MetaDORAM: Info-Theoretic Distributed ORAM with Less Communication

Brett Hemenway Falk
University of Pennsylvania, Philadelphia, PA, USA

Daniel Noble[1]
Silence Laboratories, Singapore

Rafail Ostrovsky
UCLA, CA, USA

Abstract

A Distributed Oblivious RAM is a multi-party protocol that securely implements a RAM functionality on secret-shared inputs and outputs. This paper presents two information-theoretically secure DORAMs whose communication costs are asymptotic improvements over the state of the art. Let n be the number of memory locations and let d be the bit-length of each location.

The first, MetaDORAM1, is *statistically* secure, with $n^{-\omega(1)}$ leakage. It has amortized $O(\log_b(n)d + b\omega(1)\log(n) + \log^3(n)/\log(\log(n)))$ bits of communication per memory access. Here, $b \geq 2$ is a free parameter and $\omega(1)$ is any super-constant function (in n). The most communication-efficient prior statistically secure DORAM was that of Abraham et al (PKC 2017), which has cost $O(\log_b(n)d + b\omega(1)\log_b(n)\log^2(n))$. MetaDORAM1 is a $\Theta(\omega(1)\log(\log(n)))$-factor improvement over the work of Abraham et al whenever $d = O(\log^2(n))$.

The second protocol, MetaDORAM2, achieves *perfect* security. It has amortized communication cost $O(\log_b(n)d + b\log(n) + \log^3(n)/\log(\log(n)))$ where, again, $b \geq 2$ is a free parameter. The best prior perfectly secure DORAM is that of Chan et al (ASIACRYPT 2018) which has communication cost $O(\log(n)d + \log^3(n))$. MetaDORAM2 is therefore a $\Omega(\log(\log(n)))$-factor improvement over the DORAM of Chan et al under any parameter range (by setting $b = \log(n)$) and is a $\Theta(\log(n))$-factor improvement for $d = \Omega(n^\epsilon)$ for any constant $\epsilon > 0$ (by setting $b = d/\log(n)$). Our work is the first perfectly secure DORAM with sub-logarithmic communication overhead. MetaDORAM2 comes at the cost of a once-off (for any given n) setup phase which requires exponential (in n) computation.

Both DORAMs are in the 3-party setting with security against 1 semi-honest, static corruption. By a trivial transformation, these can be transformed, respectively, into statistically and perfectly secure active 3-server ORAM protocols secure against 1 corrupt server, with the same communication costs. These multi-server ORAM protocols are likewise asymptotic improvements over the state of the art.

2012 ACM Subject Classification Security and privacy → Information-theoretic techniques; Theory of computation → Communication complexity; Theory of computation → Cryptographic primitives; Security and privacy → Management and querying of encrypted data

Keywords and phrases ORAM, MPC, DORAM, multi-server ORAM, active ORAM

Digital Object Identifier 10.4230/LIPIcs.ITC.2025.6

Related Version *Full Version*: `https://eprint.iacr.org/2024/011.pdf` [32]

Funding 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

[1] Work done while at University of Pennsylvania.

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.

Acknowledgements Daniel Noble would also like to acknowledge God for supporting him in this research.

1 Introduction

Generic secure Multi-Party Computation (MPC) has historically been designed in the *circuit model* [40, 20, 6], in which the computation is represented as a circuit on bits or group elements. However, many computations are more naturally and efficiently implemented in the *RAM model*, in which it is possible to access locations in memory according to index variables. For instance, RAM is assumed in many classic algorithms and data structures, such as implementations of dictionaries, pointers, graphs and priority queues. An efficient implementation of a RAM functionality for MPC would therefore enable the adoption of generic efficient MPC.

Distributed Oblivious RAM (DORAM) is a functionality that implements RAM for MPC. It stores n d-bit blocks of data in a secret-shared memory, and allows accesses to that memory (reads or writes) at secret-shared locations. A DORAM is secure if the views of the parties can be efficiently simulated without knowledge of any private values. See section 5 for a complete definition of the DORAM functionality.

In this work we construct information-theoretically secure DORAMs in the 3-party honest-majority setting. Our results show new asymptotic results for DORAM as a category of problem, but are also significant because the 3-party honest-majority setting is prevalent in both research and practice. An honest majority is necessary to achieve information-theoretically secure generic MPC [6], so the 3-party honest-majorty setting is the minimal setting for achieving generic information-theoretic DORAM. Furthermore, there are certain techniques which are extremely efficient in the 3-party honest majority setting (e.g. [13, 29]). Due to these factors, the 3-party honest-majority setting has received particular attention from both academia [16, 2, 18, 24, 3] and industry [7, 23].

DORAM is closely related to the problem of Oblivious RAM [21], which solves the problem of a client outsourcing memory to an untrusted server (or servers). In particular a w-party DORAM can be converted into an ORAM with w active (i.e. computation-performing) servers and vice-versa, usually without increase in the communication cost. See section 2 for more details. Therefore, efficient DORAM is intrinsically tied to efficient multi-server active ORAM. A primary metric of this efficiency is the total amount of communication per memory access. This is often measured as the *overhead*, that is the number of blocks (of size d) of communication required per memory access.

Our Contribution. In this work, we present two novel DORAM protocols for the 3-party semi-honest honest-majority setting. The first, MetaDORAM1, achieves *statistical* security. That is, the statistical distance between the adversary's view and a simulated view is negligible in n. Unlike most statistically secure DORAMs, our protocol's security does not deteriorate with the number of accesses to the DORAM; the leakage remains negligible in n even as the number of accesses tends to infinity. The statistically secure protocol achieves amortized communication cost $\Theta(\log_b(n)d + b\omega(1)\log(n) + \log^3(n)/\log(\log(n)))$. Here $b \geq 2$ is a free parameter. The best prior work is that of Abraham et al (PKC 2017) [1] which has communication cost $\Theta(\log_b(n)d + b\omega(1)\log_b(n)\log^2(n))$. MetaDORAM1 and

Abraham et al [1] are asymptotically equivalent when $d = \omega(\log^{2+\epsilon}(n))$. However, when $d = O(\log^2(n))$, the communication cost of MetaDORAM1 is $\Theta(\log^3(n)/\log(\log(n)))$ (e.g. by setting $b = \log(n)/\log(\log(n)))$ whereas the work of Abraham et al has communication cost $\Theta(\omega(1)\log^3(n))$.

