# DORAM Revisited: Maliciously Secure RAM-MPC with Logarithmic Overhead

Brett Falk[1], Daniel Noble[1](✉), Rafail Ostrovsky[2], Matan Shtepel[1], and Jacob Zhang[2]

[1] University of Pennsylvania, Philadelphia, USA
{fbrett,dgnoble}@seas.upenn.edu, matan.shtepel@ucla.edu
[2] UCLA, Los Angeles, USA
rafail@cs.ucla.edu, jacobzhang@g.ucla.edu

**Abstract.** Distributed Oblivious Random Access Memory (DORAM) is a secure multiparty protocol that allows a group of participants holding a secret-shared array to read and write to secret-shared locations within the array. The efficiency of a DORAM protocol is measured by the amount of communication required per read/write query into the array. DORAM protocols are a necessary ingredient for executing Secure Multiparty Computation (MPC) in the RAM model.

Although DORAM has been widely studied, all existing DORAM protocols have focused on the setting where the DORAM servers are semi-honest. Generic techniques for upgrading a semi-honest DORAM protocol to the malicious model typically increase the asymptotic communication complexity of the DORAM scheme.

In this work, we present a 3-party DORAM protocol which requires $O((\kappa + D) \log N)$ communication per query, for a database of size $N$ with $D$-bit values, where $\kappa$ is the security parameter. Our hidden constants in the big-O nation are small. We show that our protocol is UC-secure in the presence of a malicious, static adversary. This matches the communication complexity of the best semi-honest DORAM protocols, and is the first malicious DORAM protocol with this complexity.

## 1 Introduction

In this work, we develop the first Distributed Oblivious RAM (DORAM) protocol secure against *malicious* adversaries while matching the communication and computation costs of the best-known semi-honest constructions.

Poly-logarithmic overhead Oblivious RAM (ORAM) [Ost90, Ost92, GO96] was developed to allow a client to access a database held by an untrusted server,

M. Shtepel–Work done while at UCLA.

**Supplementary Information** The online version contains supplementary material available at https://doi.org/10.1007/978-3-031-48615-9_16.

while hiding the client's access pattern from the server itself with poly-log overhead. In this work, we focus on *Distributed* Oblivious RAM, which allows a group of servers to access a secret-shared array at a secret-shared index. The secret-shared index can be conceptualized as coming either from an external client or as the output of a previous secure computation done by the servers.

The efficiency of an ORAM protocol is usually measured by the (amortized) number of bits of communication required to process a single query. If privacy were not an issue, in order to retrieve a single $D$-bit entry from a table of size $N$, the client would need to send a $\log(N)$-bit index, and receive a $D$-bit value, so the communication would be $\log(N) + D$. In order to make the queries *oblivious*, it is known that a multiplicative communication overhead of $\Omega(\log(N))$ is required [GO96,LN18]. That is, the optimal communication in the traditional, passive-server ORAM setting is $\Omega((D + \log N)\log N)$.[1] Several ORAM protocols have achieved this "optimal" communication complexity (in slightly different settings). [LO13] achieved logarithmic amortized overhead in the two-server setting (Fig. 1b), OptORAMa [AKL+20] achieved amortized logarithmic overhead in the single-server setting (Fig. 1a) with constant $> 2^{228}$ hidden by the big-O notation. The constant was later reduced to 9405 in [DO20] and de-amortized in [AKLS21]. However, despite all these improvements, these works are of only theoretical interest, due to large constants. In Appendix A.4 we discuss why none of these semi-honest constructions can be naïvely compiled to a maliciously secure DORAM without asymptotic blowup. When a DORAM can store $N$, $D$-bit elements with security parameter is $\kappa$, we prove the following theorem:

**Theorem 1 (Malicious DORAM, Informal).** *If Pseudo-Random Functions exist with $O(\kappa + l)$ circuit size (where $l$ is the number of input bits and $\kappa$ is the computational security parameter), then there exists a (3,1)-malicious DORAM scheme (see Definition 2) with $O((\kappa + D)\log N)$ communication complexity between the servers per each query.*

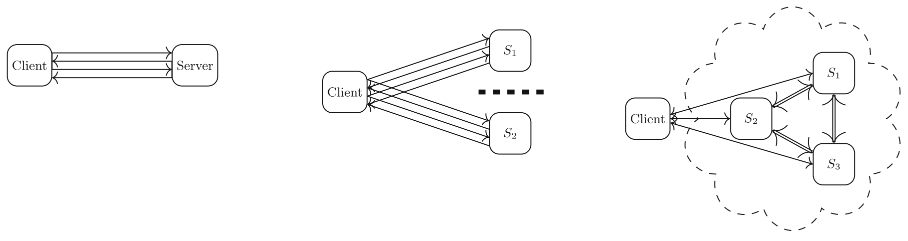
The best DORAMs in the *semi-honest* model have either $O((\kappa + D)\log(N))$ [LO13,FNO22] or $O((\log^2(N) + D)\log(N))$ [WCS15] communication complexity per query. Which of these is better depends on the parameter choices. If $D$ is large ($\Omega(\log^2(N) + \kappa)$) they are equally good. If $D$ is small, [LO13,FNO22] are better when $\log(N) = \omega(\sqrt{\kappa})$ and [WCS15] is better otherwise. Thus, our server-to-server communication overhead of $O((\kappa + D)\log(N))$ matches the best communication complexity of the best DORAM protocols in the semi-honest model [FNO22,LO13], achieving security against malicious adversaries with *no* asymptotic increase in communication costs.

Note that a non-private solution would still require communicating $\Theta(\log(N) + D)$ bits to simply send the secret-shared query and secret-shared response. Thus, our cost of $\Theta((\kappa + D)\log(N))$ has logarithmic overhead when

[1] Most ORAM works assume $D = \Omega(\log N)$, so $O((D + \log N)\log N) = O(D \log N)$ which is described as a logarithmic "overhead" or a logarithmic "blowup" over $O(D)$ communication needed to make a query in the insecure setting.

the block size, $D = \Omega(\kappa)$. Our title refers to this (common) scenario. If $D = o(\kappa)$, the overhead is $\Theta(\log(N)\kappa/D)$.

As we discuss below, one of the main motivations for studying DORAM is in service of building efficient, secure multiparty computation (MPC) protocols in the RAM model of computation.



(a) Client-server ORAM (b) Multi-server ORAM (c) DORAM

**Fig. 1.** *Abstract view of different ORAM "flavors" in the client-server model. In client-server ORAM the client and the server communicate over many rounds. In multiserver ORAM the client communicates with each server individually over many rounds. In DORAM, the client communicates a secret shared query to the servers, the DORAM servers communicate among themselves for several rounds, and respond to the client. The client's work is the lowest in the DORAM setting.*

### 1.1 MPC in the RAM Model

Secure Multiparty Computation (MPC) protocols enable a set of mutually distrusting parties, $P_1, ..., P_n$, with private data $x_1, ..., x_n$ to compute an agreed-upon (probabilistic) polynomial-time function, $f$, in such a way that each player learns the output, $f(x_1, ..., x_n)$, but no additional information about the other participants' inputs [Yao82, Yao86, GMW87, CCD88].

The majority of MPC protocols work in *the circuit model of computation* [Vol99], where the functionality, $f$, is represented as a circuit (either a boolean circuit, or an arithmetic circuit over a finite field $\mathbb{F}$). Computing in the circuit model has been advantageous for MPC protocols because circuits are naturally *oblivious*, i.e., the sequence of operations needed to compute $f$ is independent of the *private* inputs $x_1, \ldots, x_n$. This reduces the problem of securely computing an arbitrary function, $f$, to the problem of securely computing a small set of universal gates (e.g. AND and XOR).

Although the circuit model of computation is convenient for MPC, many common functionalities cannot be represented by *compact* circuits, which means generic circuit-based MPC protocols cannot compute them efficiently. A simple database lookup highlights the inefficiency of the circuit model. Consider the function $R(i, y_1, \ldots, y_N) = y_i$, which outputs the $i$th element in a list or the function $W(i, Y, y_1, \ldots, y_N)$ which produces no output but sets $y_i = Y$. These functionalities can run in constant time in the RAM-model of computation, but in the *circuit model*, both $R$ and $W$ have circuit complexity $O(N)$.

In contrast to circuit-based MPC protocols, RAM-MPC framework [OS97] provides a method of securely computing functions specified in the RAM model of computation. Efficiency is often a barrier to the deployment of MPC protocols in practice, and compilation from RAM model into circuits hurts the efficiency of programs which use random access. Thus, RAM-MPC is a critical step in making general-purpose MPC protocols that are efficient enough for practical applications.

## 1.2 Building RAM-MPC

One method for building RAM-MPC is to use a generic (circuit-model) MPC protocol to simulate the client for a client-server ORAM protocol [OS97]. For the purpose of running ORAM clients under MPC, various "MPC friendly" ORAM protocols have been developed. For example, [WCS15] developed *circuit ORAM*, an ORAM maintaining the stringent one-trusted-client one-untrusted-server security model of traditional ORAM while decreasing the circuit-complexity of the client. Another example of such efforts, are *multi-server ORAM* protocols where the trusted client's data is shared and accessed across multiple servers. Assuming some fraction of the servers are honest [OS97,GKK+12,GKW18, KM19] these works shift some of the communication burden to servers. These multi-server ORAMs can also be adapted to the MPC context by simulating the client using (circuit-based) MPC, allowing the MPC participants to play the role of the additional ORAM servers. Some of these protocols have been implemented [GKK+12,LO13,ZWR+16,WHC+14,Ds17].

