MetaDORAM: Info-Theoretic Distributed ORAM with Less Communication

Brett Hemenway Falk[1] Daniel Noble*[2] Rafail Ostrovsky[3]

[1] University of Pennslvania, fbrett@seas.upenn.edu
[2] Silence Laboratories, danielnoble@proton.me
[3] UCLA, rafail@cs.ucla.edu

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 DORAMs in the semi-honest honest-majority 3-party setting which are information-theoretically secure and whose communication costs are asymptotic improvements over previous work. 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 $O(\log_b(n)d + b\omega(1)\log(n) + \log^3(n)/\log(\log(n)))$ 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 best prior work 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 an asymptotic improvement over the work of Abraham et al whenever $d = O(\log^2(n))$.

The second protocol, MetaDORAM2, achieves *perfect* security, albeit at the cost of a computationally-expensive setup phase. It has communication cost $O(\log_b(n)d + b\log(n) + \log^3(n)/\log(\log(n)))$. The best prior work of Chan et al (ASIACRYPT 2018) has communication cost $O(\log(n)d + \log^3(n))$. When $b = \log(n)$, the communication cost of our protocol is $O(\log(n)d/\log(\log(n)) + \log^3(n)/\log(\log(n)))$, that is a $\Theta(\log(\log(n)))$ factor improvement over that of Chan et al. Our work is the first perfectly secure DORAM with sub-logarithmic communication overhead. This comes at the cost of a once-off (for any given n) setup phase which requires exponential (in n) computation.

By a trivial transformation, these can be transformed, respectively, into statistically and perfectly secure active multi-server ORAM protocols with the same communication costs. These multi-server ORAM protocols are likewise asymptotic improvements over the state of the art.

1 Introduction

Secure Multi-Party Computation (MPC) protocols allow a set of parties in a network, of which some unknown subset are dishonest, to securely simulate a trusted third party. This holds tremendous promise. It allows statistical analysis

* Work done while at University of Pennsylvania

of sensitive data from different sources without pooling data with any single party. It allows the creation of secure systems, which do not have a single point of failure; using diversification these systems can therefore tolerate attacks affecting particular operating systems, hardware or local networks. In short, it allows sensitive data to be combined and used without needing to trust any single entity or device with that data.

MPC use cases to date, though, have mostly been restricted to tailor-made protocols for specific applications, such as Private Set Intersection [48], Threshold Signature Schemes [18] and Machine Learning [33]. It was shown in the 1980's that arbitrary circuits could be evaluated securely [56, 25, 7]. However, many computations cannot be represented efficiently as circuits. Rather, it is often more natural and efficient to represent a computation in the RAM model. 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 target the 3-party honest-majority setting. An honest majority is necessary to achieve information-theoretically secure generic MPC [7] and even with computational assumptions, honest-majority MPC significantly out-performs dishonest-majority MPC. The 3-party honest-majority setting in particular has received particular attention from both academia [19, 2, 22, 31, 3] and industry [8]. This is both because it is the smallest (and therefore easiest to set up) instantiation of honest-majority MPC and also because of certain techniques which are extremely efficient in this setting (e.g. [14, 38]) but do not scale well to larger numbers of parties.

DORAM is closely related to the problem of Oblivious RAM, 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) [11] 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. 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 uses hash functions which should not only allow for a successful hash table build for some given input, but should allow for a successful hash table build on all possible inputs (chosen from a universe of size $2n$). Using $\Theta(\log(n))$ hash functions that map to disjoint spaces, causes this to occur with high probability [57]. 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 (say of size m) of $\{1, \ldots, 2n\}$ 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. Nevertheless, it clearly shows that the problem of perfectly-secure DORAM has sub-logarithmic communication overhead.

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. $\Theta(\log_b(n)d + b\log(n) + \log^3(n)/\log(\log(n)))$).

Note that a passive (non-active) ORAM has a lower bound [26, 36] of $\Omega(\log(n))$ overhead in the number of memory accesses, and therefore communication cost, even if there are multiple servers [37, 34]. In an active ORAM, the $\Omega(\log(n))$ bound applies to the number of memory accesses, but need not apply to the amount of communication. This work, like [1], achieves sub-logarithmic com-

munication 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 overhead is $\log_b(n) \leq \log(n)$. The round complexity per query is $O(\log(n)\log(\log(n)))$. The constants in all of the asymptotic notation are very small.

