# DISTRIBUTED OBLIVIOUS RAM: PROGRESS AND PITFALLS

Daniel George Noble

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2024

Supervisor of Dissertation

Brett Hemenway Falk, Research Professor, Computer and Information Science

Graduate Group Chairperson

Sebastian Angel, Raj and Neera Singh Assistant Professor, Computer and Information Science

Dissertation Committee

Sampath Kannan, Henry Salvatori Professor, Computer and Information Science
Rafail Ostrovsky, Norman E. Friedman Chair in Knowledge Sciences, Distinguished Professor of Computer Science, The University of California, Los Angeles
Tal Rabin, Rachleff Family Professor Computer and Information Science

DISTRIBUTED OBLIVIOUS RAM: PROGRESS AND PITFALLS

# ACKNOWLEDGEMENT

First of all, I would like to thank Dr. Brett Hemenway Falk for being an amazing advisor. With the right advisor, a PhD can be some of the best years of one's life, and this has definitely been the case for me. Brett introduced me to many fascinating research questions, and I always enjoyed our discussions. I am deeply grateful to Brett for giving me the freedom to choose the areas in which I worked. Brett is also a genuinely good and generous human being, who wants the best for those around him, a person wise, yet humble. I am very fortunate to have worked under him.

I am grateful for everyone with whom I had to opportunity to collaborate during my PhD studies: Rafail Ostrovsky, Tal Rabin, Andreas Haeberlin, Steve Zdancewic, Marcella Hastings, Edo Roth, Jacob Zhang and Matan Shtepel. These collaborations were essential for my growth as a researcher.

I am thankful to my former employer, Dr. William Pottenger, for introducing me to Multi-Party Computation, despite it ultimately leading to me leaving his employment to pursue this PhD.

I enjoyed my non-academic hours as a PhD student largely thanks to my housemates at the ManS1on. I am particularly grateful for them taking good care of me during my extended quarantine. I am thankful to the members of the Tundra Lab for adopting me and for our interesting but off-topic conversations. I have also received much encouragement and many good meals from the members of Philadelphia UBF church, and especially value James Roh's selfless mentorship over these many years.

I would not be here, by definition, without my parents, and am deeply grateful for the freedom and support they have given me throughout my life to pursue my interests. I am also thankful for the various kinds of support I have received from my sisters, uncles, aunts and grandparents, and appreciate them taking me at my word that the things I am researching really are important.

I am deeply blessed to be married to Maria Katanga Noble, and greatly appreciate the unwavering support she has provided in helping me to finish the doctoral program well.

Lastly, I am thankful to my lifelong advisor, boss, parent and friend, Jesus Christ, who has never left me.

# ABSTRACT

## DISTRIBUTED OBLIVIOUS RAM: PROGRESS AND PITFALLS

Daniel George Noble

Brett Hemenway Falk

Generic Secure Multi-Party Computation (MPC) was first introduced in the circuit model, using arithmetic circuits, or Boolean circuits. However, many computations are not naturally, or efficiently, representable as circuits. An easy example of this is a binary search over $n$ items. In the RAM model this requires only $\Theta(\log(n))$ RAM accesses, but in a circuit model this would require a circuit of size $\Omega(n)$.

Distributed Oblivious RAM (DORAM) is a functionality that allows reading and writing to a secret-shared memory at a secret-shared location. This is similar to the primitive of Oblivious RAM, in which a program must hide its virtual memory accesses from an adversary who can see which locations it is accessing in physical memory. Many techniques from ORAMs are applicable to DORAMs as well.

The thesis makes three contributions in the area of DORAMs and ORAMs. Firstly, it presents an attack on several prominent ORAM and DORAM protocols, and shows a solution which fixes affected protocols at little extra cost. Secondly, it presents a computationally secure DORAM which requires $\Theta(\log(n)(\kappa + d))$ bits of communication per memory access, with much smaller constants than previous work, where $\kappa$ is a computational security parameter and $d$ is the bit-length of memory blocks. Finally, it presents a statistically secure DORAM which requires $\Theta(\log(n)/\log(\log(n))(\log^2(n)+d))$ bits of communication per memory access. The latter is an asymptotic improvement over previous work, and is the first statistically secure DORAM to require $o(\log(n)d)$ communication for blocks of size $O(\log^2(n))$.

TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF ILLUSTRATIONS

# CHAPTER 1

# Introduction

## 1.1. Secure Multi-Party Computation in the RAM model

Secure Multi-Party Computation (MPC) allows a number of parties to compute a functionality using their inputs without revealing the inputs themselves. More formally, let $f$ be a functionality that takes $s$ inputs and produces $s$ outputs. If there are $s$ parties $P_1, \ldots, P_s$, with respective inputs $x_1, \ldots, x_s$, a MPC protocol allows the computation of $f(x_1, \ldots, x_s)$ with $P_i$ receiving the the $i^{th}$ output of $f$. MPC prototols guarantee that an adversary, $\mathcal{A}$, who is able to corrupt certain subsets of the parties will nevertheless not be able to learn anything besides the inputs and outputs of those parties.

The MPC problem was first formulated in the 1980's [Yao82] and MPC protocols were quickly found that could evaluate arbitrary circuits [GMW19] [BOGW19] [CCD88]. After three decades of development, these protocols are now very efficient. For instance, it is possible in the 3-party setting, where at most one party is corrupted, to evaluate 7 billion Boolean gates per second [AFL$^+$16].

However, the circuit model itself is inherently inefficient for certain computations. For instance, graph algorithms assume the ability to access the neighbors of a vertex. This is simple in the RAM model by storing the edges using an adjacency list, sorted by vertex. However, it is hard to represent such a computation efficiently in the circuit model. More generally, all data structures that are based on pointers (e.g. binary search trees, heaps) are implemented in the RAM model, and it is not simple to build such data structures efficiently using circuits.

This problem is compounded by a paradigm shift in the applications of MPC. Initially, MPC was proposed as a solution for secure function evaluation, that is there was a once-off computation with all inputs provided at the start of the computation and all outputs released at the end of the computation. However, increasingly, applications of MPC require the implementation of *reactive* functionalities, that maintain a (potentially large) state between protocol executions. For instance,

imagine a service in which people allow their medical data to be securely analyzed by researchers for a fee. Privacy guarantees are maintained by the data being secret-shared between several non-colluding servers. In this case, the servers hold a database, which is maintained over a long period of time, and periodically updated (as new data suppliers join) and frequently queried (by researchers). For this application, the amount of data may be very large (ideally millions of records). Furthermore, this data must be organized in ways that allow efficient queries; traditional techniques for making database queries efficient, such as indexing over multiple columns used for queries, assumes the existence of RAM. An efficient implementation of RAM for MPC is therefore especially pertinent for allowing such applications to be efficient and thrive in practice.

Distributed Oblivious RAM (DORAM) is a primitive that allows MPC protocols to be constructed in the RAM model. A DORAM takes a secret-shared query (either a read or a write), using a secret-shared index. For a read it returns a secret-sharing of the data in memory at the index; for a write it takes a secret-sharing of a new value and updates the memory at the index.

In this thesis we present two new DORAM protocols which have improved efficiency relative to existing approaches. The first protocol, presented in Chapter 4, is secure against a computationally-bounded adversary, and has communication cost $\Theta(\log(n)(\kappa + d))$ bits per query, where $n$ is the number of memory locations, $d$ is the bit-length of each memory location and $\kappa$ is a computational security parameter. The second protocol, presented in Chapter 5, is information-theoretically secure and requires $\Theta(\log(n)/\log(\log(n))(\log^2(n) + d))$ bits of communication per query.

The thesis proceeds as follows. We first define DORAMs formally. Following this we introduced the concept of an Oblivious RAM, define it formally and show how it (and its variants) relate to DORAMs. Following this we review the literature of ORAMs and DORAMs, focusing on techniques that have brought significant advancements. We then describe the hierarchical technique in depth and prove its security, since our protocols make use of this technique.

Chapter 3 presents an attack on several ORAMs and DORAMs. This attack hinges on the fact that these ORAMs and DORAMs broke the abstraction of the hierarchical solution, and this resulted in

a subtle security flaw. We also present a solution that fixes these ORAMs and DORAMs without increasing their asymptotic cost; nevertheless the fixed solutions still break the Hierarchical ORAM abstraction. Chapter 4 presents a DORAM that maintains the abstraction of a the Hierarchical solution. This solution is also more efficient than existing DORAMs under certain parameter ranges. This DORAM makes use of the observation that the most challenging part of the Hierarchical solution is determining at which level an item is stored. Chapter 5 presents our final DORAM. Continuing off of the observation that the most challenging part of a DORAM is knowing where each item is stored, it builds an efficient data-structure that stores only this information. This allows for efficient accesses of desired data elements, and results in the most efficient statistically-secure DORAM to date for small block sizes. Chapter 6 concludes, and highlights some interesting open questions.

## 1.2. Security

In this section, we introduce our formal security model and define what it means for a Distributed Oblivious RAM to be secure. Since DORAMs are a type of MPC protocol, we will define their security according to the simulation paradigm, as is standard [Lin17].

Simulation is a general technique for proving security of a protocol. It compares a *real* execution of the protocol to an *ideal* execution which has black-box access to the functionality which the protocol implements. In the real execution, the adversary's view consists of the inputs and received messages of the corrupted parties. In the simulated execution, a simulator who has access only to the corrupted parties' inputs and outputs to the computation, generates a simulated view for the adversary. The protocol is secure if the views of the adversary in the real and ideal executions are indistinguishable. Intuitively, this is because the adversary cannot learn anything new in the simulated execution–the simulator only has access to information the adversary already holds–and therefore $\mathcal{A}$ also cannot learn anything new in the real execution, otherwise it would know it wasn't in the simulated execution.

We will consider three types of indistinguishability. Let $A$ and $B$ be probability distribuitions. $A$ and $B$ are *computationally* indistinguishable, denoted $A \approx_c B$, if any probabilistic algorithm running in time $2^{O(\kappa)}$ has advantage $2^{-\Omega(\kappa)}$ at guessing which distribution is which, where $\kappa$ will always represent our computational security parameter. $A$ and $B$ are *statistically* indistinguishable, denoted $A \approx_s B$ if the statistical distance between them is at most $2^{-\sigma}$, where $\sigma$ always represents our statistical security parameter. $A$ and $B$ are *perfectly* indistinguishable, denoted $A \approx_p B$ if the two distributions are equal (i.e. $A = B$). Since our DORAM has an input of size $n$, it is assumed that $\kappa = \omega(\log(n))$ and $\sigma = \omega(\log(n))$.

We are now ready to define simulation-based security. Let $s$ be the number of parties. Let $f : (\{0,1\}^*)^s \to (\{0,1\}^*)^s$ be a $s$-input, $s$-output, potentially randomized functionality. Let $\pi$ be a $s$-party protocol, in which party $P_i$ has input $x_i$. Let $x = (x_0, \ldots, x_{s-1})$ and let $|x| = \sum_{i \in \{0,\ldots,s-1\}} |x_i|$ be the total length of the input. Let $OutputAndView(\pi, x, u)$ run a (typically randomized) execution

of $\pi$ on input $x$ and output a tuple consisting of (a) the output of $\pi(x)$ in that execution and (b) the view of parties $u \subset \{0, \ldots, s-1\}$ in that same execution, that is set of inputs and all received messages for parties in $u$. For $u \subset \{0, \ldots, s-1\}$, $x_u$ is the subset of $x_i$, for which $i$ is in $u$. A subset $u$ is *corruptible* if the adversary $\mathcal{A}$ is able to simultaneously corrupt all $P_i$ where $i \in u$. $u$ is a *maximal corruptible* set if it is corruptible and is not a subset of another corruptible set.

**Definition 1.2.1** (Secure MPC Implementation of a Functionality)**.** Given $U$, some set of maximal corruptible sets of parties. Let $\approx \in \{\approx_c, \approx_s, \approx_p\}$. $\pi$ is a $U$-secure implementation of $f$ if there exists a simulator $S$, such that for all inputs $x$, and all maximal corruptible sets, $u \in U$:

$$(y, view_u) \leftarrow OutputAndView(\pi, x, u) \approx (y \leftarrow f(x), view_u \leftarrow S(1^{|x|}, u, x_u, y_u))$$

If $\approx$ is $\approx_c$ (resp. $\approx_s$, $\approx_p$) then the protocol is computationally (resp. statistically, perfectly) secure.

The definition above assumes that corruptions are *static* ($\mathcal{A}$ chooses which party is corrupt at the beginning of the protocol) and are *semi-honest* (a corrupted party will still follow the protocol). This will be the case for our DORAM protocols. In fact, our protocols are all honest-majority 3-party protocols, wthich means $U$ will always be $\{\{0\}, \{1\}, \{2\}\}$. We will refer to such a protocol as a $(3, 1)$-secure protocol.

Definition 1.2.1 applies to static functionalities, in which the functionality does not retain any state between executions. However, in the context of DORAM, the state of the memory must be retained between executions. Our protocols therefore must implement a *reactive* functionality, that is the functionality is used several times and the behaviour of later queries to the functionality depend on previous queries. In this case, the security definition should allow for iterative queries of the functionality. In particular, the inputs to future queries must be able to depend on the adversary's view of previous queries. We define the security of a *reactive functionality* using the following definition, which is inspired by Definition A.1 of [AKL$^+$20].

**Definition 1.2.2** (Secure MPC Implementation of a Reactive Functionality)**.** Let $U$ be the set of

maximal corruptible sets of parties. Let $F$ be a reactive functionality, that takes $s$ inputs, returns $s$ outputs and maintains a state. Let $\Pi$ be a stateful protocol. Let $\mathcal{A}$ be a stateful adversary. Let there be two distributions of executions, $Exec_{\mathcal{A}}{}^{real,\Pi}$ and $Exec_{\mathcal{A}}{}^{ideal,F,S}$ as defined below.

| $Exec_{\mathcal{A}}{}^{real,\Pi}$ | $Exec_{\mathcal{A}}^{ideal,F,S}$ |
|---|---|
| $i = 0$ | $i = 0$ |
| $com^i, x^i \leftarrow \mathcal{A}$ | $com^i, x^i \leftarrow \mathcal{A}$ |
| While $com^i \neq \perp$ | While $com^i \neq \perp$ |
| $(y^i, view_u^i) = OutputAndView(\Pi, x^i, u)$ | $y^i = F(x_i)$ |
| | $view_u^i = S(1^{\lvert x^i \rvert}, u, x_u^i, y_u^i)$ |
| $i = i + 1$ | $i = i + 1$ |
| $(com^i, x^i) \leftarrow \mathcal{A}(view_u^{0,\ldots,i-1})$ | $(com^i, x^i) \leftarrow \mathcal{A}(view_u^{0,\ldots,i-1})$ |

$\Pi$ is a $U$-secure implementation of $F$ if there exists a (potentially stateful) simulator $S$, such that for all inputs $x$, and all maximal corruptible sets, $u \in U$, the distribution $(y^{0,\cdots}, view_u^{0,\cdots})$ of the real execution is indistinguishable from that of the ideal execution. The protocol is computationally (resp. statistically, perfectly) secure if the distributions are computationally (resp. statistically, perfectly) indistinguishable.

The above definition is quite strong. For instance, $\mathcal{A}$ is allowed to select the inputs to each call of the functionality. Furthermore, it can select these dynamically based on its view up to this point.

It is common for simulators to have oracle access to an instantiation of the adversary's code which they use in order to simulate the adversary. For our simulators this will not be necessary. In our proofs, the simulator will make use of the protocol instructions, which are public and which the adversary will follow since it is semi-honest, but it will not send requests to an instance of $\mathcal{A}$ itself. Therefore, our simulators are automatically "straight-line" and "black-box" as per the definitions of [KLR10]. Therefore, with the minor addition of start synchronization which we will omit from our protocols for conciseness, our protocols are universally composable (Theorem 1.5 of [KLR10]). In particular, our DORAMs are secure despite the fact that they use several sub-functionalities in parallel.

$F_{RAM}$: **RAM**

$Init(n, d)$: Initialize an array $A$ containing $n$ elements of size $d$, initially set to $(0^d)^n$.
$Access(op, i, y)$: If $(op = read)$ return $A_i$
If $(op = write)$ set $A_i = y$.

Figure 1.1: Functionality for RAM

The definition of DORAM follows naturally from the definition of the RAM functionality. DORAMs typically also hide whether a read or a write is performed, we define a single function *Access* which takes an argument $op \in \{read, write\}$.

A minor modification is needed to the RAM functionality above before it can be used to define DORAMs. The functionality does not receive the parameters of the Access function from all the parties, or from any one party, but rather receives secret-sharings of these parameters. A secret-sharing is defined as follows:

**Definition 1.2.3** (Secret-Sharing)**.** Let there be $s$ parties, and let $U$ represent the maximal corruptible subsets of $\{0, \ldots, s-1\}$. Let *Share* be a randomized function that maps $X \to X^s$ and let *Reconstruct* be a deterministic function that maps $X^s \to X$. *Share* and *Reconstruct* define a $U$-secure secret-sharing scheme if $\forall x \in X$, $Reconstruct(Share(x)) = x$ with probability 1 and $\forall u \in U$, $\forall x_a, x_b \in X$, $Share(x_a)_u$ and $Share(x_b)_u$ have the same distribution. We use $[[x]]$ to denote a secret-sharing of $x$.

$[[x]]$ is a *fresh* secret-sharing of $x$ if its random distribution is independent of all previous distributions.

The secret-sharing version of the RAM functionality can now be defined.

$F_{SSRAM}$ : **Secret-Shared RAM**

Let $[[\cdot]]$ represent a $U$-secure secret-sharing over $s$ parties.
$Init(n, d)$: Given public parameters $n$ and $d$, initialize an array $A$ containing $n$ elements of size $d$, initially set to $(0^d)^n$.
$Access([[op]], [[i]], [[y]]])$: If $(op = read)$ return a fresh $[[A_i]]$
If $(op = write)$ set $A_i = y$.

Figure 1.2: Functionality for Secret-Shared RAM

**Definition 1.2.4** (Distributed Oblivious RAM)**.** A Distributed Oblivious RAM (DORAM) is a Secure MPC implementation of the Secret-Sharing RAM functionality, where the Secret-Sharing RAM functionality uses a $U$-secure secret-sharing over $s$ parties.

## 1.3. Oblivious RAM

### 1.3.1. Defining the Problem

Distributed Oblivious RAM is closely related to the problem of Oblivious RAM. This problem was first formulated by Goldreich in the context of software protection [Gol87]. Consider a software vendor who wishes to hide information about the internals of the software implementation from a customer. As a first approximation to this problem, Goldreich considered the case where the customer (adversary) could view the contents of physical memory, but could not view the contents of the CPU. Goldreich considered whether it was possible to hide all information about the behavior of the program from such an adversary, apart from the number of memory accesses performed. The CPU could perform operations unknown to the adversary, including accessing a small amount of memory stored in registers: enough to hold a constant number of words of data, as well as a constant number of cryptographic keys.

Clearly the contents of the data stored in memory could be encrypted under a semantically-secure encryption scheme, with the encryption key stored in the CPU's registers. However, the adversary could still learn the pattern of accesses to memory. Goldreich proposed an Oblivious RAM (ORAM) as an intermediatary between the program and the main memory. For each memory access requested by the program, which is referred to as a *virtual access*, the ORAM would perform several accesses

to the untrusted memory, or *physical accesses*. The ORAM was correct if it could provide a RAM functionality to the program; it was secure if the physical access pattern revealed no information about the virtual access pattern, apart from the number of accesses.

While this problem may seem esoteric, as an adversary can often learn about the contents of a CPU in its possession, this exactly matches the adversarial model of secure enclaves such as Intel SGX [CD16]. Here, an adversary is unable to view data within the enclave itself, however the enclave has very limited memory and must therefore store data in the device's main memory, in which $\mathcal{A}$ can observe accesses. While the program performed by the secure enclave may be public, its behavior may depend on sensitive data, which in particular should not be leaked by the enclave's access pattern to main memory. ORAMs therefore exactly solve the problem of providing RAM to secure enclaves, and have been used for this purpose [SGF17] [AKSL18] [AJX+19].

Security of an ORAM is usually formalized (e.g. [PR10], [SvDS+13]) by stating that, for any two equal-length virtual memory access patterns, the distributions of physical access patterns should be indistinguishable (either computationally, statistically, or perfectly so). Instead, we follow [PPRY18] and [AKL+20] and define the security of ORAMs using the simulation paradigm, as we did for DORAMs. The security definition of ORAMs is somewhat simpler than DORAMs since the view of the adversary consists only of the physical access pattern. For the sake of simplicity, our definition will not allow $\mathcal{A}$ to view the *contents* of data in memory, but only the physical access pattern. Security against an adversary that can also view the contents of memory can be achieved simply by the ORAM encrypting all contents prior to storing them in memory using a semantically-secure encryption scheme.

Obliviousness can be defined by modifying and simplifying the generic MPC definition. Firstly, instead of a protocol, we are concerned with a program, $M$, running on a single (stateful) machine. $\mathcal{A}$'s view consists only of the RAM access pattern by this machine; let $Addrs(M(x))$ represent the memory access pattern by $M$ on input $x$. Let $OutputAndAddrs(M(x))$ return the output and access pattern by $M$ on input $x$.

**Definition 1.3.1** (Oblivious Implementation of a Reactive Functionality)**.** Let $F$ be a reactive functionality. Let $M$ be a stateful machine. Let $\mathcal{A}$ be a stateful adversary. Let the real and ideal executions be defined as follows:

| $Exec_{\mathcal{A}}^{real,M}$ | $Exec_{\mathcal{A}}^{ideal,F,S}$ |
|---|---|
| $i = 0$ | $i = 0$ |
| $com^i, x^i \leftarrow \mathcal{A}$ | $com^i, x^i \leftarrow \mathcal{A}$ |
| While $com^i \neq \perp$ | While $com^i \neq \perp$ |
| $\quad (y^i, view^i) = OutputAndAddrs(M, x^i)$ | $\quad y^i = F(x_i)$ |
| | $\quad view_u^i = S(1^{\lvert x^i \rvert}, com^i)$ |
| $\quad i = i + 1$ | $\quad i = i + 1$ |
| $\quad (com^i, x^i) \leftarrow \mathcal{A}(view^{0,\ldots,i-1})$ | $\quad (com^i, x^i) \leftarrow \mathcal{A}(view^{0,\ldots,i-1})$ |

$M$ is an oblivious implementation of $F$ if there exists a (potentially stateful) simulator $S$, such that the distribution of $view^{0,\cdots}$ in the real execution is indistinguishable from that in the ideal execution. The protocol is computationally (resp. statistically, perfectly) secure if the distributions are computationally (resp. statistically, perfectly) indistinguishable.

The definition of ORAM follows immediately from this definition and our previous definition of the RAM functionality.

**Definition 1.3.2** (Oblivious RAM)**.** An Oblivious RAM (ORAM) is an Oblivious Implementation of the RAM functionality.

An Oblivious RAM can easily and efficiently be converted into a DORAM. Rather than receiving memory access requests from a CPU program, the DORAM receives secret-shared access requests. Rather than accessing a physical memory on a single machine, the DORAM can make use of the physical memory of the parties. It could do this by encrypting the contents of memory and using a single party to store the contents of memory. Alternatively, the contents of memory can be secret-shared between the parties, which is usually more efficient and allows for statistical or perfect security. Finally, the ORAM itself is simulated inside of a secure computation. Since the ORAM

only uses a small amount of memory, the ORAM protocol can be efficiently represented as a circuit, and theirfore can be securely implemented using a generic circuit MPC protocol. Security comes, quite simply, from the fact that the parties learn nothing by their participation in the ORAM simulation, due to the security of the generic MPC protocol, and learn nothing from their role as data holders, since the data is encrypted or secret-shared, and the ORAM security ensures that the addresses accessed leak no information.

### 1.3.2. Efficiency metrics

Achieving memory accesses obliviously naturally incurs overhead. Assuming that memory is arranged in $d$-bit blocks, a non-oblivious access would require accessing only 1 $d$-bit block of memory. The *memory access cost* of an ORAM is the number of bits of physical memory that need to be accessed to perform one virtual memory access. Since data in memory is arranged in blocks, it often makes sense to instead consider the *overhead* of an ORAM, which is the number of *blocks* of physical memory that must be accessed to perform one virtual memory access. The memory access cost depends on $d$, $n$, the number of blocks of virtual memory, and can also depend on $\kappa$ and $\sigma$, the computational and statistical security parameters. In the case that the memory access cost has terms that do not depend on $d$, then for large block sizes these terms can become asymptotically irrelevant, as such, some ORAMs achieve certain overheads subject to certain block sizes.

It is assumed typically that $d \geq \log(n)$–that is a block at least holds enough bits to store its own address. It is also usually assumed that $\sigma = \omega(\log(n))$ (the leakage should be negligible in the size of the input) and that $\kappa = \omega(\log(n))$ (the protocol should be secure against any adversary who performs work polynomial in $n$).

ORAMs are primarily evaluated based on the amortized overhead over many (e.g. $n$) accesses. However, it is often also important to consider the worst-case overhead for an access. This may be significantly larger than the amortized overhead as some ORAM protocols have infrequent, but expensive, rebuild phases. Another metric is the *memory overhead*, that is blow-up in the amount of physical memory required, compared to a non-oblivious RAM which only needs $nd$ bits.

## 1.4. Prior Work

In this section we present a brief history of ORAM, Multi-Server ORAM and DORAM protocols. This literature is extensive, so we do not discuss all protocols in detail. Rather, we present general techniques which led to significant improvements to protocols, and also present tables which summarize significant developments.

### 1.4.1. The First ORAMs

The most basic ORAM is for the CPU to perform a linear scan of the memory during each access. Each item which is not the one queried is simply written back, for the item that is queried the value is retained (for a read) and the updated value is written back (for a write). This is clearly oblivious, and perfectly so, but has overhead $\Theta(n)$.

When Goldreich introduced the ORAM problem he also proposed a solution that has $\Theta(\sqrt{n}\log(n))$ overhead [Gol87]. Each data element was stored in memory according to a Pseudo-Random Permutation (PRP), $\pi_k$, where $k$ is the cryptographic key which was stored by the CPU. To access an item with index $i$, the CPU would simply evaluate $\pi_k(i)$ and request location $\pi_k(i)$ from memory. Due to the security of the PRP, this would be indistinguishable from a random location. However, an item could not be re-queried in the main memory without leaking the fact that the same item was being accessed again. Therefore, each queried item would be stored in a cache of size $\sqrt{n}\log(n)$. In each query, the protocol would first perform a linear scan of the cache. If the item $i$ was not found in the cache, the CPU would request location $\pi_k(i)$ from the memory. If the item *was* found in the cache, the CPU would instead request location $\pi_k(n + cnt)$ from the main memory, where $cnt \in \{1, \ldots, \sqrt{n}\log(n)\}$ was a counter incremented after each query. The protocol pre-inserted "dummy" items with indices $n+1, \ldots, n+\sqrt{n}\log(n)$ in the original build of the main memory, this ensured that each request to main memory accessed a unique element. After $\sqrt{n}\log(n)$ accesses, the cache would be full and the memory would need to be permuted again using a new PRP. The permutation could be performed using a Waksman network [Wak68], which can sort $n$ elements obliviously by performing $\Theta(n\log^2(n))$ deterministically-defined pair-wise sorts. This cost, when amortized over $\sqrt{n}\log(n)$ accesses results in $\Theta(\sqrt{n}\log(n))$ physical accesses per virtual access.

### 1.4.2. Hierarchical ORAMs

Ostrovsky then introduced the hierarchical approach to ORAM, and showed that this allowed for ORAMs with polylogarithmic overhead [Ost90]. Since the hierarchical paradigm is fundamental to our DORAM protocols of chapters 4 and 5, as well as to understanding the attack presented in chapter 3, we describe it in detail, and prove it secure, in the following section. We here give a brief overview.

Ostrovsky considered a weaker primitive that ORAM, called an Oblvious Hash Table. An Oblivious Hash Table, or OHTable, is a method of storing and retrieving data such that the phsyical access pattern leaks no information about the virtual access pattern *provided each item is accessed at most once.* An OHTable can be created, for instance, by using a regular bucket-based hash table scheme and padding all buckets to be of length $\Theta(\log(n))$. If the hash function used is a PRF, the bucket accessed during a query reveals nothing regarding which index was queried, as long as it is the first time that item has been queried.

An OHTable may seem to be a much weaker primitive than an ORAM, but Ostrovsky showed that it was possible to build an ORAM using $\log(n)$ OHTables. In short, the ORAM, initially, can be stored in a single OHTable. Then, each time the OHTable is queried, the result is cached into an instance of an ORAM[1] with smaller capacity, for instance of half the size. This sub-ORAM is always queried first, and if the item is found in the sub-ORAM, a random location is accessed in the OHTable. This ensures that the queries to the OHTable always involve a random access. Since the sub-ORAM is smaller, its contents must periodically be extracted and rebuilt into the OHTable. By implementing the sub-ORAM recursively using the same solution, an ORAM could be built, which would result in several OHTables of geometrically increasing size. These OHTables could be envisioned as arranged in a pyramid, or hierarchy, with smaller OHTables at the top [Ost92] [GO96]. This hierarchy would contain $\Theta(\log(n))$ levels. The cost was dominated by the cost of building Oblivious Hash Tables at each level. The average cost per access for each level was

[1] Actually, instead of an ORAM, data must be cached into an Oblivious Dictionary, which is similar to an ORAM but allows for queries from a larger index space than its size. See section 2.3 for a more complete explanation.

$\Theta(\log^2(n))$, The first $\log(n)$ term comes from the fact that an OHTable with $m$ real items needed $\Theta(\log(n)m)$ spaces due to the bucket padding. The second $\log(n)$ term comes from the fact that building required sorting these $\Theta(m\log(n))$ items, using the AKS sorting network [AKS83] this cost became $\Theta(m\log(n)\log(m\log(n)))$ which is dominated by the cost of the large levels $m = n^c$, which is $\Theta(m\log^2(n))$. Since a level was rebuilt every $\Theta(m)$ accesses, the cost per query was $\Theta(\log^2(n))$ per level, or $\Theta(\log^3(n))$ over the full ORAM.

The Hierarchical paradigm was built upon by several works. These replaced the OHTables that either were smaller (removing the need for asymptotic blow-up from padding) or were more efficient to build (avoiding the cost of a full oblivious sort).

Pinkas and Reinman [PR10] proposed implementing the Oblivious Hash Tables using Cuckoo Hashing, rather than bucket hashing. Cuckoo Hashing uses a small constant number of locations, typically 2 [PR04]. In order to store $m$ elements, a cuckoo hash table only needs $(2+\epsilon)m$ locations of size 1. Compared to the original Hierarchical ORAM built on bucket hashing, which had $m$ buckets of size $\Theta(\log(n))$, this was a $\Theta(\log(n))$-factor reduction in the size of the Oblivious Hash Tables, and therefore in the cost of builds and in the total ORAM overhead.

Unfortunately, the original cuckoo-hashing scheme had an error, as was observed by Goodrich and Mitzenmacher [GM11] and independently by Kushilevitz, Lu and Ostrovsky [KLO12]. Cuckoo Hashing has a non-negligible failure probabilty: when trying to place $m$ items in a table of size $\Theta(m)$, the probability that no satisfiable assignment exists is $\Theta(\frac{1}{m})$. For most applications this is not an issue as the table can simply be rebuilt in this unlikely eventuality. However, in the ORAM setting, rebuilding can result in leakage. Say Cuckoo hashing is used with 2 hash functions. Say 3 items which are not stored in a table are queried, and by chance all 3 are hashed to the same 2 locations. This will happen with probability $\Theta(m^{-4})$. In this case, it is clear that the Hash Table does not store all three of these items, breaking the properties that an OHTable should not leak information about whether queried items are stored.

Goodrich and Mitzenmacher suggested using a modification of cuckoo hashing in which any items

that could not be stored in the main table could be placed in a "stash" [GM11]. If the stash is of size $\log(n)$, the failure probability becomes negligible in $n$ for sufficiently large table sizes ([GM11] Appendix C, [Nob21]). To avoid increasing the amortized overhead, Goodrich and Mitzenmacher suggested that the stashes from multiple levels could be combined. Kushilevitz et al. proposed a similar idea, cuckoo hashing with a stash could be used and the stashes could be re-inserted to the top level of the hierarchy [KLO12]. In Chapter 3 we show that these approaches both introduce a subtle flaw which allow virtual access patterns to be distinguished with non-negligible probability. This flaw was inherited by several subsequent works [LO13a, CGLS17, PPRY18, AKL$^+$20]. We also show a simple fix that does not increase the asymptotic cost of these protocols.

Kushilevitz et al. realized that the costs of Hierarchical ORAMs were dominated by the rebuild cost [KLO12]. For instance, for cuckoo-hashing based ORAM, the overhead from accesses was only $\Theta(\log(n))$ (accessing a constant number of items from $\log(n)$ tables) however the overhead from rebuilding was $\Theta(\log^2(n))$. They observed that the cost could be reduced by *balancing* the cost of accesses and rebuilds. If each level of the hierarchy had multiple OHTables, this would reduce the number of levels, and therefore the cost of rebuilds, with the trade-off that it would increase the number of OHTables. Setting the number of OHTables per level to $\log(n)$ resulted in an amortized per-access cost of $\Theta(\log(n)\log_{\log(n)}(n)) = \Theta(\log^2(n)/\log(\log(n)))$.

Chan et al. presented an ORAM based on another type of oblivious hash table, called two-tier hashing [CGLS17]. It built a hash table of $m$ elements using two layers of bucket-hashing. Each layer held $m/\log^a(\lambda)$ buckets of size $O(\log^a(\lambda)$, where we desire failure probability negligible in $\lambda$ and $a \in (0.5, 1)$ is fixed. Items were initially hashed into the first layer, and any leftover items were hashed into the second layer. They showed that two-tier hash tables had an oblivious build process that was much more simple than that of Cuckoo hashing. Their resulting scheme had amortized per-access cost of $\Theta(\log^2(n)/\log(\log(n)))$.

Patel et al. [PPRY18] observed that employing a full shuffle to combine OHTable contents was excessive. Since the contents of each OHTable were already randomly permuted, it was enough to use a *multi-array shuffle* which produced a random output when given arrays that were each individually

shuffled. They constructed a new oblivious multi-array shuffle, and used this to construct an ORAM with overhead $\Theta(\log(n)\log(\log(n)))$.

Asharav et al. [AKL+20] then showed that it was possible to build an ORAM with $\Theta(\log(n))$ overhead. This matches the lower bound which had been proved by Goldreich and Ostrovsky in the "balls-and-bins" model [GO96] and had been proven by Larsen and Nielsen for arbitrary data representations [LN18]. As a building block, Asharov et al. showed that two randomly permuted arrays of size $m$ could obliviously be combined into a new randomly permuted array using only $\Theta(m)$ operations, effectively removing the $\log(n)$ term in the original overhead that had come from oblivious sorting.

