GigaDORAM: Breaking the Billion Address Barrier

Abstract

We design and implement GigaDORAM, a novel 3-server Distributed Oblivious Random Access Memory (DORAM) protocol. Oblivious RAM allows a client to read and write to memory on an untrusted server, while ensuring the server itself learns nothing about the client's access pattern. Distributed Oblivious RAM (DORAM) distributes the role of the ORAM server. Specifically, DORAM allows a group of servers to efficiently access a secret-shared array at a secret-shared index without reveling the index to any of the participating servers.

DORAM has two main advantages over traditional ORAM protocols: (1) it effectively outsources all the communication and computation costs to the DORAM servers, and (2) it allows secure multiparty computation (MPC) in the RAM model.

A recent generation of DORAM implementations (e.g. FLORAM, DuORAM) have focused on building DORAM protocols based on Function Secret-Sharing (FSS). These protocols have low communication complexity and low round complexity but linear computational complexity of the servers. Thus, they work for moderate size databases, but at a certain size these FSS-based protocols become computationally inefficient.

In this work, we introduce GigaDORAM, a hierarchical-solution-based DORAM which leverages several novel round-reducing tricks. GigaDORAM features poly-logarithmic computation and communication which are comparable to that of other hierarchical based (D)ORAMs, but with an over 100× reduction in rounds per query. In our implementation, we show that for small databases GigaDORAM is slightly faster than FSS-based schemes, but for moderate to large databases where FSS-based solutions become computation bound, our protocol is orders of magnitude more efficient than the best existing DORAM protocols. When $N = 2^{31}$, our DORAM is able to perform over 700 queries per second while previous constructions could not handle even 2 queries per second.

1 Introduction

To an outside observer, traditional encryption schemes can effectively hide the *contents* of a memory, yet encryption alone does not hide the memory locations being accessed. In many cases, the *access pattern* of a file system can leak sensitive information, even when the contents are encrypted.

Oblivious Random Access Memory (ORAM), introduced by Goldreich and Ostrovsky [20] is a cryptographic protocol that allows a client to read and write from memory while ensuring the *physical access pattern* (which is potentially observable to someone with sufficient access to the machine) is independent of the *virtual access pattern* (the underlying data retrieved by the client). Thus, when memory is accessed using an ORAM protocol, it is *mathematically provable* that an observer learns nothing about the client's query pattern (beyond the number of queries).

Oblivious RAM was developed in a model where a single client wishes to store and retrieve sensitive data from an untrusted data store. Originally, the untrusted data store was conceptualized as untrusted RAM on the same machine as the client, but today we usually imagine a client storing and retrieving data from an untrusted cloud provider. In this setting, encryption can hide the *data* from the cloud provider, but ORAM is necessary to hide the *access pattern*[1]. Thus, ORAM provides the strongest possible guarantee – hiding *both* the data and the access pattern.

Although ORAM was designed in the client-server setting, a slight variant of ORAM is also useful in the context of secure multiparty computation, where a group of servers need to access a secret-shared array at a secret-shared location. In this setting, the secret sharing hides the *data*, but every participating server observes the physical access pattern. *Distributed Oblivious RAM (DORAM)* provides a method for efficiently

[1]Note that most Searchable Symmetric Encryption (SSE) schemes allow a client to efficiently query encrypted data stored in an untrusted cloud, but they typically do *not* hide the access pattern from the cloud provider. SSE schemes also target a different model, where data payloads may be of drastically different size (in ORAM all entries are of size D) and queries may return different number of "matches" [33]

accessing a secret-shared array at a secret-shared index, which in turn makes it possible to do secure multiparty computation (MPC) in the RAM model (RAM-MPC). Almost all existing MPC protocols work in the *circuit-model* where the desired computation is first converted to an arithmetic/boolean circuit before being executed securely. This conversion becomes costly (e.g. in the case of private database) since every "random-access" is replaced with an $O(\text{memorySize})$ "MUX operation." RAM-MPC allows random-access programs to be executed securely without this costly conversion, which in turn enables much more flexible and efficient secure multiparty computation protocols [22, 38].

1.1 Previous DORAMs

Although there are many different ways to measure the efficiency of a (D)ORAM protocol, the theory community has emphasized the (amortized) communication complexity per query. Several recent DORAM protocols that have $(O(\log(N)))$ communication and computation overhead have been presented (e.g. [17, 31]). These asymptotically-optimal protocols are built using the ORAM's *"hierarchical solution"*, introduced in [36, 37], which we describe in Section 4. The downside of these protocols is that they have high round complexity, requiring $O(\log(N))$ rounds of communication (with large hidden constants) per query. In practice, network latency causes bottleneck in performance of these solutions.

It has been noted that the high round complexity of the hierarchical solution makes it unsuitable for practical applications [47], so most DORAM *implementations* (e.g. [10, 15, 22, 24, 25, 42–44, 47, 51]) take a different approach. These constructions focus on minimizing rounds while compromising on either asymptotic computation or asymptotic communication costs. One common technique (most recently applied by DuORAM [43]) for creating DORAM protocols with low round complexity *and* low communication complexity is Function Secret Sharing (FSS) [9, 18]. While FSS results in low *communication* complexity, FSS-based protocols require $O(N)$ computation, where N is a size of the database. Thus, protocols like DuORAM shine in high-latency, low-bandwidth networks at a small database size.

By contrast, our DORAM is designed for low-latency environments (such as of co-located servers in the same datacenter). This design choice is present in several current deployments, such as Points of Presence (POP) of mutually distrustful ISP machines present in the same Data Center in support of BGP protocols. These mutually mistrustful yet co-located machines offer a viable alternative to data clean rooms, which present a single point of failure (if security of clean room is breached). As another example, Cybernetica's commercial offering "Sharemind" MPC platform is intended to be run on three servers in nearby data centers [8]. In these low-latency environments, FSS-based DORAMs quickly becomes a bottleneck of performance for large values of N. We focus on this type of network environment because, these type of low-latency networks are necessary for RAM-MPC, which is one of the main motivations for building DORAM protocols in the first place.[2]

1.2 Our Contributions

1.2.1 Protocol contributions

In this work, we design and implement a high-performance DORAM protocol in the (3,1)-security model, i.e., where there are three semi-honest servers, and no two of them collude. Two effective techniques for building (D)ORAM protocols are "the hierarchical solution" (e.g. [3, 4, 17, 31]) which yields (D)ORAM protocols with low communication complexity, but high round complexity, and FSS-based DORAMs (e.g. FLORAM and DuORAM) which have very low round complexity, but high computational complexity. In this work, we show how to reduce the round-complexity of the hierarchical solution and give round-efficient hierarchical DORAM that scales well beyond the limits of FSS-based (D)ORAM protocols. Specifically, our DORAM requires $O((\kappa^2 + D)\log N)$ communication and computation per query. At the cost of practically significant round-cost, we can easily tweak our protocol to matches the best-known (D)ORAM complexity [3, 4, 17, 31] of $O((\kappa + D)\log N)$ (Remark 4.1). Our work is the first (D)ORAM protocol achieving these asymptotics to be implemented.

1.2.2 Code contributions

Implementing our DORAM protocol required over 9,000 lines of custom C++ code. Additionally, we provide lightweight client implementation in Python and JavaScript, enabling users to to utilize DORAM in the client-server setting.

In addition to the implementation of our DORAM protocol, we contribute (1) a from-scratch competitive implementation of the [2] general MPC framework which, in many settings, is the fastest known 3-party MPC protocol.[3] (2) A custom (3,1)-garbled circuits protocol built using EMP-toolkit's 2-party garbled circuits, which can be imported separately from ABY3's [41] large framework, and (3) The first tested circuit files of the LowMC block cipher [1], featuring a novel optimization which reduces cache misses.

2 Preliminaries

Notation. We let N be the number of elements in the DORAM database, D the size in bits of each element, and κ be the computational security parameter (in practice $\kappa = 128$, in

[2]Although MPC can run in high-latency environments [45], in a high-latency environment MPC can only compute simple functions that would not benefit from a RAM-model computation.

[3]The implementation used to benchmark the results in the [2] is proprietary, and not publicly available.

theory $\kappa = \omega(\log N)$), σ the statistical security parameter (in practice 2^{-40} or 2^{-80}).

Secret sharing. Our DORAM protocol makes heavy use of secret-sharing. Throughout this work, we use $[\![\cdot]\!]$ to denote a "replicated" (or CNF [13]) secret sharing. In a 3-party replicated sharing, a secret, x is split into three shares $x = x_1 \oplus x_2 \oplus x_3$, and participant x gets *two* of the shares – every share except x_i. We use $[\cdot]^{(i,j)}$ to denote a simple XOR-2-sharing between participants P_i, P_j

Obliviousness: A computation is data-*oblivious* if its control-flow is independent of the input data. An Oblivious RAM protocol is a protocol for accessing an array (indexed by $1,\dots,N$), where the algorithm's control flow, and in particular, the physical memory accessed, is independent of the index being queried. ORAM protocols are designed to allow a client to make a *any sequence* of queries and are often composed of simpler data structures which are only oblivious on distinct queries. A data structure (e.g. a hash table) is called *distinct-query oblivious* if the control-flow between any two sequences of *distinct* queries is indistinguishable, but a sequence with repeated queries might result in a control flow that is distinguishable from a sequence of distinct queries.

Cuckoo Hashing: A cuckoo hash table is a distinct-query oblivious data structure for storing key-value pairs [39]. The cuckoo hash table consists of two arrays ("tables") T_0 and T_1 of size $c \cdot n$, and two hash functions h_0 and h_1, $h_i : X \to [cn]$. A key-value pair, (x,y), can be stored in the table at location $h_0(x)$ in T_0 *or* $h_1(x)$ in T_1. It is possible that there is no valid way to store a series of elements (e.g. if there are x_0, x_1, x_2 such that $h_0(x_0) = h_0(x_1) = h_0(x_2)$ and $h_1(x_0) = h_1(x_2) = h_1(x_3)$). In this case we say there is a "build failure." The probability of a build failure is $1/\text{poly}(n)$, but if we allow a small "stash" that can hold at most $O(\log(n))$ elements, the probability of a build failure becomes negligible in n [26,35]. In our DORAM (as in most prior hierarchical (D)ORAM protocols) we use cuckoo hash tables as a building block for our oblivious hash tables.

DORAM: A distributed Oblivious RAM protocol is a multiparty protocol that allows a group of participants holding a secret-shared array $[\![x_1]\!],\dots,[\![x_N]\!]$ to access the array at a secret-shared index, $[\![i]\!]$, and obtain the sharing $[\![x_i]\!]$, without revealing any information about the query, i, or the database $x_1,\dots,x_n$. The theoretical efficiency of DORAM protocol is often measured by the amortized communication complexity the servers spend to respond to a single query. In practice, DORAM protocols may be bottlenecked the amortized communication per query (e.g. [44]), the amortized computation per query (e.g. [15,43]), or the amortized number of communication rounds per query (e.g. [17,31]). To compare between these constructions targeting different points in the solution space, *we measure the practical efficiency of a DORAM protocol by the number of queries per second that it can process.*

SISO-PRFs: A core building block of most DORAM protocols is a Shared-Input, Shared-Output PRF (SISO-PRF). A SISO-PRF allows the participants to compute the secret-sharing of a PRF output on a shared input, under a shared key. Any regular PRF can be converted into a SISO-PRF by implementing the PRF under a generic MPC protocol. Several PRF protocols have been designed to be "MPC friendly" (e.g. LowMC [1]). The basic idea which makes SISO-PRFs useful for DORAM is that servers can generate a random, shared key, $[\![k]\!]$, and build a hash table where the cleartext tags (of secret-shared payloads) are SISO-PRF evaluations of the elements.

3 Secure Multiparty Computation

Secure multiparty computation (MPC) [12, 19, 48, 49] is a protocol that allows a group of participants to securely compute a joint function on their private inputs *without* revealing any information beyond the output of the function. An MPC protocol is called (n,t)-secure if the protocol involves n participants, and remains secure if at most t participants collude (i.e., share private state). Our DORAM protocol works in the standard (3,1) semi-honest security model, which assumes that there are three semi-honest servers, and no collusion between servers. We use the (3,1)-"replicated" MPC protocol of [2] as a building block. In one crucial place, we also use a custom implementation of the (3,1) garbled circuit MPC protocol of ABY3 [32] to reduce round complexity. Our DORAM protocol is also an important tool in *building* efficient RAM-MPC protocols, because it allows for MPC computation in the *RAM model* of computation, whereas most current MPC protocols work in the *circuit model*.

3.1 The Arithmetic Black-Box Model

Our DORAM protocol makes use of several "basic" operations on secret shared values, e.g. addition, comparisons, and equality tests. In our protocol descriptions, we use the Arithmetic Black Box (ABB) model to abstract away the underlying implementations of these operations. In practice, we use our own implementation of [2]. A formal description of the ABB model can be found in [17, 23].