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

Distributed Oblivious RAM is closely related to the problem of Oblivious RAM (ORAM), which was first formulated by Goldreich in the 1980's [24]. Imagine 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 [13], but needs to hide 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 [24]. This was improved to $O(\log^3(n))$ by Ostrovsky [42], using the hierarchical approach, which we explain in section 3. A series of improvements reduced this first to $O(\log^2(n))$ [47, 27], then $O(\log^2(n)/\log(\log(n)))$ [35, 10], then $O(\log(n)\log(\log(n)))$ [44] and finally $O(\log(n))$ [4, 5]. This final result matches the proven asymptotic lower bound of $\Omega(\log(n))$ [26, 36, 37, 34, 46]. This asymptotic lower bound is in 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 protocols have generally followed one of two approaches. The first is the *Hierarchical* approach, initially proposed by Ostrovsky [42], 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. [50] proposed the alternative *Tree* approach, which was extended by many subsequent (D)ORAMs (e.g. [51, 54, 19]). 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 are 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.

Both Hierarchical and Tree ORAMs can benefit from "balancing" [35]. In Hierarchical ORAMs, the number of tables $b \geq 2$ that should exist before these are combined into a single table is a configurable parameter. Similarly, in a Tree ORAM the number of children of each node is also a free parameter. In both cases, these parameters can be chosen to balance the cost of accesses and the cost of table rebuilds (in the Hierarchical setting) or leafward evictions (in the Tree setting).

With the advancement of networking infrastructure in the 1990's, ORAM was quickly recognized as a solution to another challenge: outsourcing memory in a network. Here, a client with limited memory capacity wishes to store data on an untrusted server such that the access pattern on the servers' memory leaks no information about the actual memory access pattern by the client. The formalism of the original ORAM use-case was immediately applicable, with the client taking the place of the secure environment and the server replacing the device memory. However, this new application led to various extensions of the model. First, the client could feasibly store much more than $\Theta(d + \kappa)$ bits of memory–a laptop or modern smartphone has gigabytes of memory available. Second, 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*. Third, the client could easily interact with multiple servers, which could reasonably be assumed to not collude. This was referred to as *multi-server ORAM*. This new application also caused metrics to be re-evaluated. Since latency is higher in a network, the number of rounds of execution between the client and server became a very important efficiency metric. Additionally, this setting often involved much larger blocks: if the communication cost had a term that did not depend on d, this term could become asymptotically irrelevant for sufficiently large d. A large number of works arose examining various combinations of these new models of super-constant client memory overhead (e.g. [55, 52, 49, 45, 6]), active servers (e.g. [17, 12]) and multiple servers (e.g. [40, 1, 11]).

At the same time, advances in MPC protocols (e.g. [30, 16, 2]) were dramatically reducing the cost of securely evaluating generic circuits. However, many computations are not efficiently realizable as circuits. It was well-known that ORAM techniques could be applied to create efficient MPC protocols in the RAM model [43, 23, 54]. This is the problem of Distributed Oblivious RAM (DORAM), accessing secret-shared memory at secret-shared locations without leaking anything but the number of accesses. 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. There is now no trusted client, instead there is a "virtual client" that is simulated by a secure computation.

Some of the extensions to the ORAM model that resulted from the memory outsourcing application were immediately relevant to DORAMs. Since DORAMs already had multiple non-colluding parties, they could trivially take advantage

of multi-server ORAMs by having the parties simulate different servers. Furthermore, since the parties were already performing computation as part of the MPC protocol, they could naturally perform the computation needed by servers in an active ORAM. On the other hand, like in the secure program use-case, the DORAM's virtual client needed to have very limited memory in order for it to have an efficient circuit representation. Furthermore, in an ORAM, the trusted client can perform local computation essentially for free, including cryptographic operations such as PRF evaluations. In the DORAM setting, every computation performed by the virtual client needs to be evaluated inside of a MPC circuit, which can require significant communication between the parties. On the performance metric side, since the MPC protocol occurs in a network, the number of communication rounds again becomes significant, as with memory outsourcing. However, like the application of secure program evaluation, the size of data blocks is often small (say a single variable), so any terms that do not depend on d are once again significant.

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. This transformation could, potentially, increase the communication cost, depending on the circuit complexity of the virtual client. Going the other way, 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. This will not lead to any increases in asymptotic communication costs.