These hierarchical ORAMs are presented in Table 1.1. It is noteworthy that since it was introduced, all breakthroughs to the amortized complexity cost of ORAMs have come through results in the hierarchical paradigm. However, due to their reliance on oblivious shuffling (and its variants) these protocols often have very large implicit constants. These ORAMs use hash functions to build their OHTables; in the absence of a random oracle this requires introducing the (minimal) computational assumption that one-way functions exist. While hierarchical ORAMs, when implemented in a basic manner, have a high worst-case cost due to rare, but expensive, rebuilds of large tables, techniques exist to de-amortize this cost [OS97] [AKLS23].

### 1.4.3. Tree ORAMs

Another important paradigm for ORAM constructions is the Tree ORAM, which was first discovered by Shi et al [SCSL11]. A Tree ORAM stores data in a tree, where each vertex is a bucket holding a number of *items*, where an item consists of an index with its corresponding data value. Each index is assigned a random leaf of the tree, this is stored in a data structure called a position map. The protocol maintains the invariant that the data for a given index always exists in the path between the root of the tree and the index's leaf. To query an index, the ORAM would access the entire path of that index and retrieve the corresponding data element. The index would be assigned a new path and placed in the root of the tree. To avoid congestion at the top of the tree, each query would be followed by a round of *evictions*, in which certain nodes would be randomly selected and

items within those nodes moved towards their target leaves. The position map can be implemented recursively, using an ORAM with $n/c$ indexes, each of size $c \cdot \log(n)$, for any constant $c \geq 2$. Relative to Hierarchical ORAMs, Tree ORAMs are arguably simpler to construct, and automatically have low worst-case cost. Furthermore, the resulting protocols are statistically secure.

The original Tree ORAM [SCSL11] used buckets of size[2] $\omega(1)\log(n)$. To evict, it randomly selected a constant number of nodes at each depth of the tree, and obliviously moved one of its items to a child bucket. In this case the probability that any bucket overflows is negligible in $n$. This required accessing $\omega(1)\log^2(n)d$ bits in the main tree (since there were $\log(n)$ buckets, each containing $\omega(1)\log(n)$ items of size $\Theta(d)$) and accessing $\omega(1)\log^4(n)$ bits in the recursive position map implementations. This resulted in an ORAM with cost $\Theta(\omega(1)\log^2(n)(d+\log^2 n))$.

A new eviction scheme was introduced by [SS12], in which items were evicted along a path, and each item was brought leafward as far as possible. It was first shown empirically, that this allowed the buckets to be of constant size [SS12], and then proven (with some protocol modifications) that the buckets could be of size $\Theta(\log(\log(n)))$ [CLP14] and then proven that the buckets could be of size $\Theta(1)$ [SvDS$^+$13]. However, the constructions of [CLP14] and [SvDS$^+$13] assumed a CPU that had internal memory of size $poly(\log(n))$ blocks. Modifying these constructions to have a constant-memory CPU results in asymptotic costs of $\Theta(\omega(1)\log^2(n)(d+\log^2(n)))$ and $\Theta(\omega(1)\log(n)\log(\log(n))(d+\log^2(n)))$ respectively. (See section 2.2 of [WHC$^+$14] for a more complete explanation.) Wang et al. [WCS15] then introduced an eviction protocol in which the CPU only needed $\Theta(1)$ blocks of memory, and showed that this too allowed constant-sized buckets with negligible overflow probability. It was also noted that the evictions paths could be chosen deterministically [GGH$^+$13], and this resulted in concretely better bounds on the overflow probability [WCS15].

Perfect ORAMs have also have been considered [DMN11] [RS19] [CSLN21], for which the distribution of physical accesses is identical regardless of the virtual access pattern.

[2]The original paper uses buckets of size $\log(n)$, but has failure probability $n^{-c}$ for constant $c$. Achieving negligible failure in this protocol requires buckets of size $\omega(1)\log(n)$.

| Work | Cost (Amortized/Expected) | Security type |
|---|---|---|
| Linear scan | $\Theta(nd)$ | Perfect |
| Square-root [Gol87] | $\Theta(\sqrt{n}\log(n)d)$ | Computational |
| Original Hierarchical [Ost90] [GO96] | $\Theta(\log^3(n)d)$ | Computational |
| Cuckoo Hashing [PR10] [GM11] | $\Theta(\log^2(n)d)$ | Computational |
| Damgård et al. [DMN11] | $\Theta(\log^3(n)d)$ | Perfect |
| Tree ORAM [SCSL11] [CLP14] | $\Theta(\omega(1)\log^2(n)(d+\log^2(n)))$ | Statistical |
| Balanced Cuckoo Hashing [KLO12] | $\Theta(\log^2(n)/\log(\log(n))d)$ | Computational |
| PathORAM [SvDS$^+$13] (o-sort) | $\Theta(\omega(1)\log(n)\log(\log(n))(d+\log^2(n)))$ | Statistical |
| Circuit ORAM [WCS15] | $\Theta(\omega(1)\log(n)(d+\log^2(n)))$ | Statistical |
| Two-tier ORAM [CGLS17] | $\Theta(\log^2(n)/\log(\log(n))d)$ | Computational |
| PanORAMa [PPRY18] | $\Theta(\log(n)\log(\log(n))d)$ | Computational |
| OptORAMa [AKL$^+$20] | $\Theta(\log(n)d)$ | Computational |
| Chan et al. [CSLN21] | $\Theta(\log^3(n)/\log(\log(n))d)$ | Perfect |

Table 1.1: "Classic" ORAMs in which the CPU has $\Theta(d+\kappa)$ memory

#### 1.4.4. ORAM variants and DORAMs

Originally ORAMs were created to hide the access pattern by a program from an adversary who was able to view the access pattern on that machine. However, as the internet took off and data moved off of devices and onto large centrally-controlled silos, ORAM quickly became a solution to another problem: secure outsourcing of memory. Imagine a client who wishes to store a large amount of data "on the cloud". The client can secure the contents of the data from the service provider simply by using encryption. However, this still leaks the access pattern to the service provider, which may contain sensitive information. ORAM is an immediate solution to this problem, with the service-provider taking the role of the main memory, the client's software taking the role of the program and the ORAM running on the client's device. Therefore, any ORAM in the original CPU model was also a solution to secure data outsourcing.

However, in the memory outsourcing application, it was reasonable to consider some variants of the model:

**Active ORAM:** Active ORAMs allow the server to *perform computation.* In the classic ORAM setting, the memory was passive, but in the memory outsourcing setting, servers have the ability to perform computation, and may in fact have stronger computational power than the client. Examples

include [OS97] [WS12] [AKST14] [DSS14] [FNR+15] [DvDF+16].

**Multi-Server ORAM:** Multi-Server ORAM allows there to be multiple non-colluding servers, each providing memory as a service. Examples include [OS97] [LO13a] [HOY+17] [AFN+17] [GKW18]; most of these Multi-Server ORAMs are also Active.

Active and Multi-Server ORAMs can easily be converted to DORAMs. Since the parties in a DORAM already perform computation, they can easily also perform computation on behalf of the servers. Likewise, since there are already multiple non-colluding parties in a DORAM protocol, having them take on the roles of different servers in a Multi-Server ORAM protocol is usually straight-forward. A DORAM is therefore very similar to a Multi-Server Active ORAM. However, a DORAM *has no trusted client*. Also, technically the definition of ORAM only allows corrupted servers to view the *access pattern* of queries to their data, but the parties in a DORAM can also see the *contents* of the data. Therefore, converting a multi-server ORAM to a DORAM requires (a) the parties to simulate the client in a MPC protocol and (b) the contents of the data held by servers to always be encrypted. More precisely, any $s$-server $t$-threshold active ORAM can be converted to a $(s, t)$-secure DORAM by the servers simulating the client using a $(s, t)$-secure MPC protocol, by each player taking the role of one server and by encrypting data prior to storing it on a server. Any $(s, t)$-secure DORAM can be converted into a $s$-server $t$-threshold active ORAM by the client secret-sharing each query between the servers and the servers sending the client the secret-shared response, which the client reconstructs.

The line between DORAMs and Multi-Server Active ORAMs is therefore a thin one. In terms of the security model, they are equivalent as each can be converted to the other. There can, however, be differences in the cost. When there is a trusted client, the client can efficiently perform operations locally at low cost. The most significant example of this is cryptographic operations, such as PRP evaluations or encryption. This was often considered as a single operation of computation in the trusted client setting; in the absence of a trusted client it requires evaluating a circuit of size $\Theta(\kappa)$ in a generic MPC computation, which requires $\Theta(\kappa)$ communication. Therefore, converting a Multi-Server Active ORAM to a DORAM may result in an increase in the communication cost. Going

the other way, a DORAM can be converted to a Multi-Server Active ORAM at no extra cost.

Therefore, when considering the best DORAMs, one must consider not only protocols that are designed specifically as DORAMs, but also Multi-Server and single-server ORAMs (both active and passive) accounting for the added cost of simulating the client. Many of the best DORAMs to date were, in fact, not designed to be DORAMs, but are Multi-Server ORAMs that have clients which are easy to simulate. Table 1.2 contains previous DORAMs which, to date, have the lowest communication cost (under given parameter ranges), all of which require conversions from multi-server ORAMs to DORAMs. We now describe these schemes in brief, and in particular detail where needed the modifications required to (efficiently) convert these to DORAM protocols. Lu and Ostrovsky presented a 2-server ORAM with only logarithmic overhead [LO13a]. Its client only performs PRF evaluations, encryptions and decryptions so is easy to simulate. Abraham et al. [AFN$^+$17] made use of 2-server PIR and the Tree paradigm to construct a statistically secure 2-server ORAM which, for many parameters, has sub-logarithmic overhead. However, converting this to a DORAM by the generic transformation would introduce cryptographic assumptions and efficiency losses due to the cost of encrypting and decrypting data held by the servers. Instead, the asymptotic communication cost and statistical security property can both be maintained by introducing more servers. For instance, this can be done by having each server in the original protocol be simulated by 2 parties who hold a secret-sharing of the original data (for the PIR to work, the secret-sharing must be the same for both pairs). Similarly, the client's role in the TwoServerPIR can be simulated efficiently by making use of an additional party and the SSPIR protocol of chapter 5. Chan et al. present a 3-server ORAM which is perfectly secure [CKN$^+$18] Unlike Abraham et al., Chan et al. consider the server's view to include the contents of data they hold. Their protocol is perfectly secure; as such the client is simple and should be able to be simulated with no overhead.

Which protocols have the lowest cost depends on parameter choices: the relationships between $n$, $d$ and $\kappa$. For instance, [LO13a] is best when $d = \log(n)$ and $\kappa = o(\log^2(n))$. [CKN$^+$18] is best when $d = \log(n)$ and $\kappa = \omega(\log^2(n))$. When $d = \sqrt{n}$, [AFN$^+$17] is best (with $a$ set to $\sqrt{n}/\log^3(n)$). This table focuses only on total communication cost, and therefore leaves out many DORAMs which are

| Protocol | Communication Cost (Amortized/Expected) | Security |
|---|---|---|
| Lu and Ostrovsky [LO13a] | $\Theta(\log(n)(\kappa+d))$ | Computational |
| Abraham et al. [AFN+17] ($a \geq 2$) | $\Theta(\log_a(n)d + a\omega(1)\log_a(n)\log^2(n))$ | Statistical |
| Chan et al. [CKN+18] | $\Theta(\log(n)(\log^2(n)+d))$ | Perfect |
| OSet DORAM (Chapter 4, [FNO22]) | $\Theta(\log(n)(\kappa+d))$ | Computational |
| MetaDORAM (Chapter 5, [FNO24]) | $\Theta(\log(n)/\log(\log(n))(d+\log^2(n))$ | Statistical |

Table 1.2: DORAM protocols. For DORAMs which are made by simulating a client in a secure computation, it is assumed that the cost is $\Theta(1)$ communication per AND gate of the client circuit.

efficient in practice and/or have low round complexity, but which have higher communication cost (e.g [FJKW15], [Ds17], [JW18], [KM19], [BKKO20], [HV21], [VHG23]) or require $\Omega(n)$ computation per query (e.g. [GKW18]).

The first work we present (Chapter 4) has communication cost $\Theta(\log(n)(\kappa + d))$. This is the same as that of [LO13a], but the constant is about 50x lower. Significantly, it maintains the abstraction of building a hierarchical (D)ORAM from Oblivious Hash Tables, whereas [LO13a] breaks this abstraction as their use of the stash re-insersion technique means that they do not have true Oblivious Hash Tables.

The second work we present (Chapter 5) achieves communication cost $\Theta(\log(n)/\log(\log(n))(d + \log^2(n)))$. This is strictly better than [CKN+18], is better than [LO13a] when $\kappa = \omega(\log^2(n)/\log(\log(n)))$, and is better than [AFN+17] when $d = O(\log^2(n))$.

1.4.5. Other ORAM variants and metrics

In the client-server setting, many works considered a variant with *large client memory*. In the classic ORAM setting, the ORAM must be implemented inside the CPU, so only has access to a small number of registers. Classic ORAMs therefore restrict the ORAM to only store a constant number of blocks and cryptographic keys. However, in the memory outsourcing setting, the ORAM is executed by a client device, such as a smartphone or laptop. In this case the client can reasonably store a significant amount of data, less than $nd$, but perhaps $\sqrt{n}d$ or $\log(n)d$. Examples include [WSC08], [WS12], [SSS11], [SvDS+13], [GGH+13], [CLP14], [Goo17], [AKM23]. The super-constant client memory variant, while an important theme in the literature, is *not* easily applicable to the

DORAM setting, since the client must be simulated inside of a generic MPC protocol.

There are some other variations of ORAMs which have been researched, but which are beyond the scope of this review. Oblivious Parallel RAMs have the ability to handle ORAM queries from multiple CPUS/clients in parallel [BCP15] [CLT16] [CCS17] [CGLS17] [CNS18] [CSLN21] [AKL+22]. Garbled RAM allows RAM to be accessed within a garbled circuit [LO13b] [GHL+14] [GLOS15] [GLO15] [CH16] [CCHR16] [HKO22] [PLS23]. Private Information Retrieval (PIR) is a problem with some similarities to Oblivious RAM, which was introduced by Chor et al. [CKGS98] and has been studied extensively since. The main differences between ORAM and PIR are that PIR typically is concerned with only reads and not writes, assumes the server(s) are aware of the data and considers the cost of single accesses rather than the amortized cost of many accesses. Obliviousness has also been considered for data-structures which are weaker than RAM, such as stacks, queues and heaps [KS14] [MZ14] [Shi20], or RAM in which the indexes accessed are generated randomly by the protocol [HK22].

The client-server ORAM application also influenced the emphasis on efficiency metrics. Due to the high latency of distributed networks, the *round complexity*, that is the number of dependent interactions between the client and server(s) per query, became an important factor. DORAMs also operate in a network which may have high latency, so their performance in practice often depends significantly on round complexity. ORAMs have achieved constant round complexity [WS12] [GMP16], as have several DORAMs [KM19] [BKKO20] [HV21] [VHG23]. Additionally, in the client-server ORAM setting, the size of blocks is much larger than the word-size of a CPU, so it is reasonable to assume that the block sizes are large. In a DORAM, however, the block sizes are usually still small, since they represent particular variables queried by a MPC program.

# CHAPTER 2

# The Hierarchical Template

## 2.1. Overview

This section formally presents the Hierarchical Template, a protocol for achieving a (D)ORAM using a simpler protocol called a (Distributed) Oblivious Hash Table. In Chapter 3 we show an attack in which several ORAMs and DORAMs followed the general paradigm of the hierarchical solution, but deviated from the Hierarchical Template in a manner which led to non-negligible leakage. Chapter 4 uses the Hierarchical Template to build an efficient DORAM. Chapter 5 also presents an efficient and secure DORAM, which uses ideas from the hierarchical paradigm, but does not exactly follow the Hierarchical Template.

First, we introduce some useful notation which we will use throughout this work. We then present, in detail, the Hierarchical Template, in the context of ORAM, and prove it to be secure. Finally, we present the Hierarchical Template in the context of DORAM, prove it secure and discuss its efficiency.

## 2.2. Notation and Terminology

We use lower-case letters to represent parameters and variables. $n$ denotes the size of an ORAM/DORAM, or the size of the index space for an OMap/DOMap/OHTable/DOHTable.
$d$ denotes the bit length of the data values, that is each data values is from $\{0,1\}^d$.
$m$ denotes the capacity of an OMap/DOMap/OHTable/DOHTable.
$c$ denotes the cache size, $b$ the ratio of expansion for the hierarchy and $l$ the number of levels of the hierarchy.
$\kappa$ is the computational security parameter.

Arrays are represented using upper-case letters. $C$ is the cache. $X$ is an array of indices and $Y$ is an array of values. $Z$ is an array of index-value pairs.

An index-value pair, stored in a data-structure is referred to as an *item*. The item itself is thought

| Sharing type | Notation | Party Share | | | Construction |
|---|---|---|---|---|---|
| | | $P_0$ | $P_1$ | $P_2$ | |
| 3RSS (Replicated) | $[[x]]$ | $(x_0, x_1)$ | $(x_1, x_2)$ | $(x_2, x_0)$ | $x_0 \oplus x_1 \oplus x_2 = x$ |
| 3XORS (3-Party XOR) | $[[x]]_{0,1,2}$ | $x_0$ | $x_1$ | $x_2$ | $x_0 \oplus x_1 \oplus x_2 = x$ |
| 2XORS (2-Party XOR) | $[[x]]_{1,2}$ | $\emptyset$ | $x_0$ | $x_1$ | $x_0 \oplus x_1 = x$ |
| 1-2XORS (1-and-2 Party XOR) | $[[x]]_{0,(1,2)}$ | $x_0$ | $x_1$ | $x_1$ | $x_0 \oplus x_1 = x$ |
| 2-Priv (2-Party Private) | $[[x]]_{(1,2)}$ | $\emptyset$ | $x$ | $x$ | |
| 1-Priv (1-Party Private) | $[[x]]_0$ | $x$ | $\emptyset$ | $\emptyset$ | |
| Public | $x$ | $x$ | $x$ | $x$ | |

Table 2.1: Types of Secret-Sharing with Notation

of as an object that moves around due to queries and rebuilds. Even if the value is modified due to a write, it is still considered to be the same item, and if the index is $x$, the item is often referred to simply as item $x$, or just $x$.

For $a$ some positive integer, $[0, a-1]$, represents the set $\{0, \ldots, a-1\}$. $\log(\cdot)$ denotes the base-2 logarithm except where another base is explicitly provided.

We use several kinds of secret-sharing, all of which are bit-wise (Boolean) secret-sharings. These are summarized in Table 2.1.

The most common sharing we use is the 3-party replicated secret sharing (3RSS) of Araki et al. [AFL+16] (see also [CDI05]). Here, $x \in \{0,1\}^\ell$ is secret-shared by having $x_0, x_1, x_2 \in \{0,1\}^\ell$ that are uniformly random subject to $x_1 \oplus x_2 \oplus x_3 = x$. $P_i$ holds $x_i$ and $x_{(i+1) \bmod 3}$. When variable $x$ is held using this secret-sharing, it is represented as $[[x]]$. Operations (AND, OR, NOT, XOR) are performed on this secret-sharing using the methods of Araki et al [AFL+16]. Another standard sharing that we use is the 3-party XOR sharing (3XORS), that is $P_i$ holds $x_i$ where $x_0 \oplus x_1 \oplus x_2 = x$.

We also use a 2-party XOR secret-sharing (2XORS), where 2 parties hold the secret-sharing and the third party is not involved. If $P_1$ and $P_2$ hold a 2-party XOR secret-sharing of variable $x$, this is denoted as $[x]_{1,2}$. $P_1$ holds $x_0$ and $P_2$ holds $x_1$ where $x_0, x_1$ are chosen uniformly at random subject to $x_0 \oplus x_1 = x$. We also use a variant of XOR secret-sharing in which 2 parties hold one share, and the third party holds the other (1-2XORS). For instance, when $P_0$ holds one share, and $P_1$ and $P_2$

hold the other share, this is denoted $[[x]]_{0,(1,2)}$, that is $P_0$ holds $x_0$, and $P_1$ and $P_2$ both hold $x_1$ where $x_0, x_1 \leftarrow \{0,1\}^\ell$ subject to $x_0 \oplus x_1 = x$.

Sometimes a variable is held privately. If $x$ is held privately by one party (1-Priv), for instance, by $P_0$, we denote this as $[[x]]_0$. Sometimes a variable is known to 2 parties but not the third (2-Priv). If $x$ is known to $P_1$ and $P_2$, but not $P_0$, this is denoted $[[x]]_{(1,2)}$. Note that if variables are held privately or using 2XORS the involved party/parties can perform operations on them without the uninvolved party/parties knowing which variables were being operated on. For instance, if $[[i]]_{(1,2)}$ is an index held privately by $P_1$ and $P_2$, and $[[A]]_{1,2}$ is an array secret-shared between $P_1$ and $P_2$, then $P_1$ and $P_2$ can set some variable to be the $i^{\text{th}}$ value in $A$, without $P_0$ learning which index was used.

Table 2.1 holds a summary of these secret-sharings with their notation. These sharings can easily be converted between each other. A sharing of an $l$-bit variable can be converted to a fresh sharing of any other $l$-bit variable by each party creating a fresh sharing of their share in the new sharing and XORing the resulting shares. (If 2 parties hold the same share, only one needs to send a sharing.) This requires only $\Theta(l)$ bits of communication.

We present protocols on secret-sharing using regular programming pseudocode. The communication between parties needed to achieve this is implicit. The following operations are supported:

- **Bitwise XOR ($[[q]] \oplus [[r]]$):** All of our secret-sharing protocols are linear (over bitwise XOR), so allow for local bitwise XOR operations.
- **Bitshifts/Selects:** Bits can easily be selected since the secret-sharings are bitwise.
- **Bitwise AND/OR ($[[q]] \wedge [[r]]$/ $[[q]] \vee [[r]]$):** For 3RSS-shared variables, this is done using the protocol of Araki et al. [AFL+16]. For any other secret-sharing, the inputs are first implicitly cast to a 3RSS sharing. In either case, for an $l$-bit integer, this requires $\Theta(\ell)$ bits of communication
- **Equality/Inequality tests ($[[q]] = [[r]]$/ $[[q]] \leq [[r]]$):** An equality/inequality test on a

$\ell$-bit integer can be performed using a circuit of $\Theta(\ell)$ AND gates, so only needs $\Theta(\ell)$ bits of communication.

- **If/Else Statements:** Given a secret-shared condition (represented as a bit), performs certain actions if true, and perhaps performs certain other actions if false. For each variable set, this requires performing a secure select between the old value and the new value, based on the condition bit. A secure select on 2 $\ell$-bit values requires $\Theta(\ell)$ AND gates, so $\Theta(\ell)$ communication.

For instance $[[p]] \leftarrow [[q]] \wedge [[r]]$ indicates that the AND operation is performed on 3RSS-sharings of $q$ and $r$, using the protocol of Araki et al. [AFL+16], and the result is the 3RSS-sharing of $p$.

For conciseness, conversions between types of secret-sharing are typically implicit in our pseudocode, indicated by the sharing-type of the result. For instance, $[[A_i]]_{(1,2)} = [[q]] \oplus [[r]]$ means that variables $[[q]]$ and $[[r]]$, both stored using 3RSS, are first XORed to create a result that is shared using 3RSS. This result is then revealed to $P_1$ and $P_2$ (but not $P_0$), who store the result and label it $A_i$.

### 2.3. The Hierarchical Template for ORAMs

We now describe the Hierarchical template in detail in the context of ORAM, and prove it secure. In the subsequent section we describe how it can easily be modified for the context of DORAM, and show the resulting efficiency.

The Hierarchical template actually delivers a slightly stronger primitive: an oblivious dictionary. An ORAM is restricted in that the index space should be of the same size as the ORAM itself: the indices are the values $[0, n-1]$ and the ORAM stores $n$ items, one for each index. In an oblivious dictionary, there is still a restriction that the total number of items is some number $m$, however the indices[3] may be chosen from a set of size $n$, where $n$ may be much larger than $m$. For conciseness, we refer to dictionaries and their (distributed) oblivious variants as *Maps*.

[3]In a dictionary, the "indices" would normally be called "keys". However, since we elsewhere use key to refer to cryptographic keys we continue to refer to these as "indices".

**Map Functionality**

$Init(m, n, d)$: Initialize an empty dictionary $A$, that has capacity for $m$ items, where the indices are from space $[0, n-1]$ and the values are from $\{0,1\}^d$.
$Access(op, i, y)$: If $(op = read)$:
If some $(i, v)$ exists in the dictionary return $v$.
Otherwise store $(i, \perp)$ and return $\perp$.
If $(op = write)$:
If some $(i, v)$ exists in the dictionary, remove $(i, v)$.
Add $(i, y)$ to the dictionary (whether or not $(i, v)$ existed).

If the number of items in the dictionary exceeds $m$, the behavior is undefined.

Figure 2.1: Functionality for Map

If a Map is instantiated with $m = n$ this exactly matches the definition of an ORAM. However, the Map definition allows for cases where $m$ is much less than $n$.

As with a RAM, this naturally results in the following definition.

**Definition 2.3.1** (Oblivious Map)**.** An Oblivious Map (OMap) is an Oblivious Implementation of the Map functionality.

The Hierarchical approach first instantiates a weaker functionality: a Hash Table. Normally the term "Hash Table" is used to refer to a type of data structure that implements the dictionary functionality. We use Hash Table in a very different sense, to describe a *functionality* that is weaker than a dictionary. This may be confusing, but is consistent with how the term is used in the ORAM literature. A Hash Table is weaker than a Map. Firstly, all items that are to be stored in the Hash Table are added at once, during a build. After the build, it does not allow further writes. Secondly, it only allows each index to be queried (read) *at most once*.

$F_{HTable}$: **Hash Table**

*Build*$(Z, n, m, d)$: $Z$ is an array of length $m$, of pairs, such that $Z_i = (X_i, Y_i)$. For each $i \in [0, m-1]$, either $Z_i = (\perp, \perp)$ (that is the value is empty) or $X_i$ is a unique index from $[0, n-1]$, and $Y_i$ is a data values from $\{0,1\}^d$. Create a dictionary containing $(X_0, Y_0), \ldots, (X_{m-1}, Y_{m-1})$.
*Query*$(x)$: Given $x \in [0, n-1]$, if $x = X_i$ for some $i \in [0, m-1]$, return $Y_i$. Otherwise return $\perp$.
*Extract*$()$: Return an array containing the items that were *not* queried, with their corresponding values, padded to length $m$ with copies of $(\perp, \perp)$.
If some index $x$ is queried more than once (whether $x \in X$ or not) the behavior is undefined.

Figure 2.2: Functionality for Hash Table

This naturally creates the definition for Oblivious Hash Tables:

**Definition 2.3.2** (Oblivious Hash Table)**.** An *Oblivious Hash Table* (OHTable) is an oblivious implementation of the Hash Table functionality.

We now explain how the Hierarchical template can be used to build an OMap (and therefore ORAM) using Oblivious Hash Tables. This is presented in Figure 2.3. We will first describe the intuition, and then formally prove the security.

Hierarchical ORAMs maintain a hierarchy/ordering of data structues (OHTables and a cache) based on their recency. The cache is at the top of this hierarchy, followed by the tables, with more recently built tables higher than less recently built tables. (For comparison with Figure 2.3, being high in the hierarchy corresponds to having a small level index (parameterized by $i$), and within a level having a large table index (parameterized by $j$).) This ordering is also defined for the items within the data structures, which have the same location in the hierarchy as the data structure they are in. When an item is queried, it moves to the cache, to the top of the hierarchy, therefore "overtaking" all items that are in tables. However, this is the only way overtaking can occur. Rebuilds do not allow items to "overtake" each other in the hierarchy. If, before a rebuild, item $x$ is higher in the hierarchy than item $x'$, then after the rebuild, $x$ will be at least as high as $x'$, although $x$ and $x'$ may come to be at the same height by being merged into a single new data structure. This applies to the relationship between an item and a table as well. If item $x$ is higher in the hierarchy than some table $T_{i,j}$, then $x$ will remain higher in the hierarchy until the point at which $T_{i,j}$ is extracted.

When an item, $x$, is queried, it is moved to the cache. The cache is at the top of the hierarchy, so once $x$ is queried, it moves to the top of the hierarchy. In particular, $x$ is higher in the hierarchy than any OHTable, $T_{i,j}$, from which it was queried. The "no-overtaking" structure of the rebuilds ensures that if $x$ has been queried to a table $T_{i,j}$, then $x$ remains higher in the hierarchy than $T_{i,j}$ until the point at which the contents of $T_{i,j}$ are extracted. Before this point, the query order guarantees that $x$ will never be queried in $T_{i,j}$; since the query always queries higher data structures first, $x$ will be found in its higher data structure, and $n + t$ will be queried in $T_{i,j}$ instead. When $T_{i,j}$ is extracted and then rebuilt it will become safe to query $x$ again in $T_{i,j}$, because $T_{i,j}$ will be a new OHTable in which $x$ has never been queried. Therefore, any real index $x$ will never be queried more than once in any Hash Table. Furthermore, incrementing the counter after each access ensures that the fake queries (of the form $n + t$) are never repeated to any Hash Table either. Therefore, the Hierarchical ORAM template guarantees that an index is never queried more than once to any Hash Table, satisfying the required condition for Hash Tables. Since the access pattern outside of the Hash Tables is deterministic, if the Hash Tables are oblivious, the entire Map will be oblivious.

**Theorem 1.** The Hierarchical Template (Figure 2.3) is an oblivious implementation of the Map functionality, when implemented using Oblivious Hash Tables.

*Proof.* Firstly, we show that the output is correct. That is, $access(read, x, \perp)$ will return the value $y$ such that $access(write, x, y)$ was the most recent operation of the form $access(write, x, \cdot)$ (or $\perp$ if $access(write, x, \cdot)$ has never occurred).

The hierarchy only ever stores one "copy" of each item, as we now prove. Let us say that item $x$ (i.e. the item with index $x$) is *located in* a OHTable if $x$ was one of the indices with which the table was built and $x$ has not yet been queried in the table. The protocol maintains the invariant that for every value $x$ for which $access(\cdot, x, \cdot)$ has occurred, either $x$ is located in one, and only one, table (and no items with index $x$ in $C$) or there is a single item with index $x$ in $C$ (and $x$ is not located in any table.) This is true immediately after the first access: in which the item is placed in the cache (and occurs nowhere else). By induction, it is true after each rebuild: if the item is in the cache,

**Hierarchical ORAM/OMap Template**

**Parameter choices:**
$c$: The size of the cache, typically $\Theta(\log(n))$
$b$: Ratio of adjacent level sizes, typically 2. Each level has up to $b-1$ tables.
$l$: The number of levels, $\log_b(n/c)$, typically $\Theta(\log(n))$. Assume $n/c$ is a power of $b$.

Init(m, n, d):

- **Inputs:** $m$, the number of items
  $n$, the size of the index space
  $d$, the bit-length of the values
- **Create the cache:** Initialize an empty array, $C$, the cache. The cache has capacity to store $c$ items.
- **Initialize access counter:** Set $t = 0$
- **Create empty tables at each level:** Set $T_{i,j} = \perp$ for $i = 1 \dots \ell$ $j = 0 \dots b-2$

Access(op, x, y):

- **Inputs:** *op*, either read or write.
  $x$, the index to query (from $0, \dots, n-1$).
  $y$, if op = write, the value to be written.
- **Prepare:** Set $found = false$. Set $value = \perp$.
- **Scan the cache:** for $j$ in $t \mod c \dots 0$:
  If $C_j = (x, y_{old})$ for some value $y_{old}$, set $found = true$, set $value = y_{old}$ and set $C_j = (\perp, \perp)$.
- **Query the OHTables:** for $i$ in $1 \dots l$, for $j$ in $b-2 \dots 0$, if $T_{i,j} \neq \perp$:
  If $found = false$ set $q = x$, otherwise set $q = n + t$.
  $y_{old} \leftarrow T_{i,j}.Query(q)$.
  If $found = false$ AND $y_{old} \neq \perp$, set $found = true$ and set $value = y_{old}$.
- Set $t = t + 1$
- **Insert item into cache, updating if it is a write:** If $op = write$, set $C_{t \mod c} = (x, y)$
  If $op = read$, set $C_{t \mod c} = (x, value)$
- **Rebuild if needed:** If $t \mod c = 0$, Rebuild()
- **Output:** *value*. For a read, this returns the read value.

Rebuild():

- **Identify Level:** Let $\bar{i}$ be the largest value such that $t/c = 0 \mod b^{\bar{i}}$. Let $i^* = min(\bar{i}+1, \ell)$. Let $j^* = (t/c \mod b^{\bar{i}+1})/b^{\bar{i}}$. We will merge levels $0, \dots, \bar{i}$ into a new table at level $i^*$.
- **Merge Levels:** Initialize $Z = C$. For $i = 1, \dots, \bar{i}$, $j = 0, \dots, b-2$, obliviously evaluate $Z = Z \cup T_{i,j}.Extract()$. Set $T_{i^*,j^*} = OHTable.Build(Z, 2n, c \cdot b^{i^*-1}, d)$.
- **Clear Lower Levels:** Set $C$ to be an empty array of size $c$.
  For $i = 1, \dots, i^*-1$, $j = 0, \dots, b-2$, set $T_{i,j} = \perp$.
- **Reset Counter if Needed:** If $t = n$, set $t = 0$.

Figure 2.3: Hierarchical ORAM/OMap Template

or located in a table being extracted, it will be removed from the cache/table, and after the build will be located in the new table. It is true after each subsequent access: if the item was located in the cache, the location in which it was previously stored will be set to $(\perp, \perp)$ and the item will be written to a new cache location. If the item was located in a table, it will after the query, by definition, no longer be located in that table, and will instead be written to the cache.

Furthermore, $access(write, x, \cdot)$ is the only operation that can change the value associated with item $x$. A read will move the item back to the cache, but will re-write the old value. A rebuild will move the item from the cache or one table to another, but will not change any of the values associated with the item. Therefore, $access(read, x, \perp)$ will return the most recently written value.

Next, we show that the access pattern is oblivious. Most of the access sequences in the Hierarchical template are deterministic so can be easily generated by the simulator. The only accesses which are not deterministic are those of the Oblivious Hash Table. By the definition of Obliviousness 1.3.1 this requires a simulator which receives as input only the command and the size of the input. This simulator is therefore automatically "straight-line" and "black-box" (as per the definitions of [KLR10]) so can be securely composed with other protocols. In particular, the simulator for the OMap can simply run the simulator of each Oblivious Hash Table to obtain an appropriate simulation of the accesses by the Oblivious Hash Table.