A recent direction in the search for MPC-friendly ORAMs is *Distributed ORAM* (DORAM). In a DORAM protocol, both the index $i$ and the database $y_1, \ldots, y_N$ are secret shared among a number of servers. The goal of the protocol is to obtain a secret-sharing of $y_i$ at minimal communication between the servers while not exposing *any* information about $i$ or $y_1, \ldots, y_N$. DORAM has been widely studied in the semi-honest model [LO13,GHL+14,FJKW15,ZWR+16, Ds17,JW18,BKKO20,FNO22,JZLR22,VHG22]. These works have taken several interesting approaches, emphasizing different parameters, and often presenting implementations [ZWR+16,Ds17,VHG22,JZLR22].

In this paper, we study DORAM in the malicious model. In particular, we provide the *first* DORAM protocol that provides security against *malicious* adversaries while *matching the asymptotics of the best-known semi-honest construction.* We use the generic transformation to compile our DORAM into RAM-MPC, giving RAM-MPC which is secure against *malicious* adversaries with an asymptotic cost on par with *the best existing semi-honest constructions.*

## 2 Notation and Definitions

We denote the 3 parties as $P_0, P_1, P_2$, and use $\mathbb{F}_{2^l}$ to denote the finite field of $2^l$ elements. For $x = x_0 \ldots x_{n-1} = x \in \mathbb{F}^n$ we let $x[i:j] = x_i \ldots x_j$. For $a \in \mathbb{Z}^+$, $[a]$ represents the set $\{1, \ldots, a\}$. For any set $S$, $x \in_R S$ represents choosing $x$ uniformally at random from $S$.

$N$ is the number of elements in the DORAM. Each element stored in the DORAM is a pair $(X, Y)$, where $X \in [N]$ is the "virtual index" of the $D$-bit payload, $Y$. We assume that only indices in the range $[N]$ are queried. We use $\perp \notin [N] \cup \{0,1\}^D$ to represent a reserved null-value. $\kappa$ is the computational security parameter and $\sigma$ is a statistical security parameter. Since we want to achieve failures with probability negligible in $N$, we must have both $\kappa, \sigma = \omega(\log N)$.

The primary secret-sharing our protocol uses is $(3,1)$ Replicated Secret Sharing (RSS) (also called a CNF sharing [CDI05]). $[\![x]\!]$ denotes a RSS of a variable $x$. In a $(3,1)$ RSS sharing, each party holds two shares of an additive sharing:

**Definition 1 (replicated secret sharing).** *Let $x, x^{(0)}, x^{(1)}, x^{(2)} \in \mathbb{F}$ s.t $x^{(0)} + x^{(1)} + x^{(2)} = x$. we say that $P_0, P_1, P_2$ hold a replicated secret sharing of $x$ if $P_i$ hold all $x^{(j)}$ s.t $j \neq i$.*

We also use two-party additive sharings. $[x]^{(i,j)}$ denotes an additive sharing of $x$ held by parties $P_i$ and $P_j$, that is $P_i$ holds $x^{(i)}$ and $P_j$ holds $x^{(j)}$ where $x^{(i)} + x^{(j)} = x$.

We use standard Boolean operators ($\wedge$, $\vee$, $\neg$). We also represent by $x \overset{?}{=} y$ the Boolean-output operation that outputs 1 if $x$ equals $y$ and 0 otherwise, and use $(b?x : y)$ to represent an if-then-else statement which evaluates to $x$ if $b$, and $y$ otherwise.

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

$\mathcal{F}_{\text{DORAM}}$**.Init($[\![Y]\!]$):** Given a secret-shared $N$ element array, s.t for all $i \in [N]$, $Y_i \in \{0,1\}^D$, store $Y$ internally. No output.
$\mathcal{F}_{\text{DORAM}}$**.ReadAndWrite($[\![X]\!]$, $[\![Y_{\text{new}}]\!]$):** Given a secret-shared address $X \in [N]$ and secret-shared value $Y_{\text{new}} \in \{0,1\}^D \cup \perp$:

1. Output $[\![Y_X]\!]$ to the players.
2. If $Y_{\text{new}} \neq \perp$, update $Y_X = Y_{\text{new}}$.

**Fig. 2.** $\mathcal{F}_{\text{DORAM}}$: The DORAM functionality

In this work, we define security using the Universal Composability (UC) framework [Can01], which allows us to formally define DORAM.

**Definition 2 (DORAM).** *A protocol, $\Pi$, is said to be a UC maliciously-secure $(n,t)$-Distributed ORAM protocol if for all $N, D, \kappa \in \mathbb{Z}^+$, $\Pi$ UC-realizes the DORAM functionality (Fig. 2).*

## 3 Related Work

In this section we present a brief overview of related work; Appendix A contains a more detailed discussion.

Many DORAMs start with a client-server ORAM and simulate the client inside of a secure computation. This was the approach taken by [WCS15] and [ZWR+16]. The generic MPC can be achieved from Garbled Circuits (GC), which allows for 2 parties, low round complexity, but requires $\Theta(\kappa)$ communication for each AND gate. Alternatively, it can be achieved using honest-majority secret-sharing approaches derived from the BGW protocol [BOGW88], which have $\Theta(1)$ communication per AND gate, but need 3 parties and more rounds.

Similarly it is also possible to convert a multi-server ORAM, such as [LO13] to a DORAM, again by simulating the client inside of a secure computation, and having each server run by a different party.

Other protocols build DORAM directly. This includes [Ds17, HV20, FJKW15, JW18, BKKO20, VHG22, FNO22]. This allows use of techniques that are not applicable for client server ORAMs, such as Function Secret Sharing (FSS), Secret-Shared PIR (SS-PIR) and efficient shuffles.

Table 1 presents these protocols, with their communication costs. Our communication cost is asymptotically identical to [FNO22] and a BGW-style instance of [LO13]. Depending on the relationship between $\kappa$ and $\log(N)$ it may be either asymptotically better or worse than a BGW-style instance of [WCS15]. For small block sizes the communication cost is strictly better than all previous protocols. Unlike all previous protocols, it is secure against malicious adversaries.

**Table 1.** Complexity of DORAM protocols. $N$ denotes the number of records, $\kappa$ is a cryptographic security parameter, $\sigma$ is a statistical security parameter, and $D$ is the record size.

| Protocol | Communication | Parties | Security |
|---|---|---|---|
| Circuit ORAM [WCS15] (GC) | $O\left(\kappa \log^3 N + \kappa D \log N\right)$ | 2 | Semi-Honest |
| Square-root ORAM [ZWR+16] (GC) | $O\left(\kappa D \sqrt{N} \log^3 N\right)$ | 2 | Semi-Honest |
| FLORAM [Ds17] | $O\left(\sqrt{\kappa D N \log N}\right)$ | 2 | Semi-Honest |
| [HV20] | $O\left(\sqrt{\kappa D N \log N}\right)$ | 2 | Semi-Honest |
| Circuit ORAM [WCS15] (BGW) | $O\left(\log^3 N + D \log N\right)$ | 3 | Semi-Honest |
| [LO13] (BGW) | $O\left((\kappa + D) \log N\right)$ | 3 | Semi-Honest |
| [FJKW15] | $O\left(\kappa\sigma \log^3 N + \sigma D \log N\right)$ | 3 | Semi-Honest |
| [JW18] | $O\left(\kappa \log^3 N + D \log N\right)$ | 3 | Semi-Honest |
| [BKKO20] | $O\left(D\sqrt{N}\right)$ | 3 | Semi-Honest |
| DuORAM [VHG22] | $O\left(\kappa \cdot D \cdot \log N\right)$ | 3 | Semi-Honest |
| [FNO22] | $O\left((\kappa + D) \log N\right)$ | 3 | Semi-Honest |
| Our protocol | $O\left((\kappa + D) \log N\right)$ | 3 | Malicious |

## 4 Technical Overview

Our protocol is based on the Hierarchical solution [Ost90]. While this technique has primarily been applied in many client-server ORAMs [GMOT12,KLO12, LO13,PPRY18,AKLS21], we, like several other works [KM19,FNO22], apply it to DORAMs. Before understanding our protocol, it is important to understand the Hierarchical solution in general.

**The Hierarchical Solution in Client-Server ORAM:** A client-server ORAM must ensure that the physical access pattern on the server is (computationally) independent of the client's queries, regardless of the query sequence. Let us first consider a slightly weaker primitive: a protocol in which the access pattern on the server is (computationally) independent of the client's queries *provided each item is queried at most once*. This primitive is called an Oblivious Hash Table (OHTable) and is much easier to instantiate. Most common hash tables become oblivious when the hash functions themselves are pseudorandom. If the hash table can also be *constructed* on the server in a way that leaks no information about the contents, or their relation to any later queries, then a full OHTable protocol has been achieved.