Through this client simulation, efficient DORAMs could be produced from ORAMs. If a client can be represented as a Boolean circuit with q AND gates, it is possible to simulate this client in a secure computation using only $\Theta(q)$ communication [7, 2]. However, achieving statistical security for generic MPC requires an honest majority, so simulating the client without introducing cryptographic assumptions requires the use of at least 3 parties, even if the ORAM only uses 1 or 2 servers. Below we discuss the two most efficient prior multi-server ORAMs which offer statistical and perfect security respectively; Table 1 presents a summary of these works, as well as the DORAMs presented in this work. Both protocols have simple clients, so can be converted to 3-party honest-majority DORAMs without increasing the amortized asymptotic communication cost.

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 $b = 2$.

Chan et al. [11] 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 labels. These are stored in a position label ORAM, which is implemented recursively. The communication cost is $\Theta(\log(n)d + \log^3(n))$.

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. [11]	$\Theta(\log(n)d + \log^3(n))$	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 1. Communication complexity of selected info-theoretic ORAM protocols that can be converted to efficient DORAMs. Note that while the ORAM of Abraham et al. requires only 2 servers, converting it to an info-theoretic DORAM requires an honest-majority MPC protocol, therefore necessitating at least 3 parties. For statistically secure protocols, the statistical distance between the adversary's views in the real and ideal executions is $2^{-\omega(\log(n))}$.

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 [11], many had good concrete efficiency. They took advantage of the existence of multiple non-colluding servers by using Distributed Point Functions [9, 53], secret-shared PIR (SSPIR) [31] and secure shuffles/routing [21]. See Table 2 for details. There has also been work to create DORAMs that are secure against malicious adversaries [20, 29]. These works all depended on computational assumptions. In comparison, the MetaDORAM protocols are information theoretically secure. They also have strictly better communication cost than these protocols over all parameter ranges with one exception: [21] can have a lower asymptotic communication cost, and that only when $\kappa + d = o(\log^2(n)/\log(\log(n)))$.

DORAM Protocol	Amortized Communication Cost (bits)	Security
Faber et al. [19]	$O\left(\omega(1)\log^2(n)d + \kappa\omega(1)\log^4(n)\right)$	Computational
Jarecki and Wei [31]	$O\left(\log(n)d + \kappa\log^3(n)\right)$	Computational
Bunn et al. [9]	$O\left((d+\kappa)\sqrt{n}\right)$	Computational
Falk et al. [21]	$O\left(\log(n)d + \kappa\log(n)\right)$	Computational
DuORAM [53]	$O\left(\kappa \cdot d \cdot \log n\right)$	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

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 $O(2^{\omega(1)\log(n)})$.

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 pre-determined 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. However, this would require performing SSPIR over an array of size $\Omega(n)$. To avoid this, we instead store items in oblivious 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 an oblivious hash table. To limit the number of oblivious hash tables, when there are b tables of a given size, the contents of these tables are extracted and rebuilt into a single table. Furthermore, every n accesses, the contents of all tables are extracted and rebuilt into a single oblivious 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 $O(c + b\log_b(n)h) = poly(\log(n))$.

There are however several challenges with this approach. First, the exact location of an item is now no longer dependent solely on the item's index or the time the item was last accessed, but is also dependent on the hash functions and which hash function was used to store the item in a given table. Second, evaluating the hash function inside of a secure computation would be very expensive. Third, we need an efficient way to construct the oblivious hash tables.

In order to address these challenges we create intermediatary temporary identifiers for each item, which we refer to as runes (Random Unique NamEs). The metadata map will maintain the mapping between indices and runes, and an index will only be assigned a new rune when the index is accessed. We then separate the roles of the parties into a Builder (P_0) and two Holders (P_1 and P_2). The Builder selects a new rune for an item when it is accessed. It therefore knows, for each location in physical memory which rune is assigned to the item that is stored in that location. The oblivious hash tables are built based on the evaluation of the hash functions on the runes (not the indexes), which allows the Builder to evaluate the hash functions locally (rather than it being evaluated inside of a secure computation). When building smaller tables into a new large table, the Builder knows for each item its location before the build and its rune (even though it does not know the index of the item). The Builder can also calculate the new locations for all item in the new table, based (only) on their runes. The Builder can therefore locally calculate how memory needs

to be permuted to combine the tables, without ever needing to learn the indexes associated with each item. This allows oblivious hash tables to be constructed very efficiently, using a secure routing protocol.