In protocols, we use our ABB by invoking $\mathcal{F}_{\text{ABB}}$.FunctionalityName, where FunctionalityName makes it obvious what the functionality achieves. For instance, we invoke $[\![z]\!] = \mathcal{F}_{\text{ABB}}.\text{Mult}([\![x]\!],[\![y]\!])$ to multiply secret shared values x,y and obtain secret shared value z s.t $z = x \cdot y$. Although the names are usually self-explanatory, we provide a complete list of our ABB functionalities in Appendix A.

4 Construction Overview

Section 4.1 describes known techniques which enabled building communication and computation efficient "hierarchical"

DORAM which takes *many* rounds of communication to execute a single query [17, 31]. In Section 4.2 we motivate and outline several novel techniques which enable us to significantly reduce the round complexity of standard "hierarchical" DORAMs. In Section 9, we show that our round-reduced "hierarchical" DORAM, GigaDORAM, is efficient in practice.

4.1 The hierarchical solution

OHTable We call an efficient, oblivious to build, distinct-query oblivious, hash table an *OHTable*. Oblivious Hash Tables have three key functionalities: Build, Query and Extract. $\mathcal{F}_{\text{OHTable}}.\text{Build}(X)$ creates an oblivious hash table storing the elements X, where each element in X is unique. Once the hash table has been built, $\mathcal{F}_{\text{OHTable}}.\text{Query}$ queries the table obliviously, and $\mathcal{F}_{\text{OHTable}}.\text{Extract}$ extracts all elements currently stored in the table which have *not* been queried.

The hierarchical solution. The key idea of the hierarchical solution is that it is fairly easy to build a distinct-query OHTable – for example reading from a cuckoo hash table is oblivious as long as you never query the same element twice. With this insight, the hierarchical solution [37] can be seen as a compiler for bootstrapping a distinct-query OHTable construction into a full-fledged DORAM (that remains oblivious even if the user makes repeated queries). The hierarchical solution is a powerful tool, used to build many ORAM protocols, e.g. [3, 17, 20, 21, 27, 31, 36, 37, 40], and is the technique used by many of the most (asymptotically) communication efficient (D)ORAM protocols.

A hierarchical ORAM is made up of a hierarchy of OHTables, $L_0, \ldots, L_{\mathsf{numLevels}}$, of geometrically increasing size. Usually, we have $\mathsf{numLevels} = O(\log N)$ and the largest level, $L_{\mathsf{numLevels}}$, has size $O(N)$. The smallest level, L_0, is a small "cache". The cache itself needs to remain oblivious even if queries are repeated. But the cache is small, so it can be implemented inefficiently without dramatically increasing the cost of a given ORAM query. For this reason, the cache is often set be constant sized ($|L_0| = O(1)$). This means that the client can read the *entire* cache with each query (in time $O(|L_0|)$) and this is clearly oblivious. Each larger level, L_i for $i \geq 1$ holds a distinct-query OHTable of size $O\left(2^i\right)$.

When a user queries the DORAM, the user queries each level of he hierarchy sequentially. If the item is found at level i, a "dummy" element is queried at subsequent levels $L_{i+1}, \ldots, L_{\mathsf{numLevels}}$, and the retrieved item is reinserted into the cache. To maintain obliviousness, if the requested item is not found anywhere in the hierarchy, a dummy item is inserted in the cache. When the cache (or any subsequent level of the hierarchy) is full, all the elements from that level (and smaller levels) are extracted and rebuilt into the next level of the hierarchy. If $|L_{i+1}| \geq 2 \cdot |L_i|$, then level $i+1$, can accommodate all the elements in levels $0, 1, \ldots, i$. This periodic rebuild schedule guarantees that no element is ever queried twice at the same level between rebuilds, because once an element has been found at level i, it is moved to the cache, and will always stay in a level $j < i$, until level i is rebuilt. Since no element will be queried twice in any given distinct-query OHTable, the entire construction is oblivious, even if queries are repeated. The formal proof that the resulting hierarchical data structure is indeed an ORAM is now standard, and can be found for example in [37]. With this rebuild schedule, level L_i is rebuilt every $O(|L_i|)$ queries. We outline our instantiation of the hierarchical protocol in Figure 4 in Section 8.

4.2 Reducing $\mathsf{numRoundsDORAM}$

Our protocol is based on a 3-party implementation of the hierarchical solution (similar to [17]). Although the hierarchical solution has low asymptotic communication and computation complexity, it has high *round* complexity which can make it inefficient in practice.[4] This inefficiency stems from the fact that every query to the DORAM forces a query into *every* level of the hierarchy, $L_0, \ldots, L_{\mathsf{numLevels}}$. Critically, these queries must be *sequential*, because if the item is found at L_i, the protocol must query dummy elements at $L_{i+1}, \ldots, L_{\mathsf{numLevels}}$. Thus the *round complexity* of a DORAM query[5], $\mathsf{numRoundsDORAM}$, can be written as

$$\mathsf{numRoundsOHTable} \cdot \mathsf{numLevels} + \mathsf{numRoundsCache} \quad (1)$$

where $\mathsf{numRoundsOHTable}$ is the number of rounds to query L_i for $1 \leq i \leq \mathsf{numLevels}$ (which is fixed and independent of i) and $\mathsf{numRoundsCache}$ is the number rounds it takes to query the cache, L_0. The main technical contribution of our work is to optimize the hierarchical solution to reduce its round complexity.

In light of Equation 1, we can divide our efforts to reduce round complexity into four parts: (1) *reducing* $\mathsf{numRoundsOHTable}$ *via our novel OHTable, ShufTable, and SISO-PRF parallelization*, (2) *generalizing the hierarchical-solution with a tuneable parameter,* $\mathsf{baseAmpFactor}$*, to reduce* $\mathsf{numLevels}$, (3) *designing a new oblivious-cache data-structure, SpeedCache, to reduce* $\mathsf{numRoundsCache}$, and (4) *applying additional engineering/implementation-level optimizations.*

With these design improvements, we are able to significantly reduce the round complexity of the hierarchical solution, while maintaining similar asymptotic overheads to the theoretic state-of-the-art, [17]. In addition to its low asymptotic complexity and low rounds-per-query, we show that our DORAM design is actually quite fast in practice (Section 9).

[4]The (3,1)-DORAM of [17] and the 2-server ORAM of [31] use the hierarchical solution to achieve amortized communication complexity of $O((\kappa + D)\log N)$. We remark, however, that both these protocols have high round complexity, and have never been implemented.

[5]DORAM also incurs round costs when building a level L_i and extracting from a level L_i. Since the protocols are only invoked every $|L_i|$ queries and have small, constant, round costs (where our final protocol has $|L_0| = O(\kappa)$), their round cost has negligible impact on the performance of DORAM.

We briefly explain each of our optimizations below and expend on them in Section 5, 6 and 7 respectively.

(1) Reducing $\mathsf{numRoundsOHTable}$: ShufTable & SISO-PRF parallelization. To reduce $\mathsf{numRoundsOHTable}$ we present a novel, standalone OHTable, called *ShufTable*, with reduced round complexity. We also devise a method to parallelize the sequentially-dependent SISO-PRF evaluations needed to query each level of the hierarchy.

The key ideas in these optimizations revolve around how to handle queries for elements that *are not in the table*. This type of query happens frequently in a hierarchical ORAM, because although each element is only stored at one level of the hierarchy, each ORAM query results in a query to the OHTable at every level of the hierarchy. Prior solutions for handling these "dummy" queries were round-intensive (e.g. the Oblivious Sets in [17]), so we develop a new method.

As in prior works, our OHTable, ShufTable inserts dummy elements $(d_1, \perp) \ldots, (d_l, \perp)$ along with the real elements, and retrieves a dummy when the queried element is not stored in the table. Assuming the SISO-PRF has already been evaluated, ShufTable requires only 5 rounds of server-to-server communication per query. The main ingredient that enables this round-savings is a novel "persistent shuffling" trick (Section 5.1)[6] which allows to efficiently evaluate a random permutation under MPC. Our OHTable, like those in prior DORAM constructions, requires an equality-check on secret-shared values. We implement this equality check using a custom implementation of a 3-party garbled circuit [32]. This *increases* the asymptotic communication of each level's query from $O(\kappa)$ to $O(\kappa^2)$, but *decreases* the round complexity at each level by $O(\log\kappa)$. In practice, when $\kappa \in \{128, 256\}$, this dramatically improves real-world performance.

More details can be found in section 5.2 and the protocol is given in Figure 2.

Remark 4.1. If one were focused on optimizing asymptotic communication complexity, this step could be replaced by an constant-overhead-MPC-based equality check (e.g. using [2]), which would give us best known (same as [17,31]) asymptotic communication and computation complexity $O((\kappa + D)\log)$.

In the hierarchical solution, each a query into the OHTable at level i, requires evaluating a SISO-PRF, but the query into table L_{i+1} depends on whether the item was found at a lower level. Thus, hierarchical based (D)ORAMs wait until after L_i.Query is performed to evaluate the SISO-PRF for L_{i+1}.

With ShufTable and most other OHTables, there are essentially two types of queries that can be made at any level a "real" query (if the desired element has not yet been found) or a "dummy" query (if the element was found at a lower level). A simple observation we make is that we can evaluate $[\![r_i]\!]$, the PRF evaluation needed to make a real query at L_i, and $[\![d_i]\!]$, the PRF evaluation needed to make a dummy query at L_i, *in parallel for all* $i \in [\mathsf{numLevels}]$. Then, after L_i is evaluated and it is determined under MPC whether the queried element was found, we can multiplex between $[\![r_{i+1}]\!]$ and $[\![d_{i+1}]\!]$ under MPC, which takes only one round. Since the outputs of the SISO-PRF are secret-shared and since multiplexing in MPC hides whether the table is queried for a dummy or a real element, the DORAM remains oblivious. This optimization decreases the rounds per query dedicated to evaluating the SISO-PRF from $\mathsf{numRoundsPRFEvalnumLevels}$ to $\mathsf{numRoundsPRFEval}/+\mathsf{numLevels}$.

Unfortunately, the above change increases communication per query by $\approx 2\times$. To resolve this issue, we upgrade ShufTable with a "just-in-time" mechanism to detect if an element is stored in the table, replacing the retrieved element with a dummy if necessary. Due to this mechanism only $[\![r_i]\!]$ is needed to query the OHTable at L_i.

In practice this optimization is noticeable, saving ~ 100 rounds of interaction per query.

To further reduce rounds, in practice, we parallelize the query of the cache, L_0 with the evaluation of the PRFs for $L_1, \ldots, L_{\mathsf{numLevels}}$. Note that if we knew multiple queries in advance, we would be able to further batch evaluations, but we do not assume that in this work.

Overall, these optimizations have a significant impact of the efficiency of our protocol. We decrease $\mathsf{numRoundsOHTable}$ from the 45 of [17] to amortized $5 + \mathsf{numRoundsPRFEval}/\mathsf{numLevels} \approx 7$. Additionally, we reduce the number of expensive SISO-PRF evaluations necessary by a factor of four when compared to [17] which is the state-of-the-art in low SISO-PRF hierarchical DORAMs.

(2) Generalizing the hierarchical solution to reduce $\mathsf{numLevels}$. Above, we described our techniques for reducing the round complexity of queries to individual OHTables. Yet in the hierarchical solution, every ORAM query requires querying each level of the hierarchy *sequentially*. Thus a hierarchy of depth $\mathsf{numLevels}$ immediately adds a multiplicative factor of $\mathsf{numLevels}$ to the round complexity of the protocol. Since round complexity is one of the main performance bottlenecks in (D)ORAM protocols, there is a strong motivation to reduce $\mathsf{numLevels}$.

In the original hierarchical construction, and all subsequent constructions of which we are aware, level i in the ORAM hierarchy had size $2^i \cdot |L_0|$, resulting in $\mathsf{numLevels} = O(\log_2 N)$. In Section 6 we show that by introducing a new parameter, $\mathsf{baseAmpFactor} > 2$, and setting $|L_{i+1}| = \mathsf{baseAmpFactor} \cdot |L_i|$, we can reduce the *round complexity* of the protocol with minimal impact on the *communication* complexity, which we find to be less expensive in practice. This simple change immediately reduces $\mathsf{numLevels}$ from $\log_2(N)$ to $\log_2(N)/\log_2(\mathsf{baseAmpFactor})$. While this modification is conceptually simple, it requires a more nuanced rebuild schedule.