It remains only to prove that the condition is satisfied that elements are not queried more than once to any OHTable. We proceed by induction. When a real item is queried for the first time to some OHTable, $T_{i,j}$, it will be inserted into the cache. (This happens if the item is found in that particular OHTable, found in a later OHTable, or even if it is not found in the OMap at all.) After this point, until $T_{i,j}$ is extracted, the item will remain higher in the hierarchy than the OHTable. This is because the item can only "move down" in the hierarchy during rebuilds, and every rebuild combines the contents of a subset of the highest data structures. If the rebuild does not include $T_{i,j}$, the item will still remain higher than the OHTable in the hierarchy. If the rebuild *does* include $T_{i,j}$, then the contents of that OHTable will be extracted, and the item will never be queried on that OHTable ever again. (A new OHTable will later be built and labelled $T_{i,j}$, but this will be a new

instance of the OHTable data structure, and it is safe to query the item on this new OHTable.) □

## 2.4. The Hierarchical Template for DORAMs

The Hierarchical template can easily be adapted to construct a DORAM. Like with ORAM, the Hierarchical template actually creates a stronger primitive: it implements a secret-shared map:

**$F_{Map}$: Secret-Shared Map**

Let $[[\cdot]]$ represent a $U$-secure secret-sharing over $s$ parties.
$Init(m, n, d)$**:** Initialize an empty dictionary $A$, that has capacity for $m$ items, where the indices are from space $[0, n-1]$ and the values are from $\{0,1\}^d$.

$Access([[op]], [[x]], [[y]])$**:** If $(op = read)$:
If some $(i, v)$ exists in the dictionary return a fresh $[[v]]$.
Otherwise store $(i, \perp)$ and return a fresh $[[\perp]]$.
If $(op = write)$:
If some $(i, v)$ exists in the dictionary, remove $(i, v)$.
Add $(i, y)$ to the dictionary (whether or not $(i, v)$ existed).

If the number of items in the dictionary exceeds $m$, the behavior is undefined.

Figure 2.4: Functionality for Secret-Shared Map

The definition of a Distributed Oblivious Map follows automatically.

**Definition 2.4.1** (Distributed Oblivious Map)**.** A Distributed Oblivious Map (DOMap) is a Secure MPC implementation of the Secret-Sharing Map functionality, where the Secret-Sharing Map functionality uses a $U$-secure secret-sharing over $s$ parties.

The Hierarchical solution again builds a DOMap, using Distributed Oblivious Hash Tables, which are defined below.

$F_{SS-HTable}$: **Hash Table**

$Build([[Z]], n, m, d)$: $Z$ is an array of length $m$, of pairs, such that $Z_i = (X_i, Y_i)$. For each $i \in [0, m-1]$, either $Z_i = (\perp, \perp)$ (that is the value is empty) or $X_i$ is a unique index from $[0, n-1]$, and $Y_i$ is a data values from $\{0,1\}^d$. Create a dictionary containing $(X_0, Y_0), \ldots, (X_{m-1}, Y_{m-1})$.
$Query([[x]])$: Given $x \in [0, n-1]$, if $x = X_i$ for some $i \in [0, m-1]$, return $[[Y_i]]$. Otherwise return $[[\perp]]$.
$Extract()$: Return a secret-shared array containing the items that were *not* queried, with their corresponding values, padded to length $m$ with copies of $([[\perp]], [[\perp]])$.
If some index $x$ is queried more than once (whether $x \in X$ or not) the behavior is undefined.

Figure 2.5: Functionality for Secret-Shared Hash Table

**Definition 2.4.2** (Distributed Oblivious Hash Table)**.** A *Distributed Oblivious Hash Table* (DOHTable), is a secure multiparty implementation of a secret-shared Hash Table.

It is easy to transform the Hierarchical template of Figure 2.3 into a template for a DORAM/DOMap. We simply transform all sensitive variables $(op, x, y, found, y_{old}, C, value, q, Z)$ to be secret-shared and replace the OHTable with a DOHTable from the appropriate security setting. The parameters $c, b, l, m, n, d$ and the variables $t, i, j, \bar{i}, i^*, j^*$ are not sensitive so can remain public. For completeness, we present the Hierarchical template, as applied to DORAMs/DOMaps in Figure 2.6.

The argument for security is almost identical to that of the Hierarchical ORAM, so we will only summarize the argument here. All variables that can be influenced by data are held as secret-shares. Therefore, $\mathcal{A}$'s view of these is random by the security of the RSS scheme. The protocol guarantees that an index is never queried more than once to a DOHTable. Any secure DOHTable will therefore reveal no information to $\mathcal{A}$, as it is indistinguishable from the secret-shared hash table functionality, which only outputs secret shares. This leads to the following results:

**Theorem 2.** Protocol $\Pi_{DOMap}$ (Figure 2.6) is a secure MPC implementation of $\mathcal{F}_{SS-Map}$, in the $\mathcal{F}_{SS-HTable}$-hybrid model.

**Theorem 3.** When Protocol $\Pi_{DOMap}$ (Figure 2.6) is instantiated with $m = n$, it is a secure MPC implementation of $\mathcal{F}_{SS-RAM}$.

We now analyze the efficiency of the DOMap protocol. Let $Cost_Q(n, d)$ be the communication cost

**$\Pi_{DOMap}$: Hierarchical DORAM/DOMap Template**

**Parameter choices:**
$c$: The size of the cache, typically $\Theta(\log(n))$
$b$: Ratio of adjactent level sizes, typically 2. Each level has up to $b-1$ tables.
$l$: The number of levels, $\log_b(n/c)$, typically $\Theta(\log(n))$. Assume $n/c$ is a power of $b$.

Init(m, n, d):

- **Inputs:** $m$, the number of items
  $n$, the size of the index space
  $d$, the bit-length of the values
- **Create the cache:** Initialize an empty array, $[[C]]$, the cache. The cache has capacity to store $c$ items.
- **Initialize access counter:** Set $t = 0$
- **Create empty tables at each level:** Set $T_{i,j} = \perp$ for $i = 1 \dots \ell$ $j = 0 \dots b-2$

Access([[op]], [[x]], [[y]]):

- **Inputs:** *op*, either read or write.
  $x$, the index to query (from $0, \dots, n-1$).
  $y$, if op = write, the value to be written.
- **Prepare:** Set $[[found]] = [[false]]$. Set $[[value]] = [[\perp]]$.
- **Scan the cache:** for $j$ in $t \mod c \dots 0$:
  If $[[C_j]] = ([[x]], [[y_{old}]])$ for some value $[[y_{old}]]$, set $[[found]] = [[true]]$, set $[[value]] = [[y_{old}]]$ and set $[[C_j]] = ([[\perp]], [[\perp]])$.
- **Query the OHTables:** for $i$ in $1 \dots l$, for $j$ in $b-2 \dots 0$, if $T_{i,j} \neq \perp$:
  If $[[found]] = [[false]]$ set $[[q]] = [[x]]$, otherwise set $[[q]] = [[n+t]]$.
  $[[y_{old}]] \leftarrow T_{i,j}.Query([[q]])$.
  If $[[found]] = [[false]]$ AND $[[y_{old}]] \neq [[\perp]]$, set $[[found]] = [[true]]$ and set $[[value]] = [[y_{old}]]$.
- Set $t = t+1$
- **Insert item into cache, updating if it is a write:** If $[[op]] = [[write]]$, set $[[C_{t \mod c}]] = ([[x]], [[y]])$
  If $[[op]] = [[read]]$, set $[[C_{t \mod c}]] = ([[x]], [[value]])$
- **Rebuild if needed:** If $t \mod c = 0$, Rebuild()
- **Output:** $[[value]]$. For a read, this returns the read value.

Rebuild():

- **Identify Level:** Let $\bar{i}$ be the largest value such that $t/c = 0 \mod b^{\bar{i}}$. Let $i^* = min(\bar{i}+1, \ell)$. Let $j^* = (t/c \mod b^{\bar{i}+1})/b^{\bar{i}}$. We will merge levels $0, \dots, \bar{i}$ into a new table at level $i^*$.
- **Merge Levels:** Initialize $[[Z]] = [[C]]$. For $i = 1, \dots, \bar{i}$, $j = 0, \dots, b-2$, obliviously evaluate $[[Z]] = [[Z]] \cup T_{i,j}.Extract()$. Set $T_{i^*,j^*} = OHTable.Build([[Z]], 2n, c \cdot b^{i^*-1}, d)$.
- **Clear Lower Levels:** Set $C$ to be an empty array of size $c$.
  For $i = 1, \dots, i^*-1$, $j = 0, \dots, b-2$, set $T_{i,j} = \perp$.
- **Reset Counter if Needed:** If $t = n$, set $t = 0$.

Figure 2.6: Hierarchical DORAM/DOMap Template

of a DOHTable query (which we assume does not depend on the size of the table), $Cost_B(m, n, d)$ be the cost of a DOHTable build and $Cost_E(m, n, d)$ be the cost of a DOHTable Extraction.

We start by measuring the cost of a query ignoring rebuilds. The cost of scanning the cache is at most $c$ equality tests (of $\log(n)$-bit values) and at most $c$ conditional selects (of $\Theta(d)$-bit values), so the total cost is $\Theta(c(\log(n) + d))$. The protocol queries at most $l \cdot (b - 1)$ DOHtables, resulting in cost $\Theta(l \cdot b \cdot Cost_Q(n, d))$. Apart from this, querying requires $\Theta(\log(n))$ bits of communication for securely setting $q$ and a futher $\Theta(d)$ bits to securely set *value*. Finally, writing to the cache requires another secure select with $\Theta(d)$ communication. The total communication cost is therefore $\Theta(c(\log(n) + d) + l \cdot b \cdot Cost_Q(n, d))$

We now account for the cost of rebuilds. Table $T_{i,j}$ has capacity $c \cdot b^{i-1}$, so builds cost $Cost_B(c \cdot b^{i-1}, n, d)$ and extracts cost $Cost_E(c \cdot b^{i-1}, n, d)$. It is rebuilt every $c \cdot b^i$ accesses and also extracted every $c \cdot b^i$ accesses. Therefore the cost per table per access is $\frac{1}{c \cdot b^i}(Cost_B(c \cdot b^{i-1}, n, d) + Cost_E(c \cdot b^{i-1}, n, d)$. There are $b - 1$ tables per level, so the cost per access per level is $\Theta(\frac{1}{c \cdot b^{i-1}}(Cost_B(c \cdot b^{i-1}, n, d) + Cost_E(c \cdot b^{i-1}, n, d)))$. Combining these gives the amortized cost per access:

**Theorem 4.** Given a DOHTable with query, build and extract costs $Cost_Q(n, d)$, $Cost_B(m, n, d)$ and $Cost_E(m, n, d)$, the total cost of a Hierarchical DORAM with cache size $c$ and expansion ratio $b$ is: $\Theta(c(\log(n)+d)+\log_b(n) \cdot b \cdot Cost_Q(n, d)+\sum_{i=1}^{\log_b(n)} \frac{1}{c \cdot b^{i-1}}(Cost_B(c \cdot b^{i-1}, n, d)+Cost_E(c \cdot b^{i-1}, n, d)))$.

# CHAPTER 3

## Alibi: A Flaw in Cuckoo-Hashing Based Oblivious RAM and a Solution

*The chapter is based on material which was first published in the following work:*

Brett Hemenway Falk, Daniel Noble and Rafail Ostrovsky. Alibi: A flaw in cuckoo-hashing based hierarchical ORAM schemes and a solution. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 228-269, Springer, 2021. [HFNO21].



### 3.1. Introduction

*There once was a table of hashes*
*That held extra items in stashes*
*It all seemed like bliss*
*But things went amiss*
*When the stashes were stored in the caches*

This chapter presents a flaw in several ORAM/DORAM protocols which use the Hierarchical template. In particular, these protocols tried to instantiate Oblivious Hash Tables using cuckoo hashing with a stash. However, for efficiency reasons, the stashed elements were removed from the cuckoo hashing table itself, and would be queried from a different part of the hierarchy, prior to the table itself being queried. This led to a subtle error in the distribution of accesses within each table, causing the access pattern to leak non-negligible information about the query sequence. In short, removing the stash caused the tables to not actually be Oblivious Hash Tables, breaking the abstraction of the Hierarchical template, and leading to a concrete attack. We also show a method for fixing this flaw, which applies to all affected works and in most cases does so without increasing their asymptotic complexity.

## 3.2. History of the Flaw

Pinkas and Reinman first introduced the idea of instantiating the Oblivious Hash Tables in a Hierarchical ORAM with cuckoo hashing [PR10]. This had the advantage that a query only required accessing a small constant number of locations within each table, rather than $\Theta(\log(n))$ as in the original hierarchical ORAM [Ost90]. However, cuckoo hashing has a non-negligible probability of failure. Pinkas and Reinman suggested that the tables could be rebuilt with new hash functions in the case of a build failure. However, Goodrich and Mitzenmacher (ICALP 2011, [GM11]), as well as Kushilevitz et al. (SODA 2012, [KLO12]) showed that this was insecure in the ORAM setting. This is because if the ORAM is queried with a sequence of items, all of which are not stored in a given OHTable, there is a non-negligible probability that the physical access pattern in the OHTable is incompatible with the items being stored there, revealing that the queries could not have been exclusively to items in the OHTable. Goodrich and Mitzenmacher suggested that cuckoo hashing with a stash could be used instead, proving that (for $m = \Omega(\log^7(n))$ and stashes of size $\Theta(\log(n))$) this resulted in negligible (in $n$) build failure probability. They suggested that the stashes could be combined, and that this combined stash could be queried prior to querying the tables. Kushilevitz et al. instead suggested that the stashes could be re-inserted into the cache.

In this chapter, we show that the combined-stash solution of Goodrich and Mitzenmacher and the stash re-insertion solution of Kushilevitz et al. both lead to a subtle security flaw which gives an adversary non-negligible advantage in distinguishing access patterns. The problem is similar to the problem in [PR10], where rehashing in the event of a build failure leaked information about the elements being stored at that level. Removing the elements from the stash on each level, like performing a rehashing, causes the elements that would have been in the stash to no longer be in that level. Therefore, if these elements are searched for they will be found before this level is reached, so instead of accessing the locations for the stashed elements at that level, random locations will be accessed instead. This means that, if all elements that were placed in a given level are searched for (including the items that were stashed), the access pattern of that level is less likely to contain any collisions in the *physical access pattern*. In contrast, if no elements from that level are accessed,

all accessed locations will be random. The expected number of collisions will therefore be higher in the second case, and we will show that this difference is non-negligible.

This flaw affects a large number of papers [GM11, GMOT11, KLO12, LO13a, PPRY18, KM19, AKL$^+$20] which combine stashes in order to eliminate super-constant sized stashes at each level. This does not affect earlier hierarchical solutions that did not use Cuckoo Hashing, e.g. [Ost90, Ost92, GO96] or non-hierarchical ORAMs such as PathORAM [SvDS$^+$13] or Circuit ORAM [WCS15]. In addition to finding this flaw, we present a simple solution. Our solution applies to all schemes which suffer from the flaw; for most of these it does not affect their asymptotic complexity.

In Section 3.3, we review cuckoo hashing. In Section 3.4, we present our concrete attack that allows an adversary to distinguish two different access patterns with non-negligible probability in hierarchical ORAM solutions that use Cuckoo Hashing with a *combined stash*. This attack has a clean intuitive interpretation. However, this attack does not apply directly to PanORAMa and OptORAMa, so in Section 3.5 we present a generic version of our attack which does apply to these protocols. In Section 3.6 we present our solution and prove that it is correct. Finally, we present the protocols that have been affected by this flaw in Section 3.7.

### 3.3. Cuckoo hashing

Cuckoo hashing was introduced in [PR04] as a method of multiple-choice hashing with expected constant-time lookups. Since its introduction, many variants of cuckoo hashing have been proposed and analyzed (see [Mit09] for a review). In this section, we review a basic common form of cuckoo hashing, but we emphasize that our attack works for almost all types of hashing with a stash.

We view a Cuckoo Hash Table as an array, $T$, with $\gamma m$ locations, each having capacity one, and a stash $S$ containing $s$ locations, also each of capacity one. Each item, $x$, can be placed in one of $h$ locations in $T$, given by $H_i(x)$ for $i = 1, \ldots, h$ where $H_i(x) \in [0, \gamma m - 1]$. If an element cannot be placed in one of its $h$ locations, it is placed in the "stash", $S$, of size $s$. With appropriate choices of constants $\gamma$ and $h$, and a stash of size $s = \log(m)$, cuckoo hashing will succeed except with probability negligible in $m$ (Theorem 2 of [ADW14]).

Figure 3.1 shows in detail how cuckoo hashing is instantiated. Cuckoo hashing has the desirable property that the accesses during a query essentially do not depend on the index being queried, or the items in the table. Each access results in $h$ accesses to $T$, and $s$ accesses to the items in $S$. If the hash functions are PRFs (with a key unknown to $\mathcal{A}$), the locations accessed in $T$ will be indistinguishable from random.

To make a Cuckoo hash table fully oblivious would additionally require that the Build and Extract operations are performed in such a way that the combined access pattern from calls to Build, Query and Extract is simulatable without knowledge of the table contents or queried indexes. Oblivious implementations of Cuckoo hashing vary significantly in how they instantiate the Build and Extract functions, so we do not specify a particular approach here. Rather, we will present an attack that is effective regardless of how Build and Extract are implemented.

We do this by defining a weaker security definition:

**Definition 3.3.1** (Access-Oblivious)**.** Access-Obliviousness is a simplification and weakening of Obliviousness (Definition 1.3.1) that applies to implementations of the Hash Table functionality. Accessed addresses are appended to the view only during calls to the Query operation, and not during calls to Build or Extract. An implementation of a Hash Table is *access-oblivious* if a simulator exists who can generate this (more restricted) view.

Any simulator that could be used to show that an implementation was oblivious could also be used to show that the implementation was access-oblivious, simply by the simulator outputting the same values for calls to Query, and not outputting anything for calls to Build and Extract. Therefore, any oblivious implementation is also access-oblivious. Hence, any implementation that is not access-oblivious cannot be oblivious.

**Lemma 1.** Cuckoo Hash Tables, as presented in Figure 3.1 are *access oblivious*.

*Proof.* For each query, the simulator selects the accessed locations to be $h$ truly random locations from $T$ and all $s$ locations from $S$.

**Parameters:** $h \geq 2$, a number of hash functions
$\gamma$, the memory blow-up of the main tables. *e.g.,* if $h = 2$, $\gamma = 2(1+\epsilon)$ for any $\epsilon > 0$
$s$, the stash size
**Build:** Given a set $\{(X_0, Y_0) \ldots, (X_{m-1}, Y_{m-1})\}$, where $X_i \in [0, n-1]$

1. Pick $h$ hash functions $H_1, \ldots, H_h$ which map $[0, n-1] \to [0, \gamma m - 1]$.
2. Create an empty array $T$ of length $\gamma m$.
3. An item $(X_i, Y_i)$ can be stored only in $T[H_j(X_i)]$ for some $j \in [1, h]$. Find a maximum allocation of items to unique locations (*e.g.,* through a matching algorithm) and store items in $T$ according to this allocation.
4. Let $S$ be the set of items that were not able to be stored in $T$. If $|S| > s$, the build fails. Store $S$ as an array, padded to length $s$.

**Query:** Given an index $x$

1. Initialize $y_{ret} \leftarrow \perp$.
2. For $i \in 1, \ldots, s$, if $S[i] = (x, y)$, set $S[i] = (\perp, \perp)$ and set $y_{ret} \leftarrow y$.
3. For $i \in 1, \ldots, h$, if $T[h_i(x)] = (x, y)$, set $T[h_i(x)] \leftarrow (\perp, \perp)$ and set $y_{ret} \leftarrow y$.
4. Return $y_{ret}$

**Extract:**

1. Set $Z$ to contain all (non-null) items from $T$ and $S$. Pad $Z$ to length $m$ and return $Z$.

Figure 3.1: Cuckoo Hashing with a Stash (single table version)

The stash-accesses will be indistinguishable from those of the real execution since these are chosen deterministically. For the accesses to $T$, note that hash tables require that each index can be queried at most once. Therefore, since the hash functions themselves are PRFs, the output of the hash functions on the indices will be indistinguishable from $h$ values from $[0, \gamma m - 1]$, chosen uniformly at random and independently from all previous values. Hence the view of the simulated and real executions are indistinguishable. □

**Remark 1** (Set Membership in Table)**.** Note that this also means that the access pattern of a Cuckoo Hash Table does not reveal whether the queried elements were present in the table or not.

Unlike some other constructions, Cuckoo Hash Tables hide set membership *without* the insertion of dummy elements, *i.e.,* pre-inserted elements that should be searched for in the case the item is not in the table.

If Cuckoo hashing is combined with an appropriate Build and Extract construction, it can be fully oblivious. Note that this not only requires that the Build and Extract functions are oblivious in

themselves, but that when Build, Query and Extract are all performed by a single entity, that the *combined* sequence of accesses is still oblivious.

**Remark 2** (1-table vs h-table cuckoo hashing)**.** We describe a single-table cuckoo hashing scheme, where all $h$ hash functions hash into the same table. Alternatively, some cuckoo hashing constructions use $h$ tables, and hash function $H_i$ hashes into table $i$. Setting $h$ to 2 is a common choice, resulting in 2-table cuckoo hashing. For constant $h$, 1-table and $h$-table cuckoo hashing have the same asymptotic performance, although there are some differences in the details of their analyses.

A single-table Cuckoo Hash Table corresponds naturally to bipartite multigraph with $m$ left-hand nodes (corresponding to $[0, m-1]$) and $\gamma m$ right-hand nodes corresponding to the hash buckets (*i.e.,* the array $T$). Then a left hand node, $v$, is connected to $h$ right hand neighbors given by $\{H_i(v)\}_{i=1}^{h}$. It is straightforward to see that the build procedure can succeed if there is a bipartite matching that includes $m - s$ left-hand vertices: the matched elements can be placed in their right-hand neighbors (given by the matching) and the remaining $s$ elements can be placed in the stash.

This also shows that the build procedure can be implemented by building this bipartite multigraph and calculating a maximum matching. We assume that whatever build procedure is used does find such a maximum matching.

To be an Oblivious Hash Table, the functions Build and Query need to fail with probability $n^{-\omega(1)}$. If a Cuckoo Hash Table is successfully built, the locations returned by Query will always include the location of the queried item if it is stored in the table, so the probability of failure is 0. Build, however, can fail. If the stash is chosen by finding a maximum matching, the probability of failure is $\mathcal{O}(m^{-s})$ for any constant $s$ [KMW10]. A similar result holds for $s = \mathcal{O}(\log m)$, for which the probability of failure is $\mathcal{O}(m^{-\frac{s}{2}})$ [ADW14]. Therefore, if $s = \Theta(\log(m))$ the failure probability is $\mathcal{O}(m^{-\Theta(\log m)})$, which is negligible in $m$. Note that for ORAMs, the failure probability needs to be negligible not in the capacity of the Cuckoo Hash Table, $m$, but in the capacity of the ORAM, $n$. If $n$ is polynomial in $m$ this will hold. In fact, for any $m = \omega(\log(n))$, if $s = \Theta(\log(n))$ the probability of build failure is negligible in $n$ ([Nob21]). To achieve obliviousness, an ORAM therefore must use

an alternative to Cuckoo hashing for the very small table sizes (that do not satisfy $m = \omega(\log(n))$).

We have shown here that the Cuckoo Hash Table presented here, with appropriate Build and Extract functions, is an example of an oblivious hash table (with failure negligible in $n$). Cuckoo hashing with a Stash can therefore be used to build an ORAM using the Hierarchical paradigm, as shown in Section 2.3. However, we now show that if the stashes are combined this breaks obliviousness.

## 3.4. The Attack

In this section, we describe a novel attack on hierarchical ORAM protocols that use cuckoo hashing with a combined stash. This attack applies directly to [GM11, GMOT11, KLO12, LO13a] and Instantiation 2 of [KM19]. The recent works of PanORAMa [PPRY18] and OptORAMa [AKL+20] use a modified hierarchical solution with multiple cuckoo tables at each level. Since the attack presented here assumes that the adversary can know which indexes are stored in the Cuckoo Hash Table, it does not apply directly to PanORAMa and OptORAMa. In Section 3.5 we present a more general attack that also applies to PanORAMa and OptORAMa. The general attack is also simpler, but this attack has the advantage of having an intuitive interpretation.

### 3.4.1. Simplified attack

First, we describe this attack in a simplified setting, which we later show is equivalent to the ORAM setting.

Imagine the following construction of a hash table. A Cuckoo Hash Table, as defined in Figure 3.1, is modified in the following way. If an item, $x$, *is* found in the stash, rather that searching for $x$'s locations in $T$, some new unique value $x'$ will be searched for in $T$. That is $T[H_i(x')]$ will be accessed for $1 \leq i \leq h$ instead of $T[H_i(x)]$. This construction is presented in Figure 3.2. We will show that this object is no longer an Oblivious Hash Table.

Observe that previously, the locations accessed by Query depended only on the index $x$ and the hash functions $H_1, \ldots, H_h$. However, in the Stash-Resampling Cuckoo Hash Table, the locations accessed depend on the table itself, and in particular depend on which items were placed in the stash. The fact that the access pattern changes depending on how the table is constructed breaks

Parameters, Build and Extract are defined as in Figure 3.1.
**Query:** Given an index $x$

1. Initialize $y_{ret} \leftarrow \perp$.
2. For $i \in 1, \ldots, s$, if $S[i] = (x, y)$, set $S[i] = (\perp, \perp)$ and set $y_{ret} \leftarrow y$.
3. If $y_{ret} \neq \perp$, set $x$ to some new unique value, $x'$.
4. For $i \in 1, \ldots, h$, if $T[h_i(x)] = (x, y)$, set $T[h_i(x)] \leftarrow (\perp, \perp)$ and set $y_{ret} \leftarrow y$.
5. Return $y_{ret}$

Figure 3.2: Stash-Resampling Cuckoo Hash Table

the abstraction of an Oblivious Hash Table. We will next show that this break leads to a concrete vulnerability.

**Remark 3.** We describe our attack in terms of cuckoo hashing, but essentially the same argument goes through with other hashing schemes that use a stash.

Let $(T, S)$ be a Stash-Resampling Cuckoo Hash Table containing indices $X = (X_1, \ldots, X_t)$ and using hash functions $H = (H_1, \ldots, H_h)$. Let $Q_1, \ldots, Q_t$ be the addresses accessed in $T$ for queries to $X_1, \ldots, X_t$ respectively. Now, let $X'_1, \ldots, X'_t$ be the indices which are used as the inputs to the hash functions when querying $X_i$. If $X_i$ is not stored in $S$, then $X'_i = X_i$. If $X_i$ *is* stored in $S$, then $X'_i$ is a new unique random value.

Now imagine that a Cuckoo Hash Table is constructed using hash functions $H$, but with indices $X'$. All items that were already stored in the table can continue to be stored in the table. However, it is likely that if $X_i$ was stashed, $X'_i$ will not need to be stashed, since it is hashed to new locations, one of which is probably empty. Therefore the stash size of this Cuckoo Hash Table is smaller than usual. The adversary does not know $H$ or $X'$, but it *does* learn $H_j(X'_i)$ since these are the locations accessed in $T$ during a Query. Therefore, it can learn what the stash size *would have been* in a table that used hash functions $H$ and indexes $X'$.

In contrast, let $X''$ be a sequence of $t$ indices, none of which are in $X$. Since none are in the stash, $X''$ are also the inputs to the hash functions and the adversary can learn from the access pattern the size the stash would have been if the table stored $X''$. The values of $H_j(X''_i)$ will be chosen

uniformly at random, so this stash would be chosen from the usual stash size distribution. Hence, if the adversary calculates what the stash size *would have been* if a table was constructed from the hash function inputs, the distribution of stash sizes will be *smaller* if $X$ is queried than if $X''$ is queried.

We now prove formally that a Stash-Resampling Cuckoo Hash Table is not access-oblivious. We formalize the intuition above by representing the accesses as a bipartite graph, with $m$ left-vertices corresponding to the $m$ inputs to the hash functions, with $\gamma m$ right-vertices corresponding to the non-stash locations in the table and edges from a left-vertex to a right-vertex if one of the hash functions maps the left-vertex to the right-vertex. A maximum matching in the graph therefore corresponds to a possible assignment of elements to locations in the hypothetical hash table constructed by the adversary. The number of unmatched elements then will correspond to the stash size. Below, we formalize the correspondance from access sequences to graphs and show that the distribution of the number of unmatched elements in the graphs indeed differs non-negligibly.

**Definition 3.4.1** (Graph Representation of an Access-Pattern)**.** Let $Q = Q_1, \ldots, Q_m$ be the locations accessed in $T$ from a sequence of queries of length $m$. The Graph Representation of an Access Pattern, $B(m, \gamma, Q)$ is a function which, given integers $m$, $\gamma$ and $Q$ as defined above, returns a bipartite multigraph with left vertices $a_1, \ldots, a_m$, right vertices $b_1, \ldots, b_{\gamma m}$ and edges $(a_i, b_j)$ for $j \in Q_i$.

**Definition 3.4.2** (Left-regular bipartite multigraph)**.** We define a left-regular bipartite multigraph to be a graph $G = (L \cup R, E)$ with the following properties.

- It is bipartite, with vertex sets $L$ and $R$, and each edge being directed from $L$ to $R$, *i.e.,* $\forall (u, v) \in E, u \in L, v \in R$.
- Every vertex in $L$ has a constant number of edges, denoted $h$.
- $E$ is a multiset, *i.e.,* the edge $(u, v)$ may occur multiple times.

**Definition 3.4.3** (Random left-regular bipartite multigraph)**.** We define $G_0(m, \gamma, h)$ to be a func-

tion that produces a random left-regular bipartite multigraph, where $|L| = m$, $|R| = \gamma \cdot m$, $h \geq 1$ is the degree of each vertex in $L$ and where each outgoing edge from a vertex $u \in L$ has an end-point, $v \in R$, that is chosen uniformly at random from $R$ (and independent of all other choices).

If $Q = (Q_1, \ldots, Q_m)$ is the result of outputs of Query to a sequence of queries to a (Stash-Resampling) Cuckoo Hash Table with capacity $m$ and degree $h$, then $G \leftarrow B(m, c, Q)$ will be a left-regular bipartite multigraph, since every $Q_i$ will contain $h$ vertices in $[\gamma m]$. We will soon show that for a Stash-Resampling Cuckoo Hash Table, if none of the queried elements are in the table, $G$ will be sampled as a *random* left-regular bipartite multigraph, but if the table contents are queried, the left-regular bipartite multigraph will be sampled from a *different* distribution of graphs which will have fewer unmatched elements.

**Definition 3.4.4** (Matching of a bipartite multigraph)**.** For a bipartite multigraph $G = (L \cup R, E)$, a matching is a set of edges $E' \subseteq E$ such that

$$(u, v), (u', v') \in E' \Rightarrow u \neq u', v \neq v'.$$

A maximum matching is a matching of maximum size. There may be multiple such matchings, but they will all be the same size; we use $M(G)$ to denote some such matching and $|M(G)|$ to be this size, which is independent of which matching is chosen. $S(G) \stackrel{\text{def}}{=} m - |M(G)|$ is the number of unmatched elements on the left-hand side.

Note that for any $G$, $1 \leq |M(G)| \leq m$, so $0 \leq S(G) \leq m - 1$.

**Lemma 2** (Lower bound on unmatched elements)**.** For all $0 \leq s \leq m - 1$ and $G \leftarrow G_0(m, \gamma, h)$, where $h$, $\gamma$ are constants,

$$\Pr\left[S(G) \geq s\right] \geq \left(\frac{1}{\gamma m}\right)^{hs+h-1}$$

*Proof.* Pick $s+1$ elements of $L$. The probability that *all* $h \cdot (s+1)$ edges of these elements will have the same endpoint $v \in R$ is $\left(\frac{1}{\gamma m}\right)^{h(s+1)-1} = \left(\frac{1}{\gamma m}\right)^{hs+h-1}$. If this occurs, any matching can contain at most 1 of these elements, which means that at least $s$ of these elements will be unmatched. Thus

$S(G) \geq s$. Note that for any constant $h$ and $s$, this probability is non-negligible. $\square$

Next, we describe two distributions on the integers $[0, m-1]$.

**Definition 3.4.5.** Fix constants $h, m \in \mathbb{N}$, and $\gamma > 1$. Let $M(\cdot)$ be an algorithm that takes a bipartite multigraph $G$, and returns a maximum matching $M(G)$.

- **Distribution 0:** Let $s_0$ be the random variable denoting the number of unmatched elements in a random bipartite multigraph. $s_0 \stackrel{\text{def}}{=} S(G_0(m, c, d))$.
- **Distribution 1:** Define a distribution of graphs according to the following process. First construct a graph $G' \leftarrow G_0(m, \gamma, h)$. Let $G' = (L \cup R, E')$. Let $M(G')$ be a maximum matching in $G'$. Initialize $E = E'$. For every $u \in L$ s.t. $\nexists (u, v) \in M(G')$, remove every edge $(u, v) \in E'$, and replace it with a new edge $(u, v')$ where $v'$ is chosen uniformly at random from $R$. Let $G = (L_1 \cup R_1, E)$ be the modified graph. Let $G_1(m, \gamma, h, M(\cdot))$ denote the function that samples a graph from this distribution. Define $s_1$ to be the number of unmatched elements in this experiment, *i.e.,* $s_1 \stackrel{\text{def}}{=} S(G_1(m, \gamma, h, M(\cdot)))$.

Although the distributions $s_0$ and $s_1$ depend on parameters, we generally suppress these dependencies for notational convenience.

Intuitively, the expected value of $s_1$ should be smaller than the expected value of $s_0$, since the vertices which were not matched get another chance to be matched when new end-points are chosen for them. In Lemma 3 we show that this is indeed the case, and that the distributions of $s_0$ and $s_1$ are statistically different (*i.e.,* non-negligibly different).

**Lemma 3.** If $s_0$ and $s_1$ are the random variables described above, then the statistical distance between $s_0$ and $s_1$ is at least $\frac{1}{m}\left(1 - \left(\frac{1}{\gamma}\right)^h\right)\left(\frac{1}{\gamma m}\right)^{2h-1}$ which is non-negligible in $m$.