An OHTable may seem significantly weaker that an ORAM, but in fact an ORAM of size $N$ can be constructed using only $\Theta(\log(N))$ OHTables through a recursive construction known as the "hierarchical solution", first introduced in [Ost90]. Assume we have access to a sub-ORAM of capacity $N/2$. The protocol then stores all $N$ elements in a single OHTable, and each time an item is accessed, the item is cached in the sub-ORAM. When an item is queried, the sub-ORAM is queried first. If the item is not in the sub-ORAM, it has not been queried in the OHTable, so it can be queried in the OHTable and this will not be a repeated query into the OHTable. On the other hand, if the item is in the sub-ORAM, we must still query the OHTable, but in this case, we query random locations in the OHTable (independent of the client's query). This ensures that if the client makes at most $N/2$ queries, no element is ever queried twice in the OHTable, and the security of the OHTable is preserved. When the sub-ORAM becomes full, its contents can be extracted, as well as the contents of the OHTable, and the OHTable can be rebuilt, with new secret keys for the PRF/hash functions. If the sub-ORAM is implemented recursively, this results in $\Theta(\log(N))$ OHTables, and a small base-case. Typically we conceptualize the OHTables as arranged vertically in a "hierarchy" of levels of geometrically increasing size, labeled from Level 0, the base-case, also called the *cache*, to the largest level of size $N$. The cache could be of constant size, though it is often of size $\Theta(\log(N))$ and in our work is larger (of size $\Theta(\kappa) = \omega(\log N)$). Since the cache is very small it can be implemented using a less efficient "ORAM."

**OMaps:** Actually, the recursive construction requires the sub-ORAMs to be slightly more versatile than an ORAM. Notice that the sub-ORAM has capacity $N/2$ but may be required to store elements from the index space $[N]$. The ORAM definition requires the size of the ORAM to be the same size as the index space.

To implement the recursive hierarchical construction, the sub-ORAMs really need to implement an *Oblivious Map (OMAP)*. An OMap is essentially just an ORAM for storing key-value pairs instead of index-value pairs. The OMap functionality is defined formally in Fig. 3. Note that most existing ORAM protocols actually instantiate the slightly stronger OMap functionality.

**Functionality $\mathcal{F}_{\mathbf{OMap}}$**

$\mathcal{F}_{\mathbf{OMap}}$**.Init(**$[\![X]\!]$**,** $[\![Y]\!]$**,** $w$**,** $n$**,** $N$**):** Store $(X_i, Y_i)$ in a dictionary for all $1 \leq i \leq w$, where $\text{len}(X) = \text{len}(Y) = w$.

$\mathcal{F}_{\mathbf{OMap}}$**.Query(**$[\![x]\!]$**, res):** If $x$ is stored in the dictionary, store the corresponding value, $y$, as $[\![res]\!]$.
If $x$ is not stored in the dictionary, store $\perp$ as $[\![res]\!]$.

$\mathcal{F}_{\mathbf{OMap}}$**.Add(**$[\![x]\!]$**,** $[\![y]\!]$**):** Store $(x, y)$ in the dictionary (writing over an old value if need be).

$\mathcal{F}_{\mathbf{OMap}}$**.Extract(res):** Create array $Z$ consisting of the key-value pairs from the dictionary, i.e. $Z_i = (X_i, Y_i)$ for some $(X_i, Y_i)$ stored in the dictionary. Pad $Z$ to length $n$ with $(\perp, \perp)$. Shuffle $Z$ randomly and store it as $[\![res]\!]$.

**Fig. 3.** OMap Functionality

**Cuckoo Hashing:** ORAMs and DORAMs often use Cuckoo Hashing [PR01] to implement OHTables (e.g. [PR10, GMOT12, LO13, KM19, PPRY18, AKL+20, FNO22].) In Cuckoo Hashing, there are 2 hash tables, and each item can be stored in one location in each table. This makes oblivious queries efficient. The hash output for each table can be revealed and both locations accessed. However, cuckoo hashing has a non-negligible failure probability. To alleviate this, items which are unable to be stored can be placed in a super-constant sized "stash".

Unfortunately, the cuckoo stash introduces some problems in the ORAM setting. To handle the cuckoo stashes, a standard approach to use a weaker OHTable which rejects a fixed number of stash elements, and store this stash in the sub-ORAM [KLO12]. There are two challenges with this approach. First, if the table is small (say $\Theta(\log(N))$), the probability of a build failure is no longer negligible in $N$. We address by making our smallest OHTable of size $\Theta(\kappa)$, thanks to our efficient QuietCache. The second problem is more subtle. The first time a stashed item is queried, it will always be found in the sub-ORAM, and random locations will be queried in the OHTable. This effectively resamples the locations that will be queried in the OHTables, which can leak information about whether the queried items were stored in the table. We use the Alibi technique [FNO21] to solve this. When a stash item is placed in the sub-ORAM during builds, it is tagged with a bit to show that the item should still be queried in the OHTable during a query. See Supplementary Material B for more details.

**The Hierarchical Solution for DORAMs:** *Distributed* ORAMs can also be built using the Hierarchical solution. Distributed Oblivious Hash Tables (DOHTables) are multi-party protocols that implement a dictionary data structure, subject to the fact that no adversarially-controlled subset of parties can learn anything about the query pattern from their views of the protocol, *provided each item is queried at most once.* Like before, we can cache responses of queries to a large DOHTable in a sub-DORAM and query the sub-DORAM first. If the item is not found in the sub-DORAM, the item is queried at the DOHTable; if it is found, the parties execute a protocol that is indistinguishable from a real, unique query to the DOHTable. By recursively implementing the sub-DORAMs using this technique, a DORAM can be constructed using $\Theta(\log(N))$ DOHTables.

**Overview of Our Solution:** One approach to building DORAM is to take an existing ORAM and simulate the client inside of a secure computation (e.g. [WCS15]). We depart from this approach, noting that DORAM actually allows for many efficiency improvements that would not be possible in a classic client-server ORAM. While DORAM has no trusted client, it does have multiple non-colluding servers which perform local computation and can communicate between each other. In particular, we examine the (3,1)-setting, where there are 3 servers and at most one corruption. This allows us to do many things more efficiently than in the client-server ORAM setting.

**1. Efficient Shuffles:** In the (3,1) setting, similar to [LWZ11] we can secret-share a list between 2 parties. These parties can then shuffle the list using a permutation known to them but not the third party. If this process is repeated 3 times, with parties taking turns to be the uninvolved party, the final permutation will be unknown to all parties. This allows us to shuffle $n$ items of size $D$ with $\Theta(nD)$ communication and small constants.

While this protocol is simple, its significance can be appreciated when compared to the difficulty of shuffling in the classical ORAM setting. In that setting, shuffling $n$ items requires $\Theta(n \log(n) D)$ communication with huge constants, or $\Theta(n \log^2(n) D)$ communication with small constants. A core insight of recent ORAM protocols [PPRY18, AKLS21] is that full shuffles can be avoided by reusing randomness and using oblivious tight compaction instead of shuffles. This brings the cost down to $\Theta(nD)$ but the constants are still impractical [DO20].

**2. Efficient multi-select:** In the $(3, 1)$ setting, it is possible to evaluate circuits of AND-depth 1 with communication equal to that of a single AND gate. Using this, we can construct an efficient multi-select protocol. That is, given $n$ secret-shared items of size $D$, we can efficient select the $k^{\text{th}}$ item for any secret-shared $k \in [n]$ with only $\Theta(n + D)$ communication. (See Sect. 6 for more details.) To the best of our knowledge, efficient multi-selects have not been used to build DORAMs prior to our work.

**3. Separating Builders from Holders:** In the classical ORAM setting, the server can see the access patterns during both builds and queries to an OHTable. This creates a problem: for efficiency the possible locations in which an item may be stored are revealed during a query. To ensure security, the build must *obliviously* move each item to its correct location. In the $(3, 1)$ setting we can instead have a single party, the *builder*, learn the locations in which items in the table may be stored. This allows the builder to *locally and non-obliviously* calculate the allocation of items to locations. After this, the table is secret-shared between the other 2 parties, called the *holders*. During a query, the holders (but not the builder) learn the locations in which the queried item may be stored and return their shares of the items in these locations. Since the adversary controls at most one party, it can either learn the physical locations of stored items (from the builder) or the potential physical locations of queried items (from a holder) but not both, preventing it from learning information about whether the queried items were stored in the table. (Our actual protocol, in fact allows the builder to construct a useful data-structure for set queries entirely by itself, and secret-share this to the holders. This is then used to build a DOHTable.)

However, in the malicious setting it is difficult to take advantage of these techniques, especially the technique of separating builders from holders. If the builder is malicious, how can we ensure that they build data structures correctly? Naturally, zero knowledge proofs allow the builder to prove any claim, but how can it do so efficiently? Furthermore, after we secret-share the data-structure between two holders, how can we guarantee that they provide the correct shares during reconstruction? (We can use a (3,1) replicated secret sharing (Sect. 2) to detect modification of shares when all three parties are involved, but this will not work with only 2 parties.) Similarly, the multi-select and shuffle protocols described above are only secure against semi-honest adversaries.