How then do we perform Secret-Shared PIR over only $O(c+\log_b(n)bh)$ locations to access the item? Firstly, the tables must only be stored by the Holders, so the Builder will not learn which locations were accessed. To allow the hash functions to be computed efficiently during an access, the current rune of the queried index is revealed to the Holders, allowing them to compute the hash functions locally. An additional 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). The masks are stored in the metadata map.

One final challenge remains. Each time a new table is rebuilt, the items in that table must be assigned a new location, and also remasked with a new mask. This could be solved by updating the metadata data-structure for each index each time the index is rebuilt into a new table, but this would be prohibitively expensive. Instead, we observe that since the Builder can choose the assigned runes for each access, it can pre-determine which runes will be assigned at which point in time. This also pre-determines which runes will occur in each table, allowing the Builder to pre-calculate the assignments of runes to table locations, for say the first n accesses at the beginning of the protocol. The Builder can therefore pre-generate the schedule of runes to table positions at the start of the protocol, and can efficiently secret-share this data structure with the Holders. Likewise, the Builder can pre-determine the mask schedule for each rune, and secret-share this at the start of the protocol. Every n accesses, the full DORAM is refreshed, and the Builder then generates new rune, position and schedules for the subsequent n accesses.

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. 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 such that even when it is reduced modulo these ranges, the distribution is still essentially uniform.

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	

Table 3. Types of Secret-Sharing with Notation

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 $x_1 \oplus x_2 \oplus x_3 = x$. P_i holds x_i and $x_{(i+1) \bmod 3}$. When variable x is held using this secret-sharing, it is represented as $[x]$.

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 [15] 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 $\mathcal{F}_{DORAM}$

$I \leftarrow$ **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 $A_x = f([v], [y])$.
$[A] \leftarrow I$**.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 [39]. 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[4] 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.

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

$[v] \leftarrow$**SSPIR**$(m, d, q, [A]_{(1,2)}, [x])$: 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. $q \in \{1, \ldots m\}$ is a free parameter for efficiency optimization.

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 $j \notin \{Q(i)\}_{i \in [1,m]}$.

[4] Some works specify a security parameter, say σ, such that the statistical distance should be $2^{-\Theta(\sigma)}$. We choose instead for the distance negligible in n, which is equivalent to saying we set $\sigma = \omega(\log(n))$.

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 OHTable using cuckoo hashing with many ($h = \Omega(\log(n))$) 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 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. 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 $c + \ell(b-1)h$.

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 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 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).

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 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 $j \in [1, b^i c]$:
         A. $[R_{i+1,ub^i c+j}]_0 = [R_{i,j}]_0$
         B. $[V_{i+1,ub^i c+j}] = [V_{i,j}]$
         C. $[X_{i+1,ub^i c+j}] = [X_{i,j}]$
         D. Delete $[R_{i,j}]_0$, $[V_{i,j}]$ and $[X_{i,j}]$.
         E. P_0 picks a new OTP, $E_{i+1,ub^i c+j}$ from $\{0,1\}^d$.
         F. $[Z_{i+1,ub^i c+j}] = [V_{i+1,ub^i c+j}] \oplus [E_{i+1,ub^i c+j}]$
      iii. 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 $[b^i c]$ to $[2(1+\epsilon)b^i c]$ that maps $R_{i,1\ldots b^i c}$ to satisfying locations with these hash functions.
      iv. Use this mapping to build an OHTable containing the newly masked blocks:
         $[T_{i+1,u}]_{(1,2)} = \mathcal{F}_{Route}([Z_{i+1,ub^i c+1\ldots(u+1)b^i c}], [Q]_0)$
   (b) if $(t = n)$ **Refresh**()

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

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 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.

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: Init

Init(n, d, [V]):

1. P_0 creates a random permutation which determines the assignment of runes, $[M]_0 : [1, 2n] \to [1, 2n]$
2. Assign the first n of these to be the original runes for the indices. Initialize a new sub-DORAM 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 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 $M_i = r$.
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: $t = 1$.

Fig. 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 $S : [1, m] \to [1, m]$. 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$(n, d, [V])$

Fig. 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 $[v]$

Fig. 4. DORAM read protocol

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). Our main result is the following:

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.*

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 $R_{i,j}$, 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 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 [57]. 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 1 (Derived from proof of [57] Lemma 3). *For some $m \leq 2n$, let C be a disjoint-table cuckoo hash table with αm locations ($\alpha \geq 1$), and h hash 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.