Somewhat surprisingly, we find that the optimal value for $\mathsf{baseAmpFactor}$ in practice is much greater than 2, which

[6] [29] presents various efficient shuffling protocols, but none of their shuffles allows for the efficient evaluation of a reverse permutation at a point.

is the value implicitly suggested by all previous works. For instance, at $N = 2^{30}$ and the network conditions we test we found that $\mathsf{baseAmpFactor} = 128$ is optimal. That is, each level is 128 times larger than the previous (smaller) level.

(3) Optimizing the cache: SpeedCache. In the hierarchical ORAM solution, the top level (i.e. L_0, the "cache") needs to support oblivious accesses (compared to lower levels in the hierarchy which only need to be *distinct-query oblivious*). For this reason, L_0 is usually implemented as a simple read-all-to-read, append-to-write array. This is obviously oblivious, but its query complexity increases linearly with the size of the cache. In particular, if the cache stores t key-value pairs $((\llbracket x_1 \rrbracket, \llbracket y_1 \rrbracket), \dots, (\llbracket x_t \rrbracket, \llbracket y_t \rrbracket))$, querying the cache is often implemented by sequentially checking whether the query, $\llbracket x \rrbracket$, is equal to $\llbracket x_i \rrbracket$ (costs $O(\log\log|x_i|)$ sequential rounds) and if so updating return value to $\llbracket y_i \rrbracket$. Unfortunately, this simple implementation has multiplicative depth t, meaning $\mathsf{numRoundsCache} = O(|L_0| \cdot \log\log|x_i|)$.

In Section 7, we outline a simple Cache protocol SpeedCache that allows us to query the cache in $\lceil \log\log|x_i| \rceil + 1$ rounds of communication (which is *independent of* $|L_0|$). Since our SpeedCache protocol has a round complexity that is independent of the cache size, we are somewhat free to increase the cache size, which has other benefits (e.g. reducing $\mathsf{numLevels}$ by $\log_2|L_0|$).

(4) Gadget implementations: Minimizing the round complexity of our SISO-PRF is crucial for the overall efficiency of our protocol, so we provide the first circuit file of the LowMC [1] block cipher within our custom MPC implementation. Our circuit file features a novel optimization we call "wire threading" that allows us to reduce the number of L1-cache misses during evaluation. We multithread the MPC evaluation of LowMC, allowing us to evaluate the PRF 6.7M times per second. See Appendix C for further details.

We adapt the Alibi reinsertion technique [16] for "caching the stash" to the distributed setting, and we provide the first implementation of Alibi. See Appendix D for further details.

5 ShufTable: reduce $\mathsf{numRoundsOHTable}$

We present ShufTable, a novel, oblivious to build, asymptotically and practically efficient, oblivious to distinct-query, hash table (OHTable). Our chief goal with ShufTable is to minimize the round-complexity of a query, i.e., decreasing $\mathsf{numRoundsOHTable}$ without blowing up any other costs. With ShufTable we have $\mathsf{numRoundsOHTable} \approx 7$ (compared to [17] which required 45 rounds of communication for each OHTable query).

In Section 8 we use the hierarchical solution and [16] to compile ShufTable into an efficient DORAM. The resulting DORAM requires only a constant number of rounds per query, whereas prior hierarchical DORAM protocols (e.g. [17]) had a round complexity that scaled with N.

In Section 5.1 we present the "Persistent shuffle protocol," a method that enables $\Pi_{\mathsf{ShufTable}}$, the new OHTable protocol we present in Section 5.2. Leveraging features of $\Pi_{\mathsf{ShufTable}}$.Query, in Section 5.3 we show how to amortize the round cost of SISO-PRF evaluations across $L_1, \dots, L_{\mathsf{numLevels}}$.

5.1 Persistent shuffle Protocol

Like many DORAM protocols, our DORAM construction relies on an efficient oblivious shuffle, which allows players holding a secret shared list, $\llbracket X \rrbracket = \llbracket X_1 \rrbracket, \dots, \llbracket X_n \rrbracket$ to shuffle $\llbracket X \rrbracket$ under some random permutation $\pi \in S_n$ such that *nothing about* π *is learned by any player*.

As presented in [29], there is an efficient, linear-communication (3,1)-oblivious shuffle which works as follows. The players P_1, P_2, P_3 reshare $\llbracket X \rrbracket$ to P_1, P_2, who shuffle their secret shares according to some random agreed upon permutation, and reshare the shuffled list to P_2, P_3, who shuffle and reshare to P_3, P_1 who shuffle and reshare to P_1, P_2, P_3. Since the composition of permutations is not known to any single player, the final permutation is oblivious. Unfortunately, at the end of the [29] protocol, information about the permutation, π, is not accessible to the players. In our Persistent shuffle (described in Figure 1), we augment the shuffling protocol to also output a secret sharing of π.

Cost of the Persistent shuffle protocol. The Persistent shuffle protocol requires 4 rounds of communication and each round requires $n(|X_i| + \log n)$ bits of communication. For comparison, shuffling n elements using Persistent shuffle is 6 times less bandwidth than evaluating a SISO-PRF (using LowMC) on those elements, so shuffling contributes only minimally to the overall communication cost of our DORAM.

Lemma 5.1. *Persistent shuffle (described in Figure 1 fulfills the functionality $\mathcal{F}_{\mathrm{ABB}}$.ObliviousShuffle($\circ$, $\mathsf{DistributeShuffle} = True$) where the players input $\llbracket X \rrbracket$ and receive as output $\llbracket \pi(X) \rrbracket$ and $\llbracket K \rrbracket = \llbracket K_1 \rrbracket, \dots, \llbracket K_n \rrbracket$ s.t $\pi(i) = K_i$. It also guarantees that π is uniformly sampled from S_n and unknown to all players.*

Proof. In step 1, the 3 participants reshare the shares of the vector X to and the indices $L = (1, 2, \dots, n)$ to two participants. By the security of $\mathcal{F}_{\mathrm{ABB}}$.ReshareReplicatedTo2Sharing, each individual participant learns nothing about X (in fact each player's shares are uniformly random). In steps 2 & 3, pairs of players locally shuffle their shares and reshare to the next pair. In this case, the security of $\mathcal{F}_{\mathrm{ABB}}$.Reshare2To2Sharing ensures that each player learns nothing about the underlying data (the shuffled values). As in step 1, each player receives uniformly random shares and nothing else. In step 4, pairs of players call $\mathcal{F}_{\mathrm{ABB}}$.Reshare2SharingToReplicated, and again, each player's view of the protocol is a collection of uniformly random shares.

Setup: Each pair of players P_i, P_j agree on a random permutation $\pi_{\{i,j\}} \in S_n$. The players also generate a sharing $[\![L]\!] = [\![1]\!], \ldots, [\![n]\!]$.
Protocol:

1. The players reshare to the first shufflers, calling

$$[X]^{(1,2)} = \mathcal{F}_{\text{ABB}}.\text{ReshareReplicatedTo2Sharing}([\![X]\!], \{P_1, P_2\})$$

and

$$[L]^{(3,1)} = \mathcal{F}_{\text{ABB}}.\text{ReshareReplicatedTo2Sharing}([\![L]\!], \{P_3, P_1\})$$

2. Shuffle & Reshare to next shuffling pair #1:

   (a) P_1, P_2 let $[X']^{(1,2)} = [\pi_{\{1,2\}}(X)]^{(1,2)}$. Note that since P_i, P_j hold $[X]^{(i,j)}$ they can obtain $[\pi(X)]^{(i,j)}$ for a known π by locally shuffling their list of secret shares. P_1, P_2 reshare X' to the next shufflers, calling

$$[X'']^{(2,3)} = \mathcal{F}_{\text{ABB}}.\text{Reshare2To2Sharing}([X']^{(1,2)}, \{P_2, P_3\}).$$

   (b) P_3, P_1 let $[L']^{(3,1)} = [\pi^{-1}_{\{3,1\}}(L)]^{(3,1)}$. P_1, P_2 reshare L' to the next shufflers, calling

$$[L'']^{(2,3)} = \mathcal{F}_{\text{ABB}}.\text{Reshare2To2Sharing}([L']^{(3,1)}, \{P_2, P_3\})$$

3. Shuffle & Reshare next shuffling pair #2:

   (a) P_2, P_3 let $[X'']^{(2,3)} = [\pi_{\{2,3\}}(X')]^{(2,3)}$. P_2, P_3 reshare X'' to the next shufflers, calling

$$[X'']^{(3,1)} = \mathcal{F}_{\text{ABB}}.\text{Reshare2To2Sharing}([X']^{(2,3)}, \{P_3, P_1\}).$$

   (b) P_2, P_3 let $[L'']^{(2,3)} = [\pi^{-1}_{\{2,3\}}(L')]^{(2,3)}$. P_2, P_3 reshare L'' to the next shufflers, calling

$$[L'']^{(1,2)} = \mathcal{F}_{\text{ABB}}.\text{Reshare2To2Sharing}([L'']^{(1,2)}, \{P_1, P_2\}).$$

4. Shuffle & Reshare back to all players #3:

   (a) P_3, P_1 let $[X''']^{(3,1)} = [\pi_{\{3,1\}}(X'')]^{(3,1)}$. P_3, P_1 reshare X''' back to the entire group, calling

$$[\![X''']\!] = \text{Reshare2SharingToReplicated}([X''']^{(3,1)}).$$

   (b) P_1, P_2 let $[L''']^{(1,2)} = [\pi^{-1}_{\{1,2\}}(L'')]^{(1,2)}$. P_1, P_2 reshare L''' back to the entire group, calling

$$[\![L''']\!] = \text{Reshare2SharingToReplicated}([L''']^{(1,2)}).$$

Output: The output of the protocol is $[\![\pi(X)]\!], [\![\pi^{-1}(L)]\!]$.

Figure 1: The Persistent shuffle protocol, $\mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}([\![X]\!], \mathsf{DistributeShuffle} = \mathit{True})$

The final list is a sharing of $\pi(X)$, where $\pi = \pi_{\{2,3\}} \odot \pi_{\{1,2\}}$. Every player does not know either $\pi_{\{2,3\}}$ or $\pi_{\{1,2\}}$, so the resulting permutation is uniformly random from the perspective of any player. $\square$

5.2 ShufTable construction

In this section, we describe and evaluate $\Pi_{\text{ShufTable}}$.Build and $\Pi_{\text{ShufTable}}$.Query, the component of our new Oblivious Hash Table ShufTable. The pseudocode for Protocol $\Pi_{\text{ShufTable}}$ is presented in Figure 2.

Parameters. $\Pi_{\text{ShufTable}}$ is parameterized by N, the size of the address space, D, the size of each payload, κ, the computational security parameter, n, the number of real elements stored in the table, $\mathsf{numDummies}$, the number of dummy elements stored in the table, and $\mathsf{stashSize}$, the minimum size of a stash in cuckoo hash table storing n elements such that the probability of a build failure is less than our statistical security parameter, σ.

On each query, we must be able to retrieve either (1) the element being queried or (2) a new dummy element that has never been queried (if the desired element is not in the OHTable). Hence, the number of queries to each ShufTable is bounded by $\min\{\mathsf{numDummies}, n\}$. For this reason, we usually set $\mathsf{numDummies} \approx n$.

Since our construction uses a Cuckoo hash table (CHT), it as also implicitly parameterized by c, the number of slots in CHT table, and t, the number of such tables. Instantiations of these variables are discussed in Section 9.

Input-Output behavior of $\Pi_{\text{ShufTable}}$.Build. The parties, P_1, P_2, P_3, input $E = \{([\![X_1]\!], [\![Y_1]\!]), \ldots, ([\![X_n]\!], [\![Y_n]\!])\}$ where $n \leq N$ to $\Pi_{\text{ShufTable}}$.Build. Protocol $\Pi_{\text{ShufTable}}$.Build, outputs secret shares of CHT held by P_2, P_3, $[\mathsf{CHT}]^{(2,3)}$, and $[\![\hat{\hat{X}}]\!], [\![\hat{\hat{Y}}]\!]$, special shuffling of the elements under a permutation somewhat known to P_2, P_3 (Step 7 of $\Pi_{\text{ShufTable}}$.Build). The data structure, ShufTable, stores a subset of the elements sent to the build protocol $A \subset E$. $\Pi_{\text{ShufTable}}$.Build also outputs $\mathsf{Stash} = E \setminus A$, with $|\mathsf{Stash}| \leq \mathsf{stashSize}$. In the greater DORAM protocol (Section 8), Stash will be inserted into the cache, L_0. As long as $X_1, \ldots, X_n$ are distinct (as assured by the hierarchical solution), the parties learn nothing of $(X_1, Y_1), \ldots, (X_n, Y_n)$ or which elements belong to Stash.