**Core Contributions:** In this paper, we show how to take advantage of the existence of multiple non-colluding servers even when one of these parties is malicious. The primary techniques are as follows:

– **Proving in zero-knowledge a distributed statement that builder built data structures correctly:** We present a method by which it can be efficiently verified that the builder has built and secret shared their data structure correctly to the two holders without revealing any information to the holders. Our method is linear in the data-structure size and has small constants.
– **Designing QuietCache and restructuring the DORAM hierarchy:** We present a more efficient distributed, oblivious, maliciously-secure cache protocol, QuietCache (Sect. 6), which serves as a top level of our DORAM hierarchy. Querying the standard cache used in the literature when it stores $n$ elements costs $O((n + D) \log N)$ communication. For works targeting the best-known complexity of $O((\kappa + D) \log N)$, this has restricted the size of the cache to $O(\log N)$. Since Cuckoo Hash Tables with a Stash (CHTwS) of $\Theta(\log N)$ elements have non-negligible failure probability and, generally speaking, all efficient-to-query OHTables are based on CHTwS, previous constructions had to design a different type of OHTable for small levels (e.g. [LO13, FNO22]). Unfortunately, we find that a small size maliciously secure

OHTables are difficult to construct. To resolve this, we design QuietCache, which costs $O(n \log N + D)$ communication to query. This allows us to have a large cache (of size $\Theta(\kappa) = \omega(\log N)$), thus completely avoiding the need for small OHTables.

- **Mixing Boolean and** $\mathbb{F}_{2^l}$ *secret-sharings:* Our solutions to the above require a combination of large-field arithmetic (for MACs and polynomial equality checks) and bit-wise operations (for equality tests and PRFs). We therefore require efficient methods of converting between these two types of secret-sharing: by using the field $\mathbb{F}_{2^l}$ we can actually convert between these two types of sharing for free.

In addition, we design an expanded, versatile Arithmetic Black Box (Sect. 5), and prove it UC-secure against a $(3, 1)$ malicious adversary. This greatly simplifies our later protocol descriptions and proofs.

## 5 The Arithmetic Black Box (ABB) Model

In order to simplify our protocol descriptions and analysis we use the Arithmetic Black Box (ABB) model. In the words of its creators an "ABB can be thought of as a secure general-purpose computer" [DN03]. The ABB is a reactive functionality that allows secret data to be "stored" and allows other functionalities to compute on secret data. This will be implemented by the stored values being secret-shared between parties, but the ABB will extract away these details. Most functionalities will take as inputs (public) identifiers to secret variables already stored in the ABB and output (public) identifiers to secret variables added to the ABB. For instance $[\![z]\!] = [\![x]\!] + [\![y]\!]$ indicates that secret variables $x$ and $y$ are already stored in the ABB, their sum is computed securely, and the sum is stored in the ABB under the name $z$.

We use bit-wise RSS as the underlying secret-sharing scheme for the ABB. Boolean operations (AND, OR, NOT) are achieved using [FLNW17], and denoted using standard infix operators ($\vee$, $\wedge$, $\neg$). For any field $\mathbb{F}_{2^l}$ $l \in \mathbb{Z}^+$ we support the addition (bitwise XOR) and multiplication using [CGH+18]. Both of these are $(3, 1)$ UC-maliciously secure protocols. Since we use RSS, we can

**Table 2.** Communication costs of ABB operations.

| ABB operation(s) | Communication (bits) |
|---|---|
| Input(x, $P_i$)/Output($[\![x]\!]$, $P_i$)/Mult($[\![x]\!]$, $[\![y]\!]$) | $\Theta(\|x\|)$ |
| RandomElement($\ell$)/Add($[\![x]\!]$, $[\![y]\!]$)/NOT($[\![x]\!]$) | 0 |
| OR($[\![x]\!]$, $[\![y]\!]$)/AND($[\![x]\!]$, $[\![y]\!]$) | $\Theta(1)$ |
| Equal($[\![x]\!]$, $[\![y]\!]$)/IfThenElse($[\![b]\!]$, $[\![x]\!]$, $[\![y]\!]$) | $\Theta(\|x\|)$ |
| CreateMAC($[\![x]\!]$)/ CheckMAC($[\![x]\!]$)/PRFEval($[\![x]\!]$, $[\![k]\!]$) | $\Theta(\|x\| + \kappa)$ |
| ReplicatedTo2Sharing($[\![x]\!]$, i, j, varName$^{i,j}$) | $\Theta(\|x\|)$ |
| 2SharingToReplicated($[x^{i,j}]^{(i,j)}$, varName) | $\Theta(\|x\|)$ |
| ObliviousShuffle($[\![X]\!]$) ($X \in (\{0,1\}^\ell)^n$) | $\Theta(n\ell)$ |
| SilentDotProduct($[\![X]\!]$, $[\![Y]\!]$, $[\![M]\!]$) ($X, Y \in (\{0,1\}^\ell)^n$ | $\Theta(\ell + \kappa)$ |

actually switch between viewing an element as a string of bits and as a field element at no cost.

We extend the ABB to support various functionalities. The full ABB is presented in Fig. 4, with costs shown in Table 2. In Supplemental Material C, we show how all of these functionalities are instantiated. Of note, we add functionalities that allow conversion to a 2-sharing. In the context of the ABB, this means that the variable names are no longer public, but are known to only 2 parties. This, for instance, allows them to access an element of a secret array using an index known to them, but not the third party. This is critical in allowing the Holders to store and access data without the Builder learning the accessed locations.

## 6 QuietCache: Maliciously-Secure Oblivious Cache Construction

In this section, we design a novel, distributed, oblivious, "cache" protocol which we will use to instantiate the topmost level of our hierarchy.

Unlike the OHTables at all other levels of the hierarchy, the cache must allow items to be queried more than once, since there is no smaller level to which an item may be moved. Furthermore, it should allow new items to be added without requiring an expensive rebuild process. We formalize the functionality that the cache must satisfy as Functionality $\mathcal{F}_{\text{OMap}}$, (Fig. 3).

In similar works, the cache is often instantiated by executing a linear-scan under MPC [FNO22] this has append complexity $O(1)$ and query complexity $O((D + \log N)c)$ where $c$ is the number of elements in the cache.

There is a fundamental tension here regarding the size of the cache. Since every (D)ORAM query accesses the cache, performing a linear scan of the cache adds $\Omega((D + \log N)c)$ to the (D)ORAM query complexity. When $c = \Omega(\log N)$, querying the cache becomes the bottleneck for the entire (D)ORAM protocol, so most (D)ORAM protocols set $c = O(1)$. Unfortunately, there are multiple problems with a small cache. First, the "cache-the-stash" technique requires a cache of at least size $\Omega(\log(N))$. Second, small cuckoo hash tables always have a non-negligible probability of build failure [Nob21], and when the cache ($L_0$) is small, so are the smaller levels in the hierarchy ($L_1, L_2, \ldots$.) For this reason, many hierarchical (D)ORAM protocols (e.g. [LO13]) are forced to use different types of tables for the smaller levels of the (D)ORAM hierarchy.

We resolve this tension by designing a novel, distributed, oblivious cache protocol $\Pi_{\text{QuietCache}}$ that allows us to increase $c$ to $c = \kappa = \omega(\log N)$, while still maintaining efficient access to the cache. Notably, our protocol requires $O(\max D, \kappa)$ communication to store a new item and $O(D + n \log N)$ communication to query an item. This will mean that our smallest OHTables will be of size $\Omega(\kappa) = \omega(\log(N))$, allowing them to instantiate cuckoo hash tables with a stash with at most negligible build failure negligible in $N$, as required.

Our protocol, $\Pi_{\text{QuietCache}}$ works as follows. The protocol maintains an array of all items that have been added (either during initialization or later), with

**Functionality $\mathcal{F}_{\mathbf{ABB}}$**

Inputs and outputs are in $\mathbb{F}_{2^l}$ for various $l \in \mathbb{Z}$. All functionalities allow the adversary to abort the protocol, seeing their outputs first if applicable.

$\mathcal{F}_{ABB1}$: **Basic ABB Functionalities**

**Input(x, $P_i$):** Receive $x$ from party $P$ and return $[\![x]\!]$.
**RandomElement(l):** Sample $x \in_R \mathbb{F}_{2^l}$, return $[\![x]\!]$.
**OR($[\![x]\!]$, $[\![y]\!]$):** For $x, y \in \mathbb{F}_2$, compute $z = x \vee y$. Return $[\![z]\!]$.
**Add($[\![x]\!]$, $[\![y]\!]$):** Compute $z = x + y$ and return $[\![z]\!]$.
**Mult($[\![x]\!]$, $[\![y]\!]$):** Compute $z = x * y$ and return $[\![z]\!]$.
**Output($[\![z]\!]$, $P_i$):** Output $z$ to $P_i$.

$\mathcal{F}_{ABB2}$: **Circuit-based Operations**

**Equal($[\![x]\!]$, $[\![y]\!]$):** If $x \overset{?}{=} y$ set $z$ to 1, otherwise to 0. Return $[\![z]\!]$.
**IfThenElse($[\![b]\!]$, $[\![x]\!]$, $[\![y]\!]$):** If $b \overset{?}{=} 1$, set $z$ to $x$, otherwise set $z$ to $y$. Return $[\![z]\!]$.
**CreateMAC($[\![x]\!]$):** Create a "tag," $\tau$, on the data $X$, and return $[\![\tau]\!]$.
**CheckMAC($[\![x]\!]$, $[\![\tau]\!]$):** If $\tau$ is a valid tag on the message, $x$, return $[\![1]\!]$, otherwise return $[\![0]\!]$.
**PRFEval($[\![x]\!]$, $[\![k]\!]$):** Compute $z = \mathrm{PRF}_k(x)$, a pseudorandom function on input $x$ over key $k \in \mathbb{F}_{2^\kappa}$. Return $[\![z]\!]$.