Our second DORAM protocol, MetaDORAM2 achieves *perfect* security. That is, the adversary's view is chosen from an identical distribution as the simulated view. MetaDORAM2 requires communication cost $\Theta(\log_b(n)d + b\log(n) + \log^3(n)/\log(\log(n)))$. The best prior work by Chan et al. (ASIACRYPT 2018) [10] has communication cost $\Theta(\log(n)d + \log^3(n))$. MetaDORAM2 is an asymptotic improvement over all parameter ranges. When $b = \log(n)$, MetaDORAM2 has communication cost $\Theta(\log(n)d/\log(\log(n)) + \log^3(n)/\log(\log(n)))$, that is a $\log(\log(n))$-factor improvement over Chan et al for all parameter ranges. Additionally, the free parameter b allows MetaDORAM2 to have improved performance for large d. Whenever $d = \Omega(n^\epsilon)$ for some constant $\epsilon > 0$, setting $b = d/\log(n)$ yields a protocol with communication cost $\Theta(d)$, that is the overhead is constant, which is a $\Theta(\log(n))$-factor improvement over the DORAM of Chan et al. MetaDORAM2 is the first perfectly secure DORAM protocol with sub-logarithmic overhead over any parameter range.

MetaDORAM2's perfect security comes at a cost in setup computation. MetaDORAM2 requires hash tables with well chosen hash functions. These hash functions must be chosen such that whichever m inputs are chosen (from a universe of size $2n$), it must be possible to build the hash table. Using $\Theta(\log(n))$ hash functions that map to disjoint spaces, causes this to occur with high probability [41]. The problem is that it is difficult to *verify* that a given choice of hash functions satisfies this property. As such, MetaDORAM2 achieves perfect security by first verifying that all subsets of size m from $\{1, \ldots, 2n\}$ can be successfully built into a hash tables of size $\Theta(m)$ using the chosen hash functions. Each verification takes $poly(n)$ time, but there are $\binom{2n}{m} < 2^{n\log(n)}$ subsets to verify. While this computation need only occur once for any given value of n, this limits the feasibility of this construction in practice to tiny n. Note that the setup phase, while requiring exponential *computation*, does not require any *communication*, so does not affect the amortized communication cost of the protocol. Therefore, MetaDORAM2 shows for the first time that the problem of perfectly-secure DORAM has sub-logarithmic communication overhead.

Note that in the non-uniform model of computation, the expensive setup phase can be avoided. The hash functions can simply be provided in the advice string, as they do not depend on the access pattern, but only on the size of the DORAM, n.

Due to the conversion between DORAMs and ORAMs, MetaDORAM1 (resp. MetaDORAM2) can be converted to a 3-server active ORAMs that is statistically (resp. perfectly) secure and has communication cost $\Theta(\log_b(n)d + b\omega(1)\log(n) + \log^3(n)/\log(\log(n)))$ (resp. $\Theta(\log_b(n)d + b\log(n) + \log^3(n)/\log(\log(n)))$).

It has been proven that an ORAM must access $\Omega(\log(n)d)$ bits of physical memory for each d-bit virtual memory access [21, 27]. (This lower bound is normally phrased as $\Omega(\log(n))$ *overhead*, that is $\Omega(\log(n))$ blocks each of size d bits.) This bound applies even if there are multiple servers [28, 25]. If the servers are passive (non-active) this implies a $\Omega(\log(n)d)$ bound on communication. In an active ORAM, the $\Omega(\log(n)d)$ bound applies to the number of bits accessed, but need not apply to the amount of communication. This work, like [1], achieves sub-logarithmic communication overhead in the information-theoretic setting. This result is slightly surprising: it means that server-side computation is useful at reducing the asymptotic communication complexity even without the introduction of computational assumptions. Our result, taken with [1], 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 7).

MetaDORAM1 and MetaDORAM2 also have lower communication overhead than most DORAM protocols that use computational assumptions. See Table 2 in section 2 for more details. MetaDORAM1 and MetaDORAM2 also have reasonable performance in other metrics. The computation and local memory access overheads are poly-logarithmic. The persistent memory usage is $\log_b(n)d \leq \log(n)d$ bits. The round complexity per query is $O(\log(n)\log(\log(n)))$. Throughout, the hidden constants in our asymptotic notation are small: most are single-digit and all are less than 20.

Organization. Our paper is organized as follows. Section 2 provides a short history of prior ORAM and 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-Shared 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. Section 7 concludes by discussing some interesting open questions. The functionalities for SSPIR and secure routing can be implemented using standard techniques. For completeness, these are presented explicitly in appendices A and B.

2 Prior Works

Many DORAMs are constructed from instances of a related, well-studied, primitive called Oblivious RAM (ORAM). ORAMs were first identified by Goldreich and Ostrovsky in the 1980s [19, 33]. Consider a program which is running in a secure environment with very limited memory. The program wishes to make use of general memory on a device, but an adversary may be able to observe access patterns on this device. For instance, the program may be running in a secure enclave, such as Intel SGX [12], and may need to protect sensitive information even if the operating system is corrupted. The program can encrypt the data; this will hide the data's contents, but will not hide the memory locations accessed by the program, which might leak sensitive information. An ORAM is an intermediary between the program and the main memory. It provides a RAM functionality to the program using the device's memory in such a way that the access pattern on the device (the *physical* access pattern) reveals no information about the memory accesses by the program (the *virtual* access pattern), except for the number of accesses.

In a normal RAM, the cost of retrieving a block of data from memory is equal to the size of the block, denoted d. Adding obliviousness comes at a price; the *overhead* is the multiplicative increase in the number of bits that need to be accessed relative to a normal RAM. Goldreich initially presented an ORAM that had $O(\sqrt{n}\log(n))$ overhead, where n is the number of blocks of memory [19], Ostrovsky constructed the first solution with poly-logarithmic overhead of $O(\log^3(n))$ [33, 34] (the so-called "Hierarchical" solution). A series of improvements reduced the overhead to $O(\log^2(n))$ [38, 22], $O(\log^2(n)/\log(\log(n)))$ [26, 9], $O(\log(n)\log(\log(n)))$ [36] and finally $O(\log(n))$ [4, 5] (i.e. accessing $O(\log(n)d)$ *bits* of memory). This final result matches the proven asymptotic lower bound of $\Omega(\log(n))$ [21, 27, 28, 25, 37]. These ORAMs consider the setting where the untrusted memory is passive (i.e. performs no computation on behalf of the ORAM) and the ORAM only has enough memory to store $\Theta(1)$ blocks (and $\Theta(1)$ κ-bit PRF keys).