Input-Output behavior of $\Pi_{\text{ShufTable}}$.Query. For the t'th query where $1 \leq t \leq \mathsf{numDummies}$, the players input $[\![x_t]\!]$ to $\Pi_{\text{ShufTable}}$.Query. $\Pi_{\text{ShufTable}}$.Query, outputs $[\![Y_j]\!]$ if $x_t = X_j$ for $(X_j, Y_j) \in A$ and $[\![\bot]\!]$ otherwise (and outputs $[\![\mathsf{found}]\!]$ accordingly). The security guarantee is that for any sequence of *distinct* queries, $x_1, \ldots, x_t$, P_1, P_2, P_3 do not learn whether Y_j or $\bot$ was outputted, j, or Y_j.

Performance analysis. The cost of $\Pi_{\text{ShufTable}}$.Build is dominated by the cost of n SISO-PRF evaluations.[7] With our implementation of LowMC, this requires a total of $2304n$ bits of communication at $\kappa = 128$ and the computation of many XORs. Despite this cost, $\Pi_{\text{ShufTable}}$.Build is very efficient, since via multi-threading LowMC (Section C) we are able to evaluate 6.7M SISO-PRFs/s in our tests.

The time needed to evaluate $\Pi_{\text{ShufTable}}$.Query is driven by its round complexity, since we must sequentially invoke this protocol $\mathsf{numLevels}$ times during Π_{DORAM}.Query. Each step of the query requires equality checks on secret shared elements, which we execute using 3-party garbled circuits. [8]

[7]For comparison, [17] requires more than twice as many SISO-PRF evaluations ($2n + \mathsf{numDummies}$) for each OHTable build.

Lemma 5.2. *$\Pi_{\text{ShufTable}}$ implements the distinct-query oblivious hash table functionality.*

Proof. We begin by showing that the view of each player during $\Pi_{\text{ShufTable}}$.Build is independent of the input data X, Y. During $\Pi_{\text{ShufTable}}$.Build, P_2, P_3 only receive secret shares and operate on those secret shares using MPC. Since secret shares are sampled from a (individually) uniformly random distribution, P_2, P_3's views are independent of underlying data X, Y.

During $\Pi_{\text{ShufTable}}$.Build, other than operating on secret shares, P_1 receives the list $\hat{Q}$ in the clear. By assumption, X has no repetitions, so the PRF "tags" Q appear uniformly random and independent to a computationally bounded adversary. Since $\hat{Q}$ is a uniformly random shuffle of Q which P_1 does not know, P_1 cannot decipher *any* information about X from seeing $\hat{Q}$. Moreover, when deciding which elements to place in Stash (by deciding on which indexes to place there), P_1 cannot tell which elements from (X_i, Y_i) he is placing in stash (except that they aren't dummies since $\hat{Q}_i \neq \bot$). Thus, P_1's view is independent of X, Y and the stash contains uniformly random real elements.

Next we show that in $\Pi_{\text{ShufTable}}$.Query the view of each player is independent of the query X_t, the output, or X, Y previously stored. During $\Pi_{\text{ShufTable}}$.Query, other than operating on secret shares, P_2, P_3 learn q in the clear. q is either a random or a pseudorandom element which is thus computationally independent from P_2, P_3's view (because P_2, P_3 did not see $\hat{Q}$ which P_1 used to build $\mathsf{CHT} \cup \mathsf{Stash}$). Thus, P_2, P_3 do not learn anything by seeing q. Additionally, P_2, P_3 learn l, an index into the $\wedge$-shuffled list which depends on X_t's presence at the level (see step 4 of $\Pi_{\text{ShufTable}}$.Query). Yet, since l is entirely determined by $\pi_{\wedge}$ which P_2, P_3 don't know, and moreover, since unlike P_1 they do not know which indexes of $\hat{Y}$ correspond to dummies, l is sampled from a distribution indistinguishable from random relative to the individual view of P_2, P_3.

Finally, during $\Pi_{\text{ShufTable}}$.Query, other than operating on secret shares, P_1 learns l in the clear. Yet, although P_1 knows which elements from $\hat{Y}$ are dummies, due to an additional oblivious shuffle, he does not know which elements of $\hat{\hat{Y}}$ are

[8]We use 3-party garbled circuits rather than the (constant-round) 3-party BMR protocol [7] implemented in EMP because 3-party garbled circuits are significantly more bandwidth efficient than BMR. 3-party garbled circuits require honest majority, while BMR does not.

$\Pi_{\textbf{ShufTable}}$**.Build:** The players (refers to P_1, P_2, P_3) hold $([\![X_1]\!], [\![Y_1]\!]), \ldots, ([\![X_n]\!], [\![Y_n]\!])$.

1. The players create dummies to satisfy queries that were not found, letting $[\![Y_i]\!] = [\![\bot]\!]$, $[\![X_i]\!] = [\![\bot]\!]$ for $i \in \{n+1, \ldots, n+\mathsf{numDummies}\}$.
2. The players generate a κ-bit secret-shared PRF key, evaluating $[\![k]\!] = \mathcal{F}_{\text{ABB}}.\text{RandomElement}(\kappa)$.
3. The players create pseudorandom tags for all the addresses, evaluating $[\![Q_i]\!] = \mathcal{F}_{\text{ABB}}.\text{PRFEval}([\![k]\!], [\![X_i]\!])$ for $i \in \{1, \ldots, n\}$ and $[\![Q_i]\!] = [\![\bot]\!]$ for $i \in \{n+1, \ldots, n+\mathsf{numDummies}\}$.
4. Players obliviously shuffle the lists before revealing Q to P_1 to hide information about the elements in the table/stash, executing $[\![\hat{Q}]\!], [\![\hat{X}]\!], [\![\hat{Y}]\!], [\![j]\!] = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}([\![Q]\!], [\![X]\!], [\![Y]\!], \mathsf{DistributeShuffle} = \mathit{True})$ (c.f. Section 5).
    (a) The players locally create $[\![\mathsf{DI}_i]\!] = [\![j_{n+i}]\!]$ for all $i \in [\mathsf{numDummies}]$. If $\mathsf{DI}_i = k$, that implies $X_k = n+i$ and $Y_k = \bot$. That is, the kth element of $(\hat{X}, \hat{Y})$ is the ith dummy. We use DI in step 4 of query to return a dummy if needed.
5. Revealing Q_i to P_1 so that it can build a $\mathsf{CHT} \cup \mathsf{Stash}$ containing X_i using fast, local random accesses without learning X_i, the players call $\hat{Q}_i = \mathcal{F}_{\text{ABB}}.\text{RevealTo}(P_1, [\![\hat{Q}_i]\!])$ for $i \in \{1, \ldots, n+\mathsf{numDummies}\}$
6. P_1 locally constructs the Cuckoo hash table and list of stashed indices $\mathsf{CHT} \cup \mathsf{Stash} = \mathsf{BuildCHTwS}((\hat{Q}_i || i)_{i \in [n]}, \mathsf{stashSize})$. CHTstores $\hat{Q}_i || i$ for $n - \mathsf{stashSize}$ such different i's where $||$ represents bit-wise appending.
    (a) P_1 secret shares CHT between P_2 and P_3, running $[\mathsf{CHT}]^{(2,3)} = \mathcal{F}_{\text{ABB}}.\mathsf{InputTo2Sharing}(\mathsf{CHT}, P_2, P_3)$. P_2, P_3 will use CHT to satisfy queries.
    (b) P_1 sends Stash, a cleartext list of $\mathsf{stashSize}$-many indexes, s.t. $i \in \mathsf{Stash}$ indicates that $([\![\hat{X}_i]\!], [\![\hat{Y}_i]\!])$ is stashed. If $i \in \mathsf{Stash}$ then $X_i \neq \bot$. **Output** $X^{\text{stash}} = \{[\![\hat{X}_i]\!]\}_{i \in \mathsf{Stash}}$ and $Y^{\text{stash}} = \{[\![\hat{Y}_i]\!]\}_{i \in \mathsf{Stash}}$. These will be reinserted into the cache when $\Pi_{\text{ShufTable}}$.Build is called as part Π_{DORAM} (Figure 4).
7. The players shuffle the data under under a permutation, $\hat{\hat{\pi}}$, only known to P_2 and P_3, executing $[\![\hat{\hat{X}}]\!][\![\hat{\hat{Y}}]\!] = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}(([\![X]\!], [\![\hat{Y}]\!]), \text{RevealTo} = \{2,3\})$. This "rebalances the information asymmetry," allowing P_2, P_3 to guide P_1 to respond to queries (see step 5 of query) s.t. P_1 cannot use his privileged information from $\Pi_{\text{ShufTable}}$.Build, not learning anything about the query.

$\Pi_{\textbf{ShufTable}}$**.Query:** The players hold $[\![Q_{query}]\!] = \mathcal{F}_{\text{ABB}}.\text{PRFEval}([\![X_t]\!], [\![k]\!])$ (Section 5.3). ***Each step corresponds to a single round of communication.*** We pack parallelizable/silent instructions into the same step.

1. First, the players compute $[\![r]\!] = \mathcal{F}_{\text{ABB}}.\text{RandomElement}(\kappa)$ (silent generation of random κ-bit secret share) and then compute $[\![q]\!] = [\![Q_{query}]\!] + [\![\mathsf{useDummy}]\!] \cdot [\![r]\!]$ (one MPC multiplication). $\mathsf{useDummy}$ is an artifact of the hierarchical solution that indicates if X_t was already found in previous level.
2. The players reveal enable P_2, P_3 to query $[\mathsf{CHT}]^{(2,3)}$ by revealing q to them, evaluating, $q = \mathcal{F}_{\text{ABB}}.\text{RevealTo}([\![q]\!], P_2, P_3)$.
3. P_2, P_3 input $[\mathsf{CHT}[h_1(q)]]^{(2,3)}, [\mathsf{CHT}[h_2(q)]]^{(2,3)}$ their secret-shares of locations in CHT where q might be stored, calling $[\![q'_b \mid\mid i'_b]\!] = \mathcal{F}_{\text{2Share}}.\text{Reshare2to3WithoutCheck}([\![T[h_b(Q)]]\!])$ for $b = 1, 2$.
4. The players evaluate $\mathsf{QueryCircuit}$ using (3,1) garbled circuits (see [32]), evaluating $l = \mathcal{F}_{\text{ABB}}.\text{EvalCircuit(3,1)GC}(\mathsf{QueryCircuit}, \text{Inputs} = [\![q'_1]\!], [\![i'_1]\!], [\![q'_2]\!], [\![i'_2]\!], [\![q]\!], [\![\mathsf{DI}_t]\!])$, *revealing l only to P_2, P_3*. $\mathsf{QueryCircuit}$ returns i'_1 if $q'_1 = q$, returns i_2 if $Q'_2 = q$, and returns DI_t otherwise.
5. P_2, P_3 send $j = \hat{\hat{\pi}}(l)$ to P_1. The parties set $[\![Y_{output}]\!] = [\![\hat{\hat{Y}}_j]\!]$ and append j to list $\mathsf{queriedDblhatIdxs}$

$\Pi_{\textbf{ShufTable}}$**.Extract:** Output $([\![\hat{\hat{X}}_i]\!], [\![\hat{\hat{Y}}_i]\!])$ for $i \in ([n+\mathsf{numDummies}] - \mathsf{queriedDblhatIdxs})$

Figure 2: $\Pi_{\text{ShufTable}}$.Build and $\Pi_{\text{ShufTable}}$.Query.

dummies (or were stashed, or were stored in the table, etc). Hence l is independent of P_1's view.

Thus, on distinct queries and when $i \neq j \implies X_i \neq X_j$, P_1, P_2, and P_3's views are independent of the data stored and queried, and hence ShufTable is distinct-query oblivious. □

5.3 Parallelizing sequential SISO-PRF evaluations

In the hierarchical ORAM construction, each query requires searching *every* level in the hierarchy, and the OHTable query at a given level requires evaluating a SISO-PRF.

In previous constructions (e.g. [17, 31]) these SISO-PRF were evaluated sequentially, because when performing the OHTable query for an index, x, at level i, the SISO-PRF input will be $[\![x]\!]$ or a dummy element depending on whether x was found in a smaller level of the hierarchy.

Since there are only two possible SISO-PRF inputs at each level, rather than evaluating the PRF sequentially, the players could evaluate *both* the "dummy query"

$$[\![a_i]\!] = \mathcal{F}_{\text{ABB}}.\text{PRFEval}([\![N+t_i]\!], [\![k]\!])$$

and the "real query"

$$[\![b_i]\!] = \mathcal{F}_{\text{ABB}}.\text{PRFEval}([\![X_{query}]\!], [\![k]\!])$$

for all $i \in \{1, \ldots, \mathsf{numLevels}\}$ in parallel before querying L_1, then multiplex the result using MPC (which costs only a single round) before evaluating L_i.Query.

This trick will reduce the *round* complexity, but requires *doubling* the number of SISO-PRF calls per query. Since round-complexity is often the bottleneck, this improves practical performance.

