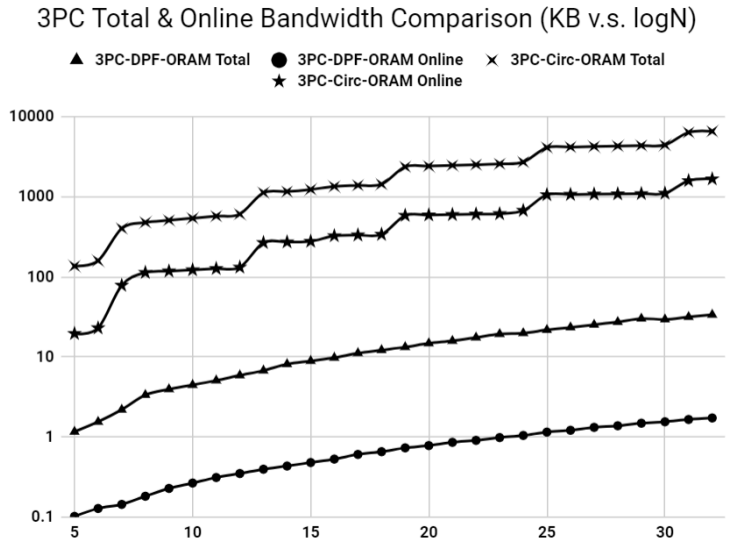


Figure 7.1: 3PC-ORAM Total and Online Bandwidth Comparison

3PC Total and Online Bandwidth. We show our 3PC-DPF-ORAM total and online bandwidth in Fig. 7.1, and compare with bandwidth of 3PC-Circuit-ORAM by [29]. Because in our 3PC construction, secure computation by garbled circuit is not necessary, that the PRF encryption on read only memory blocks are not needed, and the DPF key generation can be done as local computation instead of secure computation, our 3PC-DPF-ORAM has complexity $O(\kappa \log^2 n + B)$ and $O(\log^2 n + B)$ for total and online bandwidth, and is extremely efficient comparing to bandwidth of other 2PC/3PC schemes. For example, for $\log(n) = 30$ and $B = 4$ bytes, our 3PC-DPF-ORAM total and online bandwidth are 30KB and 1.6KB. Comparing with other 3PC-ORAM schemes, that for the same $\log(n)$ and B parameters, 3PC-Circuit-ORAM of [29] has total and online bandwidth 4.4MB and 1.1MB (shown in Fig. 7.1), which are 150x and 713x times larger than our bandwidth; 3PC ORAM of [18]

has online bandwidth 2.5MB for bucket size=32 (they require large bucket size to reduce overflow probability) which is 1500x times larger than our online bandwidth; for generic 3PC construction using [1], FLORAM has online bandwidth at least 2.5MB due to their $O(n^{1/2})$ stash linear scan and periodic refresh (which is also 1500x more comparing to ours), and Circuit-ORAM's online bandwidth will be at least 750KB based on the circuit size reported in [15], which is 468x times larger than our online bandwidth (Sqrt-ORAM's online bandwidth will be even more because of circuit size larger than Circuit ORAM).

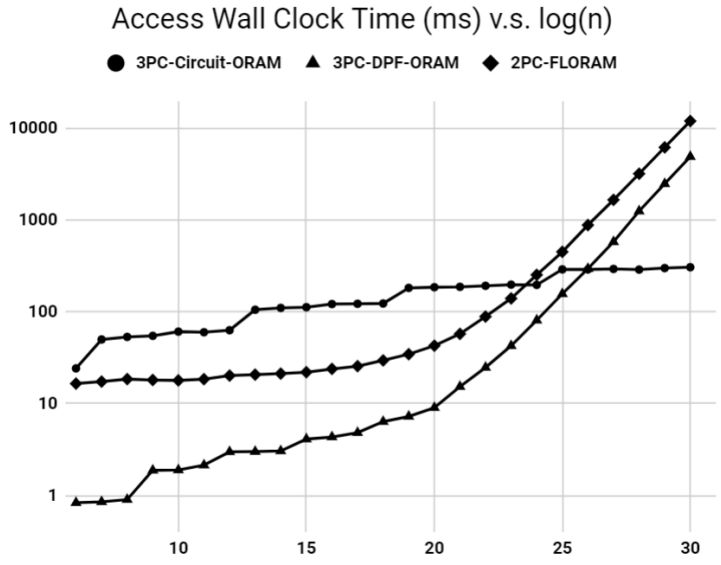


Figure 7.2: Access runtime comparison, with block size=4 bytes

Access Wall Clock Time. We compare access total runtime (single-threaded) with recent 2PC-FLORAM [15] and 3PC-Circuit-ORAM [18] in Fig. 7.2, because both of these schemes have most efficient access runtime for a wide range of $\log(n)$ for 2PC and 3PC settings. For memory size $n = 2^{10}$, our 3PC-DPF-ORAM wall clock (WC) time is 9.5x better than 2PC-FLORAM and 32x better than 3PC-Circuit-ORAM; for $n = 2^{20}$, our 3PC-DPF-ORAM WC is 4.7x better than 2PC-FLORAM and 20x better than 3PC-Circuit-ORAM. Due to $O(n)$ linear local computation complexity, for $n = 2^{30}$, our 3PC-DPF-ORAM's WC is not as good as 3PC-Circuit-ORAM's (the break-even point between the 3PC-DPF-ORAM and

3PC-Circuit-ORAM is around $n = 2^{26}$, as in Fig. 7.2), however, our WC is still 2.4x better than 2PC-FLORAM.

Chapter 8

Results and Conclusion

This chapter presents the overall results of this 3PC-ORAM study, discusses the contributions and improvements of the 3PC-ORAM schemes, and concludes this 3PC-ORAM study.

8.1 Contributions and Improvements of 3PC-ORAM

In Table 8.1 we list the bandwidth complexity for all our 3PC-ORAM schemes, together with the 2PC-ORAM bandwidth complexity from Table 2.1 in Chapter 2. Since most efficient constructions of all the listed schemes have round complexity $O(\log(n))$, we omit it in the table to avoid duplication. And note that the computation complexity of each scheme is the same as the bandwidth complexity, except for 2PC-FLORAM and 3PC-DPF-ORAM which have $O(n)$ complexity due to local computation.