ORAM was quickly recognized as a solution to another challenge: outsourcing memory in a network such as the internet. Here, a client with limited memory capacity wishes to store data on an untrusted server such that the access pattern on the server's memory leaks

no information about the actual memory access pattern by the client. The formalism of the original ORAM use-case is immediately applicable, with the client taking the place of the secure environment and the server replacing the device memory. However, this new application led several works to consider extensions of the model. First, the server is likely to also have significant computational resources, so may be able to perform computation to reduce communication overhead, a variant referred to as *active ORAM* [15, 11]. Second, some works observed that the client could easily interact with multiple servers, which under appropriate conditions could be assumed to not collude [31, 1, 10]. This was referred to as *multi-server ORAM*.

ORAM can be used to construct Distributed Oblivious RAM (DORAM). Any client-server ORAM can be transformed into a DORAM by evaluating the client's circuit inside of a secure computation and allowing one of the parties to act as a server, as was first observed in [35]. There is now no trusted client. Instead, there is a "virtual client" that is simulated by a secure computation. Furthermore, the extensions to the ORAM model that resulted from the memory outsourcing application are immediately relevant to DORAMs. Since DORAMs already have multiple non-colluding parties, they can trivially take advantage of multi-server ORAMs by having the parties act as different servers. Since the parties already perform computation as part of the MPC protocol, they can naturally perform the computation needed by servers in an active ORAM. In general, a s-server active ORAM tolerating t corrupt parties can be transformed into a (w, t)-secure DORAM protocol for any $w \geq s$ by simulating the client in a (w, t)-secure MPC and having each server's role taken by a distinct party. Note that it is also possible to convert a *DORAM* into a multi-server active ORAM by a trivial transformation. Any (w, t)-secure DORAM can also be converted to a (w, t)-secure multi-server active ORAM by each server acting as one party, and by the client initially secret-sharing their query to the servers, and the servers sending the client the shares to reconstruct the result.

While simulating an ORAM client in a MPC protocol is always possible, doing so may increase the communication costs of the protocol, or increase the number of parties required. If information-theoretic security is required, the MPC requires an honest majority so at least 3 parties are required even if the ORAM only uses 1 or 2 servers. Furthermore, any computation which was performed locally by the client, requiring no communication, must instead be evaluated within a MPC protocol. Depending on the computational complexity of the client, the communication cost of simulating the client may increase the total asymptotic communication costs. (In contrast, converting a DORAM to a multi-server ORAM will never increase the asymptotic communication, as it only requires 1 secret-sharing and 1 reconstruction.)

Table 1 presents the communication cost of the best prior information-theoretic secure DORAMs and of the DORAMs presented in this work. Note that while MetaDORAM2 requires a setup phase with exponential computation, this does not affect its communication complexity. Abraham et al. created an efficient 2-server ORAM using PIR [1]. Their ORAM is Tree-based, in which the number of children of each vertex is a configurable parameter, $b \geq 2$. This protocol achieved a communication cost of $\Theta(\log_b(n)d + b\omega(1)\log_b(n)\log^2(n))$. The parameter b should be set to reduce the amortized cost. For $d \geq 2\omega(1)\log^2(n)$ the cost is minimized by setting $b = \frac{d}{\omega(1)\log^2(n)}$, which results in a cost $\frac{\log(n)}{\log(d)}d$. For smaller d, the cost is minimal when $b = 2$. Note that while Abraham et al. constructed a 2-server ORAM, converting it to a DORAM while maintaining information-theoretic security requires a third party, to help simulate the virtual client. Chan et al. [10] created an efficient 3-server passive ORAM scheme with perfect security. Their DORAM is Hierarchical, but unlike most Hierarchical ORAMs which use PRFs, Chan et al. store items using truly random position

Table 1 Comparison with prior state of the art. Note that while the ORAM of Abraham et al. requires only 2 servers, converting it to an information-theoretic DORAM requires an honest-majority MPC protocol, therefore necessitating at least 3 parties. For statistically secure protocols, $\omega(1)$ is a super-constant function in n and the statistical leakage is $2^{-\omega(\log(n))}$.

Protocol	Amortized Communication Cost (bits)	Security
Abraham et al. [1]	$\Theta(\log_b(n)d + b\omega(1)\log_b(n)\log^2(n))$	Statistical
Chan et al. [10]	$\Theta(\log(n)d + \log^3(n))$	Perfect
MetaDORAM1 (this work)	$\Theta\left(\log_b(n)d + b\omega(1)\log(n) + \frac{\log^3(n)}{\log(\log(n))}\right)$	Statistical
MetaDORAM2 (this work)	$\Theta\left(\log_b(n)d + b\log(n) + \frac{\log^3(n)}{\log(\log(n))}\right)$	Perfect

Table 2 Complexity of select DORAM protocols. $\kappa = \omega(\log(n))$ is a cryptographic security parameter. $\omega(1)$ is any super-constant function in n and the statistical leakage is $2^{-\omega(\log(n))}$.

DORAM Protocol	Amortized Communication Cost (bits)	Security
Faber et al. [16]	$O\left(\omega(1)\log^2(n)d + \kappa\omega(1)\log^4(n)\right)$	Computational
Jarecki and Wei [24]	$O\left(\log(n)d + \kappa\log^3(n)\right)$	Computational
Bunn et al. [8]	$O\left((d+\kappa)\sqrt{n}\right)$	Computational
Lu et al. Falk et al. [31, 17]	$O\left(\log(n)d + \kappa\log(n)\right)$	Computational
DuORAM [39]	$O\left(\kappa \cdot d \cdot \log n\right)$	Computational
MetaDORAM1 (this work)	$\Theta\left(\log_b(n)d + b\omega(1)\log(n) + \frac{\log^3(n)}{\log(\log(n))}\right)$	Statistical
MetaDORAM2 (this work)	$\Theta\left(\log_b(n)d + b\log(n) + \frac{\log^3(n)}{\log(\log(n))}\right)$	Perfect

labels. These are stored in a position label ORAM, which is implemented recursively. The communication cost is $\Theta(\log(n)d + \log^3(n))$. Both prior works are multi-server ORAMs with simple clients, which can be converted to 3-party honest-majority DORAMs without increasing the amortized asymptotic communication cost.