In our protocol, however, we can leverage the design of ShufTable to parallelize the SISO-PRF calls *without* increasing communication.

The crucial observation is that we have designed ShufTable to require only $[\![b_i]\!]$, regardless of found. That is, since previous OHTables stored and retrieved their dummy elements from some data structure, obtaining the index of each dummy from a_i was needed. We observe that it is not necessary: using the $(3,1)$ garbled circuits of [32] we "just-in-time" detect if q_i is stored in the table and output a dummy index if necessary ($\Pi_{\text{ShufTable}}$.Query, step 4). Thus, if found_{i-1}, for ShufTable it is sufficient to set q_i to be some uniformly random κ-bit value r ($\Pi_{\text{ShufTable}}$.Query, step 1), which, except from with negligible probability, will not be stored in the table and will yield the desired dummy-output. This "just-in-time" trick is largely enabled by the Persistent shuffle trick we present in Section 5.1.

Thus to parallelize the SISO-PRF, we evaluate $[\![b_i]\!] = \mathcal{F}_{\text{ABB}}.\text{PRFEval}([\![X_{query}]\!], [\![k]\!])$ for all $i \in \{1, \ldots, \mathsf{numLevels}\}$ in parallel before querying L_1. Hence we can parallelize the SISO-PRF evaluations across the table without increasing the total number of SISO-PRF calls. This can be seen in Step 1.b of Π_{DORAM}.ReadAndWrite, Figure 4.

This optimization reduces $\mathsf{numRoundsDORAM}$ by $(\mathsf{numLevels}-1)\cdot\mathsf{numRoundsPRFEval}$. In our implementation $\mathsf{numLevels} \approx 5$ and $\mathsf{numRoundsPRFEval} = 9$, so this saves us ≈ 45 rounds per DORAM query.

6 Reducing the depth of the hierarchy

In most hierarchical ORAM solutions, each level is twice as large as the level above it, i.e., $|L_{i+1}| = 2\cdot|L_i|$. Since we must have $|L_{\mathsf{numLevels}}| = O(N)$ and generally $|L_0|$ is a small constant, this means that $\mathsf{numLevels} = O(\log N)$. In our protocol, we introduce a tuneable "base amplification factor" denoted $\mathsf{baseAmpFactor}$, and set $|L_{i+1}| \approx \mathsf{baseAmpFactor}\cdot|L_i|$. This change reduces $\mathsf{numLevels}$ by a factor of $\log_2(\mathsf{baseAmpFactor})$ (but increases communication by a factor of $\mathsf{baseAmpFactor}/\log(\mathsf{baseAmpFactor})$). In our testing, we find that increasing $\mathsf{baseAmpFactor}$ dramatically improves practical performance because latency is significantly more time-expensive than bandwidth (Section 9). For instance, for GigaDORAM we found that increasing $\mathsf{baseAmpFactor}$ with N to maintain that $|L_0| \approx 2^8$ and $\mathsf{numLevels} \approx 4$ yielded the best performance. For $N = 2^{30}$, this meant setting $\mathsf{baseAmpFactor} = 2^7$, which is much larger than all previous protocols, which implicitly set $\mathsf{baseAmpFactor} = 2$.

In most hierarchical ORAM schemes (where $\mathsf{baseAmpFactor} = 2$), when levels $L_0, \ldots, L_i$ are full, they are reshuffled and rebuilt into L_{i+1} Since

$$\sum_{j=0}^{i} |L_j| = \sum_{j=0}^{i} 2^j |L_0| = (2^{i+1}-1)|L_0|$$

this rebuild schedule works nicely when $\mathsf{baseAmpFactor} = 2$.

In our generalization, we also rebuild levels when they are full, but for $\mathsf{baseAmpFactor} > 2$ we have that

$$\mathsf{baseAmpFactor}^{i+1}\cdot|L_0| \gg \sum_{j=0}^{i} \mathsf{baseAmpFactor}^{j}|L_0|$$

thus, we must slightly adjust our rebuild schedule. In particular we must accommodate for "partial rebuilds." We formalize this procedure in protocol Π_{DORAM}.Rebuild presented in Section 8, Figure 4.

Since rebuilding level $i+1$ costs $O\left(\kappa\cdot|L_0|\cdot\mathsf{baseAmpFactor}^{i+1}\right)$ communication and computation, the total amortized (re)build cost of the DORAM is at most

$$\sum_{i=1}^{\frac{\log(N)}{\log(\mathsf{baseAmpFactor})}} \frac{O\left((\kappa+D)\cdot|L_0|\cdot\mathsf{baseAmpFactor}^{i+1}\right)}{|L_0|\cdot\mathsf{baseAmpFactor}^{i}}$$
$$= O\left(\frac{(\kappa+D)\cdot\mathsf{baseAmpFactor}\cdot\log(N)}{\log(\mathsf{baseAmpFactor})}\right)$$

Since the original amortized (re)build cost of all $O(\log N)$ levels of the DORAM is $O((\kappa + D)\log N)$, the cost of decreasing $\mathsf{numLevels}$ by a factor of $\log_2(\mathsf{baseAmpFactor})$ is a factor of $O(\mathsf{baseAmpFactor})/\log(\mathsf{baseAmpFactor}))$ increase in communication and computation.

7 SpeedCache: Larger, optimized cache

Most hierarchical (D)ORAMs use a constant-sized "cache," i.e., $|L_0| = O(1)$. Since each ORAM query performs a linear scan over the cache (implying a linear communication, computation, and number of rounds), a large cache can significantly hurt both asymptotics and practical performance. In our protocol, we use the Alibi reinsertion technique to "cache-the-stash" from Cuckoo hash tables at each level of the ORAM hierarchy. This means that we must have a moderately large cache size, otherwise reinserting the stashes would fill the stash and trigger an infinite chain of table rebuilds.

Thus we increase the cache size to $|L_0| = \Omega(\mathsf{stashSize})$. This has the additional advantage of eliminating "small" levels from the ORAM hierarchy since $|L_i| > |L_0|$ for every level in the hierarchy.[9]

In order to facilitate our new, larger cache without hurting round complexity, we create, SpeedCache, that can be queried using $O((D + \log N) \cdot |L_0|)$ communication and only requires $\lceil \log\log N \rceil + 1$ rounds of communication (that is, *independent of the number of elements in the cache* – only dependent on the size of the address space). We implement our SpeedCache protocol in the 3-party MPC framework of [2]. We present the protocol, $\Pi_{\mathsf{SpeedCache}}$, in Figure 3.

Since the SpeedCache protocol just uses MPC to compute on secret-shared values, the security of the SpeedCache protocol follows immediately from the security of the underlying MPC protocol.

Experimentally we find that it is optimal to maintain $|L_0| \approx 2^{10}$ a constant, rather than scaling $|L_0|$ with N. Note that $2^{10} > \log(N)$ for any conceivable value of N. Concretely, setting $|L_0| \approx 2^9$ as we do in practice allows us to reduce $\mathsf{numLevels}$ by $\approx 8/\log_2(\mathsf{baseAmpFactor})$. We found that changing $|L_0|$ useful for adapting the performance of GigaDORAM to different network settings. For instance when latency is high, a larger $|L_0|$ yields better performence.

An alternative approach would be to implement the trivial linear-scan cache (see Appendix B) via garbled circuits in a constant number of rounds. When $|L_0| \approx 2^{10}$ we estimate this would make L_0.Query up to $100\times$ slower than our implementation of SpeedCache.

[9]Because cuckoo hash tables have higher build-failure probability when the table size is small, many hierarchical (D)ORAMs had two types of tables, one for the "small" levels and one for the "large" levels (e.g. [17, 31]). Because our L_0 is sufficiently large, we do not incur this complexity.

8 Full DORAM protocol $\Pi_{\mathbf{DORAM}}$

We give the GigaDORAM protocol, Π_{DORAM}, in Figure 4.

Round-complexity: With our optimizations the final round complexity of Π_{DORAM}.Query is

$$\begin{aligned}
&\mathsf{numRoundsDORAM} = \\
&\quad \mathsf{numLevels} \cdot \mathsf{numRoundsOHTable} \\
&\quad + \max(\mathsf{numRoundsCache}, \mathsf{numRoundsPRFEval}) + 1 \\
&\quad = \mathsf{numLevels} \cdot 5 + \max(\log\log(N) + 2, 9) + 1 \\
&\quad \approx 5 \cdot 5 + 9 + 1 = 35 \qquad (\text{when } N = 2^{31})
\end{aligned}$$

The term $\max(\log\log(N) + 2, 9)$ occurs because L_0.Query ($\log\log(N) + 2$ rounds) and the SISO-PRF pre-evaluations (9 rounds – see Section 5.3, Appendix C) can be evaluated in parallel. Hence, compared to the over 2500-round-per-query [17] construction we draw inspiration from, GigaDORAM has a $\approx 70\times$ reduction in round complexity. Our overall *communication* and *computation* complexity is comparable to that of [17, 31] and previous hierarchical DORAMs.

Lemma 8.1. *Protocol Π_{DORAM} securely realizes the DORAM functionality in the semi-honest (3,1)-setting.*

Proof. Protocol Π_{DORAM} is a straightforward application of the hierarchical solution to our building blocks, $\Pi_{\mathrm{ShufTable}}$ and $\Pi_{\mathrm{SpeedCache}}$ in the distributed setting. The security of the hierarchical solution is standard (see e.g. [37]).

The proof proceeds as follows. Since ShufTable will be oblivious as long as no queries are repeated, the primary goal is to show that even if the client makes repeated queries into the DORAM, no OHTable is ever queried twice between rebuilds.

To see this, suppose the user makes a series of queries, $x_1, \ldots, x_t$ and $x_{i_1} = x_{i_2}$ with $i_1 < i_2$. Suppose x_{i_1} was initially at level ℓ. In this case, after query x_{i_1}, the key-value pair (x_{i_1}, y_{i_1}) will be inserted into the cache. As later queries come in and the cache becomes full (x_{i_1}, y_{i+1}) may get pushed to lower levels of the hierarchy. After sufficiently many queries, levels $i = 1, \ldots, \ell - 1$ will be rebuilt into level ℓ. If query x_{i_2} happens *before* level ℓ is rebuilt, x_{i_2} will be found at a level above level ℓ, and at level ℓ a (new) dummy element will be queried. If query x_{i_2} happens *after* level ℓ has been rebuilt, then at this point x_{i_1} has never been queried against this new OHTable. $\square$

9 Evaluation

In this section we evaluate the practical performance of GigaDORAM and compare its performance to previous DORAM implementations. To make the comparisons fair, whenever possible we evaluated *on the same hardware setup*. To make our results easily reproducible, we test on AWS, using c5n.metal instances with 72vCPUs and 192GiB memory.

$(\star)$ symbolizes the invariant that when $\Pi_{\text{SpeedCache}}$.Query is called $i \neq j \implies x_i \neq x_j \vee x_i = x_j = \perp$.
$\Pi_{\textbf{SpeedCache}}$.**Init**()**:** The players set the query counter $t = 0$. No communication.
$\Pi_{\textbf{SpeedCache}}$.**Store**($[\![x]\!], [\![y]\!]$)**:** Let $[\![x_t]\!] = [\![x]\!]$ and $[\![y_t]\!] = [\![y]\!]$. No communication.
$\Pi_{\textbf{SpeedCache}}$.**Query**($[\![x]\!]$)**:** $([\![x_1]\!], [\![y_1]\!]), \ldots, ([\![x_t]\!], [\![y_t]\!])$ are stored where x_i is unique. Note that this invariant, $(\star)$, inductively holds: On Init $(\star)$ trivially holds. On Store(x) at the end of Π_{DORAM}.ReadAndWrite, if x was stored at L_0, it was zeroed-out in Step 4 of $\Pi_{\text{SpeedCache}}$.Query (below) and thus $(\star)$ holds. On alibi-reinsertion from L_i since L_i.Build($[\![X]\!], [\![Y]\!]$), $(\star)$ holds because X is guaranteed to be a unique list and thus its reinserted subset also unique.

1. Using MPC, the players compute equality-indicators $[\![b_i]\!]$ for all $i \in [t]$ s.t $b_i = 1$ iff $x = x_i$. This takes $\lceil \log \log N \rceil$ rounds and $t \log N$ communication, because each x_i is $\log N$ bits.
2. Using MPC, the players zero-out all the non-queried elements, computing the bit-vector product $[\![x_i']\!] = [\![x_i']\!] \cdot [\![b_i]\!]$ and $[\![y_i']\!] = [\![y_i]\!] \cdot [\![b_i]\!]$ for all $i \in [t]$. This takes a single round and $n(\log N + D)$ communication.
3. The players output $\bigoplus_{i=1}^{n} [\![y_i']\!]$ which is $[\![y_i]\!]$ if $x = x_i$, else $[\![\perp]\!]$ This costs no communication or rounds.
4. Preparing for the next query, the players zero-out (put a dummy) at the location where x_i was found (if it was found) by setting $[\![x_i]\!] := [\![x_i]\!] \oplus [\![x_i']\!]$ for all $i \in [n]$.

