Electronic Colloquium on Computational Complexity, Report No. 5 (2024)

MetaDORAM: Breaking the Log-Overhead Information Theoretic Barrier

Daniel Noble[*] Brett Hemenway Falk[†] Rafail Ostrovsky[‡]

January 3, 2024

Abstract

This paper presents the first Distributed Oblivious RAM (DORAM) protocol that achieves sub-logarithmic communication overhead without computational assumptions. That is, given n d-bit memory locations, we present an information-theoretically secure protocol which requires $o(d \cdot \log(n))$ bits of communication per access (when $d = \Omega(\log^2(n))$.

This comes as a surprise, since the Goldreich-Ostrovsky lower bound shows that the related problem of Oblivious RAMs requires logarithmic overhead in the number of memory locations accessed. It was shown that this bound also applies in the multi-server ORAM setting, and therefore also applies in the DORAM setting. Achieving sub-logarithmic communication therefore requires accessing and using $\Omega(\log(n) \cdot d)$ bits of memory, without engaging in communication for each bit accessed. Techniques such as Fully Homomorphic Encryption and Function Secret Sharing allow secure selection of the relevant memory locations with small communication overhead, but introduce computational assumptions.

In this paper we show that it is possible to avoid a logarithmic communication overhead even without any computational assumptions. Concretely, we present a 3-party honest-majority DORAM that is secure against semi-honest adversaries. The protocol has communication cost

$$\Theta\left((\log^2(n) + d) \cdot \frac{\log(n)}{\log(\log(n))}\right)$$

For any $d = \Omega(\log^2(n))$ the overhead is therefore $\Theta(\log(n)/\log(\log(n)))$. Additionally, we show a subtle flaw in a common approach for analyzing the security of Oblivious Hash Tables. We prove our construction secure using an alternative approach.

[*] University of Pennsylvania, dgnoble@seas.upenn.edu
[†] University of Pennsylvania, fbrett@seas.upenn.edu
[‡] UCLA, rafail@cs.ucla.edu

ISSN 1433-8092

1 Introduction

A Distributed Oblivious RAM (DORAM) is a protocol that implements a RAM ideal functionality in a secure multiparty protocol. That is, it allows reads and writes to a memory, where the indices and data values (whether read or written) are secret-shared. To be secure, the information available to the adversary should reveal nothing about the indices or values that are read or written, or the data stored in the memory.

DORAM is closely related to the classic problem of Oblivious RAM (ORAM) [GO96], where there is a single client and a single server. The client stores n d-bit elements on the server in such a way that its physical memory accesses on the server reveal no information about its virtual accesses. In the original problem, there was a single server that simply provided storage and did not perform any computation. Goldreich and Ostrovsky showed that in this setting the client must access $\Omega(\log(n))$ physical memory locations for each virtual location accessed [GO96]. This was referred to as $\Omega(\log(n))$ *overhead* since a non-oblivious RAM requires accessing 1 memory location. Goldreich and Ostrovsky showed this lower bound to hold in the "balls and bins" memory model. Later, Larsen and Nielsen showed that the restriction that the client must access $\Omega(d \cdot \log(n))$ bits holds for arbitrary data encodings [LN18]. Larsen, Simkin and Yeo further showed that even if there are multiple non-colluding servers, the client still must access $\Omega(d \cdot \log(n))$ bits of memory [LSY20].

The ORAM model can also be extended to allow the server(s) to perform computation. In this case, the amount of data communicated and the amount of memory accessed are no longer the same. For instance a simple solution is to encrypt both the memory and the query using Fully Homomorphic Encryption (FHE). The server can then use FHE[1] to update the encrypted memory (for a write) and return only the encrypted result (for a read). This results in a *memory access overhead* of $\Theta(n)$, but a *communication overhead* of only $\Theta(1)$ [MG07, MG08, MCR21, MW22]. Therefore, the bound no longer applies to the communication overhead. Nevertheless, the bound still applies to the *memory access overhead.* To see this, observe that if the servers accessed $o(d \cdot \log(n))$ bits of memory, the system could simply offload all computation to the client, allowing the client to access all memory locations and perform all necessary computation, contradicting the lower bound of [LSY20].

It is possible to implement a w-server ORAM using a w-party DORAM. This can be done by each DORAM server acting as one of the servers in the multi-server ORAM protocol. The client can then secret-share the queries between the servers and then combine the secret-shares of the query response. Therefore, all lower bounds that apply to a w-server ORAM scheme must also apply to a w-party DORAM. Hence any DORAM also requires $\Theta(\log(n))$ *memory access overhead.*

The question remained open: is it possible to have a DORAM with sub-logarithmic *communication overhead* without introducing computational assumptions? It seems that DORAM papers generally expected that it was not possible. Many DORAMs aimed to approach the Goldreich-Ostrovsky bound, none aimed to surpass it. For instance Faber et al state that their protocol has "bandwidth and CPU costs which are comparable to those of the underlying Client-Server ORAM" [FJKW15] while Jarecki and Wei state that they "bridge the gap between [DORAMs] and client-server ORAM" [JW18]. It seems that a general implicit intuition existed that DORAMs could not beat the communication bounds on classic client-server ORAM, at least without introducing

[1] Even without FHE, it is possible to achieve sub-logarithmic communication ORAM under computational assumptions [KM19].

excessive computation and computational assumptions. This intuition seems justified by examining a standard approach to solving DORAMs proposed by Ostrovsky and Shoup [OS97]: to take an ORAM and simulate the client inside a secure computation. In this case, even secret-sharing the memory accessed by the client requires $\Theta(\log(n)d)$ bits of communication.

Surprisingly, we break this logarithmic barrier. Specifically, we present a statistically-secure DORAM protocol that has sub-logarithmic communication overhead. Our protocol is not based on simulating an ORAM client, but rather, we take advantage of having multiple non-colluding computing servers to achieve low communication overhead. Our protocol requires three parties and is secure against a single semi-honest corruption. The amortized communication cost is $\Theta((\log^2(n)+d)\frac{\log(n)}{\log(\log(n))})$ bits per query.

Our Contribution: This paper presents the first DORAM protocol that has information-theoretic security and obtains sub-logarithmic communication overhead. This solves an open question as to whether the Goldreich-Ostrovsky lower bound applies to communication overhead in statistically-secure DORAMs. Specifically, our result 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 10).