Several works also investigated building Distributed Oblivious RAMs directly without simulating a client-server ORAM. While these works generally did not achieve the same asymptotic efficiency as [1] and [10], many had good concrete efficiency. They took advantage of the existence of multiple non-colluding servers by using Distributed Point Functions [8, 39], secret-shared PIR (SSPIR) [24] and secure shuffles/routing [17]. See Table 2 for details. These works all depended on computational assumptions. In comparison, the MetaDORAM protocols are information-theoretically secure. The MetaDORAM protocols also have strictly better asymptotic communication cost than these protocols over all parameter ranges, except for Lu et al. and Falk et al. [31, 17], which can have a lower asymptotic communication cost, and that only when $\kappa + d = o(\log^2(n)/\log(\log(n)))$.

3 Technical Overview

In this section we provide a broad overview of how the MetaDORAM1 and MetaDORAM2 protocols work. These protocols are, in fact, almost identical and only differ in how hash functions are chosen. Therefore, in this overview, we only discuss the common framework of the solution, which we refer to simply as MetaDORAM. Section 6 later presents MetaDORAM1 and MetaDORAM2 in full detail, including the different approaches used to select hash functions.

At a very high level, MetaDORAM uses Secret-Shared Private Information Retrieval to access items, and always writes the accessed item (whether modified or not) to a predetermined new location. It uses an oblivious "metadata map" that maintains a mapping from indices to their locations. This is accessed in order to obtain shares of the location to input to the Secret-Shared PIR.

A simple way to instantiate this is to simply store the original secret-shared array in memory, and to always write new items to the next free location in memory. The metadata map would then consist only of the last access time for each index. If the index has been accessed, this uniquely determines the item's current position; if the index has never been accessed, the position is determined by the index itself. This protocol is simple, but requires performing SSPIR over an array of size $\Omega(n)$.

In order to perform SSPIR over small arrays, we instead store items in hash tables, each with $h \in \Omega(\log(n))$ hash functions. As before, items are written to the next free location in memory, but when there are c such items, these too are built into a hash table. To limit the number of hash tables, when there are b tables of a given size, the contents of these tables are extracted and rebuilt into a single table. Let $T_{1,i}, \ldots, T_{b,i}$ be the tables of the i^{th} smallest size. Furthermore, every n accesses, there is a *DORAM refresh* in which the contents of all tables are extracted and rebuilt into a single hash table. As a result, there are only ever at most $O(\log_b(n)b)$ tables, so the SSPIR need only be performed over arrays of size $O(c + b\log_b(n)h) = poly(\log(n))$.

Restricting the number of locations accessed improves efficiency, but introduces several security and logistical challenges. Let $G(x)$ be a function returning a 3-D array, such that $G(x)_{i,u,k}$ is the k^{th} location accessed in table $T_{i,u}$ when querying x.

1. The first challenge is that $G(x)$ must leak no information about x (even to a computationally unbounded adversary). This is essential, because the parties who access data during a read will inevitably learn $G(x)$ by the locations they access.
2. Additionally, each time x is accessed, $G(x)$ must be a new 3-D array, independent of the previous value(s) of $G(x)$, to avoid leaking that repeated queries occurred. This implies $G(\cdot)$ must be stateful.
3. The parties need an efficient protocol for building hash tables which does not leak any correlation between indexes and the locations in which they are stored.
4. The SSPIR-index depends not only on the index and the last access time (as in the $\Omega(n)$-sized SSPIR solution presented above) but also on *which* of the h hash functions is being used to store the item, which was determined during the hash table build.

It is standard when a hash table satisfies properties (1) and (3) above, that it is called an *oblivious hash table*. Therefore, throughout all of the hash tables in this work are oblivious hash tables (OHTables).

We solve these by separating the required properties of stateful-updates, randomness and collision-resistance. We set $G(x)_{i,u,k} = H_{i,k}(Rand(UniqueNonce(x))$, where:

- $UniqueNonce$ is a stateful function. This function returns a different value for each x. Furthermore, each time x is queried by the DORAM, the value of $UniqueNonce(x)$ is updated. This is implemented simply by defining $UniqueNonce(x) = x$ for every x which has not been accessed (since the last DORAM refresh) and defining $UniqueNonce(x) = n + LastAccess(x)$ for each x which has been accessed. (Here $LastAccess(x) \in \{1, \ldots, n\}$ is the last time x was accessed, measured from the last DORAM refresh.)
- $Rand$ is a random permutation on $\{1, \ldots, 2n\}$.
- $H_{i,k}$ are public, fixed hash functions. They are collision-resistant, but do not need to have any other security properties.

We define $Rand(UniqueNonce(x))$ as the rune (Random Unique NoncE) of x (also written $rune(x)$). Note that when x is updated, it will be assigned a new rune.

This solution simultaneously resolves challenges (1) and (2) presented above. $rune(x)$ will be information-theoretic independent of x, so $G(x)$ will be as well, solving (1). Since $UniqueNonce(x)$ will return a distinct value each time an index is queried (between each DORAM refresh), and $Rand$ is a permutation, $rune(x)$ will also be a distinct value chosen uniformly at random (without replacement) from $\{1, \ldots, 2n\}$ for each queried x. Therefore, the locations distribution will be independent of the access pattern, and in particular independent of whether indexes are repeated in queries.

To solve challenge (3), we distinguish the roles of the parties. One party, P_0, will be the *Builder*. The other two parties, P_1 and P_2, will be *Holders*. The Holders will store the tables and will learn $rune(x)$ when x is queried, allowing them to compute $G(x)$. The Builder picks and stores $Rand$. Observe that when building a new table, the items which are being built have been accessed before, and so $UniqueNonce(x)$ depends only on the last access time of x. We write items to secret-shared memory each time they are accessed, according to a fixed schedule. Let $Mem(x)$ represent the location in secret-shared memory of item with index x at a given point in time. The mapping from $Mem(x)$ to $LastAccess(x)$ is therefore public, as is the mapping from $LastAccess(x)$ to $UniqueNonce(x)$. The Builder, knowing $Rand(\cdot)$ therefore also knows the mapping from $Mem(x)$ to $Rand(UniqueNonce(x)) = rune(x)$ *without knowing* x. The Builder can therefore locally compute $H_{i,k}(rune(x))$ and locally determines how the memory must be permuted in order to build the oblivious hash table. The Builder then provides this permutation to a secure routing protocol, resulting in a very efficient protocol for constructing oblivious hash tables.