$\mathcal{F}_{ABB3}$: **Sharing to and from a 2-sharing**

**ReplicatedTo2Sharing($[\![x]\!]$, i, j, varName$^{i,j}$):** Given a handle, varName, known to (distinct) $P_i$ and $P_j$, store $x$ as $[varName]^{(i,j)}$
**2SharingToReplicated($[x^{i,j}]^{(i,j)}$, varName):** Given a handle, $x^{i,j}$, known to (distinct) $P_i$ and $P_j$, for a variable stored in $[x]^{(i,j)}$ and a public handle, varName, store $x^{i,j}$ in $[\![varName]\!]$

$\mathcal{F}_{ABB4}$: **Specialized Functionalities**

**ObliviousShuffle($[\![X]\!]$):** Let $X = X_0, \ldots, X_{n-1}$. Sample a random (secret) permutation $\pi \in_R S_n$ and return $[\![\pi(X)]\!] = [\![\pi(X_0)]\!], \ldots, [\![\pi(X_l)]\!]$. This naturally allows multiple arrays of the same length to be shuffled using the same secret shuffle, by combining elements at the same index from different arrays, shuffling, then separating the elements into their original arrays. We denote this by ObliviousShuffle having multiple inputs and outputs.
**SilentDotProduct($[\![X]\!]$, $[\![Y]\!]$, $[\![M]\!]$):** $X = X_0, \ldots, X_{n-1}$, $Y = Y_0, \ldots, Y_{n-1}$. $M_i = \alpha Y_i$ for all $0 \leq i \leq n-1$. Compute $z = \sum_{i=0}^{n-1} X_i Y_i$ and return $[\![z]\!]$.

**Fig. 4.** Arithmetic Black Box functionality.

**Protocol $\Pi_{\mathbf{QuietCache}}$**

**Hybrids:** The protocol is defined in the $\mathcal{F}_{\mathrm{ABB}}$-hybrid model.

$\Pi_{\mathbf{QuietCache}}$**.Init**$(\llbracket X \rrbracket, \llbracket Y \rrbracket, w, n, N)$**:**

```
1. Store arrays $\llbracket X \rrbracket$ and $\llbracket Y \rrbracket$.
2. Initialize counter $t$ to $w$.
3. Initialize MACs for all elements. For $i \in [w]$: $\llbracket M_i \rrbracket = \mathcal{F}_{\mathrm{ABB}}.\mathrm{CreateMAC}(\llbracket Y_i \rrbracket)$
```

$\Pi_{\mathbf{QuietCache}}$**.Store(**$\llbracket x \rrbracket$**,** $\llbracket y \rrbracket$**):**

```
1. Increment $t$.
2. Set $\llbracket X_t \rrbracket = \llbracket x \rrbracket$, $\llbracket Y_t \rrbracket = \llbracket y \rrbracket$.
3. Create MAC for new value: $\llbracket M_t \rrbracket = \mathcal{F}_{\mathrm{ABB}}.\mathrm{CreateMAC}(\llbracket y \rrbracket)$
```

$\Pi_{\mathbf{QuietCache}}$**.Query(**$\llbracket X \rrbracket$**, outName):**

```
1. First create a $t$-bit indicator array, $\llbracket b \rrbracket$ which shows the index of the copy of
   $\llbracket x \rrbracket$ that was most recently stored (or the all-zero vector if $\llbracket x \rrbracket$ has never been
   stored).
   (a) For $i = t, \dots, 1$
       i. $\llbracket isMatch_i \rrbracket = \mathcal{F}_{\mathrm{ABB}}.\mathrm{Equal}(\llbracket X_i \rrbracket, \llbracket X \rrbracket)$
       ii. $\llbracket isBeforeMatch_i \rrbracket = \llbracket isBeforeMatch_{i+1} \rrbracket \vee \llbracket isMatch_{i+1} \rrbracket$ (Except that
           $\llbracket isBeforeMatch_t \rrbracket = 0$)
       iii. $\llbracket b_i \rrbracket = \llbracket isMatch_i \rrbracket \wedge (\neg \llbracket isBeforeMatch_i \rrbracket)$
2. Then efficiently select the item based on this indicator array using
   $\mathcal{F}_{\mathrm{ABB}}$.SilentDotProduct:
   (a) $\llbracket y \rrbracket = \mathcal{F}_{\mathrm{ABB}}.\mathrm{SilentDotProduct}((\llbracket b_i \rrbracket^D)_{i=1}^t, \llbracket Y_i \rrbracket_{i=1}^t, \llbracket M_i \rrbracket_{i=1}^t)$
   (b) Return $\llbracket y \rrbracket$.
```

$\Pi_{\mathbf{QuietCache}}$**.Extract():**

```
1. For every index, $x$, set all but the latest copy of $(x, y)$ to $(\bot, \bot)$. For $i =
   1, \dots, t$:
   (a) $\llbracket \hat{X} \rrbracket_i = \llbracket X \rrbracket_i$, $\llbracket \hat{Y} \rrbracket_i = \llbracket Y \rrbracket_i$
   (b) For $i < j \le t$ :
       i. $\llbracket b_{ij} \rrbracket = \mathcal{F}_{\mathrm{ABB}}.\mathrm{Equal}(\llbracket X_i \rrbracket, \llbracket X_j \rrbracket)$
       ii. $(\llbracket \hat{X}_i \rrbracket, \llbracket \hat{Y}_i \rrbracket) = \mathcal{F}_{\mathrm{ABB}}.\mathrm{IfThenElse}(\llbracket b_{ij} \rrbracket, (\llbracket \bot \rrbracket, \llbracket \bot \rrbracket), (\llbracket \hat{X}_i \rrbracket, \llbracket \hat{Y}_i \rrbracket))$
2. Shuffle items and return the result:
   (a) $\llbracket \hat{X} \rrbracket, \llbracket \hat{Y} \rrbracket = \mathcal{F}_{\mathrm{ABB}}.\mathrm{ObliviousShuffle}(\llbracket \hat{X} \rrbracket, \llbracket \hat{Y} \rrbracket)$
   (b) Return $\llbracket \hat{X} \rrbracket, \llbracket \hat{Y} \rrbracket$.
```

**Fig. 5.** $\Pi_{\mathrm{QuietCache}}$: Protocol for the cache (implementation of smallest $\mathcal{F}_{\mathrm{OMap}}$).

items that were added later appearing later in the array. When a new item is added, $\Pi_{\text{QuietCache}}$ does not attempt to delete the old item, but merely places the new item at the end of the array to indicate it is newer. Authentication tags are also added to values each time an item is inserted, which will later allow for efficient queries. To query, we perform a linear scan of the indices, but not the values. We create a bit-array that is 1 in the location of the array where the index was most recently added (if any) and 0 elsewhere. Since the values are all authenticated, we can use our bit-array to very efficiently access the correct value using $\mathcal{F}_{\text{ABB}}$.SilentDotProduct (Fig. 4). In the honest-majority 3-party setting, this is very efficient and has essentially the same cost as a single multiplication. Leveraging the silent dot product is the key trick which enables $\Pi_{\text{QuietCache}}$'s efficiency. Finally, when items need to be extracted we need to delete old copies of items. We do this using a brute-force check under MPC.

We now show that $\Pi_{\text{QuietCache}}$ implements $\mathcal{F}_{\text{OMap}}$ securely.

**Proposition 1.** *Against a static malicious adversary controlling at most one party out of three, Protocol $\Pi_{QuietCache}$ (Fig. 5) UC-realizes functionality $\mathcal{F}_{QuietCache}$ (Fig. 5) with abort in the $\mathcal{F}_{ABB}$-hybrid model.*

The proof of Proposition 1 is in Appendix G.1. In Appendix H.1 we prove that:

**Proposition 2.** *The communication complexity of $\Pi_{QuietCache}$.Init, $\Pi_{QuietCache}$.Store, $\Pi_{QuietCache}$.Query, $\Pi_{QuietCache}$.Extract is* $\Theta(\kappa w), O(\max D, \kappa), O(D + n \log N), O(n^2 \log N + nD)$, *respectively.*

## 7 Maliciously-Secure Oblivious Set Construction

At a high level, our DORAM has a hierarchy of Oblivious Hash Tables (OHTables), one in each level. It was observed by [MZ14] that once it is known whether an item is in a given level, it is much easier to access it obliviously. We therefore adopt the approach of [FNO22] to first have a protocol exclusively to securely determine whether the item exists at a given level. We call such a protocol a *Distributed Oblivious Set* or OSet and we implement (a variant of) this functionality in this section. In the next section (Sect. 8) we use this as a sub-protocol to build (a variant of) an OHTable.

At a high level, $\Pi_{\text{OSet}}$ obtains efficiency by separating the players into the roles of "builder" and "holders" [LO13, FNO22]. The Builder constructs a data structure locally, which is secret-shared between two Holders. The Builder can learn information about where data is stored in the data structure during a build, while the Holders can learn the locations that queried items may be located during queries. If an adversary can only corrupt a single party it therefore is unable to use this information to learn whether queried items are stored in the table.