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 1, 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 1. *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, 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.

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 2. *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

In this section, we analyze the amortized communication complexity per access of MetaDORAM1 and MetaDORAM2.

Theorem 2. *The amortized per access communication complexity of MetaDORAM1 is* $\Theta(\log_b(n)d + b\omega(1)\log(n) + \log^3(n)/\log(\log(n)))$*, where* $b \geq 2$ *is a free parameter and* $\omega(1)$ *is any super-constant function.*

Theorem 3. *The amortized per access communication complexit of MetaDORAM2 is* $\Theta(\log_b(n)d + b\omega(1)\log(n) + \log^3(n)/\log(\log(n)))$*, where* $b \geq 2$ *is a free parameter.*

Proof. The only place in which the communication costs of MetaDORAM1 and MetaDORAM2 differ is in the SSPIR. Since MetaDORAM1 uses slightly more hash functions, the size of the array for the SSPIR is larger, resulting in a slightly higher communication cost. In general the complexity analysis will be common to both protocols; in the places these differ we state so clearly.

We assume that the cost of $\mathcal{F}_{BalancedSSPIR}$ is $\Theta(m/q + qd)$ for any $q \in [1, m]$ and the cost of $\mathcal{F}_{Route}$ is $\Theta(q(d + \lg(q))$ as instantiated in appendices A and B respectively. We make the standard assumption that $d = \Omega(\log(n))$, that is each item contains at least the number of bits required to store its index.

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 $\Theta(\ell) = \Theta(\log_b(n))$ comparisons of $\Theta(\log(n))$-bit values, which requires $\Theta(\log(n)\log_b(n))$ communication.
4. The correct position and mask is obtained. This requires $\Theta(\ell) = \Theta(\log_b(n))$ 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 $\Theta((\log(n)+d)\log_b(n)) = \Theta(\log_b(n)d)$ communication.
5. Finally the SSPIR is executed. The number of locations is $c + \ell(b-1)h$, where $l = \log_b(n)$ and $c = bh$. For MetaDORAM1, we set $h = \omega(1)\log(n)$, so the total number of locations is $\Theta(\log_b(n)b\omega(1)\log(n))$ for some arbitrary super-constant function $\omega(1)$. We choose parameter $q = \log_b(n)$ as the SSPIR balancing parameter, resulting in a cost of $\Theta(b\omega(1)\log(n)+\log_b(n)d)$.
   For MetaDORAM2, $h = \log(n)$, so the total number of locations is $\Theta(\log_b(n)b\log(n))$. Again we choose $q = \log_b(n)$ as the SSPIR balancing parameter, which results in a cost of $\Theta(b\log(n) + \log_b(n)d)$.

Therefore the total communication cost of a read, excluding the call to the subDORAM, is $\Theta(\log_b(n)d + b\omega(1)\log(n))$ for MetaDORAM1 and $\Theta(\log_b(n)d + b\log(n))$ for MetaDORAM2.

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, which requires $\Theta(d)$ 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 $\Theta(m(d + \log(n)) = \Theta(md)$ communication.

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

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 $\Theta(\log(n) + d) = \Theta(d)$ resulting in $\Theta(nd)$ communication.
4. The same occurs again, resulting in $\Theta(nd)$ 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 $\Theta(d)$ 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), $\Theta(\ell) = \Theta(\log_b(n))$ 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 $\Theta((\log n + d)n \log_b(n)) = \Theta(\log_b(n)dn)$ 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 $\Theta((\log(n)+d)n) = \Theta(nd)$ communication.
9. The last step is a local relabelling.

Therefore, the total cost of Init, excluding the cost of initializing the subDORAM is $\Theta(\log_b(n)dn)$, which amortizes to $\Theta(\log_b(n)d)$ per access.

Summing these up, we obtain that the cost at the first level of the recursion is $\Theta(\log_b(n)d + b\omega(1)\log(n))$ for MetaDORAM1 and is $\Theta(\log_b(n)d + b\log(n))$ for MetaDORAM2. 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 $O(\log_b(n)\log(n) + b\omega(1)\log(n))$ (in both MetaDORAM1 and MetaDORAM2). Setting $b = \log^{0.5}(n)$ and setting $\omega(1) = \log^{0.5}(n)/\log(\log(n))$ we obtain that the cost per level is $\Theta(\log^2(n)/\log(\log(n)))$ (again in both MetaDORAMs). Since there are $\lg(n)$ recursions necessary to implement the subDORAM, the total communication cost per access of the subDORAM is $\Theta(\log^3(n)/\log(\log(n)))$.