Meanwhile the mapping from x to $rune(x)$ is stored in a smaller DORAM, which we refer to as the sub-DORAM, which is implemented recursively. During an access, this is queried and $rune(x)$ is provided to the Holders, who can use this to calculate $G(x)$, and determine which locations in the oblivious hash tables are to be used for the SSPIR. Note that the Builder knows $Rand$, but does not learn the runes obtained during queries, or the memory locations accessed by the Holders in the tables, so does not learn any information about the access pattern.

We turn to challenge (4). Note that the table, $T_{i,u}$, in which an item is stored is fully determined by the item's last access time. However, to perform SSPIR, we also need to obtain a secret-sharing of which of the h hash functions was used to store the item in this table. Observe that when the Builder is determining the permutation for building an oblivious hash table, this *does not depend* on the access pattern, but only depends on $Rand(\cdot)$. Furthermore, the Builder picks $Rand(\cdot)$ at the start of each DORAM refresh. Therefore, the Builder is able to pre-determine the permutations it will use, and therefore which hash function will be used for each rune in each oblivious hash table. This may be surprising, since the mapping from runes to indexes has not even be determined at this point. This is possible because, regardless of which index is accessed at time t, the item will be stored in the same initial location in memory, will be assigned the same unique nonce, the same rune, and will be placed in the same locations in each oblivious hash table. This means that the Builder can secret-share a *position schedule* which determines the SSPIR-index of the rune at each point in time (until the next DORAM refresh).

A final challenge is that SSPIR assumes that the 2 Holders have identical copies of the tables (rather than a secret-sharing of the tables). To provide privacy from the Holders, we therefore mask each item in each table with a rune-dependent information-theoretic mask (one-time pad). During each table rebuild, new masks are applied. This is achieved by the

Builder adding (as secret-shares) the masks for the new table, prior to the secure routing protocol. The Builder pre-determines all new masks at the start of each DORAM refresh, and secret-shares a mask schedule among the Holders, akin to the position schedule.

The metadata mapping therefore consists of mappings from indices to runes, from runes to a position schedule and from runes to a mask schedule. The position schedule and mask schedule are simply secret-shared arrays (with the runes public and the schedules secret-shared). The mapping from indices to runes is implemented using a sub-DORAM, which is implemented recursively.

Novel contributions. In addition to providing the communication-efficient DORAM with information-theoretic security, 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.

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. 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 $\{1, \ldots, a\}$. We use upper-case Latin characters to represent arrays and matrices, which are indexed using standard subscript notation. lg represents the base-2 log. For asymptotic annotation, any constant base is equivalent, in which case log represents 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. Our protocols are information-theoretically secure, that is $\mathcal{A}$ may perform an arbitrarily large amount of computation.

We utilize hash functions. Our hash functions are fixed and public. 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.

Table 3 Types of Secret-Sharing with Notation.

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)	$x_0 \oplus x_1 \oplus x_2 = x$
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	

We use several kinds of secret-sharing, all of which are bit-wise (Boolean) secret-sharings. These are summarized in Table 3. Since all of these sharings are linear, they 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 of them need send a sharing.) This requires only $\Theta(l)$ communication.

The most common sharing we use is a 3-party replicated secret sharing (3RSS) [2]. 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 $x_0 \oplus x_1 \oplus x_2 = x$. P_i holds x_i and $x_{(i+1) \bmod 3}$. When variable x is held using this secret-sharing, it is represented as $[x]$.

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 $[x]_{1,2}$. Here P_1 holds x_0 and P_2 holds x_1 where x_0 and x_1 are randomly chosen subject to $x_0 \oplus x_1 = x$. 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 $[x]_{0,(1,2)}$, 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 $[x]_{(1,2)}$.

For conciseness, conversions between types of secret-sharing are typically implicit in our pseudocode, indicated by the sharing-type of the result. For instance, $[C_j]_{(1,2)} = [v_{new}] \oplus [e]$ means that $[v_{new}]$ and $[e]$, 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 C_j.

We use the Arithmetic Black Box (ABB) model [14] to formalize the guarantees provided by secret-sharing and operations on secret-shares. This treats 3RSS, 2XORS and 1-2XORS secret-shared values as stored in a reactive functionality $\mathcal{F}_{ABB}$. $\mathcal{F}_{ABB}$ can also perform operations on secret-shares (e.g. AND, XOR) with the result again being stored by $\mathcal{F}_{ABB}$. Only when a $\mathcal{F}_{ABB}$-stored value is converted to a private (2-Priv or 1-Priv) or public value (corresponding to share reconstruction) is that value released by $\mathcal{F}_{ABB}$ to the appropriate parties. The protocol of Araki et al. [2] securely implements $\mathcal{F}_{ABB}$ for any Boolean operation (AND, OR, NOT, XOR) over 3RSS-shared values. Locally XORing shares securely implements $\mathcal{F}_{ABB}$ for the XOR operation over 2XORS and 1-2XORS sharings.

5 Functionality

We wish to implement the following DORAM functionality:

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

$\boldsymbol{I} \leftarrow$ **Init(n, d, [A])**: Store array A containing n items of size d.
$[v] \leftarrow \boldsymbol{I}$**.ReadWrite([x], [y], f)**: Given an index $x \in [1, n]$, set v to A_x. Set $A_x = f([v], [y])$.
$[A] \leftarrow \boldsymbol{I}$**.Extract()**: Return the current state of the memory, A, as an array of secret-shares.

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.