There are two major challenges with this approach. The first is achieving *privacy.* The Builder must somehow build the data structures based on knowledge of the *locations* of items, without learning any information about the items

**Functionality $\mathcal{F}_{\text{OSet}}$**

$\mathcal{F}_{\text{OSet}}$**.Build(**$[\![X]\!]$**,** $n$**,** $N$**, stash):** $X$ is an array of $n$ distinct items, each chosen from $[N]$, which is stored in the ABB. Set $s = \log(N)$. It is assumed that $n = \omega(s)$.

1. Let $S$ be an arbitrary bit array of length $n$, with $s$ ones and $n - s$ zeros.
2. Store $S$ in $[\![stash]\!]$ in the ABB.

$\mathcal{F}_{\text{OSet}}$**.Query(**$[\![x]\!]$**, res):**

1. If $x \in \{X_i\}_{i\in[n]\setminus S}$, then set $z = 1$, otherwise set $z = 0$. That is, set $z$ to 1 iff $x$ is one of the $n - s$ elements that are stored.
2. Store $z$ in $[\![res]\!]$ in the ABB.

**Protocol $\Pi_{\text{OSet}}$**

**Hybrids:** $\mathcal{F}_{\text{ABB}}, \mathcal{F}_{\text{ZKPOfValidCHT}}$.
$\Pi_{\text{OSet}}$**.Build(**$[\![X]\!]$**,** $n$**,** $N$**, stash):**

1. Set public parameters for the Cuckoo Hash Table. Table size $c = 2^{\lfloor \log_2(3n) \rfloor}$, stash size $s = \log N$, and hash functions
$$h_0(x) \overset{\text{def}}{=} 0 \mid\mid x[0 : \log(c) - 1] \quad (1)$$
$$h_1(x) \overset{\text{def}}{=} 1 \mid\mid x[\log(c) : 2\log(c) - 1] \quad (2)$$
2. Generate a secret-shared PRF key: $[\![k]\!] = \mathcal{F}_{\text{ABB}}.\text{RandomEl}(\kappa)$.
3. Evaluate the SISO-PRF on inputs: $[\![Q_i]\!] = \mathcal{F}_{\text{ABB}}.\text{PRF}([\![X_i]\!], [\![k]\!])$ for $i \in [n]$.
4. Reveal PRF evaluations to $P_0$: $Q = \mathcal{F}_{\text{ABB}}.\text{Reveal}([\![Q]\!], \{0\})$
5. $P_0$ locally builds a CHTwS of the PRF evaluations $\mathsf{CHT} \cup \mathsf{Stash} = \{\mathsf{CHT}, (i_1, \ldots, i_s)\} = \mathsf{BuildCHTwS}(\hat{Q}, h_0, h_1)$
6. Define the vector $S \in \{0,1\}^n$, and set $S_i = 1$ if $i \in \{i_1, \ldots, i_s\}$.
7. $P_0$ secret shares the CHT and the stash bit-array:
   (a) $[\![\mathsf{CHT}]\!] = \mathcal{F}_{\text{ABB}}.\text{Input}(\mathsf{CHT}, 0)$.
   (b) $[\![stash]\!] = \mathcal{F}_{\text{ABB}}.\text{Input}(S, 0)$.
8. Verify that there are $s$ stash elements, and remove these:
   (a) $[\![\hat{Q}]\!], [\![\hat{S}]\!] = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}([\![Q]\!], [\![S]\!])$
   (b) For all $i \in [n]$, $\hat{S}_i = \mathcal{F}_{\text{ABB}}.\text{Output}([\![\hat{S}_i]\!])$
   (c) If there are exactly $s$ values in $\hat{S}_i$ set to 1 continue, else abort.
   (d) $[\![\tilde{Q}]\!] = \{[\![\hat{Q}_i]\!]\}_{\hat{S}_i=0}$.
9. Verify that $P_0$ built and shared a *valid* CHT on the non-stash elements:
   (a) $[\![b]\!] = \mathcal{F}_{\text{ZKPOfValidCHT}}.\textbf{Verify}([\![\tilde{Q}]\!], [\![\mathsf{CHT}]\!], c, h_0, h_1, \log(N))$
   (b) $b = \mathcal{F}_{\text{ABB}}.\text{Reveal}([\![b]\!])$. Abort if not $b$.
10. Secret-share the CHT to $P_1$ and $P_2$:
   $[\mathsf{CHT}]^{(1,2)} = \mathcal{F}_{\text{ABB}}.\text{ReplicatedTo2Sharing}([\![\mathsf{CHT}]\!], \{1,2\})$.

$\Pi_{\text{OSet}}$**.Query(**$[\![x]\!]$**, res):**

1. $[\![q]\!] = \mathcal{F}_{\text{ABB}}.\text{PRFEval}([\![x]\!], [\![k]\!])$.
2. $q = \mathcal{F}_{\text{ABB}}.\text{Reveal}([\![q]\!], \{1,2\})$
3. $P_1$ and $P_2$ access the locations in the CHT which may store $q$ and re-share their contents without revealing the locations to $P_0$:
$$[\![Q_b^*]\!] = \mathcal{F}_{\text{ABB}}.\text{2SharingToReplicated}([\mathsf{CHT}[h_b(q)]]^{(1,2)}) \text{ for } b \in \{0,1\}$$
4. $[\![res]\!] = \mathcal{F}_{\text{ABB}}.\text{Equal}([\![q]\!], [\![Q_0^*]\!]) \vee \mathcal{F}_{\text{ABB}}.\text{Equal}([\![q]\!], [\![Q_1^*]\!])$

**Fig. 6.** $\mathcal{F}_{\text{OSet}}$and $\Pi_{\text{OSet}}$: Functionality and Protocol for a Distributed Oblivious Partial Set

**Functionality $\mathcal{F}_{\mathbf{ZKPOfValidCHT}}$**

$\mathcal{F}_{\mathbf{ZKPOfValidCHT}}$**.Verify(**$[\![X]\!]$**,** $[\![\mathsf{CHT}]\!]$**,** $c$**,** $h_0$**,** $h_1$**,** $\ell$**, varName):**
$X$ is an array of length $n$ of unique elements from $\{0,1\}^\ell$, stored in the ABB. $CHT$ is an array of length $2c$ of elements from $\{0,1\}^\ell \bigcup \{\perp\}$, stored in the ABB. $h_0, h_1 : 2^\ell \to \{0,\ldots,c-1\}$ are two public hash functions.
If $CHT$ stores $X$ and $2c-n$ copies of $\perp$ the CHT stores exactly the correct set. If for every $CHT_i \neq \perp$, $h_0(CHT_i) = i$ or $h_1(CHT_1) = i + c$, the stored items are in the correct positions. If either condition is false, set $z = 0$, otherwise set $z = 1$. Store $z$ in the ABB under handle varName.

**Protocol $\Pi_{\mathbf{ZKPOfValidCHT}}$**

$\Pi_{\text{ZKPOfValidCHT}}$.Verify($[\![X]\!]$, $[\![CHT]\!]$, $c$, $h_0$, $h_1$, $S$, varName):

1. Check CHT holds correct set using Multi-Set Polynomial Check.
   (a) Append $2c-n$ copies of $[\![\perp]\!]$ to array $[\![X]\!]$.
   (b) Represent $[\![X]\!]$ and $[\![CHT]\!]$ as items from $GF(2^l)$, where $l = \max\{S+1, \sigma+\log(2c)\}$. Specifically, represent $\perp$ as $0^\ell$ and add the prefix $1||O^{\ell-S-1}$ to all real elements.
   (c) Pick a random evaluation point for the polynomial:
   $[\![w]\!] = \mathcal{F}_{\text{ABB}}$.RandomElement($\kappa$)
   (d) Using $\mathcal{F}_{\text{ABB}}$.Mult and $\mathcal{F}_{\text{ABB}}$.Add, securely evaluate the polynomial for the input elements (with copies of $\perp$):
   $$[\![u]\!] = \prod_{[\![a]\!] \in [\![X]\!]} ([\![a]\!] - [\![w]\!])$$
   (e) Likewise evaluate the polynomial for the contents of the CHT:
   $$[\![v]\!] = \prod_{[\![b]\!] \in [\![CHT]\!]} ([\![b]\!] - [\![w]\!])$$
   (f) Check that the evaluations are the same:
   $[\![check_1]\!] = \mathcal{F}_{\text{ABB}}$.Equal($[\![u]\!], [\![v]\!]$)

2. Verify CHT locations are either empty ($\perp$) or are real items in a valid position.
   (a) For all $i \in \{0,\ldots,2c-1\}$
      i. $[\![empty_i]\!] = \mathcal{F}_{\text{ABB}}.\text{Equal}(CHT_i, \perp)$
      ii. $[\![eq_{0,i}]\!] = \mathcal{F}_{\text{ABB}}.\text{Equal}(i, h_0(CHT_i))$
      iii. $[\![eq_{1,i}]\!] = \mathcal{F}_{\text{ABB}}.\text{Equal}(i, h_1(CHT_i))$
      iv. Using $\mathcal{F}_{\text{ABB}}$.OR securely evaluate
      $$[\![b_i]\!] = [\![empty_i]\!] \vee [\![eq_{0,i}]\!] \vee [\![eq_{1,i}]\!]$$
   (b) Using $\mathcal{F}_{\text{ABB}}$.AND evaluate
   $$[\![check_2]\!] = \bigwedge_{i \in \{0,\ldots,2c-1\}} [\![b_i]\!]$$
   .
3. Set $[\![varName]\!] = \mathcal{F}_{\text{ABB}}.\text{OR}(check_1, check_2)$

**Fig. 7.** $\mathcal{F}_{\text{ZKPOfValidCHT}}$ and $\Pi_{\text{ZKPOfValidCHT}}$: Functionality and Protocol for verifying in zero knowledge the correctness of a Cuckoo Hash Table.

themselves. The second is ensuring *correctness*. In the malicious setting, there must be a method to verify that the Builder constructed data structures correctly. If the Builder were to place an item in the incorrect location, the protocol would not find the item during queries.