MPC-ORAM schemes	Bandwidth
2PC-Circuit-ORAM [44]	$O(\kappa \log^3(n) + \kappa B \log(n))$
2PC-Sqrt-ORAM [51]	$O(\kappa B \sqrt{n \log^3(n)})$
2PC-FLORAM [15]	$O(\kappa B \sqrt{n})$
3PC-ORAM [18]	$O(\kappa \lambda \log^3(n) + \lambda B \log(n))$
3PC-Circuit-ORAM [29]	$O(\kappa \log^3(n) + B \log(n))$
3PC-Sqrt-ORAM	$O(\log^2(n)\sqrt{n} + B\sqrt{n})$
3PC-DPF-ORAM	$O(\kappa \log^2(n) + B)$

Table 8.1: Bandwidth per memory access for 2PC- and 3PC-ORAM schemes as a function of security parameter κ, statistical parameter λ, memory array length n, and record size B.

The contributions and the performance improvements we made on 3PC-ORAM can be summarized as following:

1. **3PC-ORAM [18]**. We see our contributions as three-fold: First, we provide an immediate improvement to any application of MPC-ORAM which can be done in the setting of three parties with an honest-majority. Secondly, the techniques we explore can be utilized in a different context, e.g., for a different "secure-computation friendly" eviction strategy for a binary-tree ORAM. Finally, the proposed protocol leaves several avenues for further improvements in 3PC-ORAM both on the level of system implementation and algorithm design. Indeed, both the idea and the actual design of the customized 3PC protocols used in [18] are adopted in all three following 3PC-ORAMs of our work. And asymptotically, as in Table 8.1, the complexity of 3PC ORAM [18] eliminates the security parameter κ factor on record size B comparing to generic 2PC implementation of the similar underlying binary-tree ORAM (i.e., 2PC Path-ORAM), and thus gives more efficient bandwidth for large B.

2. **3PC-Circuit-ORAM [29]**. 3PC-Circuit-ORAM is our practically efficient MPC ORAM with best asymptotic complexity considering both bandwidth and computation. There is often an efficiency gap between known MPC-ORAM and client-server ORAM due to client algorithm complexity. Current asymptotically best 2PC-ORAM is implied

by an "MPC-friendly" variant of *Path-ORAM* [42] called *Circuit-ORAM*, due to Wang et al. [44]. However, using garbled circuit for Circuit-ORAM's client implies MPC ORAM which matches Path-ORAM in rounds but increases *bandwidth* by $\Omega(\kappa)$ factor. With 3PC-Circuit-ORAM, we bridge the gap between MPC-ORAM and client-server ORAM by showing a specialized 3PC-ORAM protocol, which uses only symmetric ciphers and asymptotically matches client-server Path-ORAM in round complexity and for large records also in bandwidth. And comparing to our previous 3PC-ORAM of [18], we completely removes the statistical parameter λ factor from the complexity formula (as in Table 8.1), and thus achieves a lot better concrete performance. Our 3PC-Circuit-ORAM also allows for fast pipelined processing: with postponed clean-up it processes $b = O(\log(n))$ accesses in $O(b + \log(n))$ rounds with $O(B + \mathsf{poly}(\log(n)))$ bandwidth per item.

3. **3PC-Sqrt-ORAM**. Inspired and based on the work of 2PC-Sqrt-ORAM of [51] that asymptotically worse MPC-ORAM schemes can still have better concrete performance due to smaller constants, we designed 3PC-Sqrt-ORAM which improves further more on bandwidth than 2PC-Sqrt-ORAM for small n. In 3PC-Sqrt-ORAM, we simplified the round-trip of each 2PC-Sqrt-ORAM access into a single level traversal, replaced $O(n \log(n))$ shuffling network with $O(n)$ 3PC-OT variants, and eliminated the needs of using generic 2PC Yao's garbled circuit. As shown in Table 8.1, our 3PC-Sqrt-ORAM is the only non-generic MPC-ORAM with $O(\log(n))$ rounds and without security parameter κ in the bandwidth cost. Though it is indeed asymptotically worse than our previous two 3PC-ORAM schemes, 3PC-Sqrt-ORAM has the smallest bandwidth for small n in practice: comparing to 3PC-Circuit-ORAM, 3PC-Sqrt-ORAM's bandwidth can be smaller up to $n = 2^{22}$.

4. **3PC-DPF-ORAM**. In this work we first show a new, "MPC-friendly," 2-server ORAM scheme. It has communication complexity $O(B + \log^2(n))$, while the servers

perform only symmetric-key operations. It is therefore, to the best of our knowledge, the first 2-server ORAM which uses only symmetric key cryptography and achieves $O(B)$ bandwidth for $B = \Omega(\text{polylog}n)$, with $B = (\log^2(n))$. Importantly for our application to MPC-ORAM, our 2-server ORAM protocol is relatively simple, and has a lightweight client algorithm which is amenable to secure computation. We are therefore able to use it to construct 3PC-DPF-ORAM protocol that significantly improves on prior work. The bandwidth cost of our 3PC-DPF-ORAM beats all our prior 3PC ORAM work asymptotically (see Table 8.1) and concretely (see Figure 8.1). And also because 3PC-DPF-ORAM has the best asymptotic cost on record size B, it is the best 3PC-ORAM for both large n and B in practice.

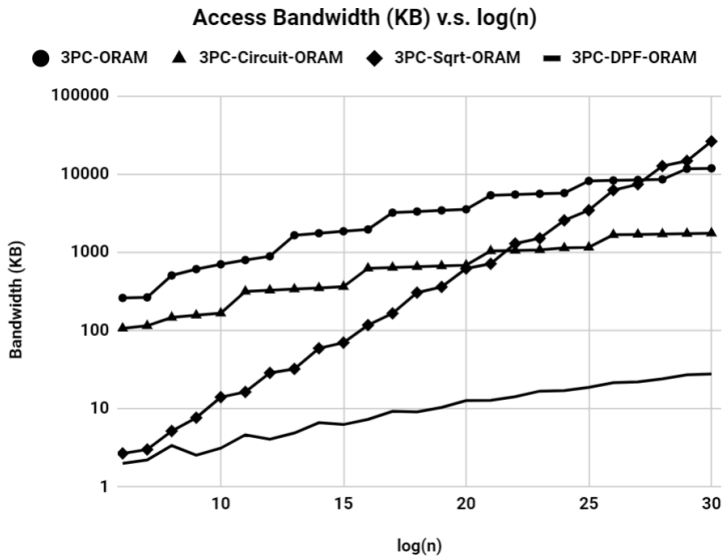


Figure 8.1: Access bandwidth comparison of 3PC-ORAM schemes

Figure 8.1 presents the access concrete bandwidth comparison graphs including all four of our 3PC-ORAM schemes, for $B = 4$ bytes. As reflected by the asymptotic cost, the concrete bandwidth of 3PC-DPF-ORAM is extremely efficient, despite the fact that it requires $O(n)$ local computation. And for large B, the bandwidth advantage of 3PC-DPF-ORAM will be even larger comparing to the other 3PC-ORAM schemes. 3PC-Sqrt-ORAM and 3PC Circuit-ORAM still have their merits that when efficient computation is required, 3PC