This completes the proof of theorems 2 and 3.

Other Performance Metrics. 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 $\Theta(\log_b(n)dn)$: this is dominated by the size of the mask matrix 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 computation cost of the setup phase for MetaDORAM2 in theory amortizes to $o(1)$ after sufficiently many memory accesses, but in practice would become the bottleneck except for tiny n. The protocols access $c + \ell(b-1)h = O(\log_b(n)b\omega(1)\log(n))$ memory locations of size d in the top level, and $\Theta(\log^3(n)/\log^2(\log(n))$ memory locations of size $\Theta(\log(n))$ in each of the recursive levels, resulting in a total of $O(\log_b(n)b\omega(1)d + \log^5(n)/\log^2(\log(n)))$ bits of memory accessed per query.

7 Future Work

It is well established that a *passive* ORAM must incur a $\Omega(\log(n))$ overhead in the amount of memory accessed [26, 36, 34], even when there are multiple servers [37, 34]. Abraham et al [1] previously showed that statistically-secure DORAMs and multi-server *active* ORAMs were possible with sub-logarithmic communication overhead. In this work we show that this is also true in the case of perfect security. This therefore begs the question: *What are the communication lower bounds in general for active multi-server ORAMs and DORAMs with information-theoretic security?* Note that Abraham et al also presented a lower bound of $\Omega(\log_a(n))$ PIR operations to a multi-server ORAM when the model allows for PIR (reads and writes) on arrays of size a. By using standard PIR protocols, this allows for active multi-server ORAMs with sub-lograthmic communication overhead. However, there may be techniques other than PIR that can make use of server computation to reduce communication overhead.

Another open question pertains to deamortization. The communication costs of the MetaDORAM protocols are amortized. In particular, the cost of rebuilding the OHTables, and refreshing the namespace is amortized across multiple queries. There are existing techniques for deamortizing the cost of building OHTables [43, 5], but 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. *Do DORAMs exist with worst-case communication costs that are equal to the amortized communication cost as MetaDORAM1 and MetaDORAM2?*

The expensive setup phase of MetaDORAM2 is a barrier to practical usage. However, this does not seem inherent. The problem of finding suitable hash functions is identical to that of constructing a highly unbalanced bipartite expander (see for instance [28, 32]). Specifically, the protocol needs a bipartite expander containing $2n$ left vertices, $\Theta(m)$ right vertices and left-degree $\Theta(\log(n))$, which has expansion of 1 for all sets of size m. While we can generate a random graph that satisfies this property with high probability [57], perfect security requires an efficient explicit construction. *Is there an efficient explicit construction of* $(2n, \Theta(m), \Theta(\log(n))$- *bipartite graphs that have* $(m, 1)$ *expansion?*

Every DORAM can be used to implement a multi-server active ORAM: the client in the multi-server ORAM can simply secret-share the query between the servers. So our construction implies that $\Theta(\log_b(n)d + b\log(n) + \log^3(n)/\log(\log(n)))$ 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. *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. Daniel Noble would also like to acknowledge God for supporting him in this research. 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

1. Abraham, I., Fletcher, C.W., Nayak, K., Pinkas, B., Ren, L.: Asymptotically tight bounds for composing ORAM with PIR. In: IACR International Workshop on Public Key Cryptography. pp. 91–120. Springer (2017)
2. Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: 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. pp. 805–817 (2016)
3. Araki, T., Furukawa, J., Ohara, K., Pinkas, B., Rosemarin, H., Tsuchida, H.: Secure graph analysis at scale. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 610–629 (2021)
4. Asharov, G., Komargodski, I., Lin, W.K., Nayak, K., Peserico, E., Shi, E.: 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. pp. 403–432. Springer (2020)
5. Asharov, G., Komargodski, I., Lin, W.K., Shi, E.: Oblivious ram with worst-case logarithmic overhead. Journal of Cryptology **36**(2), 7 (2023)
6. Asharov, G., Komargodski, I., Michelson, Y.: Futorama: A concretely efficient hierarchical oblivious ram. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 3313–3327 (2023)
7. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for noncryptographic fault-tolerant distributed computations. In: Proceedings of the 20th Annual Symposium on the Theory of Computing (STOC'88). pp. 1–10 (1988)
8. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Computer Security-ESORICS 2008: 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings 13. pp. 192–206. Springer (2008)
9. Bunn, P., Katz, J., Kushilevitz, E., Ostrovsky, R.: Efficient 3-party distributed ORAM. In: Security and Cryptography for Networks: 12th International Conference, SCN 2020, Amalfi, Italy, September 14–16, 2020, Proceedings 12. pp. 215–232. Springer (2020)