*Proof.* Consider the graph $G' = (L \cup R, E') \leftarrow G_0(m, \gamma, h)$ generated as the first step in generating distribution $s_1$, where $|R| = \gamma \cdot m$. Let $M = M(G')$. Let $S \subset L$ be the unmatched vertices in $L$. We know $|S|$ is distributed by $s_0$. When $G$ is constructed (as the second step of distribution $s_1$),

each $u \in S$ will receive $h$ new random neigbors. For $v \in L/S$ we can use the existing matching $M$ for $G$ and for $u \in S$ we can match it to a neighbor directly if this neighbor is not already matched.[4] Since at most $m$ elements of $R$ will ever be matched, the probability that a new random neighbor is already matched is at most $\frac{1}{\gamma}$. There is then at most a $\left(\frac{1}{\gamma}\right)^h$ probability that *all* $h$ right-hand neighbors of $u$ are already matched. Let $e_i'$ be the event that $v_i$ is unmatched in $G'$, and $e_i$ the event that $v_i$ is unmatched in $G$. This shows:

$$Pr[e_i] \leq \left(\frac{1}{\gamma}\right)^h Pr[e_i']$$

Thus by linearity of expectation

$$E[s_1] = \sum_{1 \leq i \leq m} Pr[e_i] \leq \sum_{1 \leq i \leq m} \left(\frac{1}{\gamma}\right)^h Pr[e_i'] = \left(\frac{1}{\gamma}\right)^h E[s_0].$$

By Lemma 2, $Pr(s_0 \geq s) \geq \left(\frac{1}{\gamma m}\right)^{hs+h-1}$. Since $s_0$ is a non-negative distribution, $E[s_0] \geq Pr(s_0 \geq 1) \geq \left(\frac{1}{\gamma m}\right)^{2h-1}$ so

$$|E[s_0] - E[s_1]| \geq \left(1 - \left(\frac{1}{\gamma}\right)^h\right)\left(\frac{1}{\gamma m}\right)^{2h-1}.$$

In particular, this means that the expected values, $E[s_0]$ and $E[s_1]$ are non-negligibly different. Let $\Delta(s_0, s_1)$ represent the statistical distance between distributions $s_0$ and $s_1$. Since $0 \leq s_0, s_1 \leq m$,

$$\Delta(s_0, s_1) \geq \frac{1}{m}|E[s_0] - E[s_1]| \geq \frac{1}{m}\left(1 - \left(\frac{1}{\gamma}\right)^h\right)\left(\frac{1}{\gamma m}\right)^{2h-1}$$

which means that $\Delta(s_0, s_1)$ is also non-negligible. □

Now we show that the Stash-Resampling Cuckoo Hash Table is not access oblivious.

**Theorem 5.** The Stash-Resampling Cuckoo table presented in Figure 3.2 is not access-oblivious.

*Proof.* We run two experiments. In one, the query sequence is disjoint from the contents of the

[4] This greedy matching assignment not give an optimal matching for $G$, but it will provide an upper bound for $s_1$ in terms of $s_0$.

Cuckoo table. In the other, the query sequence exactly matches the contents of the Cuckoo table. We show that these two cases result in access patterns that have non-negligible statistical distance.

Let $X = (1, \ldots, m)$ where $m \leq \frac{n}{2}$; this will be the indices for the table. Let $Y$ be an arbitrary sequence of $m$ $d$-bit values.

Let $(T, S) \leftarrow Build(X, Y)$ be the table in the first experiment and let $(T', S') \leftarrow Build(X, Y)$ be the table in the second experiment. (These may not be equal, but are from identical distributions.)

In experiment 0, we query $\hat{X} = (1 + m, \ldots, 2m)$. In experiment 1, we query $\hat{X}' = (1, \ldots, m)$. Let $Q_i$ be the locations accessed in $T$ when querying $\hat{X}_i$, and $Q'_i$ be the locations accessed in $T'$ when querying $\hat{X}'_i$

Let $G \leftarrow B(m, \gamma, Q)$ and $s \leftarrow S(G)$. Likewise let $G' \leftarrow B(m, \gamma, Q')$ and $s' \leftarrow S(G')$.

In the first experiment, none of the queries are in $X$, therefore none will be in the stash. Therefore $Q_i = (H_1(\hat{X}_i), \ldots, H_h(\hat{X}_i)) \ = (H_1(m + i), \ldots, H_h(m + i))$. Since the $\hat{X}_i$ are distinct from each other and the elements stored in the table, $H_j(\hat{X}_i)$ will be (computationally indistinguishable from) values chosen uniformly at random from $[0, \gamma m - 1]$ independent from all previous random choices. Hence, each left-vertex in $G$ will have $h$ neighbors, chosen uniformly at random from $\{b_1, \ldots, b_{\gamma m}\}$. Therefore $G$ is chosen exactly according to $G_0$.

In the second experiment, all of the queries are to items stored in the table. For any queried index, $\hat{X}'_j$ that is in the stash, the Stash-Resampling Cuckoo Hash Table will pick a new index to query, let us label it $\bar{X}'_j$ and access locations $(H_1(\bar{X}'_j), \ldots, H_h(\bar{X}'_j))$ which will not have been queried before so will be (computationally indistinguishable from) new random locations. Therefore, for these elements that were not in the maximum matching, the corresponding edges will be re-chosen uniformly at random. The graph from the second experiment will therefore be constructed according to distribution $G_1(m, \gamma, h, M(\cdot))$, assuming the stash was chosen by some maximum matching algorithm $M(\cdot)$.

We have already shown that distributions $G_0(m, \gamma, h)$ and $G_1(m, \gamma, h, M(\cdot))$ have a non-negligible

statistical difference. This means that the access patterns in the two experiments also have non-negligible statistical difference. It is therefore impossible for a simulatar, who does not have knowledge of which experiment is being run to create a simulated view that is statistically close to both distributions. Hence, the Stash-Resampling Cuckoo Hash Table is not access-oblivious. □

**Remark 4.** Note that the attack described above is immediately applicable in cases where the stash is accessed *before* the associated Cuckoo Hash Table, and if the target is found in the stash, the protocol access random locations in the table. For instance, our attack would apply to a hierarchical ORAM that stored a stash at the same level, but accessed the stash *first*, and queries a nonce in the rest of the table if the element is found in the stash. In the subsequent section, we show how the attack affects hierarchical ORAMs which instead move the stash away from the table to another part of the ORAM data structure.

#### 3.4.2. Hierarchical ORAM with a combined stash

We now present how hierarchical ORAMs were constructed using a combined stash. We will show that this breaks the abstraction of an Oblivious Hash Table, and results in access patterns identical to those of the Stash-Resampling Cuckoo Hash Table, which breaks obliviousness.

Beginning with the protocol of Goodrich and Mitzenmacher [GM11], a number of hierarchical ORAM schemes stored stashed items from a table construction in a shared stash or re-inserted them into the cache. Since most schemes re-insert stash items into the cache, we will present this version. There are minor variations in exactly how stash re-insersion is achieved. For concreteness, we pick a simple variant that maintains the same rebuild schedule, but doubles the cache size. While we define a concrete instantiation of stash re-insersion to make the exposition clear, our attack applies to all Hierarchical (D)ORAM protocols which re-insert the stash, as well as to those which combine stashes.

Figure 3.3 presents a concrete hierarchical RAM obtained by modifying the hierarchical ORAM protocol of Figure 2.3 from Section 2.3. The ORAM is modified to use an Oblivious Hash Table that produces a stash, which is re-inserted into the cache. All other parts of the protocol remain

A Stash-Reinserting RAM is equivalent to the ORAM of Figure 2.3 with the following modifications:

- **Rebuild:** Rather than table $T_{i*}$ storing all elements in $X$, at most $c$ of these elements can be stored in a stash. The stash is not stored at this level, but is is padded to size $c$ and inserted into the cache.
- **Cache size:** The cache size is increased from $c$ to $2c$, with locations $C_t \dots C_{2t-1}$ holding the stash from the most recent rebuild.

Figure 3.3: Stash-Reinserting Hierachical RAM

the same.

**Theorem 6.** The Stash-Reinserting RAM of Figure 3.3 is not oblivious, as per Definition 1.3.1.

*Proof.* Let $m = cb^i$, for some $i \geq 1$, where $m \leq \frac{n}{2}$. We will be constructing a Cuckoo hash table of size $m$. (Some schemes use a combination of Cuckoo hashing and other hash tables. In this case we only require that Cuckoo hashing is used for some table of size $m \leq \frac{n}{2}$, and set $m$ to this size.)

We define two sequences of RAM queries, each of length $2m$:

$$U = ((write, 1, \perp), \dots, (write, 2m, \perp))$$

$$U' = ((write, 1, \perp), \dots, (write, m, \perp),\ (write, 1, \perp), \dots, (write, m, \perp))$$

We will run two experiments using the Stash-Reinserting Hierarchical RAM. In the first we initialize the RAM and then perform query $U$. In the second, we initialize the RAM and then perform query $U'$. Since $m = cb^i$, both experiments will enact a Rebuild after $m$ accesses, which will each result in a new table, $T_{i,0}$, being built using the items $(1, \perp), \dots, (m, \perp)$. Stashed items will be re-inserted into the cache.

The stash will be re-inserted in both cases. We know that each of these stashed elements will exist at a single location at the start of each access. (See proof of Theorem 1.) Furthermore, from the build schedule, the stashed items will continue to exist higher in the hierarchy for the next $m$ accesses. This means that, until this point in time, they will always be found *before* $T_{i,0}$ is accessed. Thus,

by the ORAM query algorithm, a nonce will instead be queried in $T_{i,0}$.

Therefore, the access pattern in $T_{i,0}$ will be the same as that of the Stash-Resampling Cuckoo Hash Table in Figure 3.2, where items were searched in the stash first, and if found in the stash a nonce was searched in the remainder of the table. The only difference is that in the Stash-Resampling Cuckoo Hash Table, the algorithm also accessed a pre-assigned stash, but this is not an issue since the attack to the stash-resampling algorithm does not use the access pattern to the stash (as this access pattern is always the same). Observe that, exactly like in the attack of Theorem 5, one sequence of accesses ($U$) will only access elements that were *not* in the data table, and the other sequence ($U'$) will only access elements that *were* in the data table (including the stash). Therefore, by the same argument as Theorem 5 the statistical distance between ORAM access pattern distributions is non-negligible. Therefore, the Stash-Reinserting RAM protocol of Figure 3.3 is not oblivious. □

## 3.5. The Generic Attack

The attack described in Section 3.4 gave the adversary a significant amount of control over the operation of the protocol. While the attack assumed that $\mathcal{A}$ could only see the accesses during a Query, $\mathcal{A}$ could still completely control which items were used to build the Hash table. This allowed $\mathcal{A}$ to distinguish two sequences of accesses, one in which all of the items from the table were queried and another in which none of the items from the table were queried.

In a hierarchical ORAM protocol, an adversary may not have this level of control of the contents of the hash tables. In particular, in the PanORAMa [PPRY18] and OptORAMa [AKL$^+$20] protocols, each level contains multiple Cuckoo Hash Tables, and only some of the items from a level are placed in any given table.

As our attack is contextualized for PanORAMa and OptORAMa we use the same Cuckoo hashing as these works do. Specifically, we set the number of hash function to $h = 2$ and use 2-table instead of 1-table hashing (see Remark 2). Our attack, however, can be generalized to any constant number of hash functions, as well as to the 1-table setting.

We therefore now construct a more general attack that assumes only that the adversary knows an appropriately-sized *superset* of the elements that were used to build the table. Concretely, we assume that the adversary knows some set $P$, which is the set of *possible* indexes which might be in a given table. This clearly must be smaller than the set of all queriable indices: $N = \{1, \ldots, n\}$. Let $O = N/P$, be the set of *outside* indexes, which $\mathcal{A}$ knows are definitely not in the table. We require that $|O| \geq 3$ (or more generally $h + 1$).

Let $X$ be the set of indices which are used to build the table. The Build function will store some items using cuckoo hashing and place the others in the stash. Abusing notation slightly, let $T$ represent the set of indices which could be stored using cuckoo hashing and $S$ represent the set of indices of items which were placed in the stash. We require that $|P| = poly(|T|)$, that is the set of possible indices should be somewhat comparable to the set of items which are actually stored using cuckoo hashing.

Our attack only requires the adversary to make 3 (or more generally $h + 1$) queries to the hash-table. In short, $\mathcal{A}$ picks 3 distinct random indexes from $P$ in the first experiment and picks 3 distinct random indices from $O$ in the second experiment. The distribution of these accesses will be non-negligibly different. This will essentially be because there is a non-negligible probability that the first experiment picks 3 indexes which are in $T$, and which cannot possibly be hashed to the same 2 locations. While it is not guaranteed that indexes from $T$ will be chosen, this usually skews the total probability that all 3 indexes result in accesses to the same locations.

Formally, we will weaken our definition of access-obliviousness by restricting the information available to the adversary. We will then show that the Stash-Resampling Cuckoo Hash Table is not oblivious even under this weaker definition. We then show in Section 3.5.2 that this is sufficient to undermine the obliviousness of the PanORAMa and OptORAMa constructions.

### 3.5.1. Generic Stash-Resampling Cuckoo Hash Table Attack

We first weaken the definition of access-obliviousness in Definition 3.3.1. Rather than allowing $\mathcal{A}$ to specify the contents of the Hash table during the Build, these are chosen. $\mathcal{A}$ is only provided

an appropriately-sized superset of the possible contents to the Hash table. As in the definition of access-obliviousness, $\mathcal{A}$'s view consists only of the accesses during calls to Query.

**Definition 3.5.1.** A hash-table implementation is *access-oblivious in the knowledge of an index superset* if it is simulatable in the following experiment.

Let $P \subset [1, n]$ be a set of possible indexes, where $|P| = poly(m)$ and $|P| \leq n-3$. Let $((X_1, Y_1), \ldots, (X_m, Y_m))$ where $X_i \in [1, n]$ and $Y_i \in \{0, 1\}^d$.

We define two experiments with an adversary analogously to the definition of obliviousness (Definition 1.3.1). In both, a hash-table is already built using $X$ and $Y$. The adversary is allowed to specify Query commands, in the knowledge of $P$ (but not $X$ or $Y$). The hash-table implementation is *access-oblivious in the knowledge of an index supereset* if for every such $P$, $X$ and $Y$, and every adversary $\mathcal{A}$, there exists a simulator $S$ which given $P$ is able to create a view that is statistically indistinguishable from the access pattern in the real execution.

Before proceeding further, it is helpful to define a few other index sets. We have that $X = S \cup T$ is the set of indices that are used to build the table. Let $R = P/X$ be the set of possible indices which are not in the table, which we will call the *remaining* indices. It is also helpful to consider the set $U = P/T$. This is the set of *unallocated* possible indices, that is they are possible indices, but are not stored using cuckoo hashing. By definition, $U = R \cup S$, that is it contains the possible indices that were not in the table and those in the stash. $U$ is significant because the stash-resampling cuckoo hash table will access new random locations for indices in $U$, whereas will access according to the pre-determined allocation for indices in $T$. The various sets of indices are presented visually in Figure 3.4.

**Theorem 7.** Stash-Resampling Cuckoo Hash Tables with $h = 2$ are not access-oblivious in the knowledge of an index superset.

*Proof.* We present two experiments which can be executed by $\mathcal{A}$ which result in access patterns that have non-negligible statistical distance.

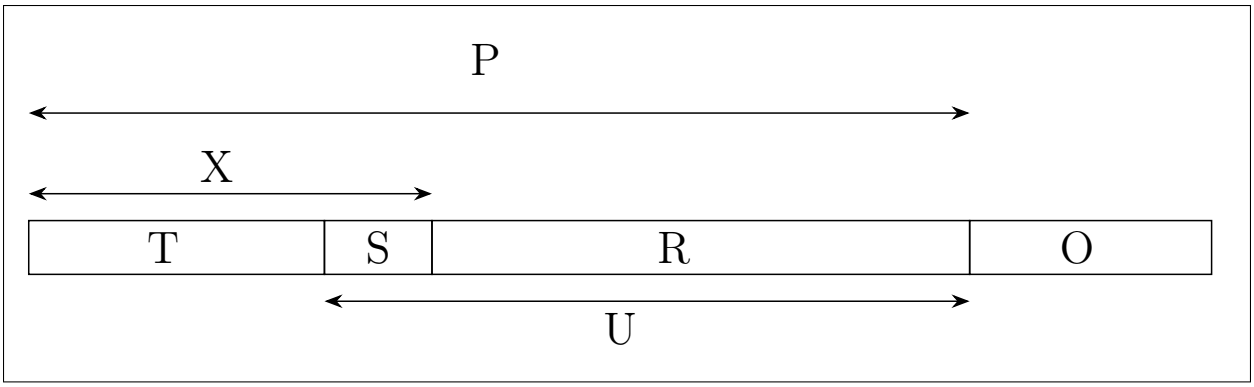


Figure 3.4: Representation of Index Sets: $X = T \cup S$, $U = S \cup R$, $P = X \cup R$.

In the first, $\mathcal{A}$ selects distinct $x_1, x_2, x_3 \xleftarrow{\$} P$. $\mathcal{A}$ defines the query sequence to be

$$A = ((read, x_1, \perp), (read, x_2, \perp), (read, x_3, \perp))$$

In the second, $\mathcal{A}$ selects distinct $x'_1, x'_2, x'_3 \xleftarrow{\$} O$ and defines the query sequence to be

$$A' = ((read, x'_1, \perp), (read, x'_2, \perp), (read, x'_3, \perp))$$

We will examine the probability that the access pattern in the table is the same for all 3 queries.

In the second experiment, the locations are accessed at random. Recall we use 2-table cuckoo hashing, and each table is of size $\Theta(m)$. Therefore, the total number of pairs of locations that may be accessed, which we denote $r$, is order $\Theta(m^2)$. Thus, in the second experiment, the probability that all 3 indices are hashed to the same two locations is $1/r^2$.

In the first experiment, there are 4 scenarios to consider. Of the values $\{x_1, x_2, x_3\}$ either 3, 2, 1, or 0 of these are in $T$.

**Case 0: 3 locations in T**. If all 3 items are in $T$, it is impossible for the all 3 queries to access the same pair of locations. Since there are only 2 locations, these cannot store all 3 items. Therefore, in this case, the probability that the query locations are identical for all 3 accesses is 0.

**Case 1: 2 locations in T**. Let $x_i, x_j$ be the indices which are in $T$, and $x_k$ the index which is not in $T$ (ie in $U$). Let $p$ be the probability that $(H_1(x_i), H_2(x_i)) = (H_1(x_j), H_2(x_j))$ for distinct $x_i$, $x_j$ chosen uniformly at random from $T$. Since $x_k \notin T$, $(H_1(x_k), H_2(x_k))$ will be distributed uniformly at random and independently of previous values. If $x_k \in S$ this is due to the stash resampling. If $x_k \in R$ this is because $x_k$ has never been queried on these hash functions previously. Therefore, the probability that all 3 queried items map to the same location is $\frac{p}{r}$.

**Case 2: 1 location in T**. Let $x_i$ be the index in $T$ and $x_j, x_k \in U$. Since $(H_1(x_j), H_2(x_j))$ and $(H_1(x_k), H_2(x_k))$ are each chosen independently and uniformly at random from $r$ values, the probability that $(H_1(x_i), H_2(x_i)) = (H_1(x_j), H_2(x_j)) = (H_1(x_k), H_2(x_k))$ is $\frac{1}{r^2}$.

**Case 3: 0 locations in T**. In this case, all accessed locations are chosen at random from the set of $r$ possible pairs of table locations. Therefore the probability that $(H_1(x_i), H_2(x_i)) = (H_1(x_j), H_2(x_j)) = (H_1(x_k), H_2(x_k))$ is $\frac{1}{r^2}$.

Hence, the probability that all 3 items hash to the same locations in the first experiment is:

$$\frac{\binom{|T|}{3}\binom{|U|}{0}}{\binom{|P|}{3}} \cdot 0 \; + \frac{\binom{|T|}{2}\binom{|U|}{1}\binom{3}{1}}{\binom{|P|}{3}} \cdot \frac{p}{r} \; + \frac{\binom{|T|}{1}\binom{|U|}{2}\binom{3}{2}}{\binom{|P|}{3}} \cdot \frac{1}{r^2} \; + \frac{\binom{|T|}{0}\binom{|U|}{3}}{\binom{|P|}{3}} \cdot \frac{1}{r^2}$$

The probability that all 3 queries hash to the same locations in the second experiment is $\frac{1}{r^2}$, which can be rewritten as:

$$\frac{\binom{|T|}{3}\binom{|U|}{0}}{\binom{|P|}{3}} \cdot \frac{1}{r^2} \; + \frac{\binom{|T|}{2}\binom{|U|}{1}\binom{3}{1}}{\binom{|P|}{3}} \cdot \frac{1}{r^2} \; + \frac{\binom{|T|}{1}\binom{|U|}{2}\binom{3}{2}}{\binom{|P|}{3}} \cdot \frac{1}{r^2} \; + \frac{\binom{|T|}{0}\binom{|U|}{3}}{\binom{|P|}{3}} \cdot \frac{1}{r^2}$$

The difference of these (latter minus former) is therefore:

$$\frac{\binom{|T|}{3}\binom{|U|}{0}}{\binom{|P|}{3}} \cdot \frac{1}{r^2} \; + \frac{\binom{|T|}{2}\binom{|U|}{1}\binom{3}{1}}{\binom{|P|}{3}} \cdot (\frac{1}{r} - p)\frac{1}{r}$$

If $|\frac{1}{r} - p| = m^{-\omega(1)}$ then the second term is negligible in $m$, but the first term is not negligible in $m$ and the sum will also not be negligible in $m$. This would mean that the distributions of the access patterns are statistically different by a non-negligible amount, breaking obliviousness.

Therefore, the only way for the entire result to be negligible in $m$, we require that $|\frac{1}{r} - p|$ is non-negligible in $m$. We now show that this would lead to an attack as well. Recall that $p$ is the probability that a randomly chosen pair of indexes from $T$, $x_i$ and $x_j$, satisfy $(H_1(x_i), H_2(x_i)) = (H_1(x_j), H_2(x_j))$. This means that for $x_1, x_2$ chosen as distinct random values from $P$, the probability that $(H_1(x_1), H_2(x_1)) = (H_1(x_2), H_2(x_2))$ is

$$\frac{\binom{|T|}{2}\binom{|U|}{0}}{\binom{|P|}{2}} \cdot p \; + \frac{\binom{|T|}{1}\binom{|U|}{1}\binom{2}{1}}{\binom{|P|}{2}} \cdot \frac{1}{r} \; + \frac{\binom{|T|}{0}\binom{|U|}{2}}{\binom{|P|}{2}} \cdot \frac{1}{r}$$

However, in the second experiment, where $x'_1, x'_2$ are distinct random values from $O$, the probability that they are hashed to the same pair of locations is $\frac{1}{r}$. The difference of these (latter minus former) is therefore:

$$\frac{\binom{|T|}{2}\binom{|U|}{0}}{\binom{|P|}{2}} \cdot (p - \frac{1}{r})$$

If $|\frac{1}{r} - p|$ is non-negligible in $m$, these probabilites are non-negligibly different.

Therefore, the distributions of the access patterns differ in the two experiments. If $p$ is statistically close to $\frac{1}{r}$ the experiments will have statistically significant differences in the probabilities that all 3 queries access the same pair of locations. If $p$ is statistically distant from $\frac{1}{r}$, the experiments will have statistically significnat differences in the probabilities that the first 2 queries access the same pair of locations. Either way, the distribution of the access patterns will have a non-negligible statistical distance. □

3.5.2. Attack against PanORAMa and OptORAMa

In PanORAMa and OptORAMa, rather than each ORAM level containing a single Cuckoo Hash Table, each level has a number of equal-size bins, an Overflow Table and a (level-specific) Combined Stash. The bins, the Overflow Table and the Combined Stash are all implemented as Cuckoo Hash Tables. The Combined Stash Table contains the combined stashes of all bins on that level. The Overflow Table and the Combined Stash additionally have their own stashes. These stashes are removed from the level and reinserted into the ORAM.

Provided that items found in the Combined Stash are still searched for at each bin, the fact that the stashes of all bins in a given level are combined is not an issue.[5] However, the fact that the stashes of the Overflow Table and of the (level-specific) Combined Stash are removed from the level and re-inserted into the ORAM makes the protocols vulnerable to the attack described in this paper.

Like in the regular ORAM attack, let $(P_1, \ldots, P_m)$ be a sequence of distinct indices of length $m$, where $n^{-\Theta(1)} \leq m \leq n - 3$ such that following a sequence of accesses to these indices, a level $L_i$ is built using the set $P = \{P_1, \ldots, P_m\}$ as input.

Let $T$ be the Overflow Hash Table,[6] and $X$ be the set of items input the the Build function. $X$ is unknown to the adversary, but it is guaranteed that $X \subseteq P$. Let $S$ be the set of stashed elements in the Overflow Hash Table.

Observe that if an index $x \in S$ is queried, PanORAMa and OptORAMa will find $x$ before reaching $L_i$ and will query a nonce in $T$ instead. Therefore, the access sequence to the Overflow Hash Table in the ORAM is the same as that of a Stash-Resampling Cuckoo Hash Table.

Since the Overflow Hash Table is not access-oblivious, to an adversary that knows $P \supseteq X$, by

[5] OptORAMa seaches in the Combined Stash *after* searching in the bins, so the access pattern in the bins will be the same for items that are later found in the Combined Stash. However, in PanORAMa, the Combined Stash is accessed *before* the bins are accessed and a random bin is chosen in the case that the data is found in the Combined Stash. Therefore, the access patterns in the individual bins are also vulnerable to a distinguishing attack based on the fact that stashed elements will not be searched for. This can simply be solved by searching the bins before searching the Combined Stash.

[6] The proof would work out the same if $T$ was the Combined Stash Hash Table.

Theorem 7, the ORAM protocols are not access-oblivious either. In particular, let the adversary choose distinct $x_1, x_2, x_3$ uniformly at random from $P$. Let $A = (P_1, \ldots, P_m, x_1, x_2, x_3)$. Let $x'_1, x'_2, x'_3 \notin P$ be distinct elements and $A' = (P_1, \ldots, P_m, x'_1, x'_2, x'_3)$. The access sequences of the ORAM on $A$ and $A'$ will have non-neglible statistical distance in $m$ (and $n$).

### 3.6. Alibi: Secure Hierarchical ORAM with Reinserted Stashes

The basic problem arises when a stashed item is found *before* the appropriate level of the ORAM hierarchy is searched. As a successful criminal needs not only to be hidden in the location where they committed a crime, but also needs an alibi who claims to have seen them enacting their everyday life, likewise the stashed items need not only hide their presence in the tables to which they are reinserted, but also need to hide their absence from the tables from which they came. To fix this problem, we need to ensure that even when an item cannot be *stored* at a certain table of the ORAM hierarchy (*i.e.,* because it falls in the cuckoo stash), it must still be *searched for* at this table. This way, the set of physical accesses in a table will always be chosen uniformly at random and be fully independent. Each item therefore needs to store a record of the locations where it would have been, and needs to be searched for in these locations if accessed.

There are some small subtleties here. First, an item needs to store the fact that it was ejected from a table not only when it is in the cache, but at least until this table is rebuilt or the item is searched for, since if it is looked up at *any* point before this table is rebuilt it needs to be searched for in this table. Second, it is entirely possible that the same item that had been stashed in some table $T_{i,j}$ could be stashed again in some other higher table $T_{i',j'}$, where $i'b - j' < ib - j$, *before* $T_{i,j}$ is rebuilt or the item queried. Therefore each item needs to store the location of all tables from which it was ejected due to having fallen in the stash. Since there are at most $bl \leq b \log_b(n)$ tables in the hierarchical ORAM, it is possible to store which tables the item was ejected from using $b \log_b(n)$ bits.

The flaw can be fixed using the following simple modification. For each item $(x, y)$ the algorithm will additionally store a bit array $\mathfrak{e}$ of length $(b-1)l$, which records from which tables the item has been "stashed."

Our solution is presented in Figure 3.5. It makes use of a weaker primitive than an OHTable, instead it uses an oblivious implementation of a Stashing Hash Table.

**Definition 3.6.1.** A *Stashing Hash Table* is a functionality similar to a Hash Table (Figure 2.2), except that, during a build, it outputs two objects: the table itself and an array of rejected items, padded to a fixed length, which is referred to as the stash. The dictionary structure only holds the non-rejected items. Like with a Hash Table, its behavior is only defined if there are no repeated queries. An *Oblivious Stashing Hash Table* (OSHTable) is an implementation of a Stashing Hash Table which is oblivious on the condition that the event of placing an item in the stash does not affect its probability of being queried to the table.

Cuckoo hashing with a stash satisfies this property. Effectively, the condition ensures that stashed items are never resampled, so the physical access pattern in the table is always uniformly random and independent of previous accesses. Many other types of hashing with a stash can be conceived which would also satisfy this property.

In order to build a hierarchical ORAM using this weaker primitive of an OSHTable, the Alibi protocol modifies the generic hierarchical ORAM protocol of Figure 2.3 in several ways. Firstly, it takes the stash, produced by the OSHTable, and stores it in the cache. Secondly, for each item, the Alibi solution stores a bit array which records which tables the item has been stashed from, but not yet queried to. Thirdly, the Alibi solution updates this array appropriately, when an item is queried, when it is placed in a stash, and when a table from which it was stashed is rebuilt. Together, these guarantee that the query pattern when a stashed item is queried is not resampled.

**Lemma 4.** In the Alibi protocol presented in Figure 3.5 there is an invariant that given a tuple $(x, y, \mathfrak{e})$ stored at some level, $\mathfrak{e}[i-1][j] = 1$ if and only if:

- $x$ was stashed in Table $T_{i,j}$ during its last rebuild
- $x$ has not been queried by the ORAM since this rebuild
- $T_{i,j}$ is not empty

**Alibi Stash-Reinserting Hierarchical ORAM/OMap Template**

**Parameter choices:**
$c$: The size of the cache is 2c, typically $c = \Theta(\log(n))$
$b$: Ratio of adjactent level sizes, typically 2. Each level has up to $b-1$ tables.
$l$: The number of levels, $\log_b(n/c)$, typically $\Theta(\log(n))$. Assume $n/c$ is a power of $b$.

Init(m, n, d):

- **Inputs:** $m$, the number of items; $n$, the size of the index space; $d$, the bit-length of the values
- **Create the cache:** Initialize an empty array, $C$, the cache, with capacity to store $2c$ items.
- **Initialize access counter:** Set $t = 0$
- **Create empty tables at each level:** Set $T_{i,j} = \perp$ for $i = 1 \dots \ell$ $j = 0 \dots b-2$

Access(op, x, y):

- **Inputs:** *op*, either read or write; $x$, the index to query (from $[0, n-1]$);
  $y$, if op = write, the value to write.
- **Prepare:** Set $found = false$. Set $value = \perp$. Set $\mathfrak{f} = (0^{b-1})^l$.
- **Scan the cache:** for $j$ in $t \mod c \dots 0$:
  If $C_j = (x, y_{old}, \mathfrak{e})$ for some values $y_{old}$, $\mathfrak{e}$ set $found = true$, set $value = y_{old}$, set $\mathfrak{f} = \mathfrak{e}$ and set $C_j = (\perp, \perp, \perp)$.
- **Query the OHTables:** for $i$ in $1 \dots l$, for $j$ in $b-2 \dots 0$, if $T_{i,j} \neq \perp$:
  If $found = false$ OR $\mathfrak{f}[i-1][j] = 1$, set $q = x$, otherwise set $q = n + t$.
  $(y_{old}, \mathfrak{e}) \leftarrow T_{i,j}.Query(q)$.
  If $found = false$ AND $y_{old} \neq \perp$, set $found = true$, set $value = y_{old}$ and set $\mathfrak{f} = \mathfrak{e}$.
- Set $t = t + 1$
- **Insert item into cache, updating if it is a write:** If $op = write$, set $C_{t \mod c} = (x, y, (0^{(b-1)})^l)$
  If $op = read$, set $C_{t \mod c} = (x, value, (0^{(b-1)})^l)$
- **Rebuild if needed:** If $t \mod c = 0$, Rebuild()
- **Output:** *value*. For a read, this returns the read value.

Rebuild():

- **Identify Level:** Let $\bar{i}$ be the largest value such that $t/c = 0 \mod b^{\bar{i}}$. Let $i^* = min(\bar{i}+1, \ell)$. Let $j^* = t/c \mod b^{\bar{i}-1}$. We will merge levels $0, \dots, \bar{i}$ into a new table at level $i^*$.
- **Merge Levels:** Initialize $Z = C$. For $i = 1, \dots, \bar{i}$, $j = 0, \dots, b-2$, obliviously evaluate $Z = Z \cup T_{i,j}.Extract()$.
  For $(x_k, y_k, \mathfrak{e}_k) \in Z$, set $\mathfrak{e}_k[i-1][j] = 0$ for all $i \leq \bar{i}$, $j \in [0, b-2]$
  Set $(T_{i^*,j^*}, S) = OSHTable.Build(Z, 2n, c \cdot b^{i^*-1}, d)$.
- **Update Alibi Bits of Stash:** For $k = 0, \dots, c-1$:
  Let $S_k = (x_k, y_k, \mathfrak{e}_k)$. If $x_k \neq \perp$, set $\mathfrak{e}_k[i^*-1][j^*] = 1$.
- **Clear Lower Levels:** For $j = 0, \dots, c-1$, set $C_j$ to be empty and set $C_{c+j} = S_j$.
  For $i = 1, \dots, i^*-1$, $j = 0, \dots, b-2$, set $T_{i,j} = \perp$.
- **Reset Counter if Needed:** If $t = n$, set $t = 0$.

Figure 3.5: Alibi Stash-Reinserting Hierarchical ORAM/OMap Template

This invariant holds initially, after each query and after each rebuild.

*Proof.* By induction.

When an item is first accessed in the ORAM, it will have $\mathfrak{e}[i-1][j] = 0$ for all $i \in [1, l], j \in [0, b-2]$ and it will not have been stashed from any table.

The only way that $\mathfrak{e}[i-1][j]$ can be set to 1 is in the building of table $T_{i,j}$, in the case that the item is placed in the stash. Therefore, the invariant will hold immediately after a build that involves the item being stashed from $T_{i,j}$.

There are only two ways that $\mathfrak{e}[i-1][j]$ can be reset to 0. The first is that the item is queried. In this case it will be found in some table (higher than $T_{i,j}$) will be queried at $T_{i,j}$, and then will be stored in the cache with $\mathfrak{e}[i-1][j]$ (and in fact all Alibi bits) set to 0.

The second way for $\mathfrak{e}[i-1][j]$ to be reset to 0 is for the contents of $T_{i,j}$ to be extracted and merged. In most cases, this will mean that $T_{i,j}$ will be set to empty, so the invariant will be preserved. The only exception is when $i = l$, in which case $j = 0$ and $T_{i,j}$ is the largest table. In this case, $T_{i,j}$ will be rebuilt, so the invariant is also preserved.

Therefore, by induction, this invariant always holds. □

**Theorem 8.** The Alibi Stash-Reinserting Hierarchical ORAM template (Figure 3.5), when instantiated with an Oblivious Stashing Hash Table, is an oblivious implementation of a RAM.