Sqrt-ORAM has most efficient bandwidth for n up to 2^{22}, and 3PC-Circuit-ORAM has more efficient bandwidth afterwards. And in practice that for $n > 2^{25}$, 3PC-DPF-ORAM has significant runtime slowdown due to the local computation bottle-neck and is actually slower than the 3PC-Circuit-ORAM. So for application that requires fast processing and response time for large n, 3PC-Circuit-ORAM is probably the best option. Comparing with the best bandwidth efficiency of state-of-art 2PC-ORAM schemes shown in Figure 2.2, for the testing range of $n = 2^6$ to 2^{30}, our best 3PC-ORAM bandwidth can be 39x to 181x times better, with similar similar number of communication rounds (due to implementation differences, the 2PC and 3PC runtime is not compared directly, but we speculate the runtime performance improvements will have the same trend as the bandwidth performance).

8.2 Conclusion

The 3PC-ORAM work accomplished in this study proves that with a slightly weaker security which requires the majority of the parties being honest, the 3PC-ORAM can be a lot more efficient on performance than the best 2PC-ORAM schemes. And our efficient and practical 3PC-ORAM implementations enables secure computation of any RAM program on large data for real world applications that rely on the RAM model. In this study we have presented multiple constructions and approaches for designing ORAMs for MPC, and demonstrated how we can utilize the 3PC setting to overcome the bottle-necks like large circuit size and inefficient crypto operations of previous ORAM designs. We believe our contributions can give insights towards efficiently implementing standard algorithms as MPC protocols.

For future work, first, we are considering semi-honest adversary model for all the 3PC-ORAM work conducted in this study, and it will be good to have efficient MPC-ORAM which is secure against malicious adversary as well. Second, many 3PC protocols presented in this work are customized protocols just for their own setting, which may require careful

adjustments when adopting to other constructions or schemes, so it will be good to also have efficient and generic MPC protocols. Third, notice that our 3PC-DPF-ORAM scheme with the best bandwidth has a trade-off of doing more expensive local computation, and it will be ideal to design some new MPC-ORAM scheme to achieve efficient bandwidth and computation at the same time, for both large memory and record sizes.

Bibliography

[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] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 535–548, New York, NY, USA, 2013. ACM.

[3] D. Beaver. Precomputing oblivious transfer. In D. Coppersmith, editor, *Advances in Cryptology — CRYPT0' 95*, pages 97–109, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[4] D. Beaver. Correlated pseudorandomness and the complexity of private computations. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 479–488, New York, NY, USA, 1996. ACM.

[5] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 503–513, New York, NY, USA, 1990. ACM.

[6] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers computation in private information retrieval: Pir with preprocessing. *J. Cryptol.*, Mar. 2004.

[7] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 784–796, New York, NY, USA, 2012. ACM.

[8] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang. Practicing oblivious access on cloud storage: The gap, the fallacy, and the new way forward. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 837–849, New York, NY, USA, 2015. ACM.

[9] E.-O. Blass, T. Mayberry, and G. Noubir. Multi-client oblivious ram secure against malicious servers. In D. Gollmann, A. Miyaji, and H. Kikuchi, editors, *Applied Cryptography and Network Security*, pages 686–707, Cham, 2017. Springer International Publishing.

[10] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 337–367, 2015.

[11] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1292–1303, 2016.

[12] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42Nd IEEE Symposium on Foundations of Computer Science*, FOCS '01, Washington, DC, USA, 2001. IEEE Computer Society.

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

[14] 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.

[15] J. Doerner and A. Shelat. Scaling ORAM for secure computation. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 523–535, 2017.

[16] Z. Dvir and S. Gopi. 2 serverr pir with subpolynomial communication. *J. ACM*, 63(4):39:1–39:15, Sept. 2016.

[17] S. Eskandarian and M. Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR*, abs/1710.00458, 2017.

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

[19] 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.

[20] 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.

[21] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. In *STOC*, 1998.

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

[23] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In L. Aceto, M. Henzinger, and J. Sgall, editors, *Automata, Languages and Programming*, pages 576–587, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[24] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 157–167, Philadelphia, PA, USA, 2012. Society for Industrial and Applied Mathematics.

[25] 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.

[26] R. Impagliazzo and S. Rudich. Limits on the provable consequences of one-way permutations. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 44–61, New York, NY, USA, 1989. ACM.

[27] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 145–161, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[28] Y. Ishai, E. Kushilevitz, S. Lu, and R. Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In *Proceedings of the RSA Conference on Topics in Cryptology - CT-RSA 2016 - Volume 9610*, pages 90–107, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[29] S. Jarecki and B. Wei. 3pc oram with low latency, low bandwidth, and fast batch retrieval. In B. Preneel and F. Vercauteren, editors, *Applied Cryptography and Network Security*, pages 360–378, Cham, 2018. Springer International Publishing.

[30] 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.

[31] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free xor gates and applications. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Automata, Languages and Programming*, pages 486–498, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[32] 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.

[33] M. Maffei, G. Malavolta, M. Reinert, and D. Schrder. Privacy and access control for outsourced personal records. In *2015 IEEE Symposium on Security and Privacy*, pages 341–358, May 2015.

[34] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder. Maliciously secure multi-client oram. In D. Gollmann, A. Miyaji, and H. Kikuchi, editors, *Applied Cryptography and Network Security*, pages 645–664, Cham, 2017. Springer International Publishing.

[35] J. C. Mitchell and J. Zimmerman. Data-Oblivious Data Structures. In E. W. Mayr and N. Portier, editors, *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 554–565, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[36] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, pages 129–139, New York, NY, USA, 1999. ACM.

[37] 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.

[38] B. Pinkas and T. Reinman. Oblivious ram revisited. In *Proceedings of the 30th Annual Conference on Advances in Cryptology*, CRYPTO'10, pages 502–519, Berlin, Heidelberg, 2010. Springer-Verlag.