$\Pi_{\textbf{SpeedCache}}$.**Extract:** Output $([\![x_1]\!], [\![y_1]\!]), \ldots, ([\![x_t]\!], [\![y_t]\!])$, set $t = 0$. No communication.

Figure 3: The SpeedCache protocol, $\Pi_{\text{SpeedCache}}$

We measure only writes in our experiments because "reads" vary across works (e.g. DuORAM separates "dependent" and "independent" reads) and for all constructions, writes are at least as expensive as reads. For each construction at each N we averaged performance across a minimum of 512 writes (and up to 100K writes in faster networks). All experiments were on $D = 64$ bit payloads as in DuORAM's testing.

To compare fairly with previous works, we *re-benchmark all previous DORAMs with* existing implementations (with the exception of proprietary [25]) *on the same hardware setup we used to test GigaDORAM*. To the best of our knowledge, this is the most comprehensive Distributed ORAM benchmark to date.[10] See Appendix E for details.

9.1 Low-latency tests

One of the primary applications of DORAM is in heavy-compute large-scale MPC computations (where random access could outperform circuit-based computations). Thus we focus on low-latency network environments where large scale MPC becomes practical.[11]

With this motivation, we begin by benchmarking GigaDORAM in a realistic low-latency regime. Specifically, we deploy to three cluster-placed AWS c5n.metal instances.[12] In this setup, In cluster placement at region `us-west-2`. Via `ping -c 20` we measure latency of `min/avg/max/mdev = 0.229/0.236/0.244/0.003 ms`.[13]

Figure 5 shows that in this low-latency environment GigaDORAM outperforms all other DORAMs for $2^{12} \leq N \leq 2^{31}$. GigaDORAM perform $700-900$ queries/sec for all measured N, with oscillations which depend on the size of the hierarchical cache L_0.[14]

9.2 Testing in restricted networks

Previous DORAM papers benchmarked [15, 25, 43] under artificial restrictions to the network. For example, [43] benchmarked restricted their bandwidth to 100Mb/s with a latency of 30ms. These network conditions are lower than you would expect from geographically separated machines. For example, we measured 20ms ping from `us-east-1` to `us-west-2`, servers in different AWS regions should enjoy several Gbit

[10]We do not benchmark [28], because it is based on the proprietary Sharemind software, and is focused on batch queries. As noted in that work, in the single-query setting those techniques are not expected to give significant improvements over straightforward "ORAM in MPC" schemes.

[11]Although MPC is possible under poor network conditions [46], it is only possible when the circuits are fairly simple and thus would not benefit significantly from random access. To put this in perspective, [46] benchmark computing the AES circuit under MPC. In GigaDORAM, we compute AES (or LowMC) under a circuit-based MPC *multiple times for each query*. Thus no DORAM that makes use of a SISO-PRF could not possibly improve the performance of a simple computations like AES.

[12]For some comparisons (e.g. [24, 43]) we used existing Docker setups to simulate three parties on a single c5.metal machine in which case we adjust the network via the `tc` command (see Section E).

[13]We observed latency fluctuating according to usage, although AWS does not make such policies public.

[14]The "oscillations" seen in our performance are due to the fact $\log_2 |L_0| = N - \mathsf{baseAmpFactor} \cdot (\mathsf{numLevels} - 1)$ and the fact that $\mathsf{baseAmpFactor}$ is a power of 2, and thus for certain values of N we cannot reasonably set $|L_0|$ as small as we'd like.

Π_{DORAM}.Init($[\![Y]\!]$):

1. The players input the address of each payload $[\![X_i]\!] = \mathcal{F}_{\text{ReplicatedMPC}}$.InputConstant$(i)$ for $i = 1$ to N. The players build the toplevel $L_{\mathsf{numLevels}} = \mathcal{F}_{\text{OHTable}}$.Build$([\![X]\!], [\![Y]\!], N)$ and initialize the cache L_0.Init(). The players set the query counter $t = 0$.

Π_{DORAM}.ReadAndWrite($[\![X_{\text{query}}]\!]$,$[\![Y_{\text{new}}]\!]$, $[\![$isWrite$]\!]$): $|L_0|$ is the size of the cache and is tracked externally.

1. The players increment the query counter, t, initialize $\mathsf{numLevels}$-bit Alibi data accumulator $[\![\mathfrak{e}_{\text{accum}}]\!] = [\![0^{\mathsf{numLevels}}]\!]$, D-bit payload accumulator $[\![Y_{\text{accum}}]\!] = [\![\perp]\!]$, and one-bit flag $[\![\mathsf{found}]\!] = 0$. Then, the players do the following in parallal (This step takes $\max\{\mathsf{numRoundsCache}, \mathsf{numRoundsPRFEval}\}$ rounds):
   (a) query the Cache, $[\![Y_{\text{accum}}]\!], [\![\mathsf{found}]\!] = L_0$.Query$([\![X_{\text{query}}]\!])$. The players extract $[\![\mathfrak{e}_{\text{accum}}]\!]$ from $[\![Y_{\text{accum}}]\!]$ (by silently copying the last $\mathsf{numLevels}$-bits).
   (b) For each level $\ell \in \mathsf{numLevels}$ evaluate $[\![a_\ell]\!] = \mathcal{F}_{\text{ABB}}$.PRFEval$([\![k_\ell]\!], [\![X_{\text{query}}]\!])$
2. For each ℓ from 1 to $\mathsf{numLevels}$, if there is an OHTable at level L_ℓ:
   (a) Set the bit $[\![\mathsf{useDummy}]\!] = [\![\mathfrak{e}]\!][\ell] \oplus [\![\mathsf{found}]\!]$, i.e., locally XOR the ℓth bit of the share of $\mathfrak{e}$ with the share of found. Query L_i, calling $[\![Y_\ell]\!], [\![\mathsf{found}_\ell]\!] = L_\ell$.Query$([\![a_\ell]\!], [\![\mathsf{useDummy}]\!])$. This step takes $\mathsf{numRoundsOHTable}$ rounds.
3. Set $[\![Y_{\text{new}}]\!] \leftarrow \mathcal{F}_{\text{ReplicatedMPC}}$.IfThenElse$([\![\text{isWrite}]\!], [\![Y_{\text{new}}]\!], [\![Y_{\text{accum}}]\!])$. Set the Alibi mask of $[\![Y_{\text{new}}]\!]$, $[\![Y_{\text{new}}.\mathfrak{e}]\!]$ to 0. Call $\mathcal{F}_{\text{Linear}}$.Store$([\![X]\!], [\![Y_{\text{new}}]\!])$. This takes 1 round.
   (a) Silently extract the $\mathsf{numLevels}$-bit value $[\![\mathfrak{e}_\ell]\!]$ from $[\![Y_\ell]\!]$ and update $[\![\mathfrak{e}_{\text{accum}}]\!] \leftarrow [\![\mathfrak{e}_{\text{accum}} \oplus \mathfrak{e}_\ell]\!]$. Update the value of $[\![\mathsf{found}]\!]$ to $\mathsf{found} \vee \mathsf{found}_\ell$ Additionally, In parallel with the first round of the next iteration of the loop, update $[\![Y_{\text{accum}}]\!] \leftarrow \mathcal{F}_{\text{ReplicatedMPC}}$.IfThenElse$([\![\mathsf{found}_\ell]\!], [\![Y_\ell]\!], [\![Y_{\text{accum}}]\!])$. This step takes one round.
4. If $t = |L_0|$, run the subroutine Π_{DORAM}.Rebuild().

Π_{DORAM}.Rebuild(): We define a level L_i for $1 \leq i \leq \mathsf{numLevels}$ in the DORAM to be *full* if it contains a ShufTable with $(\mathsf{baseAmpFactor} - 1) \cdot \mathsf{baseAmpFactor}^i \cdot |L_0|$ elements. L_0 is full if it has been written to $|L_0|$ times since it was last initialized. Suppose that ℓ is the largest number such that levels $L_0, \ldots L_\ell$ are full while $L_{\ell+1}$ is not, instead containing $A \cdot \mathsf{baseAmpFactor}^\ell |L_0|$ elements for some $A \in \{0, \ldots, \mathsf{baseAmpFactor} - 2\}$ If $\ell = \mathsf{numLevels}$, then necessarily $A = 1$.

1. By concatenating the output of L_0.Extract() and L_i.Extract() for $i \in [\ell + 1]$, the players prepare lists $[\![X]\!]$, $[\![Y]\!]$ of length $(A+1) \cdot \mathsf{baseAmpFactor}^\ell \cdot |L_0|$ containing all the elements to be (potentially) placed in $L_{\ell+1}$.
2. If $\ell < \mathsf{numLevels}$:

   In parallel, relabel the dummies, calling $[\![X_j^*]\!] \leftarrow \mathcal{F}_{\text{ReplicatedMPC}}$.ReplaceIfNull$([\![X_j]\!], [\![N + j]\!])$ and $[\![Y_j^*]\!] = [\![Y_j]\!]$ (syntactically convenient) for $j \in [(A+1) \cdot \mathsf{baseAmpFactor}^\ell \cdot |L_0|]$ (this is so P_1 does not learn how many dummies were queried at the previous level).
3. If $\ell = \mathsf{numLevels}$: The players "cleanse out the dummies" by shuffling, $[\![\hat{X}]\!], [\![\hat{Y}]\!] = \mathcal{F}_{\text{ABB}}$.ObliviousShuffle$([\![X]\!], [\![Y]\!])$, and then revealing which element is dummy by calling $\mathcal{F}_{\text{ABB}}$.Reveal$(\mathcal{F}_{\text{ABB}}$.Equals$([\![\hat{X}_j]\!], \perp))$ for all j.

   The above will reveal exactly N 0's and N 1's (because we started with N "real" elements which we have maintained and we made N queries since last build $L_{\mathsf{numLevels}}$. Since stale elements are relabeled (step 2.A) the invariant holds). Compact the $[\![\hat{X}]\!]$ and $[\![\hat{Y}]\!]$ shares corresponding to 0's into arrays $[\![X^*]\!]$ and $[\![Y^*]\!]$.
4. The players reinitialize the cache by calling $L_0 = \mathcal{F}_{\text{Linear}}$.Init$(|L_0|)$ and build $L_{\ell+1}$ by calling $[\![X^{\text{stash}}]\!], [\![Y^{\text{stash}}]\!] = L_{\ell+1}$.Build$([\![X^*]\!], [\![Y^*]\!])$
5. For i from 1 to $\mathsf{stashSize}$, update the Alibi bit $[\![\mathfrak{e}]\!][\ell + 1] =$ of $[\![Y_i^{\text{stash}}]\!]$ to 1, then call $\mathcal{F}_{\text{Linear}}$.Store$([\![X_i^{\text{stash}}]\!], [\![Y_i^{\text{stash}}]\!])$. The players reset $t = 0$.

Figure 4: The complete DORAM protocol, Π_{DORAM}

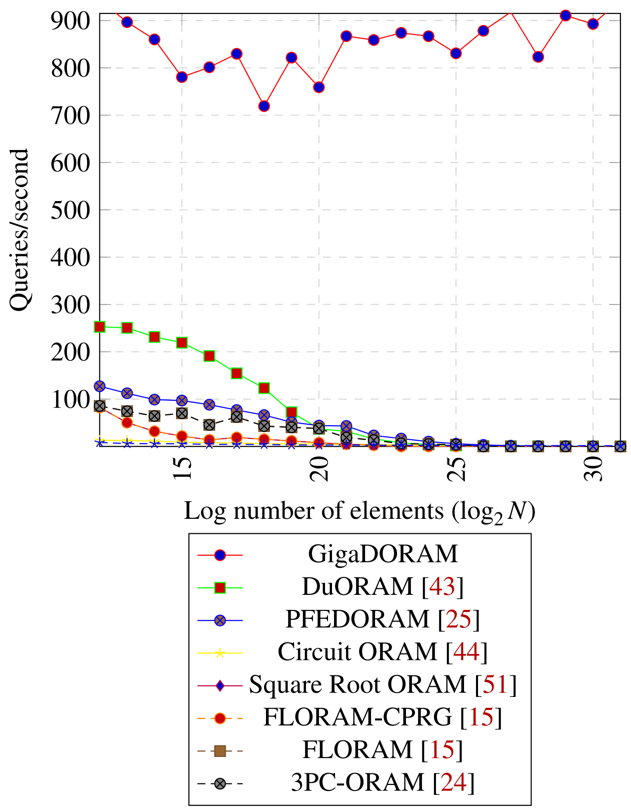


Figure 5: Number of queries/second vs. log number of elements (N). Each query is a write of random data to a random index. Preprocessing costs are accounted for. The number of queries per second is an estimate of (at least) hundreds of queries. All constructions were run in AWS cluster placement network environment.

connections [6], and it is possible to achieve 1ms ping times *across different cloud providers*[15]. Still, we compare the performance of GigaDORAM to other DORAMs under worse network conditions (which may be unavoidable in some applications).