*Proof.* Like the general Hierarchical ORAM template of Figure 2.3, the Alibi template maintains the invariant that for each index which has been queried, there is at most one "copy" of that item in the data structure. The rationale for this is almost identical to that of the proof of Theorem 1, except that items can additionally be moved by being stashed, in which case an item is removed from a table immediately during its rebuild and placed in the cache. This single copy of the item will definitely be found during a query, and stores the most recently written value for this item,

ensuring that the protocol is correct.

We now show that the protocol is oblivious, that is the access pattern is simulatable.

Firstly, the "no-overtaking" principle from the proof of Theorem 1 remains valid. An item that is queried will be removed from its table and moved to the top of the hierarchy. It will remain higher in the hierarchy than the table until the table is extracted and it is merged with the contents of the table. In the proof of Theorem 1 this allowed us to guarantee that a queried item would never be re-queried to an OHTable.

If an item was stashed from some table $T_{i,j}$, the "no-overtaking" principle guarantees that, if $T_{i,j}$ has not yet been extracted, the item will be found before $T_{i,j}$ is reached. If this is the first time that the item has been queried since $T_{i,j}$'s last rebuild then, by Lemma 4, $e[i-1][j] = 1$. Therefore the item *will* be queried in $T_{i,j}$, so the event of the item being queried in $T_{i,j}$ is not affected by the item having been placed in the stash. If this is not the first time that the item has been queried since $T_{i,j}$'s last rebuild then, by Lemma 4, $e[i-1][j] = 0$, and the item will not be re-queried in $T_{i,j}$. $T_{i,j}$ is an OSHTable, so since the probability of a stashed item being queried is unaffected by it being in the stash, and since an index will never be re-queried, $T_{i,j}$ will be an oblivious implementation of the Stashing Hash Table functionality.

The OSHTable is therefore simulatable. All other parts of the protocol have a deterministic access pattern. Therefore, there exists a simulator for the entire protocol. □

**Remark 5.** It may initially seem that the proof of security above would apply to the flawed schemes as well. However, because the schemes *resample* the queries based on whether they were stored in the stash, the access pattern of the remaining table changes, and changes specifically in a way that depends on the structure of the table. We showed that in the case of Cuckoo Hashing this change causes a change in the combined set of accesses that is distinguishable.

Note that this fix also applies to PanORAMa and OptORAMa. Even though these protocols contain multiple Cuckoo Hash Tables at each level, it is possible to view the entire level as a single Oblivious

Hash Table with a stash. (The stash of the level would be the union of the stashes of the Overflow Table and the level-specific Combined Stash Table).

**Complexity:** Since each item only needs to store one bit for each table, and there are at most $b\log(n)/\log(b)$ levels, then the additional size of each item is increased by $b\log(n)/\log(b)$ bits. In the common case, where $b = \Theta(1)$, (e.g. [GM11], [GMOT11], [LO13a], [PPRY18], [AKL+20]) this will simply be $\Theta(\log(n))$ bits. Since each item holds a $\log(n)$-bit index, this would not increase the asymptotic cost of the protocol. Likewise, a simple inspection of the steps that involve modifications to $\mathfrak{e}$ shows that these would not increase the asymptotic cost of the protocols either.

In the case where $b = \omega(1)$, (e.g. [KLO12], [KM19]) each item needs to include $b\log(n)/\log(b) = \omega(\log(n))$ Alibi bits. If the block size is large, this again does not increase the asymptotic cost of the protocol. For instance Kushilevitz and Mour consider a block size $d = \Omega(b\log(n)) = \omega(b\log(n)/\log(b))$. However, if the block size is small, the Alibi bits could result in an increase in the asymptotic communication cost.

Therefore, our fix applies to all protocols, and with the exceptions of [KLO12] and [KM19] instantiated with small blocks, does so without changing the asymptotic communication cost.

### 3.7. Summary of Affected Papers

Goodrich and Mitzenmacher (ICALP 2011. [GM11]) introduced the idea of using Cuckoo tables with combined stashes for Hierarchical ORAM. This introduced the flaw described in this paper. Kushilevitz et al. (SODA 2012, [KLO12]) later introduced the alternative approach of reinserting elements from the stash into the ORAM ("cache the stash"). While there are differences between these approaches, in either case an element that was stashed will be found prior to the the level from which it was ejected and random locations accessed at this level instead. Therefore both approaches are vulnerable to our attack.

Several works build on these ORAM protocols and inherited these flaws. Goodrich and Mitzenmacher's protocol was used as the basis for a de-amortized ORAM by Goodrich et al. (CCSW 2011, [GMOT11]). The de-amortized ORAM used the same idea of combining the stash and there-

fore is also vulnerable to the attack described in this paper.[7]

In the multi-server setting, Lu and Ostrovsky (TCC 2013, [LO13a]) used the stash-reinsertion of [KLO12] in their 2-server ORAM protocol, inheriting this vulnerability. Similarly Kushilevitz and Mour (PKC 2019, [KM19]) created a 3-server ORAM that also uses cuckoo hashing (Instantiation 2) based on [KLO12], but using a shared stash [GM11] rather than reinserting the stash. This ORAM protocol is therefore vulnerable to the attack from this paper. Kushilevitz and Mour also present other multi-party ORAM protocols based on other techniques which are not subject to this attack.

Chan et al. presented a hierarchical ORAM based on an alternative Oblivious Hash Table called two-tier hashing [CGLS17]. Two-tier hashing used two hash tables, each with bins of size $\log^{\epsilon}(\lambda)$ for some constant $\epsilon \in (0.5, 1)$ and security parameter $\lambda$. They presented an oblivious construction in which elements would be placed in the first hash table if possible and in the second if not. They showed that the probability that an element could not be placed was negligible. Since this protocol used two-tier hashing rather than Cuckoo hashing with a combined stash it is *not vulnerable* to the attack we have presented. Chan et al. also presented a concrete instantiation of Goodrich and Mitzenmacher's ORAM protocol in an appendix of the full version of the same work. The protocol they present uses a Cuckoo Hash Table at each level and a shared stash, so is vulnerable to the attack described in this paper. However, they recommend, somewhat clairvoyantly, that since Cuckoo hashing is complex and hard to prove correct, that their two-tier hash-table protocol should be used rather than the Cuckoo-hashing protocol.

The flaw also affected the the recent single-server asymptotic breakthroughs of PanORAMa (FOCS 2018, [PPRY18]) and OptORAMa (EUROCRYPT 2020, [AKL$^+$20]).[8] These achieved efficiency by storing most of the data in small bins, which are small enough to be sorted without increasing the asymptotic performance, while remaining items are placed in an overflow pile. Each of these bins is implemented as a cuckoo table and stashes are shared, but the combined stash for the bins is kept

[7] Goodrich et al. [GMOT11] also presented a de-amortization of the original square-root ORAM [Gol87], which is not vulnerable to the attack described in this paper.

[8] In response to our preprint, Asharov et al. have updated the OptORAMa paper to include a fix.

at the same level as the bins. Therefore it is possible to search the bins for the stashed elements and then to access the single-level combined stash, so the bin tables are not vulnerable to this attack. However, in both papers, the overflow and single-level combined stash cuckoo tables both have stashes that are re-inserted into the ORAM data structure. They are therefore vulnerable to the variant of our attack in Section 3.5.

Our attack *does not* affect the tree-based ORAM protocols, such as Binary Tree ORAM [SCSL11], Path ORAM [SvDS+13] and Circuit ORAM [WCS15], as these do not use cuckoo hashing.

In summary, this flaw existed in the the ORAM literature for almost a decade and has affected seven protocols in published works, including the most recent asymptotic breakthroughs. The fact that such a flaw could exist unnoticed for so long motivates the development of simpler protocols for oblivious data structures.

# CHAPTER 4

## Distributed Oblivious RAM from Oblivious Set Membership

*The chapter is based on material which was first published in the following work:*

Brett Hemenway Falk, Daniel Noble and Rafail Ostrovsky. 3-party distributed ORAM from oblivious set membership. In *International Conference on Security and Cryptography for Networks,* pages 437-461. Springer, 2022. [FNO22]

*I contributed to all aspects of the work.*

### 4.1. Introduction

In Chapter 3 we showed that the flaw in Cuckoo Hashing-based Hierarchical ORAMs/DORAMs could be remedied through the Alibi technique. This ensured that every stashed item, which had been moved to another location in the hierarchy, stored information about which table(s) it had been removed from due to stashing. The ORAM/DORAM protocol would then query the item itself in these tables, rather than performing a dummy query, ensuring the access distribution within a table was the same as in the case that the stash had not been removed. This made the full ORAM/DORAM secure without increasing the asymptotic cost.

However, the Alibi fix is not fully satisfying on a conceptual level. The Hierarchical solution is conceptually beautiful because it allows a modular solution to ORAM based on OHTables. Re-inserting a stash, even with the Alibi fix, does not manage to maintain this abstraction. Re-inserting or combining stashes means that the cuckoo hashing tables fail to implement the Hash Table functionality, since the table cannot store all of the items given to it.

In this chapter, we present a DORAM that *does* maintain this abstraction. We do this by building Distributed Oblivious Hash Tables which only require $\Theta(\kappa + d)$ communication per access. To our knowledge, this is the first DOHTable with this efficiency which does not make use of homomorphic encryption.

**Theorem 9** (Distributed Oblivious Hash Table (informal))**.** Protocol $\Pi_{DOHTable}$ (Figure 4.9) implements a 3-party Distributed Oblivious Hash Table with amortized per-query communication complexity of $\mathcal{O}(\kappa + d)$ bits. The protocol provides security in the semi-honest model against one corruption.

Using the Hierarchical DORAM/DOMap Template (Figure 2.6), instantiated with $b = 2$ and $c = \log(n)$ this automatically results in a DORAM/DOMap with $\Theta(\log(n)(\kappa + d))$ communication per access.

**Theorem 10** ((3, 1)-DORAM (informal))**.** There exists a 3-server DORAM protocol with amortized communication complexity $\mathcal{O}((\kappa + d) \log n)$ bits that maintains the abstraction of the Hierarchical template (Figure 2.6). The protocol provides security in the semi-honest model against one corruption.

In terms of asymptotics, this is equal to [LO13a]. However, as discussed above, it has the conceptual advantage that it maintains the abstraction of the Hierarchical template. Furthermore, it is significantly more efficient in practice. [LO13a] requires about 50× more secure PRP evaluations than our protocol (see Section 4.9). For certain parameter ranges our protocol is therefore the most communication-efficient DORAM protocol.

The key tool that we use to achieve this is a novel set-membership data structure that has negligible failure probability in $n$, and only reads $\mathcal{O}(\log n)$ *bits* per access.

**Theorem 11** (Set-Membership Data structure)**.** The data structure outlined in Section 4.3 can store $m = \omega(\log n)$ elements, from a universe of size $n$, with linear storage overhead ($\mathcal{O}(m \log n)$-bits), negligible false-positive rate (in $n$), zero false-negative rate, negligible probability of build failure (in $n$) and logarithmic lookup cost ($\mathcal{O}(\log n)$ bits).

Note that these properties are *not* simultaneously satisfied by existing data structures like Cuckoo Hash Tables and Bloom Filters. Cuckoo hashing has a non-negligible probability of build failure, cuckoo-hashing with a stash has $\mathcal{O}\left(\log^2 n\right)$ lookup cost, and Bloom Filters cannot simultaneously

achieve logarithmic lookup cost and negligible false-positive rates. See Section 4.3 for a more detailed discussion.

## 4.2. Construction Overview

We design a novel DOHTable that requires, on average, $\Theta(\kappa + d)$ bits of communication per access. Our starting point is the observation (e.g. from [MZ14]) that once it is known whether an element is stored in a table, DOHTables can be constructed using any non-oblivious, but secret-shared, hash table structure by searching for distinct pre-inserted dummy elements when an element is not in the set.

This essentially reduces the problem to that of designing an efficient data structure for *set membership*. We do this by building a Cuckoo Hash Table with a stash, but instantiating the stash with a Bloom Filter. Surprisingly, this simple combination increases the asymptotic efficiency of the data structure beyond what can be achieved by Cuckoo Hash Tables or Bloom Filters alone.

We first, in Section 4.3, present this data structure. We then introduce two functionalities that will be needed by our MPC protocols: Shared-Input Shared-Output PRPs (SISO-PRPs) (Section 4.4) and Secure shuffles (Section 4.5). We show how these can be instantiated securely using existing solutions. We then show how the set membership data structure of Section 4.3 can be implemented in a MPC protocol, to create a Distributed Oblivious Set (DOSet) protocol (Section 4.6). This is achieved by enabling one party to build the needed data-structures locally, avoiding the need for an *oblivious* build process which can be very challenging [CGLS17]. We use our DOSet to construct a DOHTable (Section 4.7), which in turn we use to construct the desired DORAM (Section 4.8). Lastly, Section 4.9 shows that, despite being the same asymptotically, our construction is concretely about 50 times more efficient than that of [LO13a].

## 4.3. Set Membership Data Structure

Let there be some set of $m$ elements from a universe of size $n$, each represented by $\log n \geq \log m$ bits. In this section we outline a novel data structure that supports *set membership queries* that simultaneously achieves the following properties:

1. Linear storage overhead ($\mathcal{O}(m \log n)$)

2. Negligible false-positive rate in $n$

3. Zero false-negative rate

4. Negligible probability of build failure in $n$

5. Logarithmic lookup cost ($\mathcal{O}(\log n)$)

Bloom filters and Cuckoo hash tables are widely used data structures that provide efficient storage and retrieval, but they do not satisfy all of the above design criteria simultaneously.

**Example 1** (Cuckoo Hashing (Figure 4.1))**.** Cuckoo Hashing [PR04] is a common data structure for implementing dictionaries; by storing only the index it can also be used to implement a set. Standard Cuckoo Hash Tables have linear storage overhead, zero false-positive rate, and logarithmic[9] lookup cost. Unfortunately, Cuckoo Hash Tables (without a stash) have a non-negligible probability of build failure.

**Example 2** (Cuckoo Hashing with a stash (Figure 4.1))**.** Modifying a standard Cuckoo Hash Table to include a "stash" of size $s = \Theta(\log n)$, for any $m = \omega(\log n)$ makes the failure probability negligible in $n$ [Nob21]. Unfortunately, every lookup query scans the entire stash, which requires reading $s$ locations, which means lookups require accessing $\Theta(\log^2 n)$ bits of memory.

**Example 3** (Bloom filters (Figure 4.2))**.** The false-positive rate for a Bloom filter of size $w$ storing $m$ elements (using $k$ hash functions) is about $\left(1 - e^{-\frac{km}{w}}\right)^k$. A standard analysis (e.g. [MU17][Chapter 5]) shows that the false-positive rate is minimized when $k = \log(2) \cdot (n/w)$, which makes the false-positive probability approximately $(\log 2)^{-w/m}$. Thus to make the false-positive probability negligible in $n$, we need $w = \omega(m \log n)$, which means that the storage overhead is super-linear.

Although Cuckoo Hashing and Bloom filters alone cannot achieve our five goals (linear storage

[9]Note that lookups require looking in a constant number of locations, but each location stores an identifier which must be at least $\log n$ bits, so the total lookup cost requires transmitting (at least) $\Theta(\log(n))$ bits.

**Parameters:** $\epsilon > 0$, a constant, (Stash variant: $s$, the stash size)
**Build:** Given a set $\{X_0, \ldots, X_{m-1}\}$, where $X_i \in [0, n-1]$

1. Pick 2 hash functions $H_1$, $H_2$ which map $[0, n-1] \to [0, (1+\epsilon)m - 1]$.
2. Create 2 empty tables, $T_1$ and $T_2$, each of size $(1+\epsilon)m$.
3. Try to store each $X_i$ in either $T_1[H_1(X_i)]$ or $T_2[H_2(X_i)]$. Find a maximal allocation (*e.g.,* through a matching algorithm).
4. (No stash variant) If there exist elements that were not able to be stored in either $T_1$ or $T_2$, the build fails.
5. (Stash variant) Let $S$ be the set of elements that were not able to be stored in either $T_1$ or $T_2$. If $|S| > s$, the build fails. Store $S$.

**Query:** Given an index $x$

1. Check if $x$ is stored in the Cuckoo hash table by checking whether $T_1[H_1(x)] = x$ or $T_2[H_2(x)] = x$. If so, return true.
2. (Stash variant only) Check whether $x$ is stored in $S$. If so, return true.
3. Return false.

Figure 4.1: Cuckoo Hashing (potentially with a stash), as modified for Set Membership

**Parameters:** $w$: size of Bloom filter, $k$: number of hash functions
**Build:** Given a set $\{X_0, \ldots, X_{m-1}\}$

1. Create a bit array, $B$, of length $w$ of all zeros.
2. Pick $k$ hash functions, $g_0, \ldots, g_{k-1}$, which map $[0, n-1] \to [0, w-1]$.
3. For $j \in [0, m-1]$, $i \in [0, k-1]$ set $B[g_i(X_j)] = 1$.

**Query:** Given an index $x$
Check whether $B[g_i(x)] = 1$ for all $0 \leq i \leq k-1$. If so, return true. Otherwise return false.

Figure 4.2: Bloom Filter

overhead, negligible false-positive rate, zero false-negative rate, negligible probability of build failure and logarithmic lookup cost), *combining* the Cuckoo Hashing with Bloom filters allows us to simultaneously achieve all these goals. This is achieved simply by creating a Cuckoo Hash table with a stash, but storing the stash in a Bloom filter.

**Theorem 12.** When $m = \omega(\log n)$, the Set Membership Data Structure of Figure 4.3 has linear storage overhead, negligible false-positive rate (in $n$), zero false-negative rate, negligible probability of failure (in $n$) and logarithmic lookup cost (in bits).

*Proof.* **Storage overhead:** The total storage of the data structure is $|T| + |B|$. $T$ has $\mathcal{O}(m)$

**Build:** Given a set $\{X_0, \ldots, X_{m-1}\}$

1. Create a Cuckoo Hash table with a stash as follows:
   (a) Pick 2 hash functions $H_1$, $H_2$ which map $[n] \to [(1+\epsilon)m]$ for some constant $\epsilon > 0$.
   (b) Create 2 empty tables, $T_1$ and $T_2$, each of size $(1+\epsilon)m$.
   (c) Try to store each $X_i$ in either $T_1[H_1(X_i)]$ or $T_2[H_2(X_i)]$. Find a maximal allocation (*e.g.,* through a matching algorithm). Let $S$ be the set of elements that were not able to be stored in either $T_1$ or $T_2$. If $|S| > \log n$, the build fails.
2. Store the stash in a Bloom Filter as follows:
   (a) Create an array, $B$ of length $m \log n$ of all zeros.
   (b) Pick $k = \log n$ hash functions, $g_0, \ldots, g_{k-1}$, which map $[n] \to [m \log n]$.
   (c) For each element $x \in S$, and for $0 \leq i \leq k-1$ set $B[g_i(x)] = 1$.

**Query:** Given an index $x$

1. Check if $x$ is stored in the Cuckoo hash table by checking locations $T_1[H_1(x)] = x$ or $T_2[H_2(x)] = x$. If so, return true.
2. Check if $x$ is stored in the Bloom filter, by checking whether $B[g_i(x)] = 1$ for all $0 \leq i \leq k-1$. If so, return true. Otherwise return false.

Figure 4.3: Set Membership

locations, each of size $\log n$ and $B$ has $m \log n$ bits so the total space is $\Theta(m \log n)$.

**False-positive rate:** Since Cuckoo Hash Tables have no false positives, the only way a false-positive can occur is during the Bloom Filter lookup. Given a Bloom Filter with $k$ hash functions, and a table $B$ of size $|B|$, storing $s$ elements, a standard analysis (e.g. [MU17][Chapter 5]) shows that the probability that a false positive occurs approaches

$$\left(1 - e^{-\frac{ks}{|B|}}\right)^k.$$

Here $k = \log n$, $s \leq \log n$ and $|B| = m \log n$. Therefore, the probability of a false positive is at most

$$\left(1 - e^{-\frac{\log^2 n}{m \log n}}\right)^{\log n} = \left(1 - e^{-\frac{\log n}{m}}\right)^{\log n}$$

For $m = \omega(\log(n))$, this is negligible in $n$.

**False negatives:** Any item in the set will be stored in either the Cuckoo Hash table or the Bloom

Filter. Since neither the Cuckoo hash table nor the Bloom Filter have false negatives, the probability of a false negative is 0.

**Build failure:** The build will only fail if $|S| > \log n$. For $m = \omega(\log n)$ and a stash of size $\log n$ Cuckoo Hashing with a stash of size $\log n$ will succeed except with probability negligible in $n$ [Nob21].

**Lookup cost:** The amount of memory accessed for the lookup is $2 \log n$ bits for searching the Cuckoo Tables, and $k = \log n$ bits for searching the Bloom filter, so the total amount of memory accessed is $\Theta(\log n)$. □

### 4.4. SISO-PRPs

Our protocol will need a functionality for evaluating Pseudo-Random Permutations (PRPs). A PRP is a keyed deterministic permutation that is computationally indistinguishable from a random permutation to an adversary that only learns inputs and outputs of the PRP, and in particular is not given any information about the key.

For the functionality we need, the inputs, outputs and keys are all secret-shared between the parties. Note that this is slightly different from the notion of an *Oblivious Pseudo-Random Function* (OPRF) [FIPR05]. Most OPRF protocols have focused on a 2-party evaluation of a PRF, where one party holds a key, $k$, and the other holds an input, $x$, and the output, $F_k(x)$ is delivered to one party. In our applications, however, it is critical that the inputs to the PRF are secret-shared, thus most existing OPRF protocols are not applicable. Also, for a PRF the function outputs a uniform random value, whereas in our work, we will focus on the case where the function is a pseudorandom permutation.

The definition for the *Shared-Input, Shared-Output PRP* (SISO-PRP) functionality is as follows:

**Functionality** $\mathcal{F}_{SISO-PRP}$

$[[k]] \leftarrow Keygen(\kappa, n)$: Generate a fresh sharing of a $\kappa$-bit PRP key, $k$, that maps $[0, n-1] \rightarrow [0, n-1]$.

$[[q]] \leftarrow Eval([[k]], [[x]])$: Evaluate $PRP_k(x)$. Return a fresh sharing of the result.

| Cipher | Blocksize | Data | rounds | AND gates |
|---|---|---|---|---|
| **AES** | 128 | 128 | 40 | 5120 |
| **LowMC (round optimized)** | 128 | 128 | 19 | 1824 |
| **LowMC (AND-gate optimized)** | 128 | 128 | 252 | 861 |
| **LowMC (round optimized)** | 10 | 10 | 32 | 288 |
| **LowMC (AND-gate optimized)** | 10 | 10 | 94 | 282 |

Table 4.1: Block cipher costs for 128-bit Security (AES-128 from [ARS+15][Table 2], LowMC from LowMCv3 security estimator )

Concretely, we imagine implementing our SISO-PRP using the "MPC-friendly" LowMC block cipher, which is highly optimized for evaluation as a SISO-PRP [ARS+15]. In addition to being MPC-friendly, LowMC has two additional features that make it useful in our setting. (1) LowMC has configurable block sizes, allowing us to reduce the communication and computational costs when the index space is small, and (2) when the maximum number of queries to the PRP is bounded (as is the case in our construction), LowMC can be instantiated with more aggressive parameters, increasing efficiency.

In Table 4.1, we compare the efficiency of LowMC, vs AES for 128-bit security. We present various parameter choices for LowMC using the LowMCv3 security estimator [ARS+17]. "Data" represents the log of the number of PRP evaluations the adversary will ever learn.

One appealing property of LowMC as a block cipher is that the security level can easily be configured. While [ARS+15] does not explicitly state the asymptotic relationship between the security level and the number of AND gates of a LowMC circuit, their implementation contains a script [ARS+17] to calculate the number of AND gates needed for any concrete security setting. From this it is evident that it is possible to perform $\kappa$-bit encryption (on $\kappa$-bit blocks, with at most $2^\kappa$ queries) with $\Theta(\kappa)$ AND gates (Table 4.4) with a constant of about 7. As shown in Table 4.1, the number of AND gates will be reduced if the input is fewer than $\kappa$ bits. If the bit-length of the input, $d$ is more than $\kappa$, LowMC can be converted to a stream cipher on $\kappa$-bit blocks, using standard techniques such as Counter Mode. Since the cost of encrypting each $\kappa$-bit block is $\Theta(\kappa)$, the cost becomes $\Theta(\frac{d}{\kappa} \cdot \kappa) = \Theta(d)$. Therefore, the communication cost for general $d$ is $\Theta(d + \kappa)$.

| $\kappa$ | Block Size | Data | AND gates | AND gates / $\kappa$ |
|---|---|---|---|---|
| 20 | 20 | 20 | 282 | 14.1 |
| 40 | 40 | 40 | 375 | 9.4 |
| 60 | 60 | 60 | 465 | 7.8 |
| 80 | 80 | 80 | 582 | 7.3 |
| 100 | 100 | 100 | 699 | 7.0 |
| 120 | 120 | 120 | 816 | 6.8 |
| 140 | 140 | 140 | 933 | 6.7 |
| 160 | 160 | 160 | 1050 | 6.6 |

Figure 4.4: LowMCv3 Complexity

## 4.5. Multiparty secure shuffles

Our protocol requires a secure shuffle protocol, denoted $\mathcal{F}_{Shuffle}$. Note that we sometimes use this functionality to shuffle multiple individual arrays of the same length using the same permutation. In this case this is implicitly treated as a single combined array, where the $i^{th}$ item in the combined array contains a tuple of the $i^{th}$ items from each of the individual arrays.

**$\mathcal{F}_{Shuffle}$ : Functionality for shuffle**

$[[\hat{A}]] \leftarrow Shuffle([[A]], m, d)$: Given a secret-shared array $A$, of length $m$, in which each item is of size $d$ bits, pick a uniformly random permutation, $\rho : [0, m-1] \rightarrow [0, m-1]$. Output a fresh secret-shared array $\hat{A}$, in which $A_i = \hat{A}_{\rho(i)}$ for all $i \in [0, m-1]$.

Figure 4.5: Functionality for Secure Shuffle

Although our protocol can be instantiated with any 3-party shuffle, we imagine using the multiparty shuffle of [LWZ11]. The key idea is that if a vector is secret shared among $s$ participants, with an $t$-out-of-$s$ secret sharing scheme, then for *every* subset $C$ of $s - t$ participants, the participants reshare the vector to the members of $C$, then the members of $C$ permute their shares *using* a shuffle that is public to all members of $C$. If there are only $t$ corrupt participants, there will be some subset $C$, that is completely honest, and the permutation chosen by this subset will remain hidden from the adversary.

The key benefit of this approach is that all the shuffles are done in the clear, and the only communi-

cation is the permutations and repeatedly re-sharing the vector. (If only computational security is needed, the permutations can be replaced by PRPs, and instead of sending random permutations, the parties can send PRP keys.)

In the $(3, 1)$ security setting this only requires 3 local shuffles and 4 resharings, so it is very efficient. The protocol is presented formally in $\Pi_{Shuffle}$ in Figure 4.6.

**$\Pi_{Shuffle}$: Protocol for Secure Shuffle**

**Shuffle**$([[A]], m, d)$:

1. Each party, $P_i$, picks a random permutation on $[0, m-1]$. Let $[[S_i]]_i$ be the permutation selected by $P_i$.
2. Each party $P_i$ sends $S_i$ to $P_{(i+1) \bmod 3}$. For $i = 0, 1, 2$:
   $[[S_i]]_{(i,i+1 \bmod 3)} \leftarrow [[S_i]]_i$
3. Each pair of parties, in turn, shuffles the data according to the permutation they both hold:
   (a) $[[A]]_{0,1} \leftarrow [[A]]$
   (b) $P_0$ and $P_1$ shuffle $[[A]]_{0,1}$ according to $[[S_0]]_{(0,1)}$ and store the result as $[[\hat{A}_0]]_{0,1}$
   (c) $[[\hat{A}_0]]_{1,2} \leftarrow [[\hat{A}_0]]_{0,1}$
   (d) $P_1$ and $P_2$ shuffle $[[\hat{A}_0]]_{1,2}$ according to $[[S_1]]_{(1,2)}$ and store the result as $[[\hat{A}_1]]_{1,2}$.
   (e) $[[\hat{A}_1]]_{2,0} \leftarrow [[\hat{A}_1]]_{1,2}$
   (f) $P_2$ and $P_0$ shuffle $[[\hat{A}_1]]_{2,0}$ according to $[[S_2]]_{(2,0)}$ and store the result as $[[\hat{A}_2]]_{2,0}$.
   (g) $[[\hat{A}]] \leftarrow [[\hat{A}_2]]_{2,0}$
4. Return $[[\hat{A}]]$

Figure 4.6: Secure Shuffle Protocol [LWZ11]

We show below that this protocol is a secure implementation of the shuffle functionality and has communication cost $\Theta(m(\log(m)+d))$. Note that in our applications $d = \Omega(\log(m))$, so this becomes simply $\Theta(md)$.

**Theorem 13.** Protocol $\Pi_{Shuffle}$ (Figure 4.6) is a secure MPC implementation of $\mathcal{F}_{Shuffle}$ (Figure 4.5).

*Proof.* Each party, $P_i$, receives only secret-shares and the permutations $S_i$ and $S_{i-1 \bmod 3}$. The shares can be simulated by selecting random values. The permutations $S_i$ and $S_{i-1 \bmod 3}$ are simple random, independent permutations. The simulator can therefore select these from the distribution

of all random permutations. Due to the re-randomizing effect of $S_{i+1 \bmod 3}$, the distribution of possible shuffles is still perfectly random conditioned on the knowledge of $S_i$ and $S_{i-1 \bmod 3}$, so the real and simulated executions are perfectly indistinguishable. □

**Theorem 14.** Protocol $\Pi_{Shuffle}$ required $\Theta(m(\log(m) + d))$ communication.

*Proof.* Each permutation can be represented using $m \log(m)$ bits, so the cost of sending these is $3m \log(m)$. There are 4 resharings, each of which requires $\Theta(md)$ communication. Thus, the total communication cost is $\Theta(m(\log(m) + d))$. □

## 4.6. Distributed Oblivious Set Membership

We now show how we can securely build and access the set-membership data structure presented in Section 4.3. This will be fundamental to our efficient Oblivious Hash Table construction. The functionality for the Distributed Oblivious Set data structure is presented below:

**$\mathcal{F}_{SS-Set}$: Secret-Shared Set**

**Build([[$X$]], n):**
Build an Oblivious Set data structure consisting of the elements in array $[[X]] = [[X_0]], \ldots, [[X_{m-1}]]$ where $X_i \in [0, n-1]$ are distinct for all $i \in [0, m-1]$.
**Query([[$x$]]):**
If $x \in X$ return $[[true]]$, else return $[[false]]$.
If a query is repeated, the behavior is undefined.

Figure 4.7: Functionality for Secret-Shared Oblivious Set

The core idea of the construction is that a single party, say $P_0$, can locally construct the Cuckoo Hash table and Bloom Filter objects. Since the indices must remain secret shared, the Cuckoo Hash table and Bloom Filter are constructed not from the indices $X_i$, but on PRP evaluations of the indices $Q_i = \mathrm{PRP}_k(X_i)$. This PRP is evaluated in a secure computation, and the output revealed to $P_0$, who constructs the Cuckoo Hash Table and Bloom Filter and secret-shares these between $P_1$ and $P_2$. The hash functions for the Cuckoo Hash Table and Bloom Filter can be public, since the data structures are secret-shared.

When an index $x$ is queried, the parties securely evaluate $q = \mathrm{PRP}_k(x)$ and reveal this to $P_1$ and $P_2$. The locations to be accessed in the secret-shared Cuckoo Hash table and the secret-shared Bloom Filter depend only on $q$ and public hash functions. $P_1$ and $P_2$ can therefore access the required locations of the secret-shared Cuckoo Hash Table and Bloom Filter. They can then reshare the values in these locations (without revealing the locations themselves to $P_0$). The result can be obtained by then performing, inside the secure computation, equality tests on the two Cuckoo table locations, and an AND of all of the Bloom Filter locations.

Our protocol makes use of various types of secret-sharing from Figure 2.1. Note in particular the difference between $[[x]]_{(1,2)}$ which indicates that $x$ is known by $P_1$ and $P_2$, (but not $P_0$) and $[[x]]_{1,2}$ which indicates that $x$ is *secret-shared* between $P_1$ and $P_2$. Operations on these secret-shares are performed as described in Section 2.2, based on the protocol of Araki et al. [AFL+16].

**Theorem 15.** Protocol $\Pi_{DOSet}$ (Figure 4.8) securely implements $\mathcal{F}_{SS-Set}$ (Figure 4.7) in the $(3, 1)$ semi-honest setting.

*Proof.* Observe that $\Pi_{DOSet}$.Build has a probability of failure, whereas $\mathcal{F}_{SS-Set}$.Build does not. However the failure probability is negligible (Theorem 12), so cannot be used to distinguish the real and ideal executions.

Similarly, there is a negligible probability of a false positive and zero probability of a false negative (Theorem 12). Therefore, the event of a false result does not allow the true and simulated executions to become distinguishable.

The simulators are quite simple. For the most part, the values in the protocol are secret-shared. For any fresh secret-sharing, a party's individual share is a random value, so the simulator can select a random value to show the message that the party received.