[39] M. O. Rabin. How to exchange secrets with oblivious transfer, 2005. Harvard University Technical Report 81 talr@watson.ibm.com 12955 received 21 Jun 2005.

[40] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious ram. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 415–430, Berkeley, CA, USA, 2015. USENIX Association.

[41] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with o((logn)3) worst-case cost. In *Proceedings of the 17th International Conference on The Theory and Application of Cryptology and Information Security*, ASIACRYPT'11, pages 197–214, Berlin, Heidelberg, 2011. Springer-Verlag.

[42] 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.

[43] T. Toft. Secure data structures based on multi-party computation. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 291–292, New York, NY, USA, 2011. ACM.

[44] 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.

[45] X. Wang, D. Gordon, and J. Katz. Simple and efficient two-server oram. Cryptology ePrint Archive, Report 2018/005, 2018. `https://eprint.iacr.org/2018/005`.

[46] X. S. Wang, Y. Huang, T.-H. H. Chan, A. Shelat, and E. Shi. Scoram: Oblivious ram for secure computation. In *Conference on Computer and Communications Security*, CCS '14, pages 191–202, New York, NY, USA, 2014. ACM.

[47] X. S. Wang, K. Nayak, C. Liu, T.-H. H. Chan, E. Shi, E. Stefanov, and Y. Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 215–226, New York, NY, USA, 2014. ACM.

[48] 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.

[49] A. C.-C. Yao. Protocols for secure computations (extended abstract). In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, FOCS'82, pages 160–164, 1982.

[50] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 220–250, 2015.

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

[52] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 283–298, Berkeley, CA, USA, 2017. USENIX Association.

Appendix A

Supplementary Algorithm Figures

A.1 Algorithms for Client-Server Path-ORAM [42]

For completeness we recall the access algorithm of Client-Server Path-ORAM of [42], which we briefly described in Section 5.1. As we explain in Section 5.1, we call the main Path-ORAM algorithm ORAM.Access and its main loop ORAM.ML, and here we show both algorithms as resp. Alg. A.1 and Alg. A.2.

The main point of including these algorithms here is to observe that our 3PC-ORAM protocol described in Section 5.2 is a 3PC emulation of the (client-server) Path-ORAM algorithm, except that the eviction map computation in step 6 of algorithm ORAM.ML, algorithm PathORAM-Route, is replaced by the eviction computation algorithm Route of Circuit-ORAM [44] (with some necessary modifications described in Section 5.2). In particular, observe that our top-level protocol, 3PC-ORAM.Access shown as Alg. 5.1 in Section 5.2, and its main loop 3PC-ORAM.ML, shown as Alg. 5.2 in Section 5.2, are 3PC emulations of resp. algorithms ORAM.Access, Alg. A.1, and ORAM.ML, Alg. A.2.

Algorithm A.1 ORAM.Access: Client/Server Path-ORAM

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

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

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

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

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

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

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

3: **end for**

*: On top-level ORAM tree

Algorithm A.2 ORAM.ML: Main Loop of Client/Server Path-ORAM

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

Output: (1) L_{i+1} where $\mathrm{L}_{i+1} = \mathrm{T}.\mathsf{rec}[\Delta\mathrm{N}]$ for T on $\mathsf{tree.path}(\mathrm{L})$ s.t. $\mathrm{T}.(\mathsf{fb}|\mathsf{adr}) = 1|\mathrm{N}$ (* or $\mathsf{rec}^{\mathrm{C}}$ where $\mathsf{rec} = \mathrm{T}.\mathsf{rec}$)

(2) $\mathsf{tree.path}(\mathrm{L})^{\mathsf{S}}$ modified by eviction, with $\mathrm{T}.\mathsf{lb} := \mathrm{L}'$ and $\mathrm{T}.\mathsf{rec}[\Delta\mathrm{N}] := \mathrm{L}_{i+1}'$ (* $\mathrm{T}.\mathsf{rec} := \mathsf{rec}'$)

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

Retrieval of Next Label/Record

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

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

Post-Process

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

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

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

Eviction

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

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

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

*: On top-level ORAM tree; ▷: Comments;

!: For $i = 0$, C runs steps 2-3 for $\mathrm{T} := \mathsf{tree}_0$ and sends L_{i+1} and (modified) tree_0 to S

A.2 3PC-Circuit-ORAM Auxiliary Protocols

In this section we specify all sub-protocols used in protocol 3PC-ORAM.ML, Alg. 5.2, of Section 5.2, together with the round/bandwidth characteristics of their implementation. In subsection A.2.2 we also include modified versions of protocols 3ShiftPIR, Alg. A.9, and 3ShiftXorPIR, Alg. A.11, namely protocols 3ShiftPIR-Mod, Alg. A.12, and 3ShiftXorPIR-Mod, Alg. A.13. As we explain in Section 5.2, using these protocols results in reducing the round complexity of 3PC-ORAM.ML from 4 to 3 per ORAM tree in the retrieval and post-processing phase.

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

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

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

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

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

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

Algorithm A.3 Types of Secret-Sharing

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

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

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

Algorithm A.5 Reshare: Interactive Sharing Transformation

(i) $\langle x \rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2} = (x_1^{\mathsf{P}_1}, x_2^{\mathsf{P}_2}) \quad \longrightarrow \quad \langle x \rangle = (z_1^{\mathsf{P}_2\mathsf{P}_3}, z_2^{\mathsf{P}_1\mathsf{P}_3}, z_3^{\mathsf{P}_1\mathsf{P}_2})$
Pick $z_1^{\mathsf{P}_2\mathsf{P}_3}, z_2^{\mathsf{P}_1\mathsf{P}_3} \xleftarrow{\$} \{0,1\}^{|x|}$;

P_1 and P_2 exchange $(x_1 \oplus z_2)$ and $(x_2 \oplus z_1)$, set $z_3 := (x_1 \oplus z_2) \oplus (x_2 \oplus z_1)$

(ii) $\langle x \rangle_{\mathsf{xor}} = (x_1^{\mathsf{P}_1}, x_2^{\mathsf{P}_2}, x_3^{\mathsf{P}_3}) \quad \longrightarrow \quad \langle x \rangle = (z_1^{\mathsf{P}_1\mathsf{P}_2}, z_2^{\mathsf{P}_2\mathsf{P}_3}, z_3^{\mathsf{P}_3\mathsf{P}_1})$
Generate random m-bit zero-sharing, $(s_1^{\mathsf{P}_1}, s_2^{\mathsf{P}_2}, s_3^{\mathsf{P}_3}) \xleftarrow{\$} \langle 0^m \rangle_{\mathsf{xor}}$;