Security is defined using the simulation paradigm, which is standard for proving the security of MPC protocols [30]. A simulator, given only a party's inputs and outputs from a protocol must generate a view consistent with the real view of a corrupted party during an execution. The DORAM is *perfectly* secure if the simulated view is from the same distribution as the real view. It is *statistically* secure if the distance between the distributions of the views is negligible in n. A protocol has *information-theoretic* security if it has either perfect or statistical security. It is *computationally* secure if a computationally-bounded adversary has a negligible advantage in distinguishing views in the simulated and real executions. All of our protocols have information-theoretic security. We present two DORAMs, one that is statistically secure and another that is perfectly secure. The only difference between the two protocols is the choice of hash functions: the perfectly secure protocol selects hash functions which allow for the construction of oblivious hash tables on all possible inputs, whereas the statistically secure protocol picks random hash functions which allow for the construction of oblivious hash tables on all possible inputs except with negligible probability. This is the only type of leakage in the statistically secure protocol. Apart from this all components of both protocols are perfectly secure.

Our DORAM implementation makes use of the following functionalities. These can be implemented with perfect security using standard techniques. We present explicit instantiations of the SSPIR and routing functionalities in appendices A and B respectively. The communication cost of **SSPIR** is $\Theta(\sqrt{md} + d)$ while that of **Route** is $\Theta((d + \log(q))q)$.

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 MetaDORAM1 and MetaDORAM2 in full and analyzes their security and communication costs. Since MetaDORAM1 and MetaDORAM2 have only minor differences, we first present the generic protocol, which we refer to as MetaDORAM, which does not specify how hash functions are chosen. We then show how the hash functions can be chosen to either provide statistical security or perfect security. For reference, the reader can also refer to the high-level technical overview of the MetaDORAM protocol from section 3.

The MetaDORAM protocol is presented in detail in sections 6.2 and 6.3. Section 6.4 then shows how a statistically secure DORAM (MetaDORAM1) and a perfectly secure DORAM (MetaDORAM2) can be instantiated depending on how the hash functions are chosen, and proves that these protocols achieves the desired security properties. Finally section 6.5

analyzes the complexity of these protocols. We assume the existence of functionalities for secret-shared PIR and secure routing, implementations of which are presented in appendices A and B respectively.

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 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 c. When the cache becomes full, its contents are extracted and built into an OHTable.

We implement the OHTables using cuckoo hashing with many ($h \in \Omega(\log(n))$) 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 mask the blocks using fresh masks during each build. All masks are achieved information-theoretically using one-time pads (OTPs), which are picked by the Builder.

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 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 using 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, their contents 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. 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. To simplify the OHTable builds, we store an append-only secret-shared array, $[V]$, containing all values written (including for reads) since the last refresh. We use subsets of *this array* and *not the existing OHTables* to build new OHTables, which avoids the complexity of actually extracting the contents from existing OHTables and removing previous masks. 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 $\mathbf{Read}([x])$ 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 $c + \ell(b-1)h$.

DORAM: ReadWrite Write Rebuild

Parameters:

- Cache size: $c = b \cdot h$
- Tables per level: b (Configurable parameter)
- Number of levels: $\ell = \lceil \log_b(n/c) \rceil$
- Number of hash functions: h (Configurable parameter)
- Hash functions: $H_1, \ldots, H_h$

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 sets $[r]_0 = [R_{n+t}]_0$. This will be a rune from $[1, 2n]$ which has not been used since the past refresh, and is chosen at random with replacement. (See **Init** in Figure 2.)
2. $j = t \mod c$
3. P_0 sets $[e]_0 = [E_{1,[r]_0}]_0$. That is, $[e]_0$ is a OTP from $\{0,1\}^d$, and P_0 selects it consistantly with the mask schedule it generated during **Init** (Figure 2).
4. $[C_j]_{(1,2)} = [v_{new}] \oplus [e]$
5. For future rebuilds and refreshes, the secret-shared v_{new} and x are stored in append-only arrays:
   $[V_{n+t}] = [v_{new}]$
   $[X_{n+t}] = [x]$

Rebuild():

1. if $t = 0 \mod c$:
   a. Set i to the largest value such that $t = 0 \mod cb^i$.
   b. Set $u = (t/(b^i c)) \mod b^{i+1}c$ (the number of tables in L_{i+1}).
   c. The parties obtain the runes, values and (new) masks for the cb^i most-recently written items and build these into a new table.
      for $j \in [1, b^i c]$ (in parallel):
      i. P_0 sets $[r]_0 = [R_{n+t-c \cdot b^i+j}]_0$
      ii. P_0 sets $[e]_0$ to the new mask for rune r: $[e]_0 = [E_{i+1,[r]_0}]_0$.
      iii. $[Z_j] = [V_{n+t-c \cdot b^i+j}] \oplus [e]$.
   d. P_0 locally builds an OHTable using $R_{i,1\ldots b^i c}$, and hash functions $H_1, \ldots, H_k$. Let $[Q]_0$ be the injective mapping from $[1, b^i c]$ to $[1, 2(1+\epsilon)b^i c]$ that maps $R_{i,1\ldots b^i c}$ to satisfying locations with these hash functions.
   e. Use this mapping to build an OHTable containing the newly masked blocks:
      $[T_{i+1,u}]_{(1,2)} = \mathcal{F}_{Route}([Z], [Q]_0)$
   f. Delete the old cache C and old tables $T_{1,1}, \ldots, T_{i,b-1}$.
2. if $(t = n)$ **Refresh**()

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

These locations constitute the array for the SSPIR; the protocol now needs to obtain secret-shares of the desired item's location in this array. 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 predetermine 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 exactly 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 protocol we use is explained in more detail in appendix A.

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.

Although we say above that the Builder will pre-determine all runes, locations and masks at the initialization of the protocol, in fact they will only pre-determine these for the first n queries so that the position schedule and mask schedule are not too large. The protocol will therefore have $2n$ runes (n initial index runes, and n which are assigned during the queries). The Builder only predetermines the assignment of runes and movement of blocks, 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 DORAM is refreshed and the Builder generates new runes, position schedules and mask schedules for the next n queries. We stress that the Builder pre-determines the mapping from runes to access times and does *not* pre-determine the mapping from runes to indices. The mapping from access times to indices (and therefore runes to indices) is only determined during the execution of the DORAM as queries occur.