When $P_0$ is corrupted, the simulator must generate the values $Q_i$, which are received by $P_0$. The simulator chooses $m$ distinct values, chosen at random from $[0, n-1]$. These will be indistinguishable from the actual $Q_i$, due to the security of the PRP. During a query, $P_0$ does not receive any values,

**Protocol $\Pi_{DOSet}$**

**Build($[[X]]$, n)**

1. Set $[[k]] = \mathcal{F}_{SISO-PRP}.\text{Keygen}(n)$
2. $P_0$ generates and shares with $P_1$ and $P_2$:
   (a) Public Cuckoo Hash functions $H_1, H_2 : [n] \to [(1+\epsilon)m]$
   (b) Public Bloom Filter hash functions $g_1, \ldots, g_{\log n} : [n] \to [m \log n]$.
3. For $i \in [0, m-1]$
   (a) Securely evaluate the PRP on $X_i$ and reveal to $P_0$:
   $[[Q_i]] = \mathcal{F}_{SISO-PRP}.\text{Eval}([[k]], [[X_i]])$
   $[[Q_i]]_0 \leftarrow [[Q_i]]$
4. $P_0$ locally constructs a Cuckoo Hash table with a stash for the encodings $[[Q_i]]_0$ *i.e.,* $P_0$ stores $Q_i$ in $T_1[H_1(Q_i)]$ or $T_2[H_2(Q_i)]$ for as many encodings as possible, and stores the remaining random encodings in stash $S$. If $|S| > \log n$ the build fails and $P_0$ sends **abort** to all parties, who then abort. In all empty locations in the table, $P_0$ stores $\perp$. Let $[[Cuckoo]]_0$ be the appended tables of $T_1$ and $T_2$. $P_0$ secret-shares $Cuckoo$ between $P_1$ and $P_2$, i.e. for $i \in [0, 2(1+\epsilon)m - 1]$:
   $[[Cuckoo_i]]_{1,2} \leftarrow [[Cuckoo_i]]_0$
5. $P_1$ constructs a Bloom Filter array $Bloom$ of length $m \log n$, using inputs $S$ and hash functions $g_1, \ldots, g_{\log n}$. $P_0$ then secret-shares $Bloom$ to $P_1$ and $P_2$ as follows: for $i \in [m \log n]$: $[[Bloom_i]]_{1,2} \leftarrow [[Bloom_i]]_0$

**Query($[[x]]$)**

1. Securely evaluate the PRP on the query and reveal the output to $P_1$ and $P_2$:
   $[[q]]_{(1,2)} = \mathcal{F}_{SISO-PRP}.Eval([[k]], [[x]])$
2. Securely query $q$ in the Cuckoo Table. For each hash function, $P_1$ and $P_2$ locallaly evaluate that hash function on $q$ to obtain a location in the table. They then each access their share at that location, and together reshare this value to a RSS-sharing. Finally, they check whether the item stored in that location has the correct encoding.
   For $i \in \{1, 2\}$:
   (a) $[[cLocal_i]]_{(1,2)} = H_i([[q]]_{(1,2)}) + (i-1) * m$.
   (b) $[[c_i]]_{1,2} = [[Cuckoo_{[[cLocal_i]]_{(1,2)}}]]_{1,2}$
   (c) $[[c_i]] \leftarrow [[c_i]]_{1,2}$
   (d) $[[eq_i]] \leftarrow ([[c_i]] = [[q]])$
   $[[inCuckoo]] = [[eq_1]] \vee [[eq_2]]$.
3. Similarly, $P_1$ and $P_2$ locally evaluate the Boom Filter hash functions $g_1, \ldots, g_{\log(n)}$ on $q$ to obtain the indexes to access in the secret-shared Bloom filter. They then reshare these locations, and the protocol securely computes whether all locations are set to 1.
   For $i \in [1, \log n]$:
   (a) $[[bLocal_i]]_{(1,2)} = g_i([[q]]_{(1,2)})$
   (b) $[[b_i]]_{1,2} = [[Bloom_{[[bLocal_i]]_{(1,2)}}]]_{1,2}$
   (c) $[[b_i]] \leftarrow [[b_i]]_{1,2}$
   $[[inBloom]] = \wedge_{i=1}^{\log(n)} [[b_i]]$
4. Return $[[inCuckoo]] \vee [[inBloom]]$

Figure 4.8: 3 Party Secure Set Membership

so their view consists only of secret-shares, which the simulator can choose at random.

$P_1$ and $P_2$ are symmetric, so without loss of generality we only consider the simulator needed for $P_1$. $P_1$ receives no messages during a build, so the simulator need only send random messages for each new secret-shared value. During a query, $P_1$ receives the output $q$. The simulator can replace this with a random value from $[0, n-1]$ which has never been used before. Since the query indices are never repeated, the output in the real execution will also be a new value from $[0, n-1]$. In the simulated execution these are truly random, whereas in the real execution they are the result of a PRP. However, due to the security property of the PRP the two are computationally indistinguishable. □

The communication costs of $\Pi_{DOSet}$ are presented below.

**Theorem 16.** Protocol $\Pi_{DOSet.Build}$ (Figure 4.8) requires $\mathcal{O}(\kappa m)$ communication. In particular it requires $m$ calls to $\mathcal{F}_{SISO-PRP}$.Eval.

*Proof.* Generating the secret key requires no communication. Sharing the hash functions requires sending $\mathcal{O}(\log N) = o(m)$ hash functions, each which can be represented with $\kappa$ bits, so $\mathcal{O}(\kappa m)$ bits total. Step 3 requires $m$ calls to $\mathcal{F}_{SISO-PRP}$.Eval, each of which costs $\Theta(\kappa + \log n) = \Theta(\kappa)$ bits of communication, so the cost of this step is $\Theta(\kappa m)$ bits. Outputting these results to $P_0$ requires revealing $m$ shares, each of size $\mathcal{O}(\log(n))$ bits, so the total cost of this step is $\Theta(m \log(n))$.

Constructing the Cuckoo Hash table and Bloom Filter is achieved locally by $P_0$ so needs no communication. $P_0$ then secret-shares both of these data structures. The Cuckoo Table has $2(1+\epsilon)m$ locations, each of size $\Theta(\log(n))$, so secret-sharing it requires $\Theta(m \log(n))$ communication. The Bloom Filter is of length $m \log n$ bits, so secret-sharing it requires $\Theta(m \log(n))$ communication.

Summing these up, noting that $\log(n) = o(\kappa)$, the total communication cost of a Build is $\Theta(\kappa m)$. □

**Theorem 17.** Protocol $\Pi_{DOSet.Query}$ (Figure 4.8) requires $\mathcal{O}(\kappa)$ communication. In particular it requires 1 call to $\mathcal{F}_{SISO-PRP}$.Eval.

*Proof.* The protocol begins with one call to $\mathcal{F}_{SISO-PRP}$.Eval. This costs $\mathcal{O}(\kappa + \log n) = \mathcal{O}(\kappa)$ bits of communication. Revealing the output to $P_1$ and $P_2$ requires revealing 1 message of size $\log n = o(\kappa)$, bits which requires $o(\kappa)$ communication total.

$P_1$ and $P_2$ then locally select the appropriate shares from their secret-shared data-structures, which does not require communication. $P_1$ and $P_2$ then re-share these locations to a RSS sharing. There are 2 locations in the Cuckoo Hash table, each of size $\log(n)$ bits, so sharing these requires $\Theta(\log(n))$ communication. Performing secure equality tests on these values also requires $\Theta(\log(n))$ bits of communication.

There are $\log(n)$ bits in the Bloom Filter, so sharing these also requires $\Theta(\log(n))$ communication. Computing the AND of all of these bits securely also requires $\Theta(\log(n))$ communication. The ORs of single-bit values costs $\Theta(1)$ communication.

The communication cost is therefore dominated by the single SISO-PRP evaluation at the start of the protocol, which requires $\Theta(\kappa)$ communication. □

4.6.1. Oblivious Set Membership for small $m$

Our hierarchical ORAM protocol will need Oblivious Hash Tables, and Oblivious Sets, where $m$ is not $\omega(\log n)$. In this case, the data structure presented above will have non-negligible failure probability.

To solve this, when $m$ is small we use a modified set membership protocol $\Pi_{DOSetSmall}$, which uses larger Bloom filters and no Cuckoo Hash Tables. We have some security parameter $\kappa$, where $\kappa = \omega(\log n)$. If $m < \kappa$, the Cuckoo Hash Table is not used, and $P_0$ places all $m$ PRP evaluations in the Bloom Filter, and makes the Bloom filter of size $B = m\kappa$. As before the number of hash functions is $\log n$. This makes the probability of a false positive

$$\left(1 - e^{-\frac{\log n m}{m\kappa}}\right)^{\log n} = \left(1 - e^{-\frac{\log n}{\kappa}}\right)^{\log n}$$

which is negligible in $n$. The proof of security is identical to that of the $\Pi_{DOSet}$, since the only messages revealed are the PRP evaluations, so $\Pi_{DOSetSmall}$ securely implements $\mathcal{F}_{SS-Set}$ for $m <$

$\omega(\log n)$.

The communication complexity of a build remains $\mathcal{O}(\kappa n)$ with $n$ secure PRP evaluations and the communication complexity of a query remains $\mathcal{O}(\kappa)$ with 1 secure PRP evaluation.

Therefore, in terms of security and communication cost, other protocols can call $\Pi_{DOSetSmall}$ in place of $\Pi_{DOSet}$ when $m$ is small and the behavior will be the same. One small difference, however, is that $\Pi_{DOSetSmall}$ needs superlinear storage ($\Theta(\kappa m)$ rather than $\Theta(m \log n)$). Nevertheless, in the ORAM data structure, only the smaller levels will be instantiated with this data structure, so it will not increase the asymptotic memory usage.

## 4.7. Distributed Oblivious Hash Table

We will now present how a DOSet can be used to construct an efficient DOHTable. The key idea is that once it is known whether an item is in the Hash Table, the protocol can choose whether to search for the item itself or to search for a pre-inserted dummy element. This means that the protocol need not hide where in the data structure data is stored, nor need it hide the location that is accessed. All that needs to be hidden is whether an item is a dummy element or not, and if not, to avoid revealing any information about which element it is. As such, DOSets turn out to handle the hardest part of the problem.

The DOHTable is built containing $m$ pre-inserted dummy items. Each query of an index that is not in the table needs to access a distinct dummy, so after $m$ accesses, the table must be rebuilt. In our DORAM protocol, this will not occur since a rebuild is called after $m$ accesses anyway (as we will have only one table per level, that is $b = 2$). Nevertheless, for completeness and generality we here include an instantiation that could be called an arbitrary number of times (on distinct items). Note that there may be several "empty" values included in the input (i.e. with index $\perp$). To handle these, we assign them values in the range $[n, n + m - 1]$. Additionally, the pre-inserted dummy items must also be searched for also using a PRP, so we query these using indices in the range $[2n, 2n + m - 1]$. Since the input space to the PRP must include both of these ranges, we set the input space to be of size $4n$, (so that it will still be a power of 2), i.e. PRPs will execute on indexes

with $\log(4n) = \log(n) + 2$ bits.

**Theorem 18.** $\Pi_{DOHTable}$ (Figure 4.9) securely implements $\mathcal{F}_{SS-HTable}$ (Figure 2.2) in the $\mathcal{F}_{SS-Set}$-hybrid model in the $(3, 1)$ semi-honest security setting.

*Proof.* For any secret-shared value, the simulator can simply pick random values, which will be indistinguishable from the random shares in the party's view. Therefore, we only need to show that the simulator is able to simulate any values that are revealed to parties.

$P_0$ receives the random key, $k$. This is simply a random value from the key space (normally $\{0,1\}^\kappa$). $P_0$ also evaluates $Q_{m+i}$ locally, but this is based on information which $P_0$ already has. $P_0$ does not receive any additional information during a Build.

During a query, $P_0$ learns the index $j$ of the queried element in the shuffled array. The simulator can pick a random $j$ which has not been picked before. This will cause $P_0$ to select a share of the $j^{th}$ element, which may not be the same share that was generated by the functionality as $P_0$'s share of the output. To make them consistent, the simulator chooses the messages $P_0$ receives in the refreshing in the final step of the query, such that when $P_0$ calculates its new share, it will be the same as its share in the output.

During an extract $P_0$ receives no values, though the shares during the final refresh must be chosen so as to make the output consistent with its previous shares of $\hat{X}_i, \hat{Y}_i$.

$P_1$ and $P_2$ are symmetric, so we just prove security against a corrupted $P_1$.

During a build, $P_1$ receives $\hat{Q}_0, \ldots, \hat{Q}_{2m-1}$. To simulate this, the simulator generates $2m$ random distinct $\log(4N)$-bit messages. Since these are the result of PRP evaluations on distinct inputs, any entity that could distinguish these from random distinct messages would be able to distinguish the PRP from a random permutation. Hence, by the security of the PRP, the output of $S_2$ is indistinguishable from the view of $P_2$.

Either $[[x]] \in [[X]]$, in which case $x$ is queried, or $[[x]] \notin [[X]]$, in which case $2n + t$ is queried. In

**Protocol $\Pi_{DOHTable}$**

**Build($[[Z]]$, n, m, d)**

1. Interpret $[[Z_i]]$ as a pair $([[X_i]], [[Y_i]])$ for $i \in [0, m-1]$.
2. The input may contain empty elements, which have $[[X_i]] = [[\perp]]$. We first reset these to unique values, distinct from the range of the real items ($[0, n-1]$):
   For $i = 0, \ldots, m-1$: If $[[X_i]] = [[\perp]]$, set $[[X_i]] = [[n+i]]$ and $[[Y_i]] = [[\perp]]$.
3. Build the DOSet: $\mathcal{F}_{SS-Set}$.Build($[[X]], n$).
4. Pick, $k$, a key for the PRP, and reveal it to $P_0$: $[[k]]_0 \leftarrow \mathcal{F}_{SISO-PRP}.Keygen(\kappa, 4n)$
5. For $i = 0, \ldots, m-1$: Set $[[Q_i]] = \mathcal{F}_{SISO-PRP}$. Eval($[[k]], [[X_i]]$)
6. $P_0$ creates and uploads dummies indexed from $2n+1$ to $2n+m$, to be queried when an item is not in the Oblivious Hash Table. For $i = 0, \ldots, m-1$:
   (a) $P_0$ locally evaluates the PRP for the dummy items and then secret-shares the result:
   $[[Q_{m+i}]]_0 = PRP_{[[k]]_0}([[2n+i]]_0)$
   $[[Q_{m+i}]] \leftarrow [[Q_{m+i}]]_0$
   (b) Set $[[X_{n+i}]] = [[2n+i]]$ and $[[Y_{n+i}]] = [[\perp]]$
7. Shuffle the tuples. Set $[[\hat{Q}]], [[\hat{X}]], [[\hat{Y}]] = \mathcal{F}_{ABB}$.Shuffle($[[Q]], [[X]], [[Y]]$)
8. Reveal $\hat{Q}_0, \ldots, \hat{Q}_{2m-1}$ to $P_1$ and $P_2$. This will allow $P_1$ and $P_2$ to find an item's index in the shuffled array, based on its PRP evaluation.
9. Initialize $t = 0$, $visited = \emptyset$.

**Query($[[x]]$)**

1. Rebuild if the maximum number of queries has been reached. If $t = m$:
   $[[Z]]$ = Extract()
   Build($[[Z]], n, m, d$)
2. If an index is stored in the table, query the index. Otherwise query a pre-inserted dummy:
   $[[in]] = \mathcal{F}_{SS-Set}$.Query($[[x]]$)
   If $[[in]]$, set $[[x_{used}]] = [[x]]$, otherwise set $[[x_{used}]] = [[2n+t]]$
   $[[q]] = \mathcal{F}_{SISO-PRP}$.Eval($[[k]], [[x_{used}]]$)
3. Reveal $[[q]]$ to $P_1$ and $P_2$: $[[q]]_{(1,2)} \leftarrow [[q]]$
4. $P_1$ and $P_2$ find $j$ such that $q = \hat{Q}_j$ and reveal $j$ to $P_0$.
5. Set $t = t+1$, $visited = visited \cup \{j\}$
6. Return a refreshed $[[\hat{Y}_j]]$

**Extract()**

1. Reveal an additional $m - t$ dummy items, so that only $m$ items have not been accessed.
   For $j = 1, \ldots, 2m, j \notin visited$: $[[toDelete_j]] = ([[X_j]] \geq [[n+t]]) \wedge ([[X_j]] < [[n+m]])$
   $toDelete_j \leftarrow [[toDelete_j]]$
   if toDelete, $visited = visited \cup \{j\}$
2. Built the result from the $m$ unvisited items, setting to $([[\perp]], [[\perp]])$ any dummy/empty items. For $j = 0, \ldots, m-1$:
   (a) Let $i_j$ be the $j^{th}$ index such that $i_j \notin visited$
   (b) If $([[\hat{Y}_{i_j}]] = [[\perp]])$, set $[[\hat{X}_{i_j}]] = [[\perp]]$
   (c) Set $[[Z_j]]$ to a refreshed sharing of $([[\hat{X}_{i_j}]], [[\hat{Y}_{i_j}]])$
3. Return $[[Z]]$

Figure 4.9: Distributed Oblivious Hash Table Protocol

either case, $q$ will be some value in $\hat{Q}$. The simulator therefore picks a random value from $q$ that it had not picked before. The simulator picks the messages during the refresh so that $P_1$'s previous share of $\hat{Y}_j$ is updated to the share chosen by the functionality.

$P_1$'s role in the Extract is symmetric to $P_0$'s, so the same argument applies. □

The communication costs of $\Pi_{DOHTable}$ are stated below, where $\mathcal{F}_{SS-Set}$ is implemented using $\Pi_{DOSet}$ (Figure 4.8). We first present the costs when $\Pi_{DOHTable}$.Query is called at most $m$ times, which is how $\mathcal{F}_{SS-HTable}$ is used by $\Pi_{DOMap}$. For completeness, we then show the amortized per query cost in the more general case that $\Pi_{DOHTable}$.Query is called more than $m$ times.

**Theorem 19.** $\Pi_{DOHTable}.Build$ (Figure 4.9) requires $\mathcal{O}(\kappa m + dm)$ communication. In particular, it requires $2m$ calls to $\mathcal{F}_{SISO-PRP}$.Eval.

*Proof.* $\mathcal{F}_{SS-Set}$.Build can be implemented using $\mathcal{O}(\kappa m)$ communication and $m$ calls to $\mathcal{F}_{SISO-PRP}$.Eval (Theorem 16). Inputting the PRP key requires $\mathcal{O}(\kappa)$ communication. Resetting the values of duplicates (step 2) requires inputting $m$ values of size $\log(4n)$ and $m$ of size $d$, for a total cost of $\mathcal{O}(md)$. There are $m$ more calls to $\mathcal{F}_{SISO-PRP}$.Eval, needing a total of $\mathcal{O}(\kappa m)$ communication. Inputting the dummies in step 6 requires $m$ iterations, each needing communication $\mathcal{O}(\log n + d) = \mathcal{O}(d)$, or $\mathcal{O}(md)$ total. The shuffle shuffles arrays of length $2m$, where each is of size $\mathcal{O}(d)$, leading to $\mathcal{O}(md)$) communication. Revealing $\hat{Q}$ to $P_1$ and $P_2$ requires $\mathcal{O}(2m \log n) = \mathcal{O}(md)$ communication. Therefore the entire protocol requires $\mathcal{O}((\kappa + d)m)$ communication. □

**Theorem 20.** If $\Pi_{DOHTable}.Query$ (Figure 4.9) is called at most $m$ times, then it requires $\mathcal{O}(\kappa+d)$ communication and, in particular, 2 calls to $\mathcal{F}_{SISO-PRP}$.Eval.

*Proof.* If Query is called at most $m$ times, a rebuild will never be triggered.

Calling $\mathcal{F}_{SS-Set}$.Query requires $\mathcal{O}(\kappa)$ communication and 1 call to $\mathcal{F}_{SISO-PRP}$.Eval. $\mathcal{O}(\log N) = \mathcal{O}(\kappa)$ communication is required to input the dummy index and $\mathcal{O}(\log n) = \mathcal{O}(\kappa)$ is needed to set $[[x_{used}]]$ to the appropriate value. Securely evaluating the PRP on $[[x_{used}]]$ requires an additional

call to $\mathcal{F}_{SISO-PRP}$.Eval and $\mathcal{O}(\kappa)$ communication. Revealing the result requires $\mathcal{O}(\log n) = \mathcal{O}(\kappa)$ communication. Finally, refreshing $[[\hat{Y}_j]]$ requires $\Theta(\log(n) + d) = \Theta(d)$ bits of communication.

The total cost is therefore $\Theta(\kappa + d)$. □

**Theorem 21.** $\Pi_{DOHTable}$.Extract (Figure 4.9) requires $\mathcal{O}(m(\kappa + d))$ communication and no calls to $\mathcal{F}_{SISO-PRP}$.Eval.

*Proof.* Revealing the additional $m - t$ dummy items involves at most $m$ iterations, each of which is dominated by the cost of 2 comparisons of $\Theta(\log(n))$-bit values, resulting in total cost $\Theta(m \log(n))$.

The next loop requires iterating over $m$ items. The cost of each iteration is dominated by the cost of refreshing the sharing of $([[\hat{X}_{i_j}]], [[\hat{Y}_{i_j}]])$, which costs $\Theta(\log(n) + d) = \Theta(d)$.

The total cost is therefore $\Theta(md)$. There are no calls to $\mathcal{F}_{SISO-PRP}$.Eval. □

**Theorem 22.** If $\Pi_{DOHTable}.Query$ (Figure 4.9) is called more than $m$ times, then the amortized per query communication cost is $\Theta(\kappa + d)$ and, in particular, 4 calls to $\mathcal{F}_{SISO-PRP}$.Eval.

*Proof.* The cost of queries, ignoring rebuilds (periodic extracts/builds) is $\Theta(\kappa + d)$, with 2 calls to $\mathcal{F}_{SISO-PRP}$.Eval. To find the total amortized cost we simply add this to the amortized cost of rebuilds.

Rebuilds occur every $m$ accesses. They consist of an Extract, which costs $\Theta(md)$ and has no calls to $\mathcal{F}_{SISO-PRP}$.Eval, and a Build, which costs $\Theta((\kappa + d)m)$ and has $2m$ calls to $\mathcal{F}_{SISO-PRP}$.Eval. Therefore, the amortized cost of rebuilding the DOHTable per query is $\Theta(\kappa + d)$, with 2 calls to $\mathcal{F}_{SISO-PRP}$.Eval.

The total amortized cost per query is therefore $\Theta(\kappa + d)$ with 4 calls to $\mathcal{F}_{SISO-PRP}$.Eval. □

## 4.8. Hierarchical ORAM

Our DOHTable can then be used directly to instantiate a DOMap/DORAM. Formally:

**Theorem 23.** There exists (3,1) semi-honest secure DOMap and DORAM protocols with $\Theta(\log(n)(\kappa+d))$ communitaction per query, which require only $2\log(n)$ calls to $\mathcal{F}_{SISO-PRP}$.Eval per query.

*Proof.* We use $\Pi_{DOMap}$ (Figure 2.6), setting the parameters $b = 2$, and $c = \log(n)$ (and therefore $l = \log_b(n/c) = (\log(n) - \log(\log(n)) < \log(n))$.

We instantiate $\mathcal{F}_{SS-HTable}$ with $\Pi_{DOHTable}$ (Figure 4.9). By Theorem 18, this is a secure MPC implementation, in the (3,1) setting. Security of the resulting DOMap follows automatically from the security of the Hierarchical DORAM template (Figure 2.6) and composability. By setting $m = n$, we obtain the special case where the DOMap is also a DORAM.

Note that, when we set $b = 2$, there will only be 1 table at each level. Furthermore, this table, if it is of size $m$, will only be queried $m$ times before its contents are extracted. Therefore, using Theorem 4 and setting $Cost_B$, $Cost_Q$ and $Cost_E$ to, respectively, the costs from Theorems 19, 20 and 21, we can determine the cost per query.

The total amortized asymptotic cost will be:

$$\begin{aligned}
TotalCost &= \Theta(c(\log(n)+d) + \log_b(n)\cdot b\cdot Cost_Q(n,d) \\
&\quad + \sum_{i=0}^{\log_b(n/c)} \frac{1}{c\cdot b^i}(Cost_B(c\cdot b^i, n, d) + Cost_E(c\cdot b^i, n, d))) \\
&= \Theta(\log(n)d + \log(n)(\kappa+d) + \sum_{i=0}^{\log(n)-\log(\log(n))} \frac{1}{\log(n)2^i}(\log(n)2^i(\kappa+d) + \log(n)2^i d)) \\
&= \Theta(\log(n)(\kappa+d) + \sum_{i=0}^{\log(n)-\log(\log(n))} (\kappa+d)) \\
&= \Theta(\log(n)(\kappa+d))
\end{aligned}$$

To calculate the concrete number of calls to $\mathcal{F}_{SISO-PRP}$, we will need to examine the hierarchy in slightly more detail. Since $b = 2$, there will only be at most one table per level. Moreover, each table will only exist half of the time (except the largest one). Therefore, on average, each table will result

in 1 $\mathcal{F}_{SISO-PRP}$ evaluation per query. Likewise, building a table of size $m$ requires $2m$ $\mathcal{F}_{SISO-PRP}$ evaluations, but this table is only rebuilt every $2m$ accesses (except for the table of size $n$ which is rebuilt every $n$ accesses). Therefore the cost of rebuilds will be 1 $\mathcal{F}_{SISO-PRP}$ evaluation per query per level. Therefore, the total cost for all levels except the largest is $2(\log(n) - \log(\log(n)))$ $\mathcal{F}_{SISO-PRP}$ evaluations per query. The total cost at the largest level is $4\mathcal{F}_{SISO-PRP}$ evaluations per query. Therefore, (for any $n \geq 16$), the total cost is at most $2\log(n)$ $\mathcal{F}_{SISO-PRP}$ evaluations per query. □

## 4.9. Comparison with Lu and Ostrovsky (TCC 2013) [LO13a]

As discussed in Section 1.4 the 2-Server ORAM protocol of [LO13a] can be used to implement a DORAM protocol, by simulating the client inside of an MPC protocol. The main obstacle to efficiency is that the client performs symmetric-key encryptions and decryptions, which are expensive when executed inside an MPC.

Asymptotically, using MPC to simulate the [LO13a] protocol is actually efficient. It is possible for block cipher operations on plaintexts of size $d$ to be computed using $\mathcal{O}(\kappa + d)$ AND gates. Using this and generic MPC techniques to simulate the client of [LO13a] leads to a DORAM protocol with amortized $\mathcal{O}((\kappa + d)\log n)$ communication per access. However, this protocol would be very far from practical. Specifically we show below that the number of SISO-PRPs we need is fewer than [LO13a] by a factor of about 50. These results are tabulated in Figure 4.10, and are explained verbally below.

Our protocol is more concretely efficient for several reasons. Firstly, we use secret-sharing instead of encryption, removing the need to perform encryptions and decryptions when items are moved through the hierarchy. Secondly, we use the recent result [Nob21] that cuckoo hashing can in fact be used for any level that has size $\omega(\log(n))$, whereas [LO13a] only use cuckoo hashing on levels of size $\Omega(\log^7(n))$ (based on the analysis in [GM11] Appendix C), and instantate hash tables for smaller levels using bucket hashing. Thirdly, since [LO13a] uses the cache-the-stash technique, the frequency of rebuilds is increased.

We examine Figures 3 and 4 of [LO13a] and calculate the number of SISO-PRP calls. Encryptions, decryptions and PRF evaluations will all be counted as a single PRP call. Note that encryptions and decryptions in [LO13a] are actually applied to both the index and the data payload, so will require PRPs on larger block sizes than those used by our protocol and will therefore require *more communication* than the SISO-PRPs in our protocol.

Queries in [LO13a] begin by querying the top level. Unlike our protocol, [LO13a] caches the stash, which means that the top level will always have at least $\log n$ elements. When it is full it will have $2 \log n$ elements, so on average it has $1.5 \log n$ elements. Each of these is decrypted in step 2, leading to amortized $1.5 \log n$ SISO-PRP evaluations.

[LO13a] then accesses the smaller levels, which use standard hash tables, with buckets of capacity $\frac{3 \log n}{\log \log n}$. There are approximately $7 \log \log n$ such levels, and on average half of these will have tables at a given point in time. This means that the average number of items that will need to be accessed in the small levels is $10.5 \log n$. Each item will need to be both decrypted (step 3b) and re-encrypted (step 3d) leading to $21 \log n$ SISO-PRPs per access. Additionally, a PRF will need to be executed at each level (step 3a), leading to a further $7 \log \log n$ SISO-PRPs. To simplify the analysis, we will ignore $\log \log n$ terms since these will be small relative to $\log n$ for large $n$.

The remainder of levels accessed are Cuckoo tables. These will only require accessing two locations per level. Ignoring $\log \log n$ terms, there will be $\log n$ such levels, and therefore, on average, $\frac{\log n}{2}$ such tables. Each item, again, will need to be decrypted (step 4b) and again re-encrypted (step 4d), each leading to amortized costs of $\log n$ SISO-PRPs per access. Additionally there will be a PRF call for each Oblivious Hash Table (step 4a) each costing 1 SISO-PRP. Since there are on average $\frac{\log n}{2}$ levels this costs amortized $0.5 \log n$ SISO-PRPs.

Lastly, the top level is accessed again. This time items are both decrypted and re-encrypted. Since on average there are $1.5 \log n$ items in this level, this needs, on average, $3 \log n$ SISO-PRPs.

In the build, the role that a server plays during the building of a table depends on which server will hold the built table, since the servers hold alternating levels of the hierarchy. To build a level that

will be held by $S_b$, the items of $S_a$ are first sent over to $S_b$. To do this these items are decrypted and re-encrypted by the client. Next these are combined with items held by $S_b$, permuted by $S_b$ and then sent back to $S_a$, each first being decrypted and re-encrypted by the client. The client also sends $S_a$ the PRF evaluations that $S_a$ will need to build the table. $S_a$ builds the table and the table is sent to $S_b$, but once again every item is decrypted and re-encrypted.

Additionally, since [LO13a] use the cache-the-stash technique, rebuilds will occur twice as frequently as in our construction. Specifically, in [LO13a] a table holding $n$ elements (including dummies), will be rebuilt every $m$ accesses, whereas opposed to our protocol, a table holding $n$ elements will be rebuild every $2m$ accesses.

First the elements held by $S_a$ need to be re-encrypted and send to $S_b$ (Rebuild step 2). It will be easier to count this cost based on the table that is being extracted, rather than the one being built. We assume that when a table is extracted to be placed into a lower level, that half of the time it will be placed in a level that will be held by the other party. (In reality it will be more than half, since half of the time it is placed in the level below it, which is held by the other party.) In this case, the costs described in step 2 of the reshuffle will be incurred.

For the small levels, there will be $3m\frac{\log n}{\log\log n}$ elements per level. Each will need to be encrypted and decrypted (2 SISO-PRPs). This will occur every $2m$ accesses. (Since the rebuilds occur every $m$ accesses, and for at least half of these, step 2 will need to occur.) There are $7\log\log n$ such levels. Therefore the cost of step 2 on small levels will be $3m\frac{\log n}{\log\log n}2\frac{1}{2m}7\log\log n = 21\log N$ SISO-PRPs.

Cuckoo levels are similar, except that there will be only $2(1+\epsilon)m$ elements per level and will be $\log n$ levels (ignoring $\log\log n$ terms). Therefore the cost for cuckoo levels will be $2(1+\epsilon)m2\frac{1}{2m}\log n = 2(1+\epsilon)\log n$ SISO-PRPs per level.

The reshuffle in step 3 happens locally so incurs no communication cost.

In step 4, each element in the table is again decrypted. Again we count these based on the table from which the elements came. This will happen to all elements (both $S_a$'s and $S_b$'s), so will always hap-

pen when a table is extracted. Since rebuilds occur every $m$ accesses, the amortized number of SISO-PRPs per access for decrypting items from small tables will be $3m\frac{\log n}{\log\log n}\frac{1}{m}7\log\log n = 21\log n$ SISO-PRPs. The items are then re-encrypted, but empty items need not be re-encrypted. The small tables only have $m$ items that need be re-encrypted, but there are only $7\log\log n$ small levels, so the asymptotic cost is only $m\frac{1}{m}7\log\log n = 7\log\log n$ SISO-PRPs, which will be ignored in this analysis. For larger Cuckoo tables the situation is similar, except there are $2(1+\epsilon)n$ items per level and about $\log n$ levels. This leads to $2(1+\epsilon)\log n$ SISO-PRPs for decryption. Additionally, the non-empty items will need to be encrypted. There will be $m$ of these, leading to an additional amortized $\log n$ SISO-PRPs per access for the cuckoo levels. Step 4 also simulates $m$ PRF evaluations. For each level this occurs every $m$ access, adding an amortized cost of $\log n$ SISO-PRPs.

Step 5 is again performed locally by $S_A$ so incurs no communication cost.

When a table of capacity $m$ is being built in step 6, the $m$ non-empty items given from $S_a$ need to be decrypted. Decrypting the non-empty items costs amortized $\log n$ SISO-PRPs per access (summed over all levels). Following this, every item in the resulting table (both empty and not) must be encrypted and sent to $S_b$. The total amortized cost for this over all small tables is $21\log n$ SISO-PRPs. The total amortized cost for this over all Cuckoo tables is $2(1+\epsilon)\log n$ SISO-PRPs.

Figure 4.10 shows the total costs. The total cost is at least an amortized $100\log n$ PRPs per access. Our protocol required amortized only $2\log n$ SISO-PRPs per access. Therefore [LO13a] requires about 50 times more SISO-PRPs than our protocol.

| **Step** | **SISO-PRPs per access (amortized)** |
|---|---|
| Query: 2 | $1.5 \log n$ |
| Query: 3b | $10.5 \log n$ |
| Query: 3d | $10.5 \log n$ |
| Query: 4a | $\log n$ |
| Query: 4b | $\log n$ |
| Query: 4d | $0.5 \log n$ |
| Query: 6 | $3 \log n$ |
| **Query Total:** | $28 \log n$ |
| Reshuffle: 2 (buckets) | $21 \log n$ |
| Reshuffle: 2 (cuckoo) | $2(1+\epsilon) \log n$ |
| Reshuffle: 4 (encrypt non-empty) | $\log n$ |
| Reshuffle: 4 (decrypt buckets) | $21 \log n$ |
| Reshuffle: 4 (decrypt cuckoo) | $2(1+\epsilon) \log n$ |
| Reshuffle: 4 (PRFs) | $\log n$ |
| Reshuffle: 6 (decrypt non-empty) | $\log n$ |
| Reshuffle: 6 (encrypt buckets) | $21 \log n$ |
| Reshuffle: 6 (encrypt cuckoo) | $2(1+\epsilon) \log n$ |
| **Reshuffle Total:** | $(72+6\epsilon) \log n$ |
| **Total:** | $(100+6\epsilon) \log n$ |

Figure 4.10: Communication Cost of [LO13a]: Number of SISO-PRPs

# CHAPTER 5

## MetaDORAM: Info-theoretic Distributed ORAM with Less Communication

*This chapter is based on material from the following work, which is currently in submission:*

Brett Hemenway Falk, Daniel Noble and Rafail Ostrovsky. MetaDORAM: Info-Theoretic Distributed ORAM with Less Communication.

*I contributed to all aspects of the work.*

### 5.1. Introduction

The DORAM presented in the previous chapter highlighted the value of learning (a secret sharing of) the *location* in which an item is stored. In the previous DORAM, the location information consisted only of which DOHTable actually held the item, and this information was obtained iteratively by querying the Distributed Oblivious Set protocols.

In this chapter we show that this idea can be significantly extended. We use novel data structures, which hold metadata of each item, to first obtain the *exact* location in which an item is stored. This allows the item itself to be retrieved using a secret-shared variant of Private Information Retrieval.

The resulting protocol, MetaDORAM is a novel statistically-secure 3-party DORAM protocol with sub-logarithmic communication overhead. MetaDORAM achieves amortized communication cost $\Theta((\log^2(n) + d)\frac{\log(n)}{\log(\log(n))})$ bits per query. MetaDORAM is the first information-theoretic secure DORAM to achieve sub-logarithmic communication overhead when $d = O(\log^2(n))$ (see Table 1.2 in Section 1.4).