Concretely, this article presents the first improvement in the communication overhead of information-theoretic DORAM protocols in 8 years ([WCS15]). Our DORAM is also better in terms of communication overhead even than most DORAM protocols that use computational assumptions, except those that require polynomial computation with linear overhead per access. See Table 1 in section 2 for more details.

Additionally, we point out an important subtlety in the security analysis of Oblivious Hash Tables that receive more queries than their contents. We demonstrate the security of our protocol despite this subtlety.

Organization: Our paper is organized as follows. Section 2 provides a summary of prior DORAM protocols. Section 3 provides a technical overview of our results and techniques. Section 4 explains the notation used, in particular the various types of secret-sharing used and how they are represented. The formal DORAM functionality is presented in section 5, as well as the functionalities for secret-sharing private information retrieval (SSPIR) and secure routing, which are used by our DORAM protocol. Section 6 presents our full DORAM protocol, and analyzes its security and communication costs. Our security analysis points out a subtlety that exists in analyzing Oblivious Hash Tables; section 7 explores this in more detail and shows that a common analysis approach does not actually provide the desired security guarantees. Sections 8 and 9 explain how SSPIR and secure routing, respectively, can be implemented using standard techniques. Section 10 concludes by discussing some interesting open questions.

2 Prior Works

In this section we provide a brief overview of DORAM protocols. A common approach to developing DORAMs has been to take an ORAM and implement the client inside of a secure computation [OS97]. However, ORAM clients had not initially been designed to be easy to implement in a secure computation. For instance many ORAMs make use of PRFs, but evaluating a PRF inside of a secure computation requires $\Theta(\kappa)$ operations. A new approach was suggested: to develop

Protocol	**Communication Cost (bits)**	**Security**
Circuit ORAM [WCS15]	$\omega\left((\log^2 n + d)\log n\right)$	Statistical
[LO13]	$O\left((\kappa + d)\log n\right)$	Computational
[FJKW15]	$O\left(\kappa\sigma\log^3 n + \sigma d\log n\right)$	Computational
[JW18]	$O\left(\kappa\log^3 n + d\log n\right)$	Computational
[BKKO20]	$O\left(d\sqrt{n}\right)$	Computational
[FNO22]	$O\left((\kappa + d)\log n\right)$	Computational
DuORAM [VHG23]	$O\left(\kappa \cdot d \cdot \log n\right)$	Computational
Our protocol	$\Theta((\log^2(n) + d)\log(n)/\log(\log(n)))$	Statistical

Table 1: Complexity of several DORAM protocols. κ is a cryptographic security parameter, σ is a statistical security parameter.

an ORAM in which the client was designed to have low circuit complexity, and then to create a DORAM by evaluating the client inside a secure computation [WHC$^+$14]. This idea was developed further by the Circuit ORAM protocol [WCS15]. Circuit ORAM presented an information-theoretic ORAM which had both low bandwidth overhead and low client circuit complexity. This resulted in a statistically-secure DORAM with $\omega((\log^2 + d)\log(n))$ communication per access.

Various works since have tried to take advantage of the fact that there are multiple non-colluding computing parties to build efficient protocols. This includes by using secure shuffles/routing [FNO22], using Distributed Point Functions [BKKO20] [VHG23] and secret-shared PIR (SSPIR) [JW18]. However, all of these protocols depend on computational assumptions.

3 Technical Overview

Before explaining our protocol, it is helpful to have an understanding of two general paradigms which are used in constructing (D)ORAMs.

The first is the *Hierarchical* approach, initially proposed by Ostrovsky [Ost90], in which data is stored using hash tables. In the ORAM setting the tables are held (encrypted) by a single server, in the DORAM setting, they can be secret-shared between multiple servers. The hash tables use pseudorandom hash functions, so that the physical locations accessed reveal no information about the corresponding indexes, and are hence called *Oblivious Hash Tables* (OHTables). An OHTable does not solve the ORAM problem, however, because an adversary can typically distinguish repeated queries for the *same element* into an OHTable from queries for *distinct* elements. To solve this, when an item is queried it is cached from an OHTable into a small sub-ORAM. The sub-ORAM is queried first and if the item is found there, random locations are accessed in the OHTables; this is necessary for security since re-querying an item in an OHTable would cause the same locations to be accessed, compromising security. To ensure the sub-ORAM remains small, periodically its contents are extracted and built into a new OHTable. To ensure the number of OHTables remains manageable, periodically the contents of multiple OHTables are extracted and rebuilt into a single new OHTable. Typically, the sub-ORAM and OHTables are envisioned as arranged vertically, with the sub-ORAM at the top and OHTables below it, arranged from smallest to largest, resulting in a pyramid-shaped hierarchy (hence the name) of sub-ORAM/OHTable structures.

Shi et al. [SCSL11] proposed the alternative *Tree* approach, which was extended by many subsequent (D)ORAMs (e.g. [SvDS$^+$18] [WCS15] [FJKW15]). In this solution data is arranged in a tree, each item is assigned a path from the root to the leaf and the item must remain on this path until its next access. To query an item, its path is first obtained. All locations on this path accessed and the item is removed from its location. The item is then assigned a new random path and placed in the root of the tree. The root is typically a small sub-ORAM, whereas each other node in the tree typically has capacity for a constant number of items. To prevent the root sub-ORAM from becoming too large, paths from the root to leaves are periodically accessed and each item in the path moved as far down (leafward) as it can be moved, subject to its own path restriction and congestion from other items. Analysing probability distributions shows that the congestion is unlikely to cause the sub-ORAM at the root to overflow. The assignment of indices to paths, which is called the *position map*, is stored and updated using a sub-ORAM, which is implemented recursively.

As a starting point, our protocol extends the idea of a position map to storing all of the metadata required to determine an item's exact location. Like in Tree ORAMs, a query first accesses a data-structure to gain information about an item. However, rather than just storing a random path, for each index our data structure stores a Random Unique NamE (rune), a position schedule and a mask schedule. It does this by storing the rune in a sub-DORAM, and having tables that allow the additional metadata to be obtained using only the rune. By accessing this data-structure, it is possible to obtain (a secret-sharing of) the exact location at which an item is stored. The items themselves are stored in a Hierarchy of OHTables. Unlike a typical OHTable, the potential locations of an item are based on the rune alone; no pseudorandom functions are used. During a query, the rune is revealed, which allows the servers to locally narrow down the possible query location to a small number of locations in each table, or t locations total. Then, the secret-sharing of the full location is used to securely select the correct item. In the last step, it is possible using a simple information-theoretic PIR approach to do this with only $\Theta(\sqrt{(}td) + d)$ communication, rather than td communication (see section 8).