We address the privacy challenge by storing pseudorandom "tags" (based on a PRF applied each item) rather than the items themselves. We evaluate the PRF inside of the ABB, so only the PRF output is revealed to the Builder. The security of the PRF guarantees that no information about the items themselves is leaked by their outputs. It also guarantees that collisions occur with negligible probability, so an item will be in the set only if its PRF is in the set of PRF evaluations (except with negligible probability).

We address the correctness challenge by the protocol proving, in zero-knowledge, that the data structure which the Builder constructed and shared is a valid Cuckoo Hash Table of the underlying data. First, we must prove that the set of items in the table is equal to the set of items that should be there. We prove this using a multi-set polynomial equality test. Second, we must prove that each item is in a correct location. This is done by evaluating the hash functions on each item in the table ensuring that the table location matches one of these hash functions. While we will describe our verification protocol in terms of general hash functions, in our case, since the item is itself the output of a PRF, it actually suffices for our "hash functions" to simply be bit-truncations of the items. This is very efficient: the bit-truncation itself requires no communication, after which we can evaluate a standard circuit for an equality test.

Note that we verify the first property using polynomials over large fields whereas we verify the second property using bitwise operations. We can do this efficiently due to the fact that we represent data in the field $\mathbb{F}_{2^\ell}$, which is also a valid Boolean sharing (i.e., over $\mathbb{F}_2^\ell$) (see Appendix C.1). This allows us to convert between these sharings for free. We therefore cast the data as a field for efficient polynomial evaluation, while casting it as a Boolean array for efficient bit-wise equality testing.

One final challenge in constructing our OSet is handling the stash. We will use Cuckoo Hashing with a Stash in order to ensure that the build failure probability is negligible. However, for efficiency, the stashed items will not be part of the OSet (or OHTable), but will instead be inserted into a sub-DOMap. As such, we will not implement a full Oblivious Set storing all $n$ items, but a Distributed Oblivious Partial-Set storing $n - s$ of the $n$ items, and rejecting the $s$ items in the stash. However, allowing the stash to leave the protocol/functionality is risky. If information about which queries correspond to stashed items is leaked, this breaks the obliviousness of queries. For instance, the locations of stashed elements necessary collide with elements that were stored in the OSet. This means that if a Holder is corrupted and the environment knows some queries that correspond to stashed elements, it can conclude that any other query that accesses the same locations is more likely to have been a member of the set. This coin has another side to it: if the environment can *influence* the probability of a stashed item being queried compared to a stored item it can likewise cause the accesses to be dependent. (This is exploited for instance by the Alibi attack (Appendix B) where stash items are never queried, which leaks information about whether the other queried items were in the set.) Our OSet functionality will therefore have the limitation that no information about the stash leaves the ABB *outside the protocol*, and the calls to build do not depend on which items were stashed (even conditioned on ABB-revealed values) to avoid leaking information *inside the protocol*.

Our OSet also has the limitation that it is only secure if queries are never repeated. Furthermore, we will need to limit the number of queries to the OSet data structure. We will later show that the uses of our OSet by the larger protocol obey all these restrictions. These restrictions are formalized in the following conditions:

**Condition 1 (No Repeats).** *For all $x$, $Query([\![x]\!], res)$ is called at most once.*

**Condition 2 (Limited Queries).** *Query is called at most $n$ times.*

**Condition 3 (ABB-Stash Independence).** *Let $stash_1$, $stash_2$ be two different possible values of stash. The distribution of all outputs of the ABB by the environment when $stash = stash_1$ must be computationally indistinguishable from the distribution when $stash = stash_2$.*

**Condition 4 (Query-Stash Independence).** *Let stash be the output of the Build. If $x = X_i$ for any $i \in [n]$, the probability that $Query(x, res)$ occurs/occurred, conditioned on any values revealed by $\mathcal{F}_{ABB}$ either before or after, is computationally indistinguishable from independent of $stash_i$.*

Our OSet functionality and protocol are presented in Fig. 6. This makes use of our functionality for verifying, in zero-knowledge, that the Builder ($P_0$) correctly constructed the Cuckoo Hash table (on the non-stash elements). This functionality, and the protocol that implements it, are presented in Fig. 7. We now prove that these protocols correctly implement the desired functionalities.

**Proposition 3.** *Protocol $\Pi_{ZKPOfValidCHT}$ statistically UC-securely implements $\mathcal{F}_{ZKPOfValidCHT}$ in the $\mathcal{F}_{ABB}$-hybrid model.*

*Proof.* Note that this protocol makes no assumptions about the parties or the adversary setting, as all operations are exclusively within the ABB. It inherits whichever security the ABB is implemented with. Implementing with our ABB from Fig. 4 yields a 3-party protocol with statistical UC-security with abort against a malicious adversary statically corrupting one party. Also, note that this protocol and functionality provide no guarantees that $CHT$ was chosen uniformly at random from the set of valid CHTs for $X$, only that it was one such valid CHT.

By Corollary 1, since $\Pi_{\mathrm{ZKPOfValidCHT}}$ does not reveal any values, it suffices to prove that the output stored in the ABB is correct (except with negligible probability).

Let $f(x) = \Pi_{[\![a]\!] \in [\![X]\!]}([\![a]\!] - x) - \Pi_{[\![b]\!] \in [\![CHT]\!]}([\![b]\!] - x)$. If $[\![X]\!]$ and $[\![CHT]\!]$ contain the same multiset, then $f(x)$ will be the zero polynomial. Otherwise, it will be a non-zero polynomial of degree at most $2c$. In this case, by the Schwartz-Zippel Lemma, the probability that $f(x)$ evaluates to 0 on a point chosen randomly from $GF(2^\ell)$ is at most $\frac{2c}{2^\ell}$, which is at most $2^{-\sigma}$. Note that $u = v$ if and only if $f(w) = 0$, where $w$ was chosen randomly from $GF(2^\ell)$. Therefore $check_1 = 1$ if and only if $[\![X]\!] = [\![CHT]\!]$, except with negligible probability.

Now we examine the part of the $\Pi_{\text{ZKPOfValidCHT}}$ that verifies that items are in the correct locations. If $check_1 = 1$, every item in $X$ is in the table (except with negligible probability). Assuming this is true, if every item is stored in a correct location, $check_2$ will evaluate to 1, otherwise it will evaluate to 0. (If $check_1 = 0$, then it does not matter what $check_2$ evaluates to as $varName$ will be set to 0.) Therefore $varName$ will be set to 1 if, and only if, all items in $X$ are stored in $CHT$ at a correct location.

We now prove that $\Pi_{\text{OSet}}$ realizes $\mathcal{F}_{\text{OSet}}$ subject to our conditions:

**Proposition 4.** *Against a static malicious adversary controlling at most one party out of three and an environment satisfying Conditions 1, 2, 3 and 4 Protocol $\Pi_{OSet}$ (Fig. 6) statistically (with failure probability negligible in $N$) realizes functionality $\mathcal{F}_{OSet}$ (Fig. 6) with abort in the $\mathcal{F}_{ABB}, \mathcal{F}_{ZKPOfValidCHT}$-hybrid model.*

The proof of Proposition 4 is in Appendix G.2. In Appendix H.2 we prove that:

**Proposition 5.** *$\Pi_{OSet}$.Build has complexity $O(n(\kappa + D))$ and $\Pi_{OSet}$.Query has complexity $O(\kappa)$.*

## 8 Maliciously-Secure Oblivious Hash Table Construction

In this section, we build a Distributed Oblivious Hash Table (OHTable) using the OSet protocol outlined in Sect. 7. The OHTable is a protocol for securely mapping indices to values provided each item is only queried once.

The purpose of the OSet is to check whether the item being queried is in the domain of the Hash Table. If so, the item will be accessed in the ABB based on a public tag (which is a PRF evaluation of the index). If not, a pre-inserted dummy item will be accessed based on its public tag (which is a PRF evaluation of a counter). The real items and pre-inserted dummies are shuffled prior to the tags being revealed, hiding which items are real.

The OHTable's Query function will also provide a way to do a no-op query that is indistinguishable from a real query. This will be critical in ensuring the no-repeats condition is satisfied: when the DORAM is queried multiple times for an item, it will query the item in the OHTable the first time and henceforth will ask the OHTable to perform a no-op query. Additionally, our OHTable supports an Extract functionality which returns (in the ABB) an array of the items which were not queried (padded to length $n$ with copies of $(\perp, \perp)$).