Each P_i sets $z_i := x_i \oplus s_i$ and sends z_i to $\mathsf{P}_{(i+1 \bmod 3)}$

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

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

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

A.2.1 Protocols for Retrieval

Algorithm A.6 Protocol $\mathsf{KSearch}$ (from [18]) (Step 1 in Alg. 5.2)

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

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

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

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

1: $r^{\mathrm{DE}} \xleftarrow{\$} \mathsf{array}^\kappa[\ell]$, $s^{\mathrm{DE}} \xleftarrow{\$} \mathsf{Z}_\ell$; $(a^{\mathrm{D}}, b^{\mathrm{E}}) := \langle z\rangle_{\mathsf{xor}}^{\mathrm{D-E}}$ where $\langle z\rangle_{\mathsf{xor}}^{\mathrm{D-E}}$ is locally transformed from $\langle u, v\rangle_{\mathsf{xor}}^{\mathrm{D-E}}$ s.t. $z[j] = u[j+s \bmod \ell] \oplus v$ for all j
D sends array x to C s.t. $x[j] = \mathsf{F}_{\mathsf{k}}(\, r[j] \oplus (a[j] \,|\, 0^{\kappa-c})\,)$ for all j
E sends array y to C s.t. $y[j] = \mathsf{F}_{\mathsf{k}}(\, r[j] \oplus (b[j] \,|\, 0^{\kappa-c})\,)$ for all j
2: Let $\langle i\rangle_{\mathsf{shift}}^{\mathrm{DE-C}} := (s^{\mathrm{DE}}, t^{\mathrm{C}})$ for unique t s.t. $x[t] = y[t]$
(if $x[t] = y[t]$ for ≥ 2 t's, C asks D, E to re-run from step 1 with fresh r^{DE})
3: Reshare: $\langle i\rangle_{\mathsf{shift}}^{\mathrm{DE-C}} \to \langle i\rangle_{\mathsf{shift}}$

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

Algorithm A.7 Protocol SSPIR (from [13]) (Used in Alg. A.10 and A.8)

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

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

Pick $a_1^{\mathsf{P}_1\mathsf{P}_3} \xleftarrow{\$} \{0,1\}^n$, $r^{\mathsf{P}_1\mathsf{P}_2} \xleftarrow{\$} \{0,1\}^m$;
1: P_3 sends $a_2 = a_1 \oplus \mathsf{ind}_t^n$ to P_2 (Note that $a_2 = a_1$ except $a_2[t] = a_1[t] \oplus 1$)
2: P_1 sets $z_1 := r \oplus \mathsf{XORSelect}(x, a_1)$, and P_2 sets $z_2 := r \oplus \mathsf{XORSelect}(x, a_2)$,
where $\mathsf{XORSelect}(x, a) = \bigoplus_i$ s.t. $_{a[i]=1}\ x[i]$
Output $\langle x[t]\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2} = (z_1^{\mathsf{P}_1}, z_2^{\mathsf{P}_2})$.

Bandwidth: n; Rounds: 1;
Security: This is the basis of security of PIR of Chor et al. [13]: P_2's received message a_2 is a random string because a_1 is a one-time pad. Moreover, the secret-sharing of $x[t]$ is random because r is a one-time pad.

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

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

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

1: P_1 and P_2 set $x' := x_{\mathsf{shift}[s]}$, i.e., $x'[j] = x[j + s \bmod m]$ for all j
2: SSPIR: $(x')^{\mathsf{P}_1\mathsf{P}_2}, t^{\mathsf{P}_3} \to \langle x'[t]\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$ $(= \langle x[i]\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2})$

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

Algorithm A.9 Protocol 3ShiftPIR (Step 2 in Alg. 5.2)

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

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

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

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

Algorithm A.10 Protocol ShiftXorPIR (Used in Alg. A.11)

Input: $x^{\mathsf{P}_1\mathsf{P}_2}$, $\langle i_1\rangle_{\mathsf{shift}}^{\mathsf{P}_1\mathsf{P}_2-\mathsf{P}_3} = (s_1^{\mathsf{P}_1\mathsf{P}_2}, t_1^{\mathsf{P}_3})$, $\langle i_2\rangle_{\mathsf{2,1\text{-}xor}}^{\mathsf{P}_1\mathsf{P}_2-\mathsf{P}_3} = (s_2^{\mathsf{P}_1\mathsf{P}_2}, t_2^{\mathsf{P}_3})$,
for $x \in \mathsf{array}^{\ell}[n,m]$, $i_1 \in \mathsf{Z}_n$, $i_2 \in \mathsf{Z}_m$

Output: $\langle x[i_1][i_2]\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$

1: P_1 and P_2 set $x' \in \mathsf{array}^{\ell}[nm]$ s.t. for all $(j_1, j_2) \in \mathsf{Z}_n \times \mathsf{Z}_m$,
$x'[j_1 \cdot m + j_2] = x[j_1 + s_1 \bmod n][j_2 \oplus s_2]$
P_3 computes $t = t_1 \cdot m + t_2$ (over integers)
2: SSPIR: $x'^{\mathsf{P}_1\mathsf{P}_2}, t^{\mathsf{P}_3} \rightarrow \langle x'[t]\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$ $(= \langle x[i_1][i_2]\rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2})$

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

Algorithm A.11 Protocol 3ShiftXorPIR (Step 3 in Alg. 5.2)

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

Output: $x[i_1][i_2]$

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

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

A.2.2 Protocols for Reduced-Round Retrieval

Algorithm A.12 Protocol 3ShiftPIR-Mod (Step 2 in Alg. 5.2)

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

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

Offline: Pick $a_{12}^{\mathrm{CD}}, a_{23}^{\mathrm{DE}}, a_{32}^{\mathrm{DE}} \xleftarrow{\$} \{0,1\}^n$
D picks $t_2 \xleftarrow{\$} \{1,\ldots,N\}$ and sends $a_{21} := a_{23} \oplus \mathsf{ind}_{t_2}^n$ to C
E picks $t_3 \xleftarrow{\$} \{1,\ldots,N\}$ and sends $a_{31} := a_{32} \oplus \mathsf{ind}_{t_3}^n$ to C

On input s_1^{DE}**:**

1: D sends $\delta_{12} = s_1 - t_2 \bmod n$ to C
E sends $\delta_{13} = s_1 - t_3 \bmod n$ to C