To allow efficient building of data-structures, the servers have different roles. One server is the *Builder*, while the other two servers are *Holders*. The Holders hold OHTables of masked blocks. For each rune, the Builder always knows where the block for that rune should be located, however it *does not know*, what index that rune corresponds to. Whenever an item is queried the Holders (and *not* the Builder) learn the rune for that item, which allows them to narrow down where the item can be. At the end of the query, the Builder assigns the item a new rune, without learning the associated index, and without the Holders learning the new rune. Therefore, during an access, the Builder learns (decides) the item's new rune and the Holders learn the item's previous rune, but no single party learns both, which provides obliviousness.

During a rebuild, items must be securely moved from locations in smaller tables to locations in a single larger table. The possible locations in which an item can be located within a given table are determined by its rune. Since the Builder knows all of the runes of items that are being built into the table, the Builder can *locally* compute a satisfying assignment for items in the new table. Furthermore, since the Builder knows the previous locations of all items, the Builder can engage in a secure routing protocol with the Holders to permute the elements from their original locations in the old tables to their new locations in the new table.

A query works as follows. First, the current rune for an item is determined using its index, by querying a sub-DORAM. The rune is revealed to the Holders. This *dramatically* reduces the possible space in which the Holders need to search for the item. We will later use (secret-shared)

PIR to retrieve the item from its exact location. While many (D)ORAM protocols have used PIR to access the relevant item, they commonly apply the PIR to the entire index space ($\Theta(n)$), resulting in a linear, or at least $poly(n)$ computational cost. Instead, our PIR is executed over only $o(\log^3(n))$ locations, which results in low computational cost, and allows us to use information-theoretic PIR protocols.

Next, the protocol needs to determine (secret-shares of) the exact location in which the item is located. Since the Builder pre-chooses the runes, the Builder can also pre-compute at the very beginning of the protocol the location assignments at every point in time. This schedule will be in terms of the rune, and not the indexes they correspond to, which is not yet necessarily determined even by the environment. The Builder can secret-share the position schedule between the Holders. For each rune, the Builder provides $\Theta(\log(n)/\log(\log(n)))$ secret-shared time ranges and, for each time range, the position during that time range (from among the possible locations for a given rune). During a query, the protocol uses the secret shares of the timestamp ($\log(n)$ bits), and compares these with the timestamp ranges in order to determine a secret-sharing of the item's position. This reduces the search space to a new database of polylog size, which is pairwise replicated. Finally, we use information-theoretic Private Information Retrieval (PIR), to obtain a secret-sharing of the desired masked element.

Recall that the Holders store *masked* items, rather than secret-shared items. This allows the Holders to hold exactly the same tables of masked elements (rather than secret-sharings of tables, as is normal in DORAM). This is necessary for the PIR protocol. To achieve information-theoretic security, the items are masked using One-Time Pads (OTPs). In order to prevent the Holders tracking how blocks travel through the OHTables it is necessary for blocks to use a new OTP after each build. Since the Builder never learns the masked blocks, it is safe for the Builder to pick the OTPs. The Builder therefore knows how to mask each item. The secure routing protocol can be extended to allow the Builder to specify OTPs for each block, thereby both permuting and re-masking each block. Furthermore, the Builder can pick all of the OTPs at the start of the protocol, and can create a mask schedule, akin to the position schedule. In fact, since the time-ranges are the same in both cases, the secret-shared OTP can be stored and retrieved with the secret-shared position. Locally XORing the secret-shared OTP and the secret-shared masked block results in a secret-sharing of the item.

Our protocol also makes use of "balancing" [KLO12]. Instead of combining constant numbers of small OHTables to create larger OHTables, we wait until there are $\Theta(\log(n))$ OHTables of a given size before rebuilding them into a larger OHTable. This means that the total number of tables is $\Theta(\log^2(n)/\log\log(n))$, but the number of times each item participates in a rebuild is only $\Theta(\log(n)/\log(\log(n))$. This reduces the amortized communication cost of rebuilds.

Novel contributions: In addition to the conceptual surprise that we break the logarithmic barrier overhead on communication complexity, our paper introduces a number of new techniques. Specifically, we use time-stamping and novel data structures to obtain the precise location of data blocks, we make asymmetric use of the participating compute servers to allow efficient and oblivious construction and querying of these data structures, we use as a subroutine tiny-size PIR protocols where the "databases" are constructed on the fly during query execution, and we show a novel strategy for DORAM that bridges techniques from different ORAM strategies in conjunction with ideas explained above. Additionally, we show that when an OHTable has more queries than items, it may leak information with non-negligible probability. We present the first balanced hierarchical DORAM that is proven secure against such leakage.

4 Preliminaries

We use lower-case Latin characters to represent parameters in the protocol. n is the size of the RAM, d is the bit-length of each item. t represents the (maximum) number of OHTables that may exist in the protocol. h represents the number of hash functions used by the hash tables. For an explicit integer or integral parameter, a, $[1, a]$ denotes the set of integers $\{1, \dots, a\}$. We use upper-case Latin characters to represent arrays and matrices, which are indexed using standard subscript notation. Most subscripts are one-indexed; zero-indexing is occasionally used when the first item is somehow special.

We use lg to represent the base-2 log. For asymptotic annotation, any constant base is equivalent, in which case we often use log to represent some arbitrary constant-base log.

We denote the 3 parties as P_0, P_1 and P_2. P_0 is the Builder. P_1 and P_2 are the Holders. The Adversary $\mathcal{A}$, is able to corrupt at most one of the parties. The corruption is semi-honest (passive), that is the corrupted party will still follow the protocol, but $\mathcal{A}$ is able to view all data visible to the corrupted party. The corruption is static, that is $\mathcal{A}$ cannot change which party is corrupted during the protocol. The protocol is statistically secure. Concretely, $\mathcal{A}$ learns no information about the access pattern except with probability that is negligible in n, regardless of $\mathcal{A}$'s computational powers.

We utilize hash functions. Our hash functions are fixed and public. We assume that the hash functions are $2n$-wise independent and independent of each other. The hash functions implicitly map to ranges of different sizes (depending on the size of the OHTable). In this cases, the hash functions are calculated modulo the required range. It is assumed that the output of the hash function has sufficient entropy that even when reduced modulo these ranges, the distribution is still essentially uniform.