Due to the conversion between DORAMs and ORAMs, MetaDORAM can be converted to a 3-server statistically-secure active ORAM, Meta3ORAM, with communication cost $\Theta((\log^2(n)+d)\frac{\log(n)}{\log(\log(n))})$.

Note that a passive (non-active) ORAM has a lower bound of $\Theta(\log(n))$ overhead even if there are multiple servers [LN18, LSY20]. Meta3ORAM, like [AFN$^+$17], achieves sub-logarithmic overhead

only because the servers perform computation. When the servers are active, the $\Theta(\log(n))$ lower bound can be circumvented for communication, and instead applies to the amount of memory accessed by the servers. Our result, taken with [AFN+17], shows that the asymptotic bounds on the DORAM problem are, as of yet, not well understood, and opens up many interesting questions regarding what lower bounds exist for DORAMs, as well as for active information-theoretic ORAMs in general (see Section 5.7).

MetaDORAM also has lower communication overhead than most DORAM protocols that use computational assumptions. When $\kappa = \omega(\log^2(n)/\log(\log(n)))$ the communication cost is lower than the DORAM of Lu and Ostrovsky [LO13a]. It is also strictly more communication-efficient than the DORAMs of [FJKW15], [JW18], [BKKO20] and [VHG23], which depend on computational assumptions.

**Organization:** Section 5.2 provides a technical overview of our protocol. The formal DORAM functionality is presented in Section 5.3, as well as the functionalities for Secret-Shared Private Information Retrieval (SSPIR) and secure routing, which are used by our DORAM protocol. Section 5.4 presents our full DORAM protocol, and analyzes its security and communication costs. Sections 5.5 and 5.6 explain how SSPIR and secure routing, respectively, can be implemented using standard techniques. Section 5.7 concludes by discussing some interesting open questions.

## 5.2. Technical Overview

Recall from Section 1.4 that there are two primary paradigms for constructing (Distributed) Oblivious RAMs: the Hierarchical paradigm and the Tree paradigm. The MetaDORAM protocol will make use of ideas from both of these paradigms.

Observe that there are three problems that an ORAM must solve.

1. The ORAM needs to define a set of *query locations* for every item at any given point in time. The query locations are the locations which would be accessed if the item were to be queried at that time. This must be defined in such a way that, when it is revealed for a queried index, it appears random given the locations accessed for each previous query.

2. The Query protocol must use the index to obtain the set of query locations to access, and must do so obliviously.

3. The Rebuild protocol needs to move items in such a way that each item is always located within its set of query locations. Typically this is obtained by Queries writing the item to a small capacity array (the Cache, or root bucket) and the Rebuild protocol moving items "away" from this "congested" region.

Hierarchical and Tree ORAMs solve these problems in different ways.

In a Hierarchical ORAM, the set of query locations is the PRF evaluations of the index in all tables up until the table in which it is located, and PRF evaluations of a nonce in all subsequent tables. This will be random given all previous queries. In the tables up to and including the table the item is in, this is because the tables have been rebuilt since the last time the item was queried in them, so the PRF evaluation of the item's index in these tables will be pseudorandom. For all tables after this, the PRF is evaluated on a nonce, so will also have a pseudorandom output. Note that this means that the set of query locations for an item changes after each query (for lower tables which will use nonces) and after each rebuild (for every table which is rebuilt). The Query protocol is able to obtain these locations obliviously by querying each OHTable in sequence. The Rebuild protocol combines multiple tables by storing all of the items within them using a new PRF key in a new table.

In a Tree ORAM, the set of query locations for each item is a path from the root to a random leaf. Unlike Hierarchical ORAMs, the set of query locations for an item will only change if the item is queried, in which case a new random path is selected. To obtain the path obliviously, a position map is needed, which is implemented using a smaller ORAM. The Rebuild protocol (often called Evict), picks paths and moves items within the path towards their destination leaves.

Our approach combines ideas from Hierarchical and Tree ORAMs. At a very high level, it combines the Rebuild structure of a Hierarchical ORAM, in which tables are periodically combined using shuffling, with the Query approach taken by Tree ORAMs, in which the set of query locations for

an index is updated each time it is queried. Furthermore, it uses various data structures to allow retrieval of important *metadata* of each item, most importantly the item's current exact location in the data structure. This, combined with information-theoretic Private Information Retrieval, allows items to be accessed during queries with very low communication cost. We describe our protocol in stages, based on its core ideas.

**Hierarchy with Position Map**

The first idea is to use the hierarchical table data structure from the Hierarchical paradigm, but use a position map instead of PRFs to define the locations of items within each table. The position map will assign multiple positions (e.g. 2) to each item (unlike a Tree ORAM, which assigns only one position to each index.) The locations in which the item can be located within each table will be determined by these positions, for instance by evaluating the position modulo the table size. This removes the need for PRF evaluations, which in the context of DORAMs are particularly expensive as they must be evaluated inside of a secure computation. This also means that, unlike a Hierarchical (D)ORAM which needs to access tables sequentially, the positions can be revealed at once when an item is queried and all locations in all tables accessed concurrently. Lastly, removing the PRFs results in the (D)ORAM having information theoretic security.

Note that, while the structure is very similar to a Hierarchical ORAM, this solution breaks the abstraction of the OHTable component. Rather than each table choosing its own random assignment of items to locations, the assignment is pre-determined based on the position map. Furthermore, rather than instructing each table to perform a query, the protocol directly access locations in the table, retrieves the items in those locations and determines itself whether one of those items was the one queried. Since these tables break the abstraction of a Hash Table from Definition 2.2, we instead to refer to them simply as tables. An additional consequence of this is that the table itself does not "know" when an item has been accessed, and so cannot delete this item, or ensure that when the extracted contents of this table exclude queried items. Therefore, after an item is queried, an "obsolete" version of the item continues to exist in the data structure. We will later show how obsolete versions of items can efficiently be deleted.

**Hierarchy with Rune Map**

In the above approach, the position map must hold multiple (at least 2) positions for each index. Each position requires roughly $\log(n)$ bits to be represented, as each position should result in a random location in the largest table. Instead, each index could be mapped to a unique random label, which is updated each time the index is queried. We call this label a *rune* (Random Unique NoncE). The positions can then be the output of *public hash functions* applied to the rune. This means the smaller ORAM does not need to store a full position map, but only needs to store the rune map. We choose the runes from $[1, \Theta(n)]$, so the size of each rune will be $\log(n) + \Theta(1)$ bits. This yields modest concrete improvements when the number of positions for each item is a small constant, such as 2. More importantly, it also allows the number of positions for each item to be made large (we will later use $\log^{1.5}(n)/\log(\log(n))$), with no increase to the cost of the recursive ORAM construction. Since, after each access, each item must be assigned a new rune which has not been used before, the protocol will run out of runes after $\Theta(n)$ queries. This solution therefore requires an operation every $\Theta(n)$ accesses, which refreshes the rune space by reassigning all items to new runes.

The refresh can also be used as an opportunity to delete obsolete items. Before assigning fresh runes to items, the protocol shuffles all items (obsolete and active) and reveals their runes. The obsolete items will correspond exactly to items whose runes have been revealed. These items can therefore be deleted, leaving only the active items, which can then be assigned new runes to refresh the rune space.

The previous ideas work in general for ORAMs and DORAMs, although, in the ORAM setting, the benefits mentioned are unlikely to outweigh the costs of implementing a position map and a hierarchy of tables. We now show how this approach can be combined with other techniques in the setting of DORAMs where there is at most one corrupt party, to obtain an efficient DORAM protocol.

**Role Differentiation**

The next step is to use role differentiation to allow tables to be constructed efficiently. We assign

one party, $P_0$, to be the *Builder*, who helps in the construction of tables. The other parties are referred to as *Holders*, who store the tables, for instance by secret-sharing the tables between them. When an item is queried, the Builder learns its *new rune*. Since an item is always written to a public location in the cache, the Builder also learns the item's location after the access. The Builder therefore knows the mapping from runes to locations, for that particular item, even though the Builder doesn't know which item (index) the rune refers to. When it comes time to build the cache into a table, the Builder will know the runes of all items in the table, and also knows the public hash functions which determine their permissible locations in the new table. Therefore, the Builder can *locally* construct an allocation of runes to locations in the new table. The parties can then engage in a secure routing protocol, in which the Builder provides the mapping from original locations in the cache to locations in the new table, the Holders provide the secret-shared cache and the Holders obtain a secret-sharing of the new table. This same technique can be applied to all table rebuilds. We allow the Builder to maintain a mapping from runes to current locations for all runes in the data structure (cache + tables). The Builder is therefore able to always compute how items must be routed to build new tables, and the movement of the data itself can be achieved efficiently and securely using a secure routing protocol. When an item is queried, the current rune is revealed to the Holders *but not to the Builder*, which allows the Holders to access all query locations in which the item may be located, and the correct item is then selected securely by a MPC protocol. When an item is queried, the Holders learn its current rune, and the Builder learns its new rune, but no party learns both, so to any single party the revealed runes are simply a sequence of distinct random values.

Note that when an item becomes obsolete, it still has a rune associated with it. These obsolete items will continue to percolate through the data structure, as the Builder should not learn which runes have been accessed. For the purposes of rebuilds, therefore, obsolete items will be treated the same as active items. Only during the refresh phase will the obsolete items be removed. Note that even then the Builder should not learn the set of runes which have been queried. Instead, the Holders can shuffle the items, reveal the rune of each item and delete the runes which have been queried.

**Clairvoyant Builder**

Note that the rune map stores a mapping from indexes to runes and the Builder knows the mapping from runes to their current locations. We would therefore like to use this to obliviously obtain a secret-sharing of the current location of the index being queried. However, this is not straightforward. The Holders learn the rune being queried, but they cannot reveal it to the Builder, as the Builder knows when the item with this rune was last accessed. The Builder knows the mapping from runes to locations, but cannot reveal this to the Holders, as they should only obtain a secret sharing of the item's current location.

To solve this, consider how the Builder calculates the mapping from runes to locations. The location of the runes at a given point in time is a function of runes revealed to the Builder, the public hash functions, the time and perhaps some random decisions made by the Builder. If the Builder were able to instead *pick* the runes, setting the new rune of the queried item, then the Builder can predetermine the sequence it will use to assign runes. The Builder can then pre-compute the locations of all runes at every point in time (until the next refresh). Note that, if there are $l$ levels, a rune will be stored in at most $l + 1$ distinct locations (one for each level, and one in the cache). The Builder is therefore able to construct a *position schedule* which, for each rune, stores the $l + 1$ different locations in which the rune will be stored and the time range for each one. It can construct the position schedule *at the beginning of the protocol* before any queries occur. It can then secret-share the position schedule between the Holders (with the runes public, but in a permuted order, and the schedule secret-shared). During a query, the Holders can therefore access the secret-shared position schedule of the queried rune and, inside of a MPC protocol, securely select the location that matches the current time range.

Note that the Holders already know a set of query locations in which the item may be located based on the rune and the public hash functions. If there are $b - 1$ tables per level, $l$ levels, $h$ hash functions and $c$ cache locations, there are only (at most) $c + (b - 1)lh$ query locations in which the rune may be located. The position schedule therefore need not store the rune's location in the entire data structure, but can simply record which of these $c + (b - 1)lh$ locations is used.

**Secret-Shared PIR**

The problem now closely resembles a multi-server Private Information Retrieval (PIR) problem, but not quite. In PIR, the servers hold identical copies of some array, $A$, and a client holds some index $x$ it wishes to query privately. In our case, however, the Holders have a *secret-shared array* of the $c+(b-1)lh$ items which could be the queried item, and they also have a *secret-sharing* of an index to query. However, if the right secret-sharing scheme is used, PIR can still be used to achieve this very efficiently. If there are 3 Holders, and data is held using a RSS scheme, then each share is held by 2 parties. For each share, a standard 2-server information-theoretic PIR protocol can therefore be employed, with the third Holder acting as the client. (The protocol must be modified so that the input index, and the output, are both secret-shared. An example is given in Figure 5.5.) Using standard information-theoretic PIR, a $d$-bit value can be selected from an array of length $w$ using $w+d$ communication. In contrast, a regular Hierarchical DORAM would need to communicate $w \cdot d$ bits in order to securely select the desired item. This *dramatically* reduces the communication cost of Queries. By balancing the cost of queries and rebuilds ([KLO12], Section 1.4.2), a DORAM with low amortized communication cost can be achieved.

**Reducing to 3 Parties**

The PIR solution described above requires 3 Holders, and therefore, including the Builder, requires 4 parties in total. The protocol can be modified to only require 2 Holders, or 3 parties total.

Rather than holding secret-shared items, the 2 Holders will instead hold identical masked items. That is, each item will be masked by a One-Time Pad (OTP), and both Holders will hold the same masked item. The masks are generated by the Builder. Each time a new table is built, the items are re-masked; the Builder knows both the old and new mask for each rune. The routing protocol is therefore modified to allow the Builder to apply the new mask to the routed items before they are revealed to the Holders. The Builder pre-determines the mask schedule which records, for each rune, which mask will be applied to it at each location. The Builder secret-shares the mask schedule between the Holders before any queries occur, in the same way as the position schedule was shared. Again, the runes are revealed to the Holders (after a random permutation) but the mask schedule

itself is secret-shared. When a rune is revealed, the Holders can re-share the portion of the mask schedule that corresponds to that rune, and the parties can retrieve a secret-sharing of the correct mask inside of a MPC protocol.

Since the Holders store *identical* masked items, the protocol can engage in a 2-server PIR with the Holders acting as the server, and the Builder acting as the client. Again, the PIR protocol must be modified to allow the input index and output to be secret shared (see Figure 5.5).

Note that the PIR protocol will only be applied to arrays of length $c + (b-1)lh$ which will be poly-logarithmic in $n$, so the total computation remains poly-logarithmic in $n$.

## 5.3. Functionality

The definition we need for the Secret-Shared RAM functionality differs slightly from that of Figure 2.4. This is largely because we implement the rune map using a smaller instance of a DORAM. In particular, we will modify the definition in the following ways:

- The functionality will not be initialized to be empty, but will rather be initialized based on a secret-shared array.
- The functionality allows both Reads and Writes to be performed in a single operation: the function returns (reads) the old value and at the same time writes the provided new value.
- We only consider the case where the number of items in the array is equal to the size of the index space. That is, this is strictly a RAM; we do not consider the Map variant.

The second modification is necessary as it is essential that each call to the main DORAM protocol result in only one call to the sub-DORAM implementing the rune map. Imagine, instead, that for each query to the main DORAM, the protocol made 2 queries to the sub-DORAM, one for reading the old value and another for writing the new value. This sub-DORAM would then make 4 queries to its sub-DORAM and so on. This would mean the total number of queries in the lowest instance of the recursion would be *exponential* in the depth of the recursion.

To differentiate our functionality from that of Figure 2.4 we refer to this modified functionality as Secret-Shared Read-and-Write RAM, or SS-RAW-RAM.

**Functionality $\mathcal{F}_{SS-RAW-RAM}$**

$I \leftarrow$ **Init(n, d, [[A]])**: Store array $A$ containing $n$ items of size $d$.

$[[y_{old}]] \leftarrow I$**.ReadWrite(**$[[x]]$**,** $[[y]]$**)**: Given an index $x \in [1, n]$, set $y_{old} \leftarrow A_x$. Update $A_x \leftarrow [[y]]$.

If only a read is desired, the functionality can be called twice, writing back the original value.

Our DORAM implementation makes use of the following functionalities. We show how to implement these using standard techniques in Sections 5.5 and 5.6 respectively.

**Functionality $\mathcal{F}_{SSPIR}$**

$[[v]] \leftarrow$**SSPIR**$(m, d, [[A]]_{(1,2)}, [[x]])$: Given an array $A$ held (duplicated) by $P_1$ and $P_2$, containing $m$ elements of size $d$, and a share of $x \in [1, m]$, return a fresh secret-sharing of $A_x$.

**Functionality $\mathcal{F}_{Route}$**

$[[B]] \leftarrow$ **Route**$([[A]], [[Q]]_0)$: Given a secret-sharing of array $A$, of length $m$, and an injective mapping $Q : [1, m] \rightarrow [1, q]$, for $q \geq m$, held by $P_0$, create a fresh secret-shared array $B$ such that $B_{Q(i)} = A_i$ for all $i \in [1, m]$ and $B_j$ is distributed uniformally at random for all $j \notin \{Q(i)\}_{i \in [1,m]}$.

## 5.4. DORAM Protocol

This section presents the DORAM protocol in full and analyzes its security and communication complexity. The description is based closely on the technical overview of Section 5.2. We proceed with the description of protocols by functions, starting with initialization (Section 5.4.1), followed by Reads (Section 5.4.2), Writes and Rebuilds (Section 5.4.3) and finally Refreshes (Section 5.4.4). Section 5.4.5 then demonstrates that the protocol achieves the desired security properties. Finally section 5.4.6 analyzes the complexity of the protocol. We assume the existence of functionalities for secret-shared PIR and secure routing, implementations of which are presented in Sections 5.5 and 5.6 respectively.

#### 5.4.1. Initialization

In the initialization phase the Builder:

- Pre-determines the allocation of runes that it will use ($M$). $n$ of these are the initial runes assigned to indexes. The other $n$ it will assign to items which were queried during the execution of the protocol.

- Pre-computes the location of every rune at every point in time until the next refresh. Recall from the technical overview (Section 5.2) that since the Builder has pre-determined when each rune will be assigned, and knows the public hash functions, the Builder is able to pre-compute how the tables should be built to ensure satisfying assignments. The Builder creates the position schedule $(S, F, P)$ containing this information.

- Pre-determines the mask schedule $(S, F, E)$, which records which mask will be used for each location.

- Secret-shares the position schedule and mask schedule (sorted by rune) to the Holders.

The parties then coordinate to build the first table, which consists of items containing the initial state of the RAM.

The rune map is stored in a sub-DORAM. For simplicitly, our protocol will treat the sub-DORAM as providing a memory containing $n$ items of size $\log(n) + 1$, mapping indices to runes. In fact, the sub-DORAM will be implemented by a DORAM containing $\frac{n}{2}$ items of size $2(\log(n)+1)$ by packing adjacent index pairs together. During a query, the queried half of the pair is selected and returned, and only the queried half of the pair is updated by the write (the unqueried half retains its original value). This allows the rune map to be implemented using $\log(n)$ recursive calls to DORAMs, each containing $O(n)$ items of size $\Theta(\log(n))$ bits. Each of the recursive DORAMs will be implemented using the same protocol as the main DORAM. This recursion and packing technique is simple and is standard in the position maps of Tree ORAMs (e.g. [SvDS+13]), so we omit a formal description of it.

The protocol parameters are chosen to achieve efficiency and security. The instantiations of the DORAM used to recursively implement the rune map must use *the same parameter values* as the top-level DORAM, even though they store smaller numbers of items. The cache and number of hash functions per table are both chosen to be super-logarithmic ($c = \log^2(n)/\log(\log(n))$ and $h = \log^{1.5}(n)/\log(\log(n))$) so that the probability that the hash functions result in table that has no satisfying assignment, for any choice of inputs, is negligible in $n$ (see the proof in Section 5.4.5). We balance the cost of queries and rebuilds [KLO12] by storing many tables in each level, specifically $b = \log^{0.5}(n)$. This means that the total number of levels, and therefore the the number of times that an item is rebuilt into a new table is $l = \lceil \log_b(n/c) \rceil = \Theta(\log(n)/\log(\log(n)))$.

**DORAM: Init**

**Parameters**:

- Cache size: $c = \log^2(n)/\log(\log(n))$
- Tables per level: $b = \log^{0.5}(n)$
- Number of levels: $\ell = \lceil \log_b(n/c) \rceil = \Theta(\log(n)/\log(\log(n)))$
- Number of hash functions: $h = \log^{1.5}(n)/\log\log(n)$
- Hash functions: $H_1, \ldots, H_h$

**Init(n, d, [[A]]):**

1. $P_0$ creates a random permutation which determines the assignment of runes, $[[M]]_0 : [1, 2n] \to [1, 2n]$
2. Assign the first $n$ of these to be the original runes for the indices. Initialize a new sub-DORAM to store this initial rune map.
   (a) for $i \in [1, n]$, $[[R_i]] \leftarrow [[M_i]]_0$
   (b) runeMap = $\mathcal{F}_{SS-RAW-RAM}$**.Init(**$n, (\log(n) + 1), [[R]]$**)**
3. $P_0$ locally builds all the tables for the next $n$ queries, based on its knowledge of the runes involved, and the hash functions.
   If there *is no satisfying assignment* for one of the tables, $P_0$ tells $P_1$ and $P_2$ to **abort** the protocol. Otherwise, $P_0$ can determine where each rune's item will be when, and it creates the *position schedule* which consists of these three matrices:
   - $[[S_{i,r}]]_0$ contains the time rune $r$'s item will start to be in its $i^{th}$ position.
   - $[[F_{i,r}]]_0$ contains the time rune $r$'s item will finish to be its $i^{th}$ position.
   - $[[P_{i,r}]]_0$ contains the $i^{th}$ position of rune $r$'s item.
4. $P_0$ creates a mask-schedule. Note that the start and finish times will be the same as the position schedule. Therefore all that is needed is one additional matrix containing the OTPs:
   $[[E_{i,r}]]_0$ contains the OTP used to mask rune $r$'s item when it is in its $i^{th}$ position.
5. $P_0$ XOR secret-shares the position schedule and mask schedule between $P_1$ and $P_2$:
   $([[S]]_{1,2}, [[F]]_{1,2}, [[P]]_{1,2}, [[E]]_{1,2}) \leftarrow ([[S]]_0, [[F]]_0, [[P]]_0, [[E]]_0)$
6. $P_0$ provides the masks to the items, based on his previous selection:
   for $i \in 1, \ldots, m$: $[[E_i]]_0 \leftarrow [[E_{0,[[R_i]]_0}]]_0$
7. Based on the Builder's previous assignment of the initial locations of the initial runes, he sets $[[Q]]_0$ to be the injection from $[1, n]$ to $[1, 2(1 + \epsilon)n]$ that builds the initial table.
8. The parties create the table containing the initial items, and $P_1$ and $P_2$ store the masked items:
   $[[T_{\ell+1}]]_{(1,2)} \leftarrow \mathcal{F}_{Route}([[A]] \oplus [[E]]_0, [[Q]]_0)$
9. The runes, values and indices of the initial items are stored for future reference. That is, for $i \in [1, n]$: $[[R_{\ell+1,i}]] = [[R_i]]$
   $[[V_{\ell+1,i}]] = [[A_i]]$
   $[[X_{\ell+1,i}]] = [[i]]$
10. Initialize the query counter: $t = 0$.

Figure 5.1: DORAM: Init functionality

### 5.4.2. Reads

Our ReadWrite proceeds in the expected way. First, the previous value is read, that is it is obtained from the hierarchy of tables by first obtaining a secret-sharing of its exact location and then performing secret-shared PIR on the set of query locations. Then, it writes the new value to the cache. If the cache has become full, then a rebuild occurs, which build new tables and, every $n$ accesses, also performs a refresh.

The read is achieved by first obtaining the current rune of the queried index. The rune is revealed only to the Holders. The rune is used for two purposes. It allows the Holders to determine the set of query locations in which the item could be located: the query locations within each table are the result of the public hash functions applied to the rune. Secondly it allows the Holders to access the relevant entry in the position schedule and the mask schedule. This allows the parties to obtain a secret- sharing of the item's exact location within the set of query locations, as well as a secret-sharing of the item's current mask. Using secret-shared PIR, the parties are able to efficiently obtain a secret-sharing of the queried item.

**MetaDORAM: ReadWrite and Read functions**

**ReadWrite**($[[x]], [[y]]$):

1. $[[y_{old}]]$ =**Read**($[[x]]$)
2. **Write**($[[x]], [[y]]$)
3. **Rebuild**()
4. $t = t + 1$
5. Return $[[y_{old}]]$

**Read**($[[x]]$):

1. Access the rune map to obtain the (secret-shared) rune of $x$. At the same time, update the rune map to store $x$'s new rune, as pre-determined by $P_0$.
   (a) $P_0$ supplies the new rune: $[[r_{new}]] \leftarrow [[M_{n+t}]]$
   (b) $[[r_{old}]] \leftarrow$ runeMap.**ReadWrite**($[[x]], [[r_{new}]]$)
   (c) Reveal $x$'s (old) rune to $P_1$ and $P_2$: $[[r]]_{(1,2)} \leftarrow [[r_{old}]]$
   (d) Append $[[r]]_{(1,2)}$ to $[[D]]_{(1,2)}$, the set of runes which $P_1$ and $P_2$ have already observed.
2. $P_1$ and $P_2$ access rune $r$'s entries in the position schedule and mask schedule (which is secret shared between them) and reshares these entries to a 3-party RSS (without revealing to $P_0$ which rune was accessed). For $j \in [0, \ell + 1]$:
   (a) $[[S_j]] \leftarrow [[S_{j,[[r]]_{(1,2)}}]]_{1,2}$
   (b) $[[F_j]] \leftarrow [[F_{j,[[r]]_{(1,2)}}]]_{1,2}$
   (c) $[[P_j]] \leftarrow [[P_{j,[[r]]_{(1,2)}}]]_{1,2}$
   (d) $[[E_j]] \leftarrow [[E_{j,[[r]]_{(1,2)}}]]_{1,2}$
3. The parties obtain secret-shares of the correct position and mask. For $j \in [0, \ell + 1]$:
   (a) Set $[[J_j]] \leftarrow ([[S_j]] \leq [[t]]) \ \wedge ([[F_j]] > [[t]])$
   (b) If $[[J_j]]$ set $[[p]] \leftarrow [[P_j]]$ and set $[[e]] \leftarrow [[E_j]]$
4. $P_1$ and $P_2$ create an array $Y$ containing all of the (masked) blocks which may hold rune $r$'s block:
   (a) $[[Y_{1,\ldots,c}]]_{(1,2)}$ contains the blocks from the cache. These are padded to length $c$ with empty blocks if the cache is not full.
   (b) For $i \in [1, l + 1]$, $u \in [1, b - 1]$, $k \in [1, h]$:
   Set $[[Y_{c+(i-1)bh+(u-1)h+k}]]_{(1,2)} \leftarrow [[T_{i,u,H_k([[r]]_{(1,2)})}]]_{(1,2)}$.
   This is the $H_k([[r]])^{th}$ location in table $T_{i,u}$. If table $T_{i,u}$ does not exist, set location to an empty block.
5. $[[y_{old}]] \leftarrow \mathcal{F}_{BalancedSSPIR}(c + l(b-1)h, d, [[Y]]_{(1,2)}, [[p]]) \oplus [[e]]$
6. Return $[[y_{old}]]$

Figure 5.2: DORAM read protocol

### 5.4.3. Writes and Rebuilds

An item is written simply by storing it in the cache.

Recall that the Builder pre-computes, before any accesses occur, the location of each rune at each point in time, based on its pre-determining of when runes will be assigned, and the public hash functions. During a Rebuild, the Builder provides the satisfying assignment which it pre-computed, which is recorded in the position schedule, and this is used to build the new table.

The Builder knows the location of items in their old tables, so the protocol could first extract the the items from the old tables by "unpermuting" them. We find it simpler to consider that each level stores a secret-sharing of all items which are in that level, arranged in the order which they were queried. These are stored as matrices $V$ (for the values), $X$ (for the indexes) and $R$ (for the runes). Extracting tables is therefore unnecessary, and the items from multiple levels can be easily combined. The Builder then provides new masks for each item, consistent with the Builder's pre-determined mask schedule. The masked items are then used as the input of a secure routing protocol, in which the Builder provides the appropriate routing needed to place each item in a valid location and the Holders obtain the resulting table of masked items.

**MetaDORAM: Write Rebuild**

**Write**($[[x]], [[v_{new}]]$):

1. $P_0$ sets $r$ to be the pre-determined rune for this time: $[[r]]_0 \leftarrow [[M_{n+t}]]_0$.
2. $j = t \mod c$
3. $P_0$ sets $e$ to be the pre-determined initial mask for the newly-written rune: $[[e]] \leftarrow [[E_{0,[[r]]_0}]]_0$.
4. $[[C_j]]_{(1,2)} \leftarrow [[v_{new}]] \oplus [[e]]$
5. To aid future rebuilds and refreshes, the secret-shared $v_{new}$, $r$ and $x$ are stored in a matrix:
   $[[V_{0,j}]] = [[v_{new}]]$
   $[[R_{0,j}]] = [[r]]_0$
   $[[X_{0,j}]] = [[x]]$

**Rebuild():**

1. for $i \in [0, \ell - 1]$:
   (a) if $t = 0 \mod b^i c$ (i.e. $L_i$ is full):
      i. $u = (t/(b^i c)) \mod b^{i+1} c$ (the number of tables in $L_{i+1}$).
      ii. for $j \in [1, b^i c]$:
         A. $[[R_{i+1,ub^i c+j}]] = [[R_{i,j}]]$
         B. $[[V_{i+1,ub^i c+j}]] = [[V_{i,j}]]$
         C. $[[X_{i+1,ub^i c+j}]] = [[X_{i,j}]]$
         D. $P_0$, knowing the mask schedule and the rune of each item, applies the appropriate mask to each value:
            $[[Z_{i+1,ub^i c+j}]] = [[V_{i+1,ub^i c+j}]] \oplus [[E_{i+1,[[R_{i+1,ub^i c+j}]]_0}]]$
         E. Delete $[[R_{i,j}]]$, $[[V_{i,j}]]$ and $[[X_{i,j}]]$.
      iii. $P_0$ provides $[[Q]]_0$, the injective mapping from $[1, b^i c]$ to $[1, 2(1+\epsilon)b^i c]$ which maps each item to a satisfying location. (This was pre-computed during the Init stage.)
      iv. Use this mapping to build a table containing the newly masked blocks:
         $[[T_{i+1,u}]]_{(1,2)} = \mathcal{F}_{Route}(\ [[Z_{i+1,ub^i c+1 \ldots (u+1)b^i c}]], [[Q]]_0)$
   (b) $if (t = n)$ **Refresh**()

Figure 5.3: MetaDORAM write and rebuild functions

### 5.4.4. Refreshes

The Builder pre-determines the runes to be used for $n$ accesses at a time, and likewise pre-computes the position schedule and mask schedule for $n$ accesses at a time. After $n$ accesses, the DORAM therefore needs to be refreshed.

The refresh can be divided into two parts. First, the contents of the up-to-date memory is extracted. This is achieved by randomly permuting all blocks and revealing their runes to the Holders. The Holders know which runes have been queried, so can identify these blocks as obsolete, leaving only the blocks which contain the most recently written value for each index. The extract protocol returns a secret-shared array of the current memory; that is using the same format as that provided for the Init function. The refresh protocol then simply calls the Init function using this secret-shared array to create all of the data-structures necessary for a further $n$ queries. The Extract functionality is useful in its own right, as it allows the memory contents of the MetaDORAM to be returned as a secret-shared array.

The Refresh and Extract protocols are presented formally in Figure 5.4 below.

**MetaDORAM: Refresh and Extract**

**Refresh():**

1. $[[V]] \leftarrow \text{Extract}()$
2. $\text{Init}(n, d, [[V]])$

$[[V]] \leftarrow$ **Extract()**:

1. Concatenate all (non-deleted) $R$, $V$ and $X$ into a single secret-shared array. This will contain all runes that have been used thus far, the index they corresponded to, and the value that was assigned to that index at the time that the rune was assigned:
   $[[R]] = [[R_0]]||[[R_1]]\ldots||[[R_{\ell+1}]]$, $[[V]] = [[V_0]]||[[V_1]]\ldots[[V_{\ell+1}]]$, $[[X]] = [[X_0]]||[[X_1]]\ldots[[X_{\ell+1}]]$
2. Let $m$ (where $n \leq m \leq 2n$) be the length of these arrays.
3. $P_1$ picks a random permutation $S : [1, m] \to [1, m]$. Let all items be securely routed according to $[[S]]_1$:
   $[[R]] = \mathcal{F}_{Route}([[R]], [[S]]_1)$, $[[V]] = \mathcal{F}_{Route}([[V]], [[S]]_1)$, $[[X]] = \mathcal{F}_{Route}([[X]], [[S]]_1)$
4. $P_2$ similarly picks a random permutation, $U : [1, m] \to [1, m]$ which is used to permute all items:
   $[[R]] = \mathcal{F}_{Route}([[R]], [[U]]_2)$, $[[V]] = \mathcal{F}_{Route}([[V]], [[U]]_2)$, $[[X]] = \mathcal{F}_{Route}([[X]], [[U]]_2)$
5. The values $R$ are revealed to $P_1$ and $P_2$. Note that $R$ will contain a random subset of $m$ items from $[1, 2n]$: $[[R]]_{(1,2)} \leftarrow [[R]]$.
6. $P_1$ and $P_2$ identify all runes which have already been revealed to them. The locations of these items in the permuted arrays are made public, and the items are deleted:
   For $i \in [1, m]$, $I_i = 0$ if $[[R_i]]_{(1,2)} \in [[D]]_{(1,2)}$, else 1
   If $I_i = 0$, delete $[[X_i]]$ and $[[V_i]]$ (and re-assign indices).
7. Reveal $[[X]]$ to all parties. (This will contain all indices in $[1, n]$ in a random order.) Sort $[[V]]$ locally according to $[[X]]$.
8. Return $[[V]]$.
9. Delete all variables and the sub-DORAM storing the rune map.

Figure 5.4: Refresh and Extract functionalities

5.4.5. Security Analysis

In this section we show that the DORAM protocol is secure. That is, the views of all participants in the protocol can be efficiently simulated without knowledge of any private values. We show that this security holds in the $\mathcal{F}_{SSPIR}$, $\mathcal{F}_{Route}$-hybrid model.

All steps of the protocol are one of four cases. Either:

- It is an operation on secret-shares (with a secret-shared output).
- It is an operation on public, pre-determined values (e.g. $t$).
- A secure functionality is being accessed, that only outputs secret-shared results (e.g. $\mathcal{F}_{SSPIR}$).
- A value is revealed to some party, or subset of the parties.

The first three cases are easily simulatable. We therefore only need to examine all revealed values and show that they can be simulated without knowledge of the private inputs.

**Init:** No information is revealed to $P_0$, rather all private variables it holds are the result of its own random choices (the runes and OTPs) and public parameters (the hash functions).
If a Build Failure occurs, this is revealed to $P_1$ and $P_2$. This event happens with negligible probability, as will be proven later. The simulator can therefore choose that this event does not occur, and the executions will be statistically indistinguishable.
$P_1$ and $P_2$ learn $T_{\ell+1}$. All of these blocks have been masked by fresh OTPs, so this is simulatable by generating a uniformly random string.