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

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

C sets $c_1 := \mathsf{XORSelect}((x_2)_{\mathsf{shift}[s_2]}, a_{21})$
C sets $c_2 := \mathsf{XORSelect}((x_3)_{\mathsf{shift}[s_3]}, a_{31})$
D sets $d_1 := \mathsf{XORSelect}((x_1)_{\mathsf{shift}[s_1]}, a_{12})$
D sets $d_2 := \mathsf{XORSelect}((x_3)_{\mathsf{shift}[s_3]}, a_{32})$
E sets $e_1 := \mathsf{XORSelect}((x_1)_{\mathsf{shift}[s_1]}, a_{13})$
E sets $e_2 := \mathsf{XORSelect}((x_2)_{\mathsf{shift}[s_2]}, a_{23})$

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

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

Bandwidth: on-line: $\approx n + 3\ell$, off-line: $\approx 2n$ (assuming s_1^{DE} known off-line) ;
Rounds: 2;
Security: Party C receives only δ_{21} and δ_{31}, but these are random in $\{1,\ldots,N\}$ because of one-time pads t_2 and t_3. These one-time pads were used also in computing a_{21} and a_{31} (resp. by D and E) but nevertheless a_{21}, a_{31} are uniform random strings in C's view because of onetime pads a_{23} and a_{32}. Party D receives $s_3 = t_1 + \delta_{13}$ from C, but $t_1 + \delta_{13} = t_1 + (s_1 - t_3) = i - t_3$ where t_3 is E's one-time pad, so s_3 is random in $\{1,\ldots,N\}$ in D's view. Party E receives a_{13} and $s_2 = t_1 + \delta_{12}$ from C, but $t_1 + \delta_{12} = t_1 + (s_1 - t_2) = i - t_2$ where t_2 is D's one-time pad, so s_2 is random in $\{1,\ldots,N\}$ in E's view. Value $a_{13} = a_{12} \oplus \mathsf{ind}_{t_1}^n$ is also random in E's view because of one-time pad a_{12}.
Finally, by correctness of Reshare, the final sharing is a *random* sharing of $x[i]$.

Algorithm A.13 Protocol 3ShiftXorPIR-Mod (Step 3 in Alg. 5.2)

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

Output: $x[i][j]$

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

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

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

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

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

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

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

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

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

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

A.2.3 Protocols for PostProcess

Algorithm A.14 Protocol ULiT - Update Labels in Tuple (Step 4, Alg. 5.2)

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

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

Offline:

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

Online:

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

Bandwidth: Online: $\approx 4|X|$, Offline: $\approx 4|X|$;
Rounds: 2 (the first round requires only input $\langle \Delta\mathrm{N}\rangle$, see Alg. A.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 A.15 Protocol InsertLbl - Inserting Label (Used in Alg A.14)

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

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

Offline:

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

Online:

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

Correctness: Observe that $z_2 = p \oplus b_{\mathsf{rot}[\beta_2 \oplus w]}$ and $p^* = p \oplus a_{\mathsf{rot}[\alpha_1 \oplus v]}$.
Note that $(s_2)_{\mathsf{rot}[u_1]} = (a^{\mathsf{xor}[\,\mathrm{L}_2 \,@\, \alpha_2 \oplus v]})_{\mathsf{rot}[\alpha_1 \oplus v]}$
$= (a_{\mathsf{rot}[\alpha_1 \oplus v]})^{\mathsf{xor}[\,\mathrm{L}_2 \,@\, (\alpha_2 \oplus v) \,\oplus\, (\alpha_1 \oplus v)\,]}$
$= (a_{\mathsf{rot}[\alpha_1 \oplus v]})^{\mathsf{xor}[\,\mathrm{L}_2 \,@\, \Delta \mathrm{N}\,]}$.
Likewise $(s_1)_{\mathsf{rot}[u_2]} = (b_{\mathsf{rot}[\beta_2 \oplus w]})^{\mathsf{xor}[\,\mathrm{L}_1 \,@\, \Delta \mathrm{N}\,]}$.
It follows that
$z_3 = p^* \,\oplus\, (s_2)_{\mathsf{rot}[u_1]} \,\oplus\, (s_1)_{\mathsf{rot}[u_2]}$
$= p \,\oplus\, a_{\mathsf{rot}[\alpha_1 \oplus v]} \,\oplus\, (a_{\mathsf{rot}[\alpha_1 \oplus v]})^{\mathsf{xor}[\,\mathrm{L}_2 \,@\, \Delta \mathrm{N}\,]} \,\oplus\, (b_{\mathsf{rot}[\beta_2 \oplus w]})^{\mathsf{xor}[\,\mathrm{L}_1 \,@\, \Delta \mathrm{N}\,]}$
By xor-ing z_2 and z_3 observer that pad p and rotated pads a and b cancel out and we get $M = z_2 \oplus z_3 = [0, ..., 0]^{\mathsf{xor}[\,\mathrm{L} \,@\, \Delta \mathrm{N}]}$ where $[0, ..., 0]$ is an all-zero array.
Bandwidth: Online: $2 \cdot (2^{\tau}\ell + \tau) \approx 2 \cdot 2^{\tau}\ell$, Offline: $\approx 2^{\tau}\ell$;
Rounds: 2 (the first round requires only input $\langle \Delta \mathrm{N} \rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$);
Security:
(1) For $\mathsf{P}_1, \mathsf{P}_2$: Let $(\Delta \mathrm{N}_1^{\mathsf{P}_1}, \Delta \mathrm{N}_2^{\mathsf{P}_2})$ denote input $\langle \Delta \mathrm{N} \rangle_{\mathsf{xor}}^{\mathsf{P}_1-\mathsf{P}_2}$. P_2 receives $\Delta \mathrm{N}_1 \oplus \alpha_1$ and P_1 receives $\Delta \mathrm{N}_2 \oplus \beta_2$ in in Reshare in step 1. Bot values are random because α_1, β_2 are randomly chosen resp. by P_1 and P_2, and neither value affects the distribution of protocol outputs (z_2, z_3), because z_2 is uniform by randomness of a, and z_3 is a deterministic function of z_2 and protocol inputs.