We use several kinds of secret-sharing, all of which are bit-wise (Boolean) secret-sharings. We can explicitly state the sharing using the following notation: $(\langle y_0 \rangle_0, \langle y_1 \rangle_1, \langle y_2 \rangle_2)$, which means that P_0 holds share y_0, P_1 holds share y_1 and P_2 holds share y_2. The most common sharing we use is a 3-party replicated secret sharing (3RSS) [AFL+16]. 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}. (All such subscript operations are implicitly modulo 3.) When variable x is held using this secret-sharing, it is represented as $[x]$. Using the notation above, we can say $[x] = (\langle (x_0, x_1) \rangle_0, \langle (x_1, x_2) \rangle_1, \langle (x_2, x_0) \rangle_2)$. Operations (AND, OR, NOT, XOR) are performed on this secret-sharing using the methods of Araki et al [AFL+16].

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 $[x]_{1,2} = (\langle\rangle_0, \langle x_1 \rangle_1, \langle x_2 \rangle_2)$ where $x_1, x_2 \leftarrow \{0,1\}^\ell$ subject to $x_1 \oplus x_2 = x$. We also use a variant of XOR secret-sharing in which 2 parties hold one of the shares, and the third party holds the other. For instance, when P_0 holds one share, and P_1 and P_2 hold the other share, this is denoted $[x]_{0,(1,2)} = (\langle x_1 \rangle_0, \langle x_2 \rangle_1, \langle x_2 \rangle_2)$ where $x_1, x_2 \leftarrow \{0,1\}^\ell$ subject to $x_1 \oplus x_2 = x$. Sometimes a variable is held privately. If x is held privately, for instance, by P_0, we denote this as $[x]_0 = (\langle x \rangle_0, \langle\rangle_1, \langle\rangle_2)$. Sometimes a variable is known to 2 parties but not the third. If x is known to P_1 and P_2, but not P_0, this is denoted $[x]_{(1,2)} = (\langle\rangle_0, \langle x \rangle_1, \langle x \rangle_2)$.

It can be easily shown that for any $x \in \{0,1\}^\ell$ it is possible to convert a sharing of x from any of the above secret-sharings to a fresh resharing of any other of the above sharings with only $\Theta(\ell)$ bits of communication. For brevity, we provide here only a single example. To covert

$[x]_{0,(1,2)} = (\langle x_1 \rangle_0, \langle x_2 \rangle_{1,2})$ to a fresh $[x]$, P_0 picks $a_1, a_2 \leftarrow \{0,1\}^\ell$ and sets $a_3 = x_1 \oplus a_1 \oplus a_2$. P_1 picks $b_1, b_2 \leftarrow \{0,1\}^\ell$ and sets $b_3 = x_2 \oplus b_1 \oplus b_2$. P_2 does nothing. P_0 sends a_i, a_{i+1} to P_i and likewise P_1 sends b_i, b_{i+1} to P_i. Each party P_i locally calculates $c_i = a_i \oplus b_i$ and $c_{i+1} = a_{i+1} \oplus b_{i+1}$ and the new sharing is $[x] = (\langle (c_0, c_1) \rangle_0, \langle (c_1, c_2) \rangle_1, \langle (c_2, c_0) \rangle_2)$.

5 Functionality

We wish to implement the following DORAM functionality:

Functionality $\mathcal{F}_{DORAM}$

$I \leftarrow$ **Init(n, d, [A])**: Store array A containing n items of size d.
$[v] \leftarrow I$**.ReadWrite(**$[x]$**,** $[y]$**,** f**)**: Given an index $x \in [1, n]$, set v to A_x. Set $A_x = f([v], [y])$.

Our definition of a DORAM combines the Read and Write functionalities, allowing for reads, writes, or more complex functionalities. This is done by setting the public function f appropriately. For a read, define $f(v, y) = v$. For a write, define $f(v, y) = y$. Allowing the written value to be a function of the input provides additional flexibility, such as writing to only particular bits of the data-value or applying a bit-mask to the memory value. Implicitly, f must be representable using a Boolean circuit containing $\Theta(d)$ AND gates.

Our DORAM implementation makes use of the following functionalities. We show how to implement these using standard techniques in sections 8 and 9 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, held by P_0, of length $q \geq m$, create a fresh secret-sharing B such that $B_{Q(i)} = A_i$ for all $i \in [1, m]$ and B_j is distributed uniformally at random for all $j \notin \{Q(i)\}_{i \in [1,m]}$.

6 DORAM Protocol

6.1 Overview

This section presents the DORAM protocol in full and analyzes its security and communication complexity. The protocol is first presented at a high level in section 6.1 and then presented in detail in sections 6.2 and 6.3. Section 6.4 then demonstrates that the protocol achieves the desired security properties. Finally 6.5 analyzes the security and complexity of the protocol respectively. We assume the existence of functionalities for secret-shared PIR and secure routing, implementations of which are presented in sections 8 and 9 respectively.

Our protocol maintains a data-structure composed of multiple Oblivious Hash Tables (OHTables). The data-structure is stored by two parties, called the Holders (P_1 and P_2). They hold identical copies of the data-structure (this will facilitate use of PIR to access elements). To provide privacy, all items in these Oblivious Hash Tables are masked. (The masks are information-theoretic, based on one-time pads.)

We refer to the items in the data structure as blocks. Each block is labelled by a random tag, chosen from $[1, 2n]$, which we refer to as a rune (Random Unique NamE). Although the Holders store the blocks, they are not aware of the runes for these blocks. The other party, called the Builder (P_0), *does* know the rune of every block, and knows the block's location in the data-structure, but never stores any blocks itself.

Every index is assigned a rune. The block for that rune holds the (masked) data value at that index's location in memory. When an index is read or written to, it is assigned a new rune and a new block is created which holds the, potentially updated, value. However, the old block, and the old rune, continue to exist in the system. The Builder does not learn that that rune was used, and the Holders do not learn that that block was accessed. Therefore, the data-structure holds both active and obsolete blocks, and the way that blocks are moved through the data-structure does not depend on whether the block is active or obsolete. These obsolete blocks will periodically be deleted, during a refresh phase which happens every n accesses, which we explain in more detail later.

The mapping of indices to runes is stored in a sub-DORAM. In the sub-DORAM, adjacent indices are stored together, so there are only $n/2$ indices in the sub-DORAM. Each data-value in the sub-DORAM therefore stores 2 runes, which needs $\Theta(\log(n))$ space. The sub-DORAM is implemented recursively, so there are $\Theta(\log(n))$ levels to the recursion.