**Read:** No information is revealed to $P_0$.
$P_1$ and $P_2$ learn the rune queried. The runes are distributed uniformly at random from $[1, 2n]$, subject to the fact that they are each unique. Nevertheless, the revealed runes, combined with the knowledge that every table was built successfully, could leak information. It will later be shown that such leakage occurs with negligible probability.

**Write:** No information is revealed to $P_0$.

$P_1$ and $P_2$ learn $C_j$. This has been masked using a fresh OTP, so can be simulated by generating a random string.

**Rebuild:** No information is revealed to $P_0$.

$P_1$ and $P_2$ learn $T_{i,u}$. This contains blocks which have been masked under fresh OTPs, so can be simulated by generating random strings.

**Extract:** $P_0$ learns $X$. This will contain the items $[1, n]$ in a randomly permuted order. This can be seen by induction. The protocol maintains the invariant that at each point in time, each index $x$ has a single rune assigned to it which has not been observed by $P_1$ and $P_2$. In other words, there is a single rune $R_{i,j}$, such that $X_{i,j} = x$ and $R_{i,j} \notin D$. Therefore, when the indices corresponding to viewed runes are deleted, a single instance of each index will remain. They will be in a random order because they have been shuffled according to a permutation known to no parties.

$P_0$ also learns $I$. This contains $n$ 1s and $m - n$ 0s in a random order, for the reasons explained above.

$P_1$ and $P_2$ additionally learn $R$. This contains a subset of $m$ runes from $[1, 2n]$. It will necessarily include all $m - n$ runes from $D$, since these runes are definitely stored in the system. The other $n$ runes are distributed uniformly at random from the set of the remaining $2n - (m - n)$ runes, so are efficiently simulatable. The ordering must be consistent with $I$, that is the $m - n$ previously observed runes must have $I_i = 0$.

Therefore, the only challenging part of the security proof is showing that the distribution of the queried runes (revealed to the Holders), combined with the knowledge that all tables were built successfully, does not leak information except with negligible probability. Leakage could occur if some queried set of runes resulted in a set of hash function outputs that was incompatible with that set being stored in a given table. We show that this does not occur by showing that, for all table capacities $m \leq n$, the probability that there exists *any* subset of size $m$ of the $2n$ runes that would result in a build failure is negligible.

We prove this making use of Yeo's analysis of Robust Cuckoo Hashing [Yeo23]. Yeo considered an adversary who could pick the indices of items in a hash table, and attempted to pick these such that would cause a build failure, given the predetermined hash functions. His analysis works in general for determining the probability that, given a set of elements there exists some subset of these that would result in a build failure. Specifically, we can rephrase his Lemma 3 with our notation. Let there be a cuckoo hash table of size $\Theta(m)$ with $h$ hash functions. Then the probability that there exists some subset of $[1, 2n]$ of size $m$ that results in this cuckoo hash table to have a build failure is at most:

$$\left(\frac{2n}{2^{h-3}}\right)^{h+1}$$

This probability does not depend on $m$, except for requiring that $m \leq 2n$. We would like this probability to be negligible in $n$, i.e. with probability $2^{-\omega(\log(n))}$ Setting $h = \log^{1.5}(n)/\log(\log(n)) = \omega(\log(n))$ achieves this.

This indicates that, for any given table, there is a negligible probability that there exists a set of runes that would be incompatible with this hash table. Since there are poly(n) different tables ever constructed (in fact only polylog(n), since all tables at all times within a level can re-use the same hash functions), the probability that there is any table in the protocol that has any incompatible set of runes is also negligible in $n$. Note that the subDORAMs, even though they have smaller sizes, they should use the same parameter $h$ as the top level, so that the failure probability remains negligible in the size of the top DORAM, $n$.

Therefore, except with negligible probability (over the choice of hash functions), a build failure cannot occur with any choice of runes. This means that, except with negligible probability, there will never be observed a set of runes queried that is incompatible with any allocation of runes to tables. This completes the proof.

**Remark 6.** An interesting corollary of this is that, for most choices of hash functions, the protocol

is actually *perfectly* secure, that is *no information* is leaked about the access pattern. Given an exp(n)-time setup, it would be possible to test whether a certain choice of hash functions allows for successful builds under all appropriately-sized rune sets, and therefore achieves perfect security. It is not clear whether such a setup for a perfectly-secure protocol can be achieved in poly(n) time. Instead this section shows that, over the randomness of the choice of hash functions, the protocol is statistically secure, that is the adversary is unable to distinguish any access patterns, except with probability negligible in $n$.

5.4.6. Complexity Analysis

In this section, we show that the amortized communication complexity per access is $\Theta((\log^2(n) + d)\log(n)/\log(\log(n))$ bits. We assume that the cost of $\mathcal{F}_{BalancedSSPIR}$ is $\Theta(\sqrt{md} + d)$ and the cost of $\mathcal{F}_{Route}$ is $\Theta(q(d + \log(q))$ as instantiated by our implementations in Sections 5.5 and 5.6 respectively.

First we analyze the parts of the protocol that have the same cost per-access: reads and writes. We initially analyze only the first level of the recursion. We analyze the number of bits of communication by section, using the same enumeration as the protocols.

**Read:**

1. The rune of the index is accessed and a new rune written. Apart from the call to the subDO-RAM, which will be analyzed later, this involves only operations on runes of size $\Theta(\log(n))$ bits, so this step requires $\Theta(\log(n))$ communication.

2. The rune's entries in the position schedule and the mask schedule are reshared. The position schedule entry contains $l = \Theta(\log(n)/\log(\log(n)))$ values, each of size $\Theta(\log(n))$ bits (to store the timestamps). The mask schedule entry contains $l$ masks, each of size $\Theta(d)$ bits. Therefore, resharing these requires $\Theta((\log(n) + d)\log(n)/\log(\log(n)))$ bits of communication.

3. The correct time slot is obtained and the position and mask for that time slot retrieved. Obtaining the time slot requires $\Theta(l) = \Theta(\log(n)/\log(\log(n)))$ comparisons of $\Theta(\log(n))$-bit

values, which requires $\Theta(\log^2(n)/\log(\log(n)))$ communication. Selecting the correct position and mask requires $\Theta(l) = \Theta(\log(n)/\log(\log(n))$ secure if-then-else statements on $\Theta(\log(n))$-bit and $\Theta(d)$-bit values for the positions and masks respectively. The total communication cost is therefore $\Theta((\log(n)+d)\log(n)/\log(\log(n)))$.

4. The Holders arrange the blocks which may hold the rune's block. This requires only local operations and no communication.

5. Finally the SSPIR is executed. The number of locations is $c+l(b-1)h = \Theta(\log^3(n)/\log^2(\log(n)))$. Therefore, the cost of the Balanced SSPIR protocol is $\Theta(\sqrt{\log^3(n)d/\log^2(\log(n))} + d) = \Theta(\log(n)/\log(\log(n))\sqrt{\log(n)d} + d)$. For $d = \Omega(\log(n))$, $\sqrt{\log(n)d} = O(d)$, so the cost above simplifies to $O(\log(n)d/\log(\log(n)))$.

**Write:** The only two steps which require communication are when the new mask is re-shared (at cost $\Theta(d)$) and when the new rune is reshared (at cost $\Theta(\log(n)) = O(d)$) resulting in a total cost of $\Theta(d)$.

We next analyze the communication cost of the Rebuild function (excluding the refresh function). The communication cost of this function is variable, so we calculate the amortized cost per access.

**Rebuild:**

A level of capacity $m$ is rebuilt every $m$ accesses. Most steps are simply relabelling of variables, which require no communication. The steps that require communication are:

- The Builder secret-shares the new OTP for each item, which costs $\Theta(md)$.
- The Routing protocol, which requires $\Theta(m(d+\log(n))$ communication.

Therefore, the amortized cost per access per level is $\Theta(\log(n)+d)$. Since there are $\Theta(\log(n)/\log(\log(n)))$ levels, the total communication cost per access is $\Theta((\log(n)+d)\log(n)/\log(\log(n)))$.

**Extract:**

1. Concatenating the arrays requires only local relabelling of variables, except for the runes which are reshared from $P_0$ to being shared by all parties, at communication cost $\Theta(n \log(n))$.

2. Setting $m$ is a local operation.

3. $m = \Theta(n)$ elements are routed, each of size $\Theta(\log(n) + d)$ resulting in $\Theta(n(\log(n) + d))$ communication.

4. The same occurs again, resulting in $\Theta(n \log(n) + d)$ communication.

5. Revealing all runes to Holders requires $\Theta(n \log(n))$ communication.

6. Holders reveal $m = \Theta(n)$ bits, hence $\Theta(n)$ communication.

7. Revealing all (permuted) indices requires $\Theta(n \log(n))$ communication.

8. The last 2 steps are local operations.

Since this occurs every $n$ accesses, the cost is $\Theta(\log(n) + d)$ communication per access.

**Init:**

1. The rune assignment is local, so has no communication.

2. The cost of initializing the subDORAM will be evaluated as part of the cost of recursion.

3. Creating the position schedule is a local operation

4. Creating the mask schedule is a local operation

5. The position schedule has $\Theta(n)$ columns (for the runes), $\Theta(\ell) = \Theta(\log(n)/\log\log(n))$ rows (for the levels) and has $O(\log(n))$ bits per cell, for both the timestamp representations and the position representations. Each cell of the mask schedule is $\Theta(d)$ bits. Therefore the total cost of secret-sharing the position and mask schedules is $\Theta((\log n + d)n \log(n)/\log(\log(n)))$ communication.

6. Selecting the pre-chosen OTPs is a local operation.

7. Assigning the mapping to build the OHTable is a local operation.

8. Secret-sharing the mask, and routing the blocks requires a total of $\Theta((\log(n) + D)n)$ communication.

9. The last step is a local relabelling.

Therefore, the total cost is $\Theta((\log(n)+d)n \log(n)/\log(\log(n)))$, or $\Theta((\log(n)+d)\log(n)/\log(\log(n)))$ per access.

Summing these up, we obtain that the cost at the first level of the recursion is $\Theta((\log(n) + d)\log(n)/\log(\log(n)))$. In the first level of the recursion, the block size $d$ can be arbitrary. However, for the recursively implemented subDORAM, the block size is always $\Theta(\log(n))$. Therefore, each level of the recursion has cost $\Theta(\log^2(n)/\log(\log(n)))$. There are $\Theta(\log(n))$ such levels, so the cost of the recursive calls is $\Theta(\log^3(n)/\log(\log(n)))$. Hence, the total communication cost per access is $\Theta((\log^2(n) + d)\log(n)/\log(\log(n)))$.

While our focus is amortized total communication per query, for completeness we also provide below the performance of our protocol by other metrics. The total memory required by the protocol is $\Theta(\log(n)dn/\log(\log(n)))$: this is dominated by the size of the mask matrix (assuming $d = \Omega(\log(n))$) which must be held in memory by $P_1$ and $P_2$. The round-complexity is dominated by the cost of evaluating inequality tests (in step 3 of **Read**) which uses a circuit with AND-depth $\Theta(\log(\log(n)))$ and therefore needs $\Theta(\log(\log(n)))$ rounds. This is done sequentially in all $\Theta(\log(n))$ recursions of the subDORAM, leading to a total round complexity per query of $\Theta(\log(n)\log(\log(n)))$. The computation cost depends on the hash function implementation, and in most cases would be dominated by the evaluation of $\ell h = \Theta(\log^{2.5}(n)/\log^2(\log(n)))$ hash functions per recursion level, or a total of $\Theta(\log^{3.5}(n)/\log^2(\log(n)))$ hash function evaluations per query. The protocol accesses $c+\ell(b-1)h = \Theta(\log^3(n)/\log^2(\log(n))$ memory locations of size $d$ in the top level, and $c + \ell(b-1)h$ memory locations of size $\Theta(\log(n))$ in each of the recursive levels, resulting in a

total of $\Theta(\log^3(n)d/\log^2(\log(n)) + \log^5(n)/\log^2(\log(n)))$ bits of memory accessed per query.

## 5.5. Secret-Shared Private Information Retrieval

This section presents a simple protocol for secret-shared Private Information Retrieval that is optimized for our use-case.

In general Private Information Retrieval protocols are designed for the case that a single bit is to be retrieved. However in our protocols we need to retrieve $d$ bits, which all occur in a single location in memory. We therefore use the following "naïve" PIR protocol. Let $x$ be the secret location, and $m$ the length of the memory, that is $1 \leq x \leq m$. The secret location is represented using a $m$-bit array, which is 0 everywhere except for the $x^{th}$ bit, which is 1. This array is secret-shared between the two parties, who can locally compute a dot-product of this string with their memory, to obtain a secret-sharing of the desired element. While this PIR protocol has a query of length $m$, a single query can be used regardless of the bit-length $d$. That is the same query string is used for all $d$ bits of the data. The cost is therefore $\Theta(m + d)$.

The above protocol assumes that there is a PIR client who can safely learn the location $x$. It is possible to apply a transformation to obtain a *secret-shared* PIR protocol. This technique has been used before, for instance, in the "Data-Rotations" of [FJKW15]. The PIR servers ($P_1$ and $P_2$) are given a location mask $x_2$, and locally permute their array according to this mask, such that each item is moved from location $i$ to location $i \oplus x_2$. The PIR client ($P_0$) then searches for a location $x_1 = x \oplus x_2$. This will clearly hold the index that was at location $x$. The security of the PIR protocol hides the query from the PIR servers. The client only receives $x_1$ which is a uniform random value.

The protocol is presented in full in Figure 5.5.

The SSPIR protocol (Figure 5.5) is secure. $P_0$ receives only $x_0$ which is a uniform random value. $P_1$ and $P_2$ receive only shares of $Q$, which are uniform random bit arrays. The protocol is deterministic and secure.

The protocol presented above has communication cost $\Theta(m + d)$. For some situations this is suffi-

**SSPIR**

**UnbalancedSSPIR**$(m, d, [[A]]_{(1,2)}, [[x]])$

1. Convert $[[x]]$ to a XOR sharing in which $P_0$ holds one share ($x_1$) and $P_1$ and $P_2$ both hold the other share ($x_2$):
   $[[x]]_{0,(1,2)} \leftarrow [[x]]$
2. $P_0$ creates a bit-array, $Q$, of length $m$ such that $Q_i = 1$ for $i = x_1$ and is 0 elsewhere.
3. $P_0$ XOR-shares this array between $P_1$ and $P_2$: $[[Q]]_{1,2} \leftarrow [[Q]]_0$
4. $P_1$ and $P_2$ permute this array according to $x_2$, that is they create an array $[[W]]_{1,2}$ such that $W_i = Q_{i \oplus x_2}$.
5. $P_1$ and $P_2$ compute $[[v]]_{1,2} = \oplus_{i=1}^{m} [[A_i]]_{(1,2)} [[W_i]]_{1,2}$. Note that the $A_i$ are public to $P_1$ and $P_2$, so the multiplication is simply multiplication of a secret by a public value, which is a local operation.
6. Return $[[v]]$

**BalancedSSPIR**$(m, d, [[A]]_{(1,2)}, [[x]])$

1. Let $\log(q) \leftarrow \lceil \frac{\log(m) - \log(d)}{2} \rceil$. That is $q$ is the smallest power of 2 such that $q^2 \geq m/d$.
2. Modify the memory from containing $m$ blocks of length $d$ bits, to containing $m/q$ blocks of length $dq$. Let $[[B]]_{(1,2)}$ be the updated memory, that is $B_i = A_{qi} || \ldots || A_{qi+q-1}$
3. Let $[[x]]$ be split into its upper-order $\log(m) - \log(q)$ bits, labelled $[[y]]$ and its lowest-order $\log(q)$ bits, labelled $[[z]]$.
4. Call the main SSPIR protocol to obtain the secret-shared $y^{th}$ large block:
   $[[u]] \leftarrow SSPIR(m/q, dq, [[B]]_{(1,2)}, [[y]])$.
5. Inside of a secure computation, access the $z^{th}$ small block in this big block:
   for $i \in [1, q]$ if $i = [[z]]$, $[[v]] \leftarrow [[u_{id \ldots id+i-1}]]$.
6. Return $[[v]]$.

Figure 5.5: Implementation of SSPIR

cient. However, when $m = \omega(d)$ it is possible to increase the size of data-blocks to achieve improved complexity, effectively "balancing" the $m$ and $d$ terms. We do this by increasing the block size from $d$ to $qd$, for some balancing factor $q > 1$, where $q$ is a power of 2.

We present the balanced PIR protocol in the second part of Figure 5.5. All operations are inside of a secure computation, so the protocol is secure. The cost of the call to the main SSPIR protocol is $\Theta(m/q + dq)$. Additionally, there is a cost of $\Theta(qd)$ to securely select the relevant small block. The total cost is therefore $\Theta(m/q + dq)$. Our protocol picks the optimum $q = \Theta(\sqrt{m/d} + 1)$ which results in a communication cost of $\Theta(\sqrt{md} + d)$. (The last term in both equations comes from the case when $m = O(d)$.)

## 5.6. Secure Routing

We here present an implementation of a secure 3-party routing protocol. That is, there is some secret-shared array $A$ of length $m$ and one party knows an injective mapping $Q$ from $[1, m]$ to $[1, q]$, (where $q \geq m$). The items are moved to a new secret-shared array $B$ such that $A_i$ is moved to some location $B_j$ where $j = Q(i)$. See Section 5.3 for a formal definition of the functionality. Variants of this protocol have occurred before, for instance as the protocol $\Pi_{SWITCH}$ in [MRR20]. We include the protocol here for clarity and completeness. The protocol is presented in Figure 5.6, and is analyzed below.

**Security:** $P_0$, knowing a desired permutation, secret-shares this permutation between $P_1$ and $P_2$, providing them permutation shares $R$ and $S$ respectively. Each of these permutation-shares is distributed as a uniformly random permutation, and leaks no information about the true permutation $Q$. Apart from that, parties only receive secret-shares, which are distributed uniformly at random.

**Complexity:** Communicating the permutations requires $\Theta(q \log(q))$ communication. There are a constant number of resharings of arrays, each of which contains $q$ elements of size $d$ bits, resulting in $\Theta(qd)$ communication. The total communication cost is therefore $\Theta((d + \log(q))q)$.

**Routing**

**Route**$([[A]], [[Q]]_0, d)$:

1. Pad $[[A]]$ to length $q$ (if $q > m$) with random values:
   for $i \in [1, m]$, $[[B_i]] = [[A_i]]$
   for $i \in [m+1, q]$, $[[B_i]] \leftarrow \{0,1\}^d$.
2. $P_0$ picks permutations $R$ and $S$ which are chosen uniformly at random subject to $R \cdot S = Q$ (over domain $[1, m]$ and is an arbitrary permutation elsewhere). $P_0$ sends $S$ to $P_1$ and sends $R$ to $P_2$:
   $[[S]]_{(0,1)} \leftarrow [[S]]_0$
   $[[R]]_{(0,2)} \leftarrow [[R]]_0$
3. Reshare $B$ to $P_0$ and $P_1$: $[[B]]_{0,1} \leftarrow [[B]]$
4. $P_0$ and $P_1$ locally permute $[[B]]_{0,1}$ according to $[[S]]_{(0,1)}$ to obtain $[[C]]_{0,1}$.
5. Reshare $C$ to $P_0$ and $P_2$: $[[C]]_{0,2} \leftarrow [[C]]_{0,1}$
6. $P_0$ and $P_2$ locally permute $[[C]]_{0,2}$ according to $[[R]]_{(0,2)}$ to obtain $[[D]]_{(0,2)}$
7. Return $[[D]]$

Figure 5.6: Secure Routing protocol

## 5.7. Conclusion and Future Work

In this work, we construct an information-theoretic DORAM with $O((d + \log^2(n)) \log(n) / \log(\log(n)))$ bits of communication per query, which is *below* the lower bound on communication for passive ORAMs.

This begs the question: *What is the communication lower bound for active DORAMs?* Abraham et al. [AFN+17] show that constant overhead is possible for polynomial-sized blocks. Is constant overhead also possible for polylogarithmic sized blocks? Also, does the lower-bound introduce trade-offs between the amount of communication and the number of memory locations accessed, the total memory required, the computation cost, the randomness consumed or the round complexity?

Another open question pertains to deamortization. MetaDORAM achieves *amortized* communication cost $\Theta((d + \log^2(n)) \log(n) / \log(\log(n)))$. In particular, the cost of rebuilding the tables, and refreshing the rune namespace is amortized across multiple queries. There are standard techniques for deamortizing the cost of building tables, but it seems more challenging to deamortize the cost of refreshing, in particular the cost of refreshing the rune namespace by reassigning new runes to all

indices. This leads to the question: *Is it possible to have a DORAM with* worst-case *communication cost* $\Theta((d + \log^2(n)) \log(n) / \log(\log(n)))$ *(or better)?*

Every DORAM can be used to implement a multi-server active ORAM: the client in the multi-server ORAM can simply secret-share the query between the servers. So our construction implies that $\Theta((\log^2 n + d) \log(n) / \log \log(n))$ communication can be achieved, without computational assumptions. An interesting final open question raised by this work is whether this is possible for a single-server active ORAM. Due to existing lower bounds, such an ORAM would need to access at least a logarithmic overhead in memory and therefore perform a logarithmic overhead of computation, but this computation could, perhaps, avoid the introduction of computational assumptions. Specifically, this leads to the question: *Is it possible for a single-server active ORAM to have sub-logarithmic communication overhead and be information-theoretically secure?*

# CHAPTER 6

## Conclusion

Distributed Oblivious RAM is a crucial research area for the development of efficient, generic MPC. It is, however, far from simple, for several reasons.

Firstly, the security of oblivious protocols is often subtle. The challenges in the correct use of Cuckoo Hash tables in the Hierarchical (D)ORAM paradigm is the clearest example of this. In the case of a build failure, tables cannot be rebuilt, as observed by [GM11] and [KLO12]. Furthermore, tables must have a build failure probability that is not only negligible in the size of the table itself, but is negligible in the size of the (D)ORAM (or depends on an independent statistical security parameter). The Alibi attack presented in this thesis shows yet another subtlety: even if oblivious data structures are constructed securely such that build failure is negligible, if they are accessed in a non-standard way, this can lead to statistical leakage.

Secondly, DORAM is closely related to, yet distinct from, ORAMs. The ORAM literature is extensive and contains many variants, including active ORAMs and multi-server ORAMs. These can typically be converted to DORAM protocols, but the conversion is not necessarily without overhead, due to the fact that ORAMs have a trusted CPU/client, and that some ORAMs assume an adversary who can only see the locations accessed in memory, and cannot actually see the contents. Therefore, positioning DORAM results is non-trivial, as it requires comparing with many previous works, some of which are not explicitly presented as DORAMs.

Lastly, and most interestingly, existing lower bounds do not apply directly to DORAMs. Specifically, in the DORAM setting, the Goldreich-Ostrovsky lower bound [GO96] [LN18] applies only to the amount of memory that must be accessed, but does not apply to the amount of communication performed, as shown by [AFN$^+$17] and the MetaDORAM protocol from this thesis. It remains an open question to show a tight asymptotic bound on the total communication cost of information-theoretic DORAMs. It is also not clear whether such a bound would be identical for all security

settings, such as malicious security or dishonest majority. For instance, the MetaDORAM protocol does not easily generalize to more than one corruption, since security depends on the fact that the adversary cannot corrupt both the Builder and a Holder. The trade-offs between communication and other important parameters, such as the round complexity and the amount of computation, is also not yet well understood.

This thesis presented some progress in the state of the art of DORAM, presenting two new DORAM protocols with improved communication efficiency. It also described a pitfall, the Alibi attack, showing that (D)ORAM security is subtle, and presenting a remedy for the attack that applied to existing protocols.

MPC will likely become an increasingly mainstream technology, with profound impact on the way data is shared, accessed and monetized. As this occurs, the question of how to implement efficient DORAMs, allowing efficient generic RAM-based MPC, will become all the more pertinent.

# BIBLIOGRAPHY


[ADW14] Martin Aumüller, Martin Dietzfelbinger, and Philipp Woelfel. Explicit and efficient hash families suffice for cuckoo hashing with a stash. *Algorithmica*, 70(3):428–456, 2014.

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma 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*, pages 805–817, 2016.

[AFN+17] Ittai Abraham, Christopher W Fletcher, Kartik Nayak, Benny Pinkas, and Ling Ren. Asymptotically tight bounds for composing oram with pir. In *IACR International Workshop on Public Key Cryptography*, pages 91–120. Springer, 2017.

[AJX+19] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A commodity obfuscation engine on intel sgx. In *Network and Distributed System Security Symposium*, 2019.

[AKL+20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: optimal oblivious ram. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II 30*, pages 403–432. Springer, 2020.

[AKL+22] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Optimal oblivious parallel ram. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2459–2521. SIAM, 2022.

[AKLS23] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. Oblivious ram with worst-case logarithmic overhead. *Journal of Cryptology*, 36(2):7, 2023.

[AKM23] Gilad Asharov, Ilan Komargodski, and Yehuda Michelson. Futorama: A concretely efficient hierarchical oblivious ram. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 3313–3327, 2023.

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An o (n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9, 1983.

[AKSL18] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *NDSS*, 2018.

[AKST14] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable

oblivious storage. In *Public-Key Cryptography–PKC 2014: 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings 17*, pages 131–148. Springer, 2014.

[ARS+15] Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. 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 I 34*, pages 430–454. Springer, 2015.

[ARS+17] Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. LowMC implementation. https://github.com/LowMC/lowmc/blob/master/determine_rounds.py, 2017.

[BCP15] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram and applications. In *Theory of Cryptography Conference*, pages 175–204. Springer, 2015.

[BKKO20] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed oram. In *Security and Cryptography for Networks: 12th International Conference, SCN 2020, Amalfi, Italy, September 14–16, 2020, Proceedings 12*, pages 215–232. Springer, 2020.

[BOGW19] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali*, pages 351–371. 2019.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, 1988.

[CCHR16] Ran Canetti, Yilei Chen, Justin Holmgren, and Mariana Raykova. Adaptive succinct garbled ram or: How to delegate your database. In *Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part II 14*, pages 61–90. Springer, 2016.

[CCS17] T-H Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel ram. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 567–597. Springer, 2017.

[CD16] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom

secret-sharing and applications to secure computation. In *Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005. Proceedings 2*, pages 342–362. Springer, 2005.

[CGLS17] T-H Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient oram and opram. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*, pages 660–690. Springer, 2017.

[CH16] Ran Canetti and Justin Holmgren. Fully succinct garbled ram. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 169–178, 2016.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[CKN+18] T-H Hubert Chan, Jonathan Katz, Kartik Nayak, Antigoni Polychroniadou, and Elaine Shi. More is less: Perfectly secure oblivious algorithms in the multi-server setting. In *Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part III 24*, pages 158–188. Springer, 2018.

[CLP14] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure oram with overhead. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 62–81. Springer, 2014.

[CLT16] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel ram: improved efficiency and generic constructions. In *Theory of Cryptography: 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II 13*, pages 205–234. Springer, 2016.

[CNS18] T-H Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel ram. In *Theory of Cryptography Conference*, pages 636–668. Springer, 2018.

[CSLN21] TH Hubert Chan, Elaine Shi, Wei-Kai Lin, and Kartik Nayak. Perfectly oblivious (parallel) ram revisited, and improved constructions. In *2nd Conference on Information-Theoretic Cryptography*, 2021.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious ram without random oracles. In *Theory of Cryptography: 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings 8*, pages 144–163. Springer, 2011.

[Ds17] Jack Doerner and Abhi shelat. Scaling oram for secure computation. In *Proceedings of*

*the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 523–535, 2017.

[DSS14] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. Burst {ORAM}: Minimizing {ORAM} response times for bursty access patterns. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 749–764, 2014.

[DvDF+16] Srinivas Devadas, Marten van Dijk, Christopher W Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion oram: A constant bandwidth blowup oblivious ram. In *Theory of Cryptography: 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II 13*, pages 145–174. Springer, 2016.

[FIPR05] Michael J Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005. Proceedings 2*, pages 303–324. Springer, 2005.

[FJKW15] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-party oram for secure computation. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 360–385. Springer, 2015.

[FNO22] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. 3-party distributed oram from oblivious set membership. In *International Conference on Security and Cryptography for Networks*, pages 437–461. Springer, 2022.

[FNO24] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. Metadoram: Info-theoretic distributed oram with less communication. *Cryptology ePrint Archive*, 2024.

[FNR+15] Christopher Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket oram: single online roundtrip, constant bandwidth oblivious ram. *Cryptology ePrint Archive*, 2015.

[GGH+13] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing oram and using it efficiently for secure computation. In *Privacy Enhancing Technologies: 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings 13*, pages 1–18. Springer, 2013.

[GHL+14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled ram revisited. In *Advances in Cryptology–EUROCRYPT 2014: 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings 33*, pages 405–422. Springer, 2014.

[GKW18] S Dov Gordon, Jonathan Katz, and Xiao Wang. Simple and efficient two-server oram. In *Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the*

*Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part III 24*, pages 141–157. Springer, 2018.

[GLO15] Sanjam Garg, Steve Lu, and Rafail Ostrovsky. Black-box garbled ram. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 210–229. IEEE, 2015.

[GLOS15] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. Garbled ram from one-way functions. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 449–458, 2015.

[GM11] Michael T Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *International Colloquium on Automata, Languages, and Programming*, pages 576–587. Springer, 2011.

[GMOT11] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 95–100, 2011.

[GMP16] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. Tworam: efficient oblivious ram in two rounds with applications to searchable encryption. In *Annual International Cryptology Conference*, pages 563–592. Springer, 2016.

[GMW19] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 307–328. 2019.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194, 1987.

[Goo17] Michael T Goodrich. Bios oram: improved privacy-preserving data access for parameterized outsourced storage. In *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society*, pages 41–50, 2017.

[HFNO21] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. Alibi: A flaw in cuckoo-hashing based hierarchical ORAM schemes and a solution. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 338–369. Springer, 2021.

[HK22] Shai Halevi and Eyal Kushilevitz. Random-index oblivious ram. In *Theory of Cryptog-*

*raphy Conference*, pages 33–59. Springer, 2022.

[HKO22] David Heath, Vladimir Kolesnikov, and Rafail Ostrovsky. Epigram: Practical garbled ram. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2022.

[HOY+17] Thang Hoang, Ceyhun D Ozkaptan, Attila A Yavuz, Jorge Guajardo, and Tam Nguyen. S3oram: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 491–505, 2017.

[HV21] Ariel Hamlin and Mayank Varia. Two-server distributed oram with sublinear computation and constant rounds. In *IACR International Conference on Public-Key Cryptography*, pages 499–527. Springer, 2021.

[JW18] Stanislaw Jarecki and Boyang Wei. 3pc oram with low latency, low bandwidth, and fast batch retrieval. In *Applied Cryptography and Network Security: 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings 16*, pages 360–378. Springer, 2018.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[KLR10] Eyal Kushilevitz, Yehuda Lindell, and Tal Rabin. Information-theoretically secure protocols and security under composition. *SIAM Journal on Computing*, 39(5):2090–2112, 2010.

[KM19] Eyal Kushilevitz and Tamer Mour. Sub-logarithmic distributed oblivious ram with small block size. In *IACR International Workshop on Public Key Cryptography*, pages 3–33. Springer, 2019.

[KMW10] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2010.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for mpc. In *Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20*, pages 506–525. Springer, 2014.

[Lin17] Yehuda Lindell. How to simulate it–a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pages 277–346, 2017.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *Annual International Cryptology Conference*, pages 523–542. Springer, 2018.

[LO13a] Steve Lu and Rafail Ostrovsky. Distributed oblivious ram for secure two-party computation. In *Theory of Cryptography Conference*, pages 377–396. Springer, 2013.

[LO13b] Steve Lu and Rafail Ostrovsky. How to garble ram programs? In *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*, pages 719–734. Springer, 2013.

[LSY20] Kasper Green Larsen, Mark Simkin, and Kevin Yeo. Lower bounds for multi-server oblivious rams. In *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part I 18*, pages 486–503. Springer, 2020.

[LWZ11] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-efficient oblivious database manipulation. In *Information Security: 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings 14*, pages 262–277. Springer, 2011.

[Mit09] Michael Mitzenmacher. Some open questions related to cuckoo hashing. In *European Symposium on Algorithms*, pages 1–10. Springer, 2009.

[MRR20] Payman Mohassel, Peter Rindal, and Mike Rosulek. Fast database joins and psi for secret shared data. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1271–1287, 2020.

[MU17] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[MZ14] John C Mitchell and Joe Zimmerman. Data-oblivious data structures. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[Nob21] Daniel Noble. Explicit, closed-form, general bounds for cuckoo hashing with a stash. *Cryptology ePrint Archive*, 2021.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In Frank Thomson Leighton and Peter W. Shor, editors, *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 294–303. ACM, 1997.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious rams. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 514–523, 1990.

[Ost92] Rafail Ostrovsky. *Software protection and simulation on oblivious RAMs*. PhD thesis, Massachusetts Institute of Technology, 1992.

[PLS23] Andrew Park, Wei-Kai Lin, and Elaine Shi. Nanogram: Garbled ram with o˜(log n) overhead. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 456–486. Springer, 2023.

[PPRY18] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious ram with logarithmic overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882. IEEE, 2018.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[PR10] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30*, pages 502–519. Springer, 2010.

[RS19] Michael Raskin and Mark Simkin. Perfectly secure oblivious ram with sublinear bandwidth overhead. In *Advances in Cryptology–ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8–12, 2019, Proceedings, Part II 25*, pages 537–563. Springer, 2019.

[SCSL11] Elaine Shi, T H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o ((log n) 3) worst-case cost. In *Advances in Cryptology–ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings 17*, pages 197–214. Springer, 2011.

[SGF17] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. *Cryptology ePrint Archive*, 2017.

[Shi20] Elaine Shi. Path oblivious heap: Optimal and practical oblivious priority queue. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 842–858. IEEE, 2020.

[SS12] Emil Stefanov and Elaine Shi. Path o-ram: An extremely simple oblivious ram protocol. *arXiv preprint arXiv:1202.5150*, 2012.

[SSS11] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. *arXiv preprint arXiv:1106.3652*, 2011.

[SvDS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310, 2013.

[VHG23] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A bandwidth-efficient distributed oram for 2-and 3-party computation. In *32nd USENIX Security Symposium*,

2023.

[Wak68] Abraham Waksman. A permutation network. *Journal of the ACM (JACM)*, 15(1):159–163, 1968.

[WCS15] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 850–861, 2015.

[WHC+14] Xiao Shaun Wang, Yan Huang, TH Hubert Chan, Abhi shelat, and Elaine Shi. Scoram: oblivious ram for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 191–202, 2014.

[WS12] Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 293–304, 2012.

[WSC08] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 139–148, 2008.

[Yao82] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[Yeo23] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. In *Annual International Cryptology Conference*, pages 197–230. Springer, 2023.