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

Algorithm A.16 Protocol $\mathsf{FlipFlag}$ (Step 5, Alg. 5.2)

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

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

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

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

Algorithm A.17 Protocol Shift (based on [18]) (Used in Alg A.16)

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

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

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

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

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

Bandwidth: $2n\ell$; Rounds: 1;
Security: Sharing $\langle y \rangle_{\mathsf{xor}}^{\mathrm{C-E}}$ 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.2.4 Protocols for Eviction

Algorithm A.18 Protocol $\mathsf{GC}(\mathsf{circ})$ (Step 6, Alg. 5.2)

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

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

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

1: C and E on input $\langle x \rangle_{\mathsf{xor}}^{\mathrm{C-E}} = (x_1^{\mathrm{C}}, x_2^{\mathrm{E}})$, select input wire keys according to their respective input $x_1^{\mathrm{C}}, x_2^{\mathrm{E}}$ and send to D resp. $\{\mathsf{iwk}_1 : x_1\}$ and $\{\mathsf{iwk}_2 : x_2\}$
2: D evaluates the garbled circuit circ' starting given the received sets of wire keys; Because the garbled circuit contains the wire-key-to-bit translation table only for the wires corresponding to variable z, D outputs the z part of the output in the clear, but for variable y it can only output the wire key set $\{\mathsf{owk} : y\}$ corresponding to value y.

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

Algorithm A.19 Protocol $\mathsf{PermTuples}$ (Step 8, Alg. 5.2)

Param: Number of buckets d, bucket size w

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

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

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

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

1: D sends $z = \mathsf{t} \oplus p$ to C.
2: C sends $g = \rho \oplus \pi(z \oplus r)$ to D.
3: D outputs $\mathsf{t}^{\circ} = a \oplus g$.

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

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

Algorithm A.20 Protocol $\mathsf{PermBuckets}$ (Step 7, Alg. 5.2)

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

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

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

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

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

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

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

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

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

Online:

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

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

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

Correctness:

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

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

Algorithm A.21 Protocol SSXOT (Step 9, Alg. 5.2)

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

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

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

$\mathsf{Extract}$: $\langle \mathsf{path} \rangle \to \langle \mathsf{path} \rangle^{\mathrm{C-E}}_{\mathsf{xor}}$. Parties sets $\mathsf{EM}^{\circ \mathrm{D}} := \mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ}$, $\Pi^{\mathrm{CE}} := \tilde{\rho} \cdot \ddot{\pi} \cdot \tilde{\delta}$, and $(x_1^{\mathrm{C}}, x_2^{\mathrm{E}}) = \langle \mathsf{path}^\circ \rangle^{\mathrm{C-E}}_{\mathsf{xor}} := \langle \Pi(\mathsf{path}) \rangle^{\mathrm{C-E}}_{\mathsf{xor}}$. (see eq. (5.1) in Sec. 5.2)

1: The following two steps are performed in parallel:

$\mathsf{HalfXOT}$: $x_1^{\mathrm{C}}, (\mathsf{EM}^\circ, p)^{\mathrm{D}} \to y_1^{\mathrm{E}}$ ▷ $y_2 = p \oplus \mathsf{EM}^\circ(x_1)$

$\mathsf{HalfXOT}$: $x_2^{\mathrm{E}}, (\mathsf{EM}^\circ, p)^{\mathrm{D}} \to y_2^{\mathrm{C}}$ ▷ $y_1 = p \oplus \mathsf{EM}^\circ(x_2)$

Parties set $\langle y \rangle^{\mathrm{C-E}}_{\mathsf{xor}} := (y_1^{\mathrm{C}}, y_2^{\mathrm{E}})$, ▷ $y = \mathsf{EM}^\circ(x) = \mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ}(\Pi(\mathsf{path}))$

and set $\langle \mathsf{path}' \rangle^{\mathrm{C-E}}_{\mathsf{xor}} := \langle \Pi^{-1}(y) \rangle^{\mathrm{C-E}}_{\mathsf{xor}}$ ▷ $\mathsf{path}' = \Pi^{-1}(y) = \mathsf{EM}_{\sigma,\mathsf{t}}(\mathsf{path})$

2: $\mathsf{Reshare}$: $\langle \mathsf{path}' \rangle^{\mathrm{C-E}}_{\mathsf{xor}} \to \langle \mathsf{path}' \rangle$

Bandwidth: Online: $4|\mathsf{path}| + 2m\log(m)$, $m = \#$ tuples; Offline: $2|\mathsf{path}|$; Rounds: 3;
Security: Because p is a one-time pad known only to D, both y_1 and y_2 are individually random, and by security of $\mathsf{HalfXOT}$, the protocol leaks nothing beyond (locally random) values y_1 to E and y_2 to C. Moreover, by correctness of $\mathsf{HalfXOT}$ we have that (y_1, y_2) is an xor-sharing of $y = \mathsf{EM}^\circ(x)$. Then by eq. (5.1) in Sec. 5.2, we have $\mathsf{path}' = \Pi^{-1}(y) = \Pi^{-1} \cdot \mathsf{EM}_{\sigma^\circ,\mathsf{t}^\circ} \cdot \Pi(\mathsf{path}) = \mathsf{EM}_{\sigma,\mathsf{t}}(\mathsf{path})$.

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

Param: n, k, ℓ s.t. $k \le n$.

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

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

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

1: On P_2's input p: P_2 sends $s = p \oplus \delta(r)$ to P_3

2: On P_2's input σ: P_2 sends $\pi = \delta^{-1} \cdot \sigma$ to P_1

3: On P_1's input x (and message π): P_1 sends $a = r \oplus \pi(x)$ to P_3
P_3 outputs $y = s \oplus \delta(a)$

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

A.3 3PC-Circuit-ORAM Routing Circuit

In this section we explain the construction of the routing circuit Route used in the eviction phase of protocol 3PC-ORAM.ML (see Step 6 in Alg. 5.2, Section 5.2).

A.3.1 Main Routing Circuit

Circuit Route determines eviction map by generating a dp array (PrepareDeepest), computing the eviction array σ from dp (PrepareTarget), and making the eviction map σ into a cycle (MakeCycle). Circuits PrepareDeepest and PrepareTarget are, with minor variations, the same circuits which implemented the original Circuit-ORAM eviction computation CircORAM-Route [44], but circuit MakeCycle is new. Because of some small differences in the implementation of PrepareDeepest and PrepareTarget, the combined size of circuit Route is virtually identical to the size of CircORAM-Route reported in [44]. We discuss these three circuits in the following subsections.

Algorithm A.23 Circuit Route (Used in Step 6, Alg. 5.2)

Param: Tree height d, bucket size w.