We now present the full DORAM protocol. We first describe how writes and rebuilds occur. Reads depend on data-structures for the metadata, so we then show how to create these during the initialization and refresh phases. We then show how reads are achieved. Finally we analyze the communication complexity and security of the protocol.

6.2 Writes and Rebuilds

We first show how the data-structure storing the blocks is written to and rebuilt. Initially, all blocks are stored in a single, large, OHTable. When an index is queried, it is assigned a new rune, which is picked from the correct distribution by the Builder, and the sub-DORAM is updated with this information. A new block is then created which holds the new value for that index. This block is placed in an area called the *cache*. The cache is filled sequentially. The cache is of size $\lg^2(n)/\lg\lg(n)$. When the cache becomes full, its contents are extracted and built into an OHTable.

We implement the OHTable using cuckoo hashing with many hash functions. The block may be stored in locations corresponding to the output of the hash functions *on the block's rune.* Since the Builder knows the runes of every block, the Builder is able to *locally* compute an assignment from runes to locations. It can then collaborate with the Holders to securely route the blocks to their correct locations. It is important for the Holders not to be able to tell how the blocks were permuted. It is therefore necessary to re-mask them. All masks are achieved information-theoretically using one-time pads (OTPs). The Builder picks the random OTPs each time a block is masked.

We periodically combine multiple OHTables into a single OHTable. Once there are b OHTables of a given size, the contents of all of these OHTables are extracted and then are built into a single new OHTable. We refer to the OHTables as being arranged in levels. The first, or top, level, L_0, contains the cache. The next level, L_1, contains OHTables that were built by extracting the contents of the cache. We label these tables $T_{1,1}, \ldots, T_{1,b}$. Since the cache is of capacity c, each OHTable in L_1 will also be of capacity c. L_1 will contain at most b such OHTables; when there are b such tables, they will be combined into an OHTable of size bc which will be placed in L_2, and so on. Note that, once the bth OHTable in a level is built, it is immediately combined with all other OHTables in that level to construct an OHTable in the next level. Therefore, during queries there are only at most $b - 1$ tables at any level. Since each level's capacity is b times larger than that of the level before it, a total of $\Theta(\log_b(n/c))$ levels will be needed to store the blocks created by n queries. For the parameters we choose, $\log_b(n/c) = \Theta(\lg(n)/\lg(\lg(n)))$. After n queries, the refresh occurs, the contents of all OHTables (and the cache) are extracted, and the active blocks are rebuilt into a single, large OHTable of size n, as at the start of the protocol. The Rebuild protocol is presented in Figure 1, together with the overall DORAM ReadWrite function and the Write function.

6.3 Reads and Refreshes

The question remains as to how the function **Read**$([x])$ can be implemented efficiently. Firstly, we reveal the rune of x to the Holders, let it be called r. This greatly simplifies the problem. It is known that the block is stored either in the cache, or in location $H_k(r)$ of some table $T_{i,j}$, for some $i \in [1, \ell], j \in [1, b - 1], k \in [1, h]$. This reduces the number of possible locations to $c + \ell(b - 1)h$.

This is still a significant number of locations. P_0 knows, for each rune and each time, the location at which each item is stored. However, r cannot be revealed to P_0 during a read, since P_0 knows when the index with rune r was last accessed, which would allow P_0 to link access times of indices. In short, the Builder knows the location of each rune, but there seems no way to make use of this without leaking information about the current rune being queried.

Recall that in the description of the DORAM write protocol, P_0 gets to *pick* the rune. P_0 should pick the runes such that each rune is unique, but apart from this runes are chosen uniformly at random from $[1, 2n]$. Therefore, the choice of runes does not depend on any other activity in the protocol. Hence, P_0 is able to pick *all of the runes at the beginning of the protocol.* In other words, P_0 can pre-choose the runes that it will assign at each point in time, and during the protocol can assign runes consistently with this original assignment.

Observe, further, that P_0 builds the OHTables based solely on the hash functions and the runes. Since these are both known at the start of the protocol, P_0 can also pre-calculate all assignments in all hash tables at the beginning of the protocol. This allows P_0 to locally create a *position schedule*, that is a data-structure storing where each rune will be located at each point in time.

This allows us to sidestep the conundrum described above. The Builder can secret-share the position schedule containing all information about the locations of all of the runes, once, at the start of the protocol. The Holders can then access the relevant parts of the position schedule dynamically as they learn the rune of each queried block. Note that this location is the location among all of the possible locations that the block may have been located based on the rune (up to c cache locations, and up to $\ell(b - 1)h$ table locations).

Given secret-shares of the location of the block, the protocol now engages in a secret-shared PIR (SSPIR) to obtain a secret-sharing of the block. SSPIR can be implemented using a simple modification of any 2-party PIR protocol. The SSPIR protocols we use are explained in more detail in Section 8.

DORAM: ReadWrite Write Rebuild

Parameters:

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

```
DORAM.ReadWrite($[x], [y], f$):
  1. $[v]$ =Read($[x]$) (Defined in Figure 4)
  2. $[v_{new}] = f([v], [y])$
  3. Write($[x], [v_{new}]$)
  4. Rebuild() (Performs rebuilds, if needed)
  5. $t = t + 1$ (Counter indicating the number of queries)
  6. Return $[v]$
Write($[x], [v_{new}]$):
  1. $P_0$ picks a new, unused, rune $r$ from $[1, 2n]$
  2. $j = t \mod c$
  3. $P_0$ picks a OTP, $e$ from $\{0,1\}^d$, to mask the block.
  4. $[C_j]_{(1,2)} = [v_{new}] \oplus [e]$
  5. For future rebuilds and refreshes, the secret-shared $v_{new}$, $r$ and $x$ are stored in a matrix:
     $[V_{0,j}] = [v_{new}]$
     $[R_{0,j}]_0 = [r]$
     $[X_{0,j}] = [x]$
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 $j \in [b^i c]$:
             A. $[R_{i+1,ub^ic+j}]_0 = [R_{i,j}]_0$
             B. $[V_{i+1,ub^ic+j}] = [V_{i,j}]$
             C. $[X_{i+1,ub^ic+j}] = [X_{i,j}]$
             D. Delete $[R_{i,j}]_0$, $[V_{i,j}]$ and $[X_{i,j}]$.
             E. $P_0$ picks a new OTP, $E_{i+1,ub^ic+j}$ from $\{0,1\}^d$.
             F. $[Z_{i+1,ub^ic+j}] = [V_{i+1,ub^ic+j}] \oplus [E_{i+1,ub^ic+j}]$
         iii. $P_0$ locally builds an OHTable using $R_{i,1\dots b^ic}$, and hash functions $H_1, \dots, H_k$. Let $[Q]_0$
              be the injective mapping from $[b^ic]$ to $[2(1+\epsilon)b^ic]$ that maps $R_{i,1\dots b^ic}$ to satisfying
              locations with these hash functions.
         iv. Use this mapping to build an OHTable containing the newly masked blocks:
             $[T_{i+1,u}]_{(1,2)} = \mathcal{F}_{Route}([Z_{i+1,ub^ic+1\dots(u+1)b^ic}], [Q]_0)$
     (b) $if(t = n)$ Refresh()
```