Performance without cluster-placement. Figure 5 shows the performance of GigaDORAM when the machines are cluster-placed. We also benchmarked GigaDORAM when the machines were not cluster-placed (but still in the same AWS region). Table 1 shows that the number of queries/sec drops by about 33%.

Performance and breakeven points in varying latency/bandwidth. Using the `tc` (as in [15, 43]) we vary the latency and bandwidth to search for the breakeven point between GigaDORAM and DuORAM (which is the best previous DORAM in high-latency low-bandwidth environments).

Figure 6.a shows that the performance of GigaDORAM degrades dramatically with latency while the performance of DuORAM degrades dramatically with $\log(N)$. In particular, we see that at $\log(N) = 20$, for latency 8ms GigaDORAM outperforms DuORAM while for 8ms latency DuORAM edges out GigaDORAM. At $\log(N) = 25$ GigaDORAM outperforms DuORAM for all sub-40ms latency.

[15] https://medium.com/@sachinkagarwal/public-cloud-inter-region-network-latency-as-heat-maps-134e22a5ff19

$\log(N)$	Standard Q/s	Cluster Q/s	Slowdown
20	543	812	33%
23	573	910	37%
26	597	877	32%
29	589	891	34%

Table 1: Benchmarking GigaDORAM using 3 different c5n.metal machines standard placement vs cluster placement in `us-west-2`. We benchmarked over 100,000 queries, using lowMC as our SISO-PRF.

Figure 6.b shows that the performance of neither construction degrades substantially with bandwidth. Our experiments show DuORAM outperforming GigaDORAM for $\log(N) = 20$ and GigaDORAM outperforming DuORAM for $\log(N) = 25$. Again, we see the performance of DuORAM degrading significantly with N while GigaDORAM's performance hardly changes.

9.3 The cost of GigaDORAM

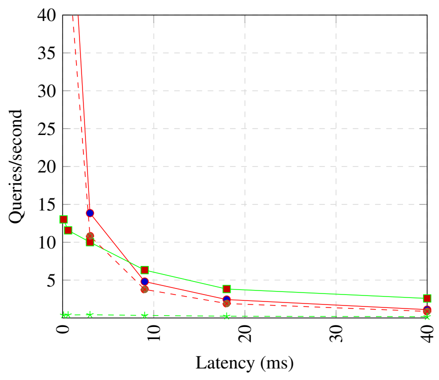
Running large computations in AWS can be quite expensive, but due to its high query/sec, GigaDORAM is extremely cost efficient. Including both network and compute costs, GigaDORAM can handle about over 120,000 queries per dollar with current AWS pricing, which is 10× DuORAM [43], which is to the best of our knowledge, the previously most cost efficient DORAM. See Appendix F for the calculations.

9.4 Replacing LowMC with AES

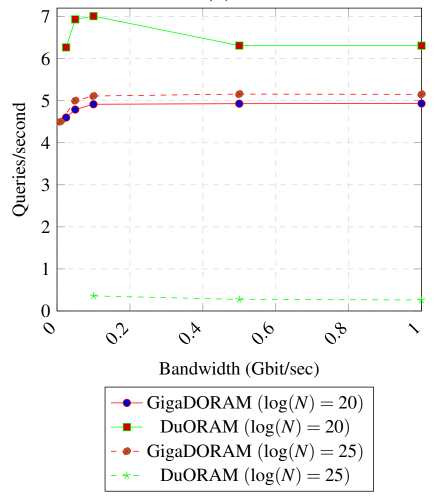
We implemented our SISO-PRFs by evaluating the LowMC block cipher under MPC. For comparison we evaluated GigaDORAM using AES instead of LowMC. The AES circuit we has about 10× the ANDs and 10× the AND-depth as the LowMC circuit we use, and is thus less efficient to evaluate under MPC. Table 2 shows approximately a 2× slowdown to GigaDORAM when using AES instead of LowMC. We note this might change in worse network settings where the extra communication and rounds may become more pronounced.

10 Conclusion

In this work we introduce GigaDORAM, the most efficient and scalable DORAM construction to date. At $N = 2^{31}$, our DORAM can perform over 700 queries per second, making GigaDORAM orders of magnitude faster than prior DORAM



(a)



(b)

Figure 6: (a) Number of queries/second vs. latency in milliseconds for fixed 1Gbit network (b) Number of queries/second vs. Bandwidth in Gbit for fixed 8ms latency. Executed on a single c5.metal via multiple processes restricted in communication via the `tc`. Preprocessing costs are accounted for. The number of queries per second is an estimate of 5000 queries for GigaDORAM and 128 for DuORAM.

constructions in low-latency environments. We give a custom C++, open source implementation of GigaDORAM. We hope GigaDORAM will enable the first somewhat practical RAM-MPC applications and open a new realm of possibilities for privacy-preserving cloud data-stores.

References

[1] Albrecht, M. R., Rechberger, C., Schneider, T., Tiessen, T., and Zohner, M. Ciphers for MPC and FHE. In *EUROCRYPT* (2015), Springer, pp. 430–454.

[2] Araki, T., Furakawa, J., Lindell, Y., Nof, A., and Ohara, K. High-throughput semi-honest secure three-party computation with an honest majority. In *CCS* (2016).

$\log(N)$	AES Q/s	LowMC Q/s	Slowdown
20	393	812	51%
23	405	910	55%
26	401	877	54%
29	417	891	53%

Table 2: Benchmarking GigaDORAM using AES to implement a SISO-PRF instead of LowMC. We run the benchmark in the same setting as Figure 5.

[3] Asharov, G., Komargodski, I., Lin, W.-K., Nayak, K., Peserico, E., and Shi, E. OptORAMa: Optimal oblivious RAM. In *EUROCRYPT* (2020).

[4] Asharov, G., Komargodski, I., Lin, W.-K., and Shi, E. Oblivious RAM with worst-case logarithmic overhead. In *CRYPTO* (2021), Springer, pp. 610–640.

[5] Banik, S., Barooti, K., Vaudenay, S., and Yan, H. New attacks on LowMC instances with a single plaintext/ciphertext pair.

[6] Bar, J. The floodgates are open – increased network bandwidth for EC2 instances. `https://aws.amazon.com/blogs/aws/the-floodgates-are-open-increased-network-bandwidth-for-ec2-instances/`, 2018.

[7] Beaver, D., Micali, S., and Rogaway, P. The round complexity of secure protocols. In *STOC* (1990), pp. 503–513.

[8] Bogdanov, D., Kamm, L., Kubo, B., Rebane, R., Sokk, V., and Talviste, R. Students and taxes: a privacy-preserving study using secure computation. *Proc. Priv. Enhancing Technol. 2016*, 3 (2016), 117–135.

[9] Boyle, E., Gilboa, N., and Ishai, Y. Function secret sharing. In *EUROCRYPT* (2015).

[10] Bunn, P., Katz, J., Kushilevitz, E., and Ostrovsky, R. Efficient 3-party distributed ORAM. In *SCN* (2020).

[11] Chase, M., Derler, D., Goldfeder, S., Orlandi, C., Ramacher, S., Rechberger, C., Slamanig, D., and Zaverucha, G. Post-quantum zero-knowledge and signatures from symmetric-key primitives. In *CCS* (2017), pp. 1825–1842.

[12] Chaum, D., Crépeau, C., and Damgård, I. Multiparty Unconditionally Secure Protocols. In *STOC* (1988).

[13] Cramer, R., Damgård, I., and Ishai, Y. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *TCC* (2005).

[14] Dinur, I., Liu, Y., Meier, W., and Wang, Q. Optimized interpolation attacks on lowmc. In *ASIACRYPT* (2015), Springer, pp. 535–560.

[15] Doerner, J., and shelat, a. Scaling ORAM for secure computation. In *CCS* (2017).

[16] Falk, B. H., Noble, D., and Ostrovsky, R. Alibi: A flaw in cuckoo-hashing based hierarchical ORAM schemes and a solution. In *EUROCRYPT* (2021), Springer, pp. 338–369.

[17] Falk, B. H., Noble, D., and Ostrovsky, R. 3-party distributed ORAM from oblivious set membership. In *SCN* (2022), Springer, pp. 437–461.

[18] Gilboa, N., and Ishai, Y. Distributed point functions and their applications. In *EUROCRYPT* (2014).

[19] Goldreich, O., Micali, S., and Wigderson, A. How to play any mental game. In *STOC* (1987).

[20] Goldreich, O., and Ostrovsky, R. Software protection and simulation on oblivious RAMs. *JACM 43*, 3 (1996).

[21] GOODRICH, M. T., MITZENMACHER, M., OHRIMENKO, O., AND TAMASSIA, R. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA* (2012).

[22] GORDON, S. D., KATZ, J., KOLESNIKOV, V., KRELL, F., MALKIN, T., RAYKOVA, M., AND VAHLIS, Y. Secure two-party computation in sublinear (amortized) time. In *CCS* (2012).

[23] GRASSI, L., RECHBERGER, C., ROTARU, D., SCHOLL, P., AND SMART, N. P. MPC-friendly symmetric key primitives. In *CCS* (2016), pp. 430–443.

[24] JARECKI, S., AND WEI, B. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In *ACNS* (2018).

[25] JI, K., ZHANG, B., LU, T., AND REN, K. Multi-party private function evaluation for RAM. Cryptology ePrint Archive, Paper 2022/939, 2022. `https://eprint.iacr.org/2022/939`.

[26] KIRSCH, A., MITZENMACHER, M., AND WIEDER, U. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing* (2009).

[27] KUSHILEVITZ, E., LU, S., AND OSTROVSKY, R. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *SODA* (2012).

[28] LAUD, P. Privacy-preserving minimum spanning trees through oblivious parallel RAM for secure multiparty computation. IACR ePrint Archive 2014/630, 2014.

[29] LAUR, S., WILLEMSON, J., AND ZHANG, B. Round-efficient oblivious database manipulation. In *ISC* (2011).

[30] LIU, F., ISOBE, T., AND MEIER, W. Cryptanalysis of full LowMC and LowMC-M with algebraic techniques. In *CRYPTO* (2021), pp. 368–401.

[31] LU, S., AND OSTROVSKY, R. Distributed oblivious RAM for secure two-party computation. In *TCC* (2013).

[32] MOHASSEL, P., AND RINDAL, P. ABY3: A mixed protocol framework for machine learning. In *CCS* (2018), pp. 35–52.

[33] NAVEED, M. The fallacy of composition of oblivious RAM and searchable encryption. IACR ePrint 2015/688, 2015.

[34] NIST. Post-quantum cryptography PQC: Round 3 submissions. `https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`, 2021.

[35] NOBLE, D. An intimate analysis of cuckoo hashing with a stash. IACR ePrint 2021/447, 2021.

[36] OSTROVSKY, R. Efficient computation on oblivious RAMs. In *STOC* (1990).

[37] OSTROVSKY, R. *Software Protection and Simulation On Oblivious RAMs*. PhD thesis, Massachusetts Institute of Technology, 1992.

[38] OSTROVSKY, R., AND SHOUP, V. Private information storage. In *STOC* (1997), vol. 97.

[39] PAGH, R., AND RODLER, F. F. Cuckoo hashing. In *ESA* (2001).

[40] PATEL, S., PERSIANO, G., RAYKOVA, M., AND YEO, K. PanORAMa: Oblivious RAM with logarithmic overhead. In *FOCS* (2018).

[41] RINDAL, P. The ABY3 Framework for Machine Learning and Database Operations. `https://github.com/ladnir/aby3`.

[42] SHI, E., CHAN, T.-H. H., STEFANOV, E., AND LI, M. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT* (2011).

[43] VADAPALLI, A., HENRY, R., AND GOLDBERG, I. DuORAM: A bandwidth-efficient distributed ORAM for 2- and 3-party computation. IACR ePrint 2022/1747, 2022.

[44] WANG, X., CHAN, H., AND SHI, E. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *CCS* (2015).

[45] WANG, X., RANELLUCCI, S., AND KATZ, J. Global-scale secure multiparty computation. In *CCS* (2017), pp. 39–56.

[46] WANG, X., RANELLUCCI, S., AND KATZ, J. Global-scale secure multiparty computation. In *CCS* (2017), pp. 39–56.

[47] WANG, X. S., HUANG, Y., CHAN, T.-H. H., SHELAT, A., AND SHI, E. SCORAM: oblivious RAM for secure computation. In *CCS* (2014).