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

Output: $\sigma \in \mathsf{perm}_{\mathrm{d}}$ and $\mathsf{t}' \in \mathsf{array}^{\log(w+1)}[\mathrm{d}]$, where σ extends Φ into a cycle and $\mathsf{t}' = \delta \oplus \mathsf{t}$ for Eviction Map Φ and Tuple Index t computed as in Circuit-ORAM [44]

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

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

A.3.2 Prepare Array dp

Algorithm PrepareDeepest in Alg. A.24, based on the same name algorithm in [44], outputs an array dp where $\text{dp}[i] < i$ is the index of the first bucket in the path which contains a tuple that can be evicted to the i-th bucket. (If no tuples in higher levels can be evicted to the i-th bucket then $\text{dp}[i] = \perp$.) In addition, PrepareDeepest outputs three other arrays $j_{\mathsf{d}}, j_{\mathsf{e}}, e$, where $j_{\mathsf{d}}[i]$ is the index of the "deepest tuple" in the i-th bucket, i.e., a tuple which could be evicted furthest down from that bucket, $e[i] = 1$ if and only if there is an empty tuple at this level, and $j_{\mathsf{e}}[i]$ is the index of that empty tuple. (If $e[i] = 0$ then $j_{\mathsf{e}}[i]$ is meaningless.)

Algorithm A.24 Circuit PrepareDeepest [44] (Used in Alg. A.23)

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

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

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

1: $\text{dp} := [\perp, \perp, ..., \perp]$, $src := \perp$, $goal := -1$
2: **for** $i := 0$ **to** $d - 1$ **do** ▷ cycle: d
3: **if** $goal \geq i$ **then** ▷ cost: $log(d)$
4: $\text{dp}[i] := src$ ▷ cost: $log(d)$
5: **end if**
6: $(l, j_{\mathsf{d}}[i], j_{\mathsf{e}}[i], e[i]) := \mathsf{FDAE}(i, \text{L}, \mathsf{fb}[i], \mathsf{lbl}[i])$ ▷ cost: Alg. A.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: $(3w + \log d) \cdot d^2 + (2w \log w + 5 \log d) \cdot d$

Find the Deepest and Empty Tuples. Algorithm FDAE in Alg. A.25 (which stands for FindDeepestAndEmpty) is adopted from [44], and it is a sub-procedure of Alg. A.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_{d} in the bucket together

with the target level l'. FDAE also determines if there is an empty tuple in this bucket, and outputs its index j_{e} and a flag e which is set to 1 if an empty tuple was found. If no tuple can be moved down from the i-th bucket then FDAE returns $(j_{\mathsf{d}}, l') = (0, i)$, and if there is no empty tuple then $(j_{\mathsf{e}}, e) = (0, 0)$.

Algorithm A.25 Circuit FDAE [44] (Used in Alg. A.24)

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

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

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

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

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

A.3.3 Prepare Arrays σ and t

Algorithm PrepareTarget in Alg. A.26 is an extended version of the corresponding algorithm in [44], which determines the final eviction pattern. PrepareTarget outputs a σ array which contains the same eviction movement as in [44], plus the eviction jumps filling up the possible gaps. PrepareTarget also outputs an array t where $\mathsf{t}[i]$ is the index of the tuple that will be evicted on level i. Note that each $\mathsf{t}[i]$ is selected from one of $j_{\mathsf{d}}[i], j_{\mathsf{e}}[i]$, or w (fake/empty

tuple index) depending on what kind of eviction movement level i is doing. And this `t` will be finally masked by δ so the real indexes will be hidden to D.

Algorithm A.26 Circuit $\mathsf{PrepareTarget}$ (following [44]) (Used in Alg. A.23)

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

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

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

$nTop, nBot, eTop, eBot, src, \mathrm{dest} := \bot$
$\sigma, \mathsf{t} := [\bot, \bot, ..., \bot]$
for $i := d-1$ **to** 0 **do**
  if $i = src$ **then** ▷ cost: $log(d)$
    $\sigma[i] := \mathrm{dest}$ ▷ cost: $log(d)$
    $\mathsf{t}[i] := j_{\mathsf{d}}[i]$ ▷ cost: $log(w)$
    $src := \bot$ ▷ cost: $log(d)$
    if $\mathrm{dp}[i] = \bot$ **then** ▷ cost: $log(d)$
      $\mathrm{dest} := i$ ▷ cost: $log(d)$
    else
      $\mathrm{dest} := \bot$ ▷ cost: $log(d)$
    end if
  end if
  if $\mathrm{dp}[i] \neq \bot$ **then**
    if $\mathrm{dest} \neq \bot$ **and** $src = \bot$ **then** ▷ cost: $2log(d)$
      $\sigma[i] := \mathrm{dest}$ ▷ cost: $log(d)$
      $\mathsf{t}[i] := j_{\mathsf{e}}[i]$ ▷ cost: $log(w)$
    end if
    if $(\mathrm{dest} = \bot$ **and** $e[i] = 1)$ **or** $\sigma[i] \neq \bot$ **then** ▷ cost: $log(d)$
      $src := \mathrm{dp}[i]$ ▷ cost: $log(d)$
      $\mathrm{dest} := i$ ▷ cost: $log(d)$
      $eTop := src$ ▷ cost: $log(d)$
      if $eBot = \bot$ **then** ▷ cost: $log(d)$
        $eBot := \mathrm{dest}$ ▷ cost: $log(d)$
        $\mathsf{t}[i] := j_{\mathsf{e}}[i]$ ▷ cost: $log(w)$
      end if
    end if
  end if
  if $\mathsf{t}[i] = \bot$ **then** ▷ cost: $log(w)$
    $\mathsf{t}[i] := w$ ▷ cost: $log(w)$
    $nTop := i$ ▷ cost: $log(d)$
    if $nBot = \bot$ **then** ▷ cost: $log(d)$
      $nBot := i$ ▷ cost: $log(d)$
    end if
  end if
  $\mathsf{t}'[i] = \mathsf{t}[i] \oplus \delta[i]$
end for

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

A.3.4 Making the Eviction Map into A Cycle

Algorithm MakeCycle in Alg. A.27 adds upwards spurious jumps to the eviction jump array σ output from PrepareTarget and makes the final eviction map as a cycle.

Algorithm A.27 Circuit MakeCycle (Used in Alg. A.23)

Param: Tree height d.

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

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

```
1:  nPrev :=⊥
2:  for i := 0 to d − 1 do
3:      if nTop = ⊥ then                                  ▷ cost: log(d)
4:          if i = eBot then                              ▷ cost: log(d)
5:              σ[i] := eTop                              ▷ cost: log(d)
6:          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: $11log(d) \cdot d$