Figure 1: DORAM protocol overview, write function and rebuild function

This allows us to obtain secret-sharings of the masked value, but how can this be unmasked? P_0 knows which rune is masked using which OTP, but this information somehow needs to be accessed without revealing to P_0 which rune is being queried. This is the same problem we had with the location mapping, and it can be solved using the same solution! Since the Builder gets to *pick* the OTPs, he can pre-determine, at the initialization of the protocol, which OTPs it will use. He can then secret-share the OTPs that will be used *for all runes at all points in time.* Recall that each time a block is moved, it will be masked using a new OTP. Therefore, P_0 will secret-share a *mask schedule*, analogous to the position schedule, that contains the OTP used to mask each block at each point in time, and which can be accessed dynamically during reads to unmask blocks. This allows us to obtain a secret-sharing of the queried value, performing a read. The read protocol is presented formally in Figure 4.

While we say above that the Builder will pre-determine all runes, locations and ciphertexts at the initialization of the protocol, this is not quite true. A new rune is needed for each query, so an arbitrarily large number of queries would necessitate an arbitary large number of runes, which cannot be achieved with fixed memory and communication bounds. Instead, we initially set up the system to handle only n queries. It will therefore have $2n$ runes (n initial index runes, and n which are assigned during the queries). The Builder only predicts the future, so to speak, for the next n queries, and therefore only creates and shares schedules for locations and masks over n points in time. After n queries, the entire system will be refreshed.

We now describe the method for refreshing in more detail. 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 call 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, and may be called by the environment at an arbitrary time (i.e. when there have been fewer than n queries since the last refresh). The Refresh and Extract protocols are presented formally in Figure 3, while the Init protocol is presented in Figure 2.

6.4 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 $\mathcal{F}_{ABB}$, $\mathcal{F}_{SSPIR}$, $\mathcal{F}_{Route}$-hybrid model.

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

- A secure functionality is being accessed, that only outputs secret-shared results. This can either be a basic ABB functionality, like $\oplus$, or a more sophisticated functionality like **SSPIR**.
- The operations are on public, predetermined values (e.g. t, u).
- Some value is revealed to some party, or subset of the parties.

We need to examine all revealed values and examine whether they can be simulated without knowledge of the private inputs.

DORAM: Init

Init(n, d, [V]):

1. P_0 creates a random permutation which determines the assignment of runes, $[M]_0 : [1, 2n] \rightarrow [1, 2n]$
2. Assign the first n of these to be the original runes for the indices. Initialize a new sub-DORAM containing these runes (with adjacent pairs appended together into a single entry).
   (a) for $i \in [1, n]$, $[R_i]_0 = [M_i]_0$
   (b) for $i \in [1, n/2]$, $[B_i] = [M_{2i-1}]_0 || [M_{2i}]_0$
   (c) subDORAM $= \mathcal{F}_{DORAM}$.**Init**($\frac{n}{2}, 2(\lg(n) + 1), [B]$)
3. P_0 locally builds all the OHTables 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 OHTables, P_0 tells P_1 and P_2 to **abort** the protocol.
   Otherwise, P_0 can determine where each rune's block 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 block started to be in its i^{th} position.
   - $[F_{i,r}]_0$ contains the time rune r's block finished to be its i^{th} position.
   - $[P_{i,r}]_0$ contains the i^{th} position of rune r's block.
4. P_0 creates an mask-schedule. Note that the times will be the same as the position schedule. Therefore all that is needed is one addition matrix containing the OTPs:
   $[E_{i,r}]_0$ contains the OTP used to mask rune r's block 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: $[S]_{1,2}, [F]_{1,2}, [P]_{1,2}, [E]_{1,2}$.
6. P_0 provides the masks to the blocks, based on his previous selection: $[E_i]_0 = [E_{0,r}]_0$ for $M_i = r$.
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 OHTable containing the initial items, and P_1 and P_2 store the masked blocks:
   $[T_{\ell+1}]_{(1,2)} \leftarrow \mathcal{F}_{Route}([V] \oplus [E]_0, [Q]_0)$
9. The runes, values and indices of the initial items are stored for future reference. That is, for $i \in [1, n]$: $[R_{\ell+1,i}]_0 = [R_i]_0$
   $[V_{\ell+1,i}] = [V_i]$
   $[X_{\ell+1,i}] = [i]$
10. Initialize the query counter: $t = 1$.

Figure 2: DORAM: Init functionality

DORAM: Extract and Refresh

$[V] \leftarrow$ **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:
   $[R] = [R_0]_0||[R_1]_0 \dots ||[R_{\ell+1}]_0$, $[V] = [V_0]||[V_1] \dots [V_{\ell+1}]$, $[X] = [X_0]||[X_1] \dots [X_{\ell+1}]$
2. Let m (where $n \leq m \leq 2n$) be the length of these arrays.
3. P_1 picks a random permutation $S : [1, m] \rightarrow [1, m]$. 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, $U : [1, m] \rightarrow [1, m]$ 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 subDORAM.

Refresh():

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

Figure 3: Extract and Refresh functionalities

DORAM: Read

Read($[x]$):

1. Access the subDORAM to learn the rune of x. Note that indices are stored in the subDORAM in pairs, so the subDORAM will return a share of both x's rune and a share of x's neighbor's rune. The protocol reveals (only) x's rune to P_1 and P_2. Also, in order to access the subDORAM only once per query, the protocol takes the opportunity to use this access to also write the new rune that is being assigned to x.

   (a) Let $[x_{\ln(n)}]$ be the least significant bit of $[x]$ (i.e. if x is odd it is 1, otherwise 0).

   (b) Set $[x_{sig}]$ to be the $\lg(n) - 1$ most significant bits of $[x]$, (i.e. drop the last bit).