**Functionality $\mathcal{F}_{\mathbf{OHTable}}$**

**Build(($\llbracket X \rrbracket$,$\llbracket Y \rrbracket$, $n$, $N$, $X^{\mathbf{stash}}$, $Y^{\mathbf{stash}}$):**
$X$ is an array of $n$ distinct elements from $[N]$. $Y$ is an array of $n$ elements of length $D$ bits. Let $s = \log(N)$. An arbitrary array of $s$ distinct items from $X$, and their corresponding values from $Y$ are stored in the ABB under handles $X^{\text{stash}}$ and $Y^{\text{stash}}$ respectively.
**Query($\llbracket x \rrbracket$, res):**
$x \in [N] \cup \{\perp\}$. If $\exists j \in [n]; x = X_j$ and $x \notin X^{\text{stash}}$, set $z = \llbracket Y_j \rrbracket$. Else, set $z = \llbracket \perp \rrbracket$. Store $z$ in the ABB under handle $res$.
**Extract(res):**
For all $i \in [n]$ such that $X_i \notin X^{\text{stash}}$ and Query$(X_i, res)$ was never called store $(X_i, Y_i)$ in an array $Z$. Pad $Z$ to length $n - s$ with copies of $(\perp, \perp)$. Randomly shuffle $Z$. Store $Z$ in the ABB under handle $res$.

**Protocol $\Pi_{\mathbf{OHTable}}$**

**Hybrids:** $\mathcal{F}_{\text{ABB}}$, $\mathcal{F}_{\text{OSet}}$.
**Build($\llbracket X \rrbracket$, $\llbracket Y \rrbracket$, $n$ $N$, $X^{\mathbf{stash}}$, $Y^{\mathbf{stash}}$):**

1. Build OSet from the indices. Create arrays of the stashed indices and values:
   - (a) $\llbracket S \rrbracket = \mathcal{F}_{OSet}.\text{Build}(\llbracket X \rrbracket, n)$
   - (b) $\llbracket \tilde{S} \rrbracket, \llbracket \tilde{X} \rrbracket, \llbracket \tilde{Y} \rrbracket = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}(\llbracket S \rrbracket, \llbracket X \rrbracket, \llbracket Y \rrbracket)$
   - (c) $\tilde{S} = \mathcal{F}_{\text{ABB}}.\text{Output}(\llbracket \tilde{S} \rrbracket)$
   - (d) $\llbracket X^{\text{stash}} \rrbracket, \llbracket Y^{\text{stash}} \rrbracket = (\llbracket \tilde{X}_i \rrbracket, \llbracket \tilde{Y}_i \rrbracket)_{\tilde{S}_i = 1}$
2. Tag items with PRF evaluations of the indices under a new PRF key:
   - (a) $\llbracket k \rrbracket = \mathcal{F}_{\text{ABB}}.\text{RandomElement}(\kappa)$
   - (b) $\llbracket Q \rrbracket = \{\mathcal{F}_{ABB}.\text{PRFEval}(\llbracket X_i \rrbracket, \llbracket k \rrbracket)\}_{i \in [n], \tilde{S}_i = 0}$
   - (c) For $i \in [n - s]$, let $\llbracket Q_{n-s+i} \rrbracket = \mathcal{F}_{\text{ABB}}.\text{PRFEval}(\llbracket N + i \rrbracket, \llbracket k \rrbracket)$, $\llbracket X_{n-s+i} \rrbracket = \llbracket \perp \rrbracket$ and $\llbracket Y_{n-s+i} \rrbracket = \llbracket \perp \rrbracket$
3. Build data structure allowing items to be accessed based on their tags:
   - (a) $\llbracket \hat{Q} \rrbracket, \llbracket \hat{X} \rrbracket, \llbracket \hat{Y} \rrbracket = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}(\llbracket Q \rrbracket, \llbracket X \rrbracket, \llbracket Y \rrbracket)$
   - (b) For $i \in [2n - 2s]$, set $\hat{Q}_i = \mathcal{F}_{\text{ABB}}.\text{Output}(\hat{Q}_i)$
   - (c) Store $(\hat{Q}_i, i)_{i \in [2n-2s]}$ in a local dictionary.
4. Initialize local query counter: $t = 0$

**Query($\llbracket x \rrbracket$, res):**

1. Locally increment counter: $t = t + 1$.
2. Query $x$ to the OSet or, if $x = \perp$, query a counter (not in $[N]$) to the OSet.
   - (a) $\llbracket x_{OSet} \rrbracket = \mathcal{F}_{\text{ABB}}.\text{IfThenElse}(\llbracket x \rrbracket \overset{?}{=} \llbracket \perp \rrbracket, \llbracket N + t \rrbracket)$
   - (b) $\llbracket in \rrbracket = \mathcal{F}_{OSet}.\text{Query}(\llbracket x_{OSet} \rrbracket)$
3. If the item was found in the OSet, access the item's value using its tag, otherwise access a pre-inserted dummy:
   - (a) $\llbracket x_{used} \rrbracket = \mathcal{F}_{ABB}.\text{IfThenElse}(\llbracket in \rrbracket, \llbracket x \rrbracket, \llbracket N + t \rrbracket)$
   - (b) $\llbracket q \rrbracket = \mathcal{F}_{ABB}.\text{PRFEval}(\llbracket x_{used} \rrbracket, \llbracket k \rrbracket)$
   - (c) $q = \mathcal{F}_{ABB}.\text{Output}(\llbracket q \rrbracket)$
   - (d) Find $i$ such that $q = \hat{Q}_i$.
   - (e) Set $\llbracket res \rrbracket = \llbracket \hat{Y}_i \rrbracket$
4. Delete the accessed item: Delete $\llbracket \hat{Y}_i \rrbracket$ from $\llbracket \hat{Y} \rrbracket$, $\llbracket \hat{X}_i \rrbracket$ from $\llbracket \hat{X} \rrbracket$ and $\llbracket \hat{Q}_i \rrbracket$ from $\llbracket \hat{Q} \rrbracket$.

**Extract(res):**

1. Shuffle the remaining items of $\llbracket \hat{X} \rrbracket$ and $\llbracket \hat{Y} \rrbracket$ and return them:
   - (a) $\llbracket \bar{X} \rrbracket, \llbracket \bar{Y} \rrbracket = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}(\llbracket \hat{X} \rrbracket, \llbracket \hat{Y} \rrbracket)$.
   - (b) Store $\llbracket \bar{X} \rrbracket, \llbracket \bar{Y} \rrbracket$ in $\llbracket res \rrbracket$.

**Fig. 8.** $\mathcal{F}_{\text{OHTable}}$ and $\Pi_{\text{OHTable}}$: Functionality and Protocol for Distributed Oblivious Partial Hash-Table

Since the OHTable uses our OSet construction which generates a stash, our OHTable will also generate a stash. The stash elements will be not be stored in the table; they will be rejected and returned by the Build functionality. Like the OSet, our OHTable will only be secure if stashed items are queried with probability equal to items stored in the set.

Like our OSet protocol, our OHTable protocol has a limit on the number of times Query is executed. It has an additional Extract function which must be called so the OHTable can be rebuilt when this limit has been reached.

Our protocol is subject to similar conditions as that of our OSet protocol, but with some modifications. While OSet did not allow repeated queries, OHTable does not allow repeated queries of real items, but does allow repeated queries of the null-value $\perp$, which is used for the no-op queries. Like in the OSet protocol we need to limit to $n$ queries. We also need independence from the stash, both for values revealed by the ABB by the environment and for queries to the OHTable. However in this case, the stash consists of an array of both indices and values. In addition, we have a condition that the Extract function will only be called after the queries have been depleted. We formally state our conditions below:

**Condition 5 (No Repeats of Real Items).** *For all $x \in [N]$, Query($[\![x]\!], res$) is called at most once. (Query($[\![\perp]\!], res$) may be called many times.)*

**Condition 6 (Limited Queries).** *Query is called $n - s$ times.*

**Condition 7 (ABB-Stash Independence).** *Let $(stashX_1, stashY_1)$, $(stashX_2, stashY_2)$ be two different possible values of $(X^{stash}, Y^{stash})$. The distribution of all outputs of the ABB by the environment when $(X^{stash}, Y^{stash}) = (X_1^{stash}, Y_1^{stash})$ must be computationally indistinguishable from the distribution when $(X^{stash}, Y^{stash}) = (X_2^{stash}, Y_2^{stash})$.*

**Condition 8 (Query-Stash Independence).** *Let $(X^{stash}, Y^{stash})$ be the output of Build. If $x = X_i$ for any $i \in [n]$, the probability that Query($x, res$) is called at any time, conditioned on any values revealed by the ABB either before or after, is computationally indistinguishable from independent of whether $x \in X^{stash}$.*

**Condition 9 (Extract at End).** *The function Extract will only be called at most once, and only after $n - s$ calls to Query.*

We present our OHTable protocol ($\Pi_{\text{OHTable}}$) and functionality ($\mathcal{F}_{\text{OHTable}}$) in Fig. 8. We now prove its security. Firstly, we need to demonstrate that if $\Pi_{\text{OHTable}}$ is accessed consistently with its conditions, it will also access $\