10. Chan, T.H.H., Guo, Y., Lin, W.K., Shi, E.: 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. pp. 660–690. Springer (2017)
11. Chan, T.H.H., Katz, J., Nayak, K., Polychroniadou, A., Shi, E.: 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. pp. 158–188. Springer (2018)
12. Chen, H., Chillotti, I., Ren, L.: Onion ring oram: Efficient constant bandwidth oblivious ram from (leveled) tfhe. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 345–360 (2019)
13. Costan, V., Devadas, S.: Intel SGX explained. Cryptology ePrint Archive (2016)
14. Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In: Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005. Proceedings 2. pp. 342–362. Springer (2005)
15. Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Annual international cryptology conference. pp. 247–264. Springer (2003)
16. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Annual Cryptology Conference. pp. 643–662. Springer (2012)
17. Devadas, S., van Dijk, M., Fletcher, C.W., Ren, L., Shi, E., Wichs, D.: 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. pp. 145–174. Springer (2016)
18. Doerner, J., Kondi, Y., Lee, E., shelat, A.: Threshold ecdsa from ecdsa assumptions: The multiparty case. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 1051–1066. IEEE (2019)
19. Faber, S., Jarecki, S., Kentros, S., Wei, B.: Three-party ORAM for secure computation. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 360–385. Springer (2015)
20. Falk, B., Noble, D., Ostrovsky, R., Shtepel, M., Zhang, J.: Doram revisited: Maliciously secure ram-mpc with logarithmic overhead. In: TCC (2023)
21. Falk, B.H., Noble, D., Ostrovsky, R.: 3-party distributed ORAM from oblivious set membership. In: International Conference on Security and Cryptography for Networks. pp. 437–461. Springer (2022)
22. Furukawa, J., Lindell, Y., Nof, A., Weinstein, O.: High-throughput secure three-party computation for malicious adversaries and an honest majority. In: Annual international conference on the theory and applications of cryptographic techniques. pp. 225–255. Springer (2017)
23. Gentry, C., Goldman, K.A., Halevi, S., Julta, C., Raykova, M., Wichs, D.: Optimizing ORAM and using it efficiently for secure computation. In: Privacy Enhancing Technologies: 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings 13. pp. 1–18. Springer (2013)
24. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing. pp. 182–194 (1987)

25. Goldreich, O., Micali, S., Wigderson, A.: 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, pp. 307–328 (2019)
26. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Journal of the ACM (JACM) **43**(3), 431–473 (1996)
27. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. In: International Colloquium on Automata, Languages, and Programming. pp. 576–587. Springer (2011)
28. Guruswami, V., Umans, C., Vadhan, S.: Unbalanced expanders and randomness extractors from parvaresh–vardy codes. Journal of the ACM (JACM) **56**(4), 1–34 (2009)
29. Ichikawa, A., Komargodski, I., Hamada, K., Kikuchi, R., Ikarashi, D.: 3-party secure computation for rams: Optimal and concretely efficient (2023)
30. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Annual International Cryptology Conference. pp. 145–161. Springer (2003)
31. Jarecki, S., Wei, B.: 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. pp. 360–378. Springer (2018)
32. Kalev, I., Ta-Shma, A.: Unbalanced expanders from multiplicity codes. In: Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2022). Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2022)
33. Knott, B., Venkataraman, S., Hannun, A., Sengupta, S., Ibrahim, M., van der Maaten, L.: Crypten: Secure multi-party computation meets machine learning. Advances in Neural Information Processing Systems **34**, 4961–4973 (2021)
34. Komargodski, I., Lin, W.K.: 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. pp. 579–609. Springer (2021)
35. Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in) security of hash-based oblivious RAM and a new balancing scheme. In: SODA (2012)
36. Larsen, K.G., Nielsen, J.B.: Yes, there is an oblivious RAM lower bound! In: Annual International Cryptology Conference. pp. 523–542. Springer (2018)
37. Larsen, K.G., Simkin, M., Yeo, K.: 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. pp. 486–503. Springer (2020)
38. Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In: Information Security: 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings 14. pp. 262–277. Springer (2011)
39. Lindell, Y.: How to simulate it–a tutorial on the simulation proof technique. Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich pp. 277–346 (2017)
40. Lu, S., Ostrovsky, R.: Distributed oblivious RAM for secure two-party computation. In: Theory of Cryptography Conference. pp. 377–396. Springer (2013)
41. Mohassel, P., Rindal, P., Rosulek, M.: Fast database joins and psi for secret shared data. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 1271–1287 (2020)