DORAM: Init

Init(n, d, [V]):

1. P_0 creates a random permutation which determines the assignment of runes, $[R]_0 : [1, 2n] \to [1, 2n]$. The first n of these are the initial runes of the indices. $[R_{n+t}]_0$ is the rune which will be assigned to the index that is accessed at time t.
2. Initialize a new sub-DORAM containing these runes (with adjacent pairs appended together into a single entry).
   a. for $i \in [1, n/2]$, $[B_i] = [R_{2i-1}]_0 || [R_{2i}]_0$
   b. subDORAM $= \mathcal{F}_{DORAM}.\mathbf{Init}(\frac{n}{2}, 2(\lg(n) + 1), [B])$
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 starts to be in its i^{th} position.
   - $[F_{i,r}]_0$ contains the time rune r's block finishes to be its i^{th} position.
   - $[P_{i,r}]_0$ contains the i^{th} position of rune r's block.
4. P_0 creates a 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 $R_i = r$.
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 values of the initial items are stored for future reference in $[V_{1,\ldots,n}]$.
10. The indexes are stored in an array $[X]$, that is for $i \in [1, n]$:
    $[X_i] = [i]$
11. Initialize the query counter: $t = 1$.

Figure 2 DORAM: Init functionality.

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.

DORAM: Extract and Refresh

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

1. Set $m = n + t$. Observe that $[V]$ and $[X]$ are both of length m, due to the initial n values stored during the refresh, and the t writes which have occured since.
2. Set $[R] = [R_{1,\ldots,m}]$. This remove any runes from $[R]$ which have not been assigned to an index since the last refresh.
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 subDORAM.

Refresh():

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

Figure 3 Extract and Refresh functionalities.

6.4 Security Analysis

In this section, we show how to instantiate the MetaDORAM protocol so as to achieve statistical security (MetaDORAM1) and perfect security (MetaDORAM2). In short, MetaDORAM1 picks hash functions at random, and is secure in the event (which occurs except with negligible probability) that the hash functions are suitably chosen. MetaDORAM2 instead manually verifies that the hash functions are suitably chosen. We first prove the security of the protocol in the event that the hash functions are suitably chosen.

▶ **Theorem 1.** *Let $H_1, \ldots, H_h$ satisfy the property that for all $i \in [0, l]$, for $m = cb^i$ and for all subsets M of size m of $[1, 2n]$, there exists an assignment $a_1, \ldots, a_m \in [1, h]^m$ such that $H_{a_i}(M_i) \bmod 2(1 + \epsilon)m$ are distinct. In this case, the MetaDORAM protocol, as presented in Figures 1, 2, 3 and 4 is perfectly secure in the $\mathcal{F}_{ABB}$, $\mathcal{F}_{SSPIR}$, $\mathcal{F}_{Route}$-hybrid model.*

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. $[v_{sub}] \leftarrow$ 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 $[v_{sub}]$, otherwise securely set it to be the second half of $[v_{sub}]$.
   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.

Proof. 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.

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). In the case that $H_1, \ldots, H_h$ have satisfying assignments for all subsets of size m of $[1, 2n]$, then P_0 will never abort. Therefore, the only information P_1 and P_2 learn during the Init function are the values $T_{\ell+1}$. 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.

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, such that $X_i = x$ and $R_i \notin D$. 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 after it has been permuted. 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 views of all parties are perfectly simulatable, so the protocol is secure in the semi-honest setting against an adversary that corrupts any one of the parties. ◀

In the case that $H_1, \ldots, H_h$ are such that there is no satisfying assignment for some subset of $[1, 2n]$ of some size $m = cb^i$, then there is some small leakage. In the case that this subset is chosen, this leads to an abort, so does not leak any information. However, if such a subset is not chosen, P_1 and P_2 learn that such a subset was not chosen. Since P_1 and P_2 learn the rune of items when they are accessed, they can therefore conclude that certain access patterns were impossible, as they would have led to tables that were unconstructable. Note that this type of leakage can occur even if the probability of P_0 actually choosing a rune assignment that would lead to an abort is negligible: P_1 and P_2 could learn of some access pattern which for certain did not occur.

We present two solutions to this problem. MetaDORAM1 selects $\omega(1)\log(n)$ hash functions at random, for any super-constant function $\omega(1)$. We show that, except with negligible probability in n, this results in a choice of hash functions which have a satisfying assignment for all subsets of size m of $[1, 2n]$, for all $m = cb^i$. This results in a protocol which has a negligible probability of any leakage, and is therefore statistically secure.

In MetaDORAM2, the protocol instead selects $\Theta(\log(n))$ hash functions at random and manually verifies that this choice of hash functions result in a satisfying assignment for all subsets of size m of $[1, 2n]$, for all $m = cb^i$. The verification stage requires an exponential time setup phase (which need only be done once for any value n). This allows for a perfectly secure protocol.

We prove both protocols secure by making use of Yeo's analysis of Robust Cuckoo Hashing [41]. 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. Specifically, from his proof of Lemma 3 we can derive the following:

▶ **Lemma 2** (Derived from proof of [41] Lemma 3). *For some $m \leq 2n$, let C be a disjoint-table cuckoo hash table with αm locations ($\alpha \geq 1$), and h random hash functions $H_1, \ldots, H_h$. Furthermore, each location in C is of capacity $l = 1$ and C does not have a stash ($s = 0$). Then all subsets of $[1, 2n]$ of size m can be successfully built by C, except with probability:*

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

Note that this probability does not depend on m, except for requiring that $m \leq 2n$. For $h = \lg(n) + 5$, this simplifies to

$$\left(\frac{1}{2}\right)^{\lg(n)+6} = \frac{1}{64n}$$

For any $h = \omega(\log(n))$ this is negligible in n.

6.4.1 MetaDORAM1

For MetaDORAM1, we set $h = \lg^{1.5}(n)/\lg(\lg(n)) = \omega(\log(n))$. We select h independent random hash functions. By Lemma 2, this means that the failure probability is negligible in n. Note that this gives the failure probability for a given m, but as there are fewer than n such values of m to consider (even given the recursive implementation of the sub-ORAM) the probability that there is any m for which a subset of size m could not have a satisying assignment is also negligible. Therefore MetaDORAM1 is perfectly secure, except in the case of an event (poorly chosen hash functions) which occurs with probability negligible in n. This leads to our desired result:

▶ **Corollary 3.** *MetaDORAM1 is a statistically secure implementation of functionality $\mathcal{F}_{DORAM}$ in the $\mathcal{F}_{ABB}$, $\mathcal{F}_{SSPIR}$, $\mathcal{F}_{Route}$-hybrid model.*

Note that the subDORAMs, even though they have smaller sizes, use the same parameter h as the top level, so that the failure probability remains negligible in the size of the top DORAM, n.