   (c) P_0 supplies the new rune $[r_{new}]_0$ which will be assigned to x when it is re-written.

   (d) We define f to overwrite x with its new rune, while leaving x's neighbor as is. Formally $f(v, y)$, $v \in \{0,1\}^{2(\lg n+1)}$, $y \in \{0,1\}^{\lg(n)+2}$ is defined such that if $y_0 = 0$ (which will happen when x is even) $f(v, y) = v_{1,\ldots,\lg(n)+1}||y_{1,\ldots,\lg(n)+1}$ (the second half of the value is overwritten with the remaining bits of y) and if $y_0 = 1$ (x is odd), $f(v, y) = y_{1,\ldots,\lg(n)+1}||v_{\lg(n)+2,\ldots,2\lg(n)+1}$ (the first half is overwritten).

   (e) $[v] \leftarrow$ subDORAM.**ReadWrite**($[x_{sig}]$, $[x_{\ln(n)}]||[r_{new}]$, f).

   (f) If $[y_0] = 1$, securely set $[r_{old}]$ to be the first half of $[v]$, otherwise securely set it to be the second half of $[v]$.

   (g) Reveal x's (old) rune to P_1 and P_2: $[r]_{(1,2)} \leftarrow [r_{old}]$.

   (h) 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 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 $i \in [1, \ell+1]$, $u \in [1, b-1]$, $k \in [1, h]$, 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.

3. Securely determine which time-slot is being used. That is, for $j \in [0, \ell+1]$:

   (a) Set $[S_j] \leftarrow [S_{j,[r]_{(1,2)}}]_{1,2} \geq t$

   (b) Set $[F_j] \leftarrow [F_{j,[r]_{(1,2)}}]_{1,2} < t$

   (c) Set $[J_j]_{1,2} \leftarrow [S_j] \wedge [F_j]$

4. Securely select the correct location and OTP from the position and mask schedules:

   (a) For $j \in [0, \ell+1], [P_j] \leftarrow [P_{j,[r]_{(1,2)}}]_{1,2}$

   (b) For $j \in [0, \ell+1], [E_j] \leftarrow [E_{j,[r]_{(1,2)}}]_{1,2}$

   (c) For $j \in [0, \ell+1]$, securely set $[p]$ to $[P_j]$ if $[J_j] = 1$

   (d) For $j \in [0, \ell+1]$, securely set $[e]$ to $[E_j]$ if $[J_j] = 1$

5. $[v] \leftarrow \mathcal{F}_{BalancedSSPIR}(c + \ell(b-1)h, d, [Y]_{(1,2)}, [p]) \oplus [e]$

6. Return $[v]$

Figure 4: DORAM read protocol

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).

It is revealed to P_1 and P_2 whether P_0 was able to successfully build all OHTables given his choice of the rune assignment. If P_0 is unable to, the protocol aborts, and no information is leaked as this event happens with fixed probability. If P_0 is able to build all OHTables, P_1 and P_2 learn only this fact, which again occurs with fixed probability. This leaks no information at this point, but has the potential to leak information later, as will be discussed.

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 OHTable was built successfully, can leak information. This will be analyzed in more detail.

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 OHTables 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 functions that was incompatible with that set being stored in a given OHTable. 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 was concerned with an adversary that 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 large set of elements there exists some subset of these that would result in a build failure. Speficially, 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. $\log(1/\epsilon) = \omega(\lg(n))$ Setting $h = \lg^{1.5}(n)/\lg(\lg(n)) = \omega(\lg(n))$ achieves this.

This indicates that, for any given OHTable, 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 OHTables 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 OHTable 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.

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.

6.5 Complexity Analysis

In this section, we show that the amortized communication complexity per access is $\Theta((\lg^2(n) + d)\lg(n)/\lg(\lg(n))$ bits. We assume that the cost of $\mathcal{F}_{BalancedSSPIR}$ is $\Theta(\sqrt(md) + d)$ and the cost of $\mathcal{F}_{Route}$ is $\Theta(q(d + \lg(q))$ as instantiated by our implementations in sections 8 and 9 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 subDORAM, which will be analyzed later, this involves only operations on runes, each of which requires at most $\Theta(\log(n))$ AND gates, or revealing $\Theta(\log(n))$ bits, so $\Theta(\log(n))$ communication.

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

3. The time slot is obtained. This requires $\Theta(\ell) = \Theta(\log(n)/\log(\log(n)))$ comparisons of $\Theta(\log(n))$-bit values, which requires $\Theta(\log^2(n)/\log(\log(n)))$ communication.

4. The correct position and mask is obtained. This requires $\Theta(\ell) = \Theta(\log(n)/\log(\log(n))$ secure if-then-else statements on $\Theta(\log(n))$-bit and $\Theta(d)$-bit values for the positions and masks respectively. The total is therefore $\Theta((\log(n) + d)\log(n)/\log(\log(n)))$ communication.

5. Finally the SSPIR is executed. The number of locations is $c+\ell(b-1)h = \Theta(\log^3(n)/\log^2(\log(n)))$. 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:

1. The first 3 steps are either local to P_0, or on public values, so require no communication

2. The masked block is created, required $\Theta(d)$ communication

3. The final steps consist only of re-labelling variables and operations on public values, so require no communication.

Therefore the communication cost of the write is $\Theta(d)$.

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 average 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 $\Theta(m(d + \lg(n))$ communication.

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

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 $\Theta(n\lg(n))$.

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.

7 The Constellation Attack

In our security analysis, rather than examining the probability that a build failure occurred in building an OHTable, we examined the probability that there existed any appropriately-sized subset of runes such that the OHTable build would fail. This may have seemed excessive, but in fact it is essential for security. In this section we show that failing to do so can lead to a subtle leakage attack.

It is possible that, given the hash functions, and the choice of runes, that it is impossible to construct an Oblivious Hash Table. Such build failures can leak information about the access pattern.

One option is to build an incorrect table when a failure occurs. This means that no information is leaked and the protocol is actually perfectly private. This causes the outputs of the protocol to be incorrect, albeit with negligible probability. Note that this problem only occurs when the build itself fails.