[48] YAO, A. Protocols for secure computations (extended abstract). In *FOCS* (1982).

[49] YAO, A. How to generate and exchange secrets. In *FOCS* (1986).

[50] ZAHUR, S., AND EVANS, D. Obliv-C: A language for extensible data-oblivious computation. IACR ePrint 2015/1153, 2015.

[51] ZAHUR, S., WANG, X., RAYKOVA, M., GASCÓN, A., DOERNER, J., EVANS, D., AND KATZ, J. Revisiting square-root ORAM: efficient random access in multi-party computation. In *S & P* (2016).

A ABB Functionalities

The basic ABB operations we use are described in Figure 7.

B The naïve cache protocol

The naïve cache protocol works as follows

```
$r \leftarrow \perp$
for $i = 1, \ldots, t$ do
    if $x = x_i$ then
        $r \leftarrow y_i$
    end if
end for
Return: $r$
```

This protocol has multiplicative depth, t, thus implementing this under the MPC of [2] leads to a protocol with $\mathsf{numRoundsCache} = |L_0|$. It is possible to implement this with a garbled-circuit-based approach, (e.g. 3-party garbled circuits [32] or BMR [7]). This results in a constant-round MPC protocol, but the communication complexity is extremely large – too inefficient for our application.

$\Pi_{\text{SpeedCache}}$, (described in Figure 3), parallelizes the equality tests of the naïve protocol leading to a low-depth circuit, that we implement using the 3-party MPC of [2].

C LowMC

In this section, we discuss LowMC, an MPC-friendly block-cipher we use to instantiate $\mathcal{F}_{\text{ABB}}$.PRFEval.

LowMC. LowMC (Low Multiplicative Complexity) [1] is a family of block cipher that is built with MPC, ZK, and FHE in mind. LowMC has a variety of instantiations which trades low number of AND gates and a low circuit AND-depth. [16] The instantiation of LowMC we use has 46837 total gates, out of which 1134 are ANDs, stacked into 9-AND-depth circuit. By

[16] The authors of LowMC provide a script to exactly calculate the tradeoff between the number of AND gates, and the AND-depth of the circuit `https://github.com/LowMC/lowmc/blob/master/determine_rounds.py`.

- $[x]^{(i,j)} = \mathcal{F}_{\text{ABB}}.\text{InputTo2Sharing}(x, P_i, P_j)$. A single player shares creates an additive sharing of a secret, x, among participants i and j.
- $[x]^{(i,j)} = \mathcal{F}_{\text{ABB}}.\text{ReshareReplicatedTo2Sharing}([\![x]\!], \{P_i, P_j\})$. Convert a 3-party CNF sharing of a secret, x, to a two-party DNF sharing of the same secret, x, held by participants i and j.
- $[x]^{(i',j')} = \mathcal{F}_{\text{ABB}}.\text{Reshare2To2Sharing}([x]^{(i,j)}, \{P_{i'}, P_{j'}\})$. Convert a two-party DNF sharing of a secret, x, held by participants i and j, to a two-party DNF sharing of the same secret, x, held by participants i' and j'.
- $[\![x]\!] = \mathcal{F}_{\text{ABB}}.\text{Reshare2SharingToReplicated}([x]^{(i,j)})$. Convert a two-party DNF sharing of a secret, x, held by participants i and j to a three-party CNF sharing of the same secret, x.
- $[\![k]\!] = \mathcal{F}_{\text{ABB}}.\text{RandomElement}(\kappa)$. Generate a three-party CNF sharing of a uniformly random field element (whose value is unknown to the participants).
- $x = \mathcal{F}_{\text{ABB}}.\text{RevealTo}(P_i, [\![x]\!])$. Reveal a secret-shared value, $[\![x]\!]$, to participant P_i.

We also abstract away a few more sophisticated operations:

- $[\![y]\!] = \mathcal{F}_{\text{ABB}}.\text{PRFEval}([\![x]\!], [\![k]\!])$. Evaluate a SISO-PRF with secret-shared key, $[\![k]\!]$, on secret-shared input, $[\![x]\!]$, to obtain a secret-shared output, $[\![y]\!]$. In our instantation, we instantiate the $\mathcal{F}_{\text{ABB}}.\text{PRFEval}(\cdot,\cdot)$ functionality by evaluating the LowMC block cipher [1] under MPC. Specifically, in our custom implementation of a (3,1)-MPC protocol (based on [2]).
- $[\![\mathsf{QueryCircuit}(x_1,\ldots,x_l)]\!] = \mathcal{F}_{\text{ABB}}.\text{EvalCircuit(3,1)GC}(\mathsf{QueryCircuit}, \text{Inputs} = ([\![x_1]\!],\ldots,[\![x_l]\!]))$. Evaluate a 3-party garbled circuit on secret-shared inputs $[\![x_1]\!],\ldots,[\![x_l]\!]$, and returns a *sharing* of the output of the circuit computation. We use a custom implementation of the 3-party Garbled Circuit protocol outlined in ABY3 [32].
- $[\![Y]\!] = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}([\![X]\!])$. This functionality implements a linear-communication, three-party shuffle of secret shared values [29]. So $Y = \pi(X)$ for some random permutation, unknown to the participants. We also use a modified protocol to output a sharing of the permutation as well. $[\![Y]\!], [\![L]\!] = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}([\![X]\!], \text{DistributeShuffle} = True)$. In this setting, $Y = \pi(X)$ as before, and $L = \pi^{-1}(1,2,\ldots,n)$. We describe how to implement this novel "Persistent shuffle" in Section 5.1.

Figure 7: The ABB functionalites used in our DORAM protocol.

$\log(N)$	# of bytes sent
20	$5.86 \cdot 10^9$
25	$5.64 \cdot 10^9$
30	$6.73 \cdot 10^9$

Figure 8: number of bytes sent by a single GigaDORAM machine for 100,000 queries, using LowMC as the PRF at varying values of $\log(N)$ at the same values of $\mathsf{baseAmpFactor}$, $\mathsf{numLevels}$, in which our main benchmark was conducted (see Figure 5).

contrast, AES has a total of 36663 gates, out of which 6400 are ANDs, stacked into a 60-AND-depth-circuit. [17] Since the AND-depth of the circuit is equal to the number of rounds to evaluate the circuit under MPC, using LowMC instead of AES saves 51 rounds of communication for each query.

We provide the first circuit files for LowMC and encode them in the popular Bristol fashion.[18] The traditional Bristol fashion requires that each virtual "wire" to be only assigned once, requiring significantly more memory for the computation than necessary. By threading wires through the circuit (i.e., reusing memory) we are able speed up the computation of LowMC under the [2] MPC by a factor of 2.

Although LowMC has not received as much analysis as AES, it's security has been been adapted into the Picnic signature scheme [11], a 3rd round candidate in the NIST post-quantum digital signature contest [34]. Additionally, there have been several thorough cryptanalysis attempts [5, 14, 30] motivated by an ongoing Microsoft-funded challenge, none of which discovered an attack which made the community doubt the security LowMC.[19]

D Alibi reinsertion

As discussed in Section 5.2, since we use Cuckoo hash tables to store ShufTable's data, we must reduce the failure probability of $\Pi_{\text{ShufTable}}$.Build from noticeable to negligible (in N) by outputting a small stash, Stash, which we reinsert into the cache. Naïvely, for all $([\![X_i]\!], [\![Y_i]\!]) \in$ Stash we could call L_0.Append$([\![X_i]\!], [\![Y_i]\!])$, "reinserting" $([\![X_i]\!], [\![Y_i]\!])$ which could not fit in L_i's CHT into the cache. Yet, as proved in [16], the naïve reinsertion technique above compromises the obliviousness of (D)ORAM. Intuitively, if we reinsert $([\![X_i]\!], [\![Y_i]\!])$ to L_0 from L_j and then query $[\![X_i]\!]$, in the aggregate, an eavesdropper could tell that we should have queried $[\![X_i]\!]$ to L_j, but instead queried a dummy because we found $([\![X_i]\!], [\![Y_i]\!])$ at some previous level (it was reinserted). Roughly speaking, our goal $(*)$ is to append some data to $([\![X_i]\!], [\![Y_i]\!])$ such that when $([\![X_i]\!], [\![Y_i]\!])$ is reinserted from L_j and found at L_k for $k < j$, we will know to continue querying $L_{k+1}, \ldots, L_j$ *as if* we did not find $([\![X_i]\!], [\![Y_i]\!])$, and query $L_{j+1}, \ldots, L_{\mathsf{numLevels}}$ *as if* we found $([\![X_i]\!], [\![Y_i]\!])$ at L_j. To do this, we store $[\![\mathfrak{e}_{X_i}]\!] \in \{0,1\}^{\mathsf{numLevels}}$ as the last numLevels bits of Y_i (it is assumed that $D >$ numLevels) where $\mathfrak{e}_{X_i}[j] = 1$ iff $([\![X_i]\!], [\![Y_i]\!])$ was reinserted from L_j. We set $\mathfrak{e}_{X_i} = 0^{\mathsf{numLevels}}$ every time we query $([\![X_i]\!], [\![Y_i]\!])$. When querying for $[\![x]\!]$ and finding it at L_k where $\mathfrak{e}_x[j] = 1$ from L_j, under MPC we "decide" to query according to $(*)$. The full details of how the Alibi bits are used can be found in the description of our full DORAM protocol (Figure 4).

[17]We refer to Bristol Fashion AES circuit file from `https://homes.esat.kuleuven.be/~nsmart/MPC/MAND/aes_128.txt`

[18]`https://homes.esat.kuleuven.be/~nsmart/MPC/`

[19]`https://lowmcchallenge.github.io/`

E Benchmarking previous DORAMs

We benchmark the (3,1) semi-honest (most efficient) DuORAM variant [43] via their convenient Docker setup on a single c5n.metal machine. We use the `set-networking.sh` script they provide to set .229ms latency and 25Gbit bandwidth simulated network between their Docker containers. We do not restrict the number of cores their process can use, and DuORAM used all 96 vCPUs during their preprocessing stage.

We benchmark FLORAM, FLORAM-CPRG, Circuit ORAM, and Square Root ORAM [15, 44, 51] via the obliv-c [50] based setup given by [15].[20] We benchmark the above 2-party constructions between two cluster-placed c5n.metal machines. For backwards compatibility with obliv-c, we run tests on Ubuntu 18.04.6 LTS. We do not restrict the network via the `tc` command as was done in [15].

Bingsheng Zhang kindly benchmarked the proprietary PFE-DORAM [25] on a comparable network to ours. Zhang executed the protocol via separate processes on the same Intel(R) Core i7 8700 CPU 3.2 GHz, 6 CPUs, 32 GB Memory, 1TB SSD machine running Ubuntu 18.04.2 LTS. Bandwidth between the processes was not limited and latency was restricted to 0.05ms.

We benchmark 3PC-ORAM [24] via the dockerization graciously provided by the DuORAM [43] team[21]. Like other constructions, we ran 3PC-ORAM with 0.229ms latency and 25Gbit bandwidth.

F The cost of running GigaDORAM

The c5n.metal machines we rent cost \$3.888 per hour, and running DORAM requires 3 different machines. GigaDORAM. Given that we get ≈ 800 queries per second, (Figure 5), we get $800 \cdot 3600/(3.888 \cdot 3) = 246914$ queries per USD.

If we benchmark in the same region, *all communication is free*. If we benchmark in different regions, according to AWS the charge per Gigabyte in and out of AWS is \$0.01.[22] According to Table 8 it is reasonable to conservatively estimate that GigaDORAMrequires $3 \cdot 7 \cdot 10^9/100{,}000 = 210{,}000$ bytes per query $= 210{,}000/2^{30} = 2 \cdot 10^{-4}$ Gigabytes per query. Multiplying by the dollar cost, we get that GigaDORAM requires $2 \cdot 10^{-4} \cdot .02 = 4 \cdot 10^{-6}$ USD per query.

Summing up communication and computation, we get that we have $1/246914 + 4 \cdot 10^{-6} = 8.05 \cdot 10^{-6}$ which gives approximately $120{,}000$ queries per dollar.

By comparison, DuORAM [43] reports $8900 \cdot 10^{-6}$ dollars for 128 queries (computation cost) giving $128/8900 \cdot 10^{-6}$ queries per dollar. For communication DuORAM gets $5 \cdot 10^{-6}$ dollars per 128 queries, giving $128/5 \cdot 10^{-6}$. This gives $128/8905 \cdot 10^6 = 14{,}374 \approx 15{,}000$ queries per dollar.

[20]See `https://gitlab.com/neucrypt/floram/`

[21]`https://git-crysp.uwaterloo.ca/iang/circuit-oram-docker`

[22]`https://aws.amazon.com/ec2/pricing/on-demand/`