6.4.2 MetaDORAM2

For MetaDORAM2, we set $h = \lg(n) + 5$. This means that the choice of hash functions satisfies all subsets of a given size m with probability $\frac{1}{64n}$. Therefore, it also satisfies all subsets for all $m \leq n$ except with probability at most $\frac{1}{64}$. The protocol selects a random $H_1, \ldots, H_h$ and then attempts to build the hash tables using all subsets of $[1, 2n]$ of size m,

for all $h \leq m \leq n$. If any subset does not have a satisfying assignment, new random hash functions are selected and the process is repeated. If all subsets have a satisfying assignment, these hash functions are used for the protocol.

Unfortunately, iterating over all subsets of $[1, 2n]$ of size m requires $\binom{2n}{m}$ iterations. Further iterating over all $m \in [h, n]$ results in nearly 2^{2n-1} iterations. This exponential-time setup phase makes the protocol infeasible for practical applications. Nevertheless, it does not affect the communication cost of the protocol, nor does it undermine its perfect security.

By thus choosing the hash functions, the condition of Theorem 1 is satisfied:

▶ **Corollary 4.** *MetaDORAM2 is a perfectly secure implementation of functionality $\mathcal{F}_{DORAM}$ in the $\mathcal{F}_{ABB}$, $\mathcal{F}_{SSPIR}$, $\mathcal{F}_{Route}$-hybrid model.*

6.5 Complexity Analysis

The complexity analysis is presented in the full version of the paper [32].

7 Future Work

A discussion of future work is included in the full version [32].

References

1 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. doi:10.1007/978-3-662-54365-8_5.

2 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. doi:10.1145/2976749.2978331.

3 Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 610–629. ACM, 2021. doi:10.1145/3460120.3484560.

4 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. doi:10.1007/978-3-030-45724-2_14.

5 Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. Oblivious ram with worst-case logarithmic overhead. *Journal of Cryptology*, 36(2):7, 2023. doi:10.1007/s00145-023-09447-5.

6 Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Janos Simon, editor, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10. ACM, 1988. doi:10.1145/62212.62213.

7 Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, Springer, 2008. doi:10.1007/978-3-540-88313-5_13.

8 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. doi:10.1007/978-3-030-57990-6_11.

9 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. doi:10.1007/978-3-319-70694-8_23.

10 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. doi:10.1007/978-3-030-03332-3_7.

11 Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring oram: Efficient constant bandwidth oblivious ram from (leveled) tfhe. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 345–360, 2019. doi:10.1145/3319535.3354226.

12 Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016. URL: http://eprint.iacr.org/2016/086.

13 Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362. Springer, Springer, 2005. doi:10.1007/978-3-540-30576-7_19.

14 Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *Annual international cryptology conference*, pages 247–264. Springer, 2003. doi:10.1007/978-3-540-45146-4_15.

15 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. doi:10.1007/978-3-662-49099-0_6.

16 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. doi:10.1007/978-3-662-48797-6_16.

17 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. doi:10.1007/978-3-031-14791-3_19.

18 Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 225–255. Springer, 2017. doi:10.1007/978-3-319-56614-6_8.

19 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. doi:10.1145/28395.28416.

20 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. ACM, 2019. doi:10.1145/3335741.3335755.

21 Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996. doi:10.1145/233551.233553.

22 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. doi:10.1007/978-3-642-22012-8_46.

23 Yuval Ishai, Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. Private large-scale databases with distributed searchable symmetric encryption. In Kazue Sako, editor, *Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings*, volume 9610 of *Lecture Notes in Computer Science*, pages 90–107. Springer, Springer, 2016. doi:10.1007/978-3-319-29485-8_6.

24 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. doi:10.1007/978-3-319-93387-0_19.

25 Ilan Komargodski and Wei-Kai Lin. A logarithmic lower bound for oblivious ram (for all parameters). In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part IV 41*, pages 579–609. Springer, 2021. doi:10.1007/978-3-030-84259-8_20.

26 Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 143–156. SIAM, 2012. doi:10.1137/1.9781611973099.13.

27 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. doi:10.1007/978-3-319-96881-0_18.

28 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. doi:10.1007/978-3-030-64375-1_17.

29 Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-efficient oblivious database manipulation. In Xuejia Lai, Jianying Zhou, and Hui Li, editors, *Information Security, 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings*, volume 7001 of *Lecture Notes in Computer Science*, pages 262–277. Springer, Springer, 2011. doi:10.1007/978-3-642-24861-0_18.

30 Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. *Electron. Colloquium Comput. Complex.*, TR17-112:277–346, 2017. URL: https://eccc.weizmann.ac.il/report/2017/112.

31 Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography Conference*, pages 377–396. Springer, 2013. doi:10.1007/978-3-642-36594-2_22.

32 Daniel Noble, Brett Hemenway Falk, and Rafail Ostrovsky. MetaDORAM: Info-theoretic distributed ORAM with less communication. *IACR Cryptol. ePrint Arch.*, page 11, 2024. URL: https://eprint.iacr.org/2024/011.

33 Rafail Ostrovsky. An efficient software protection scheme. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*. Springer, 1989. doi:10.1007/0-387-34805-0_55.

34 Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 514–523, 1990. doi:10.1145/100216.100289.

35 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. doi:10.1145/258533.258606.

36 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. doi:10.1109/FOCS.2018.00087.

37 Giuseppe Persiano and Kevin Yeo. Limits of breach-resistant and snapshot-oblivious rams. In *Annual International Cryptology Conference*, pages 161–196. Springer, 2023. doi:10.1007/978-3-031-38551-3_6.

38 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. doi:10.1007/978-3-642-14623-7_27.

39 Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A bandwidth-efficient distributed ORAM for 2- and 3-party computation. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 3907–3924. USENIX Association, 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/vadapalli.

40 Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society, 1982. doi:10.1109/SFCS.1982.38.

41 Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 197–230. Springer, 2023. doi:10.1007/978-3-031-38551-3_7.

A Secret-Shared Private Information Retrieval

The full version [32] presents an instantiation of the SSPIR protocol, using standard techniques.

B Secure Routing

The full version [32] presents an instantiation of the secure routing protocol using standard techniques.