42. Ostrovsky, R.: Efficient computation on oblivious RAMs. In: Proceedings of the twenty-second annual ACM symposium on Theory of computing. pp. 514–523 (1990)
43. Ostrovsky, R., Shoup, V.: Private information storage (extended abstract). In: Leighton, F.T., Shor, P.W. (eds.) Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997. pp. 294–303. ACM (1997). https://doi.org/10.1145/258533.258606, https://doi.org/10.1145/258533.258606
44. Patel, S., Persiano, G., Raykova, M., Yeo, K.: PanORAMa: Oblivious RAM with logarithmic overhead. In: 2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS). pp. 871–882. IEEE (2018)
45. Patel, S., Persiano, G., Yeo, K.: Recursive orams with practical constructions. Cryptology ePrint Archive (2017)
46. Persiano, G., Yeo, K.: Limits of breach-resistant and snapshot-oblivious rams. In: Annual International Cryptology Conference. pp. 161–196. Springer (2023)
47. Pinkas, B., Reinman, T.: Oblivious RAM revisited. In: Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30. pp. 502–519. Springer (2010)
48. Pinkas, B., Rosulek, M., Trieu, N., Yanai, A.: Spot-light: lightweight private set intersection from sparse ot extension. In: Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39. pp. 401–431. Springer (2019)
49. Ren, L., Fletcher, C.W., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., Devadas, S.: Ring oram: Closing the gap between small and large client storage oblivious ram. IACR Cryptol. ePrint Arch. **2014**, 997 (2014)
50. Shi, E., Chan, T.H.H., Stefanov, E., Li, M.: Oblivious RAM with $o((logn)^3)$ 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. pp. 197–214. Springer (2011)
51. Stefanov, E., van Dijk, M., Shi, E., Chan, T.H.H., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol. Journal of the ACM (JACM) **65**(4), 1–26 (2018)
52. Stefanov, E., Shi, E., Song, D.: Towards practical oblivious ram. arXiv preprint arXiv:1106.3652 (2011)
53. Vadapalli, A., Henry, R., Goldberg, I.: DuORAM: A bandwidth-efficient distributed ORAM for 2-and 3-party computation. In: 32nd USENIX Security Symposium (2023)
54. Wang, X., Chan, H., Shi, E.: Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 850–861 (2015)
55. Williams, P., Sion, R.: Single round access privacy on outsourced storage. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 293–304 (2012)
56. Yao, A.: Protocols for secure computations (extended abstract). In: FOCS (1982). https://doi.org/10.1109/SFCS.1982.88, http://dx.doi.org/10.1109/SFCS.1982.88
57. Yeo, K.: Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. arXiv preprint arXiv:2306.11220 (2023)

Appendix

A Secret-Shared Private Information Retrieval

This section presents a simple protocol for secret-shared Private Information Retrieval that is suitable 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 [19]. 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.

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

1. 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 $B_i = A_{qi} || \ldots || A_{qi+q-1}$
2. Let $[x]$ be split into its upper-order $\lg(m) - \lg(q)$ bits, labelled $[y]$ and its lowest-order $\ln(q)$ bits, labelled $[z]$.
3. Call the main SSPIR protocol to obtain the secret-shared y^{th} large block: $[u] \leftarrow UnbalancedSSPIR(m/q, dq, [B]_{(1,2)}, [y])$.
4. Inside of a secure computation, access the z^{th} small block in this big block: for $i \in [1, q]$ if $i = [z]$, $[v] = [u_{id \ldots id+i-1}]$.
5. Return $[v]$.

Fig. 5. Implementation of SSPIR

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 $\Theta(m/q + dq)$.

B 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 [41]. 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 $[D]$

Fig. 6. Secure Routing protocol