Alternatively, if a build fails, this can be reported. In this case, either the protocol can abort, or the build can be re-attempted with new hash functions. In this case, an abort by itself leaks no information. In fact, the abort would occur at the beginning of an initialization phase, before any data has been revealed, and has a fixed probability of occurring, depending on the hash functions and the assignment of runes. Similarly, if the build is re-attempted, at the time that this occurs, no information is leaked. The privacy issue actually occurs during queries. It is known that the contents of the OHTable are such that there is a satisfying assignment. Therefore, if it is observed that a certain sequence of queries result in hash functions that *do not have a satisfying assignment*, then it is clear that those items *cannot all be stored in the OHTable.*

Surprisingly, this creates a serious challenge when the number of queries to an OHTable is larger than its size. For instance, imagine there is an OHTable with m items. Now imagine the OHTable be queried some $q > m$ times. If there is *any* subset of size m of these q queries that has an access pattern that is incompatable with these m items being located in the table, then there is leakage. It is leaked that, for certain, this subset of items is not stored in the OHTable, for if they were an abort/rebuild would have occurred.

This means that a standard analysis of the build failure probability is insufficient. It is typical in most works on OHTables to show that the probability that a build failure occurs is negligible (in some parameter n) and conclude that the probability of leakage is therefore also negligible. If the number of queries is equal to the capacity of the OHTable this is true, as leakage will only occur if the queried items would have resulted in a build failure. However, in general it is not true. In particular, the technique of balancing hierarchical ORAM protocols allows there to be multiple OHTables per level, which means that an OHTable of capacity m may be queried by $m \lg(n)$ times. There are therefore $\binom{m \lg(n)}{m}$ subsets of the queried elements that are of size m, and it by no means follows that the probability of leakage remains negligible.

This is analogous to searching for constellations in a large clear night sky. If we wish to find m stars that are approximately in a given arrangement, it is unlikely that we will be able to do so if there are only m stars in the sky. However, if there are significantly more than m stars, and we are free to ignore stars in making our constellation, it is likely we can find some subset of m stars that create an image close to our desired arrangement.

It could be argued that this leakage is unlikely to be significant. In particular, for any desired single subset of size m the probability that that subset would be incompatable with the choice of hash functions is the same as the build failure probability. Nevertheless, some leakage does occur, which may be exploitable in certain cases.

In our case, the challenge is even greater. Since the hash functions are public, and the Holders eventually learn n different runes that were used, the Holders can locally calculate the result of the hash functions on n different runes. For an OHTable of capacity m, if any subset of size m of those n runes resulted in an incompatible assignment, a Holder can learn that those m runes were certainly not in that OHTable at that time. Nevertheless, our security analysis in section 6 demonstrates that over all subsets of size m of the $2n$ possible runes, there is a negligible probability that any subset would result in an OHTable build failure.

8 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 was used, 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.

The SSPIR protocol (figure 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 sufficient. 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. 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 $m = O(d)$.)

SSPIR

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

1. Convert $[x]$ to a XOR sharing in which P_0 holds one share and P_1 and P_2 both hold the other share:
   $[x]_{0,(1,2)} = (\langle x_1 \rangle_0, \langle x_2 \rangle_1, \langle x_2 \rangle_2) \leftarrow [x]$
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 $\lg(q) = \lceil \frac{\lg(m) - \lg(d)}{2} \rceil$. 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 $\ln(m) - \ln(q)$ bits, labelled $[y]$ and its lowest-order $\ln(q)$ 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 $i = [z], [v] = [u_{id \ldots id+i-1}]$.
6. Return $[v]$.

Figure 5: Implementation of SSPIR

9 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 for a formal definition of the functionality. Variants of this protocol have occurred before, for instance as the protocol Π_{SWITCH} in [MRR20]. We include the protocol here for clarity and completeness. The protocol is presented in figure 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 $\Theta(q \lg(q))$ 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 6: Secure Routing protocol

10 Conclusion and Future Work

This paper shows that the classic logarithmic overhead lower bound of ORAMs does not apply to the communication cost of DORAMs in the information theoretic setting. This begs the question: what is the lower bound in this setting? In particular is it possible to construct a DORAM with $o((d + \log^2(n)) \log(n)/\log(\log(n)))$ communication complexity? Also, does the lower-bound introduce trade-offs between the amount of communication and the number of memory locations accessed, the total memory utilized, the computation cost, the randomness consumed or the round complexity?

Another open question pertains to deamortization. This protocol deamortizes the costs of building OHTables, as is standard in the hierarchical model, but also deamortizes the cost of refreshes. While there are standard techniques for deamortizing the cost of building OHTables, it seems more challenging to deamortize the cost of refreshing, in particular the cost of refreshing the namespace for runes by reassigning runes to all indices. So the question remains: is it possible to have a DORAM with *worst-case* communication cost $\Theta((d+\log^2(n))\log(n)/\log(\log(n)))$?

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. This paper therefore also shows that logarithmic communication overhead can be avoided in the multi-server active ORAM setting, and $\Theta((\log^2 n+d)\log(n)/\log\log(n))$ 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. Concretely is it possible to have a single-server active ORAM that has sub-logarithmic communication overhead, but is information-theoretically secure?

Acknowledgements

This research was supported in part by DARPA under Cooperative Agreement HR0011-20-2-0025, the Algorand Centers of Excellence programme managed by Algorand Foundation, NSF grants CNS-2246355, CCF-2220450 and CNS-2001096, US-Israel BSF grant 2022370, Amazon Faculty Award, Cisco Research Award, Sunday Group, ONR grant (N00014-15-1-2750) “SynCrypt: Automated Synthesis of Cryptographic Constructions” and a gift from Ripple Labs, Inc. Any views, opinions, findings, conclusions or recommendations contained herein are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, the Department of Defense, the Algorand Foundation, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes, notwithstanding any copyright annotation therein.

References

[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.

[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.

[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.

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

[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.

[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.

[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.

[LO13] Steve Lu and Rafail Ostrovsky. Distributed oblivious ram for secure two-party computation. In *Theory of Cryptography Conference*, pages 377–396. 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.

[MCR21] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response efficient single-server PIR. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2292–2306, 2021.

[MG07] Carlos Aguilar Melchor and Philippe Gaborit. A lattice-based computationally-efficient private information retrieval protocol. Cryptology ePrint Archive, 2007.

[MG08] Carlos Aguilar Melchor and Philippe Gaborit. A fast private information retrieval protocol. In *2008 IEEE international symposium on information theory*, pages 1848–1852. IEEE, 2008.

[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.

[MW22] Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server PIR via FHE composition. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 930–947. IEEE, 2022.

[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.

[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.

[SvDS$^+$18] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. *Journal of the ACM (JACM)*, 65(4):1–26, 2018.

[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.

[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.

[Yeo23] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. *arXiv preprint arXiv:2306.11220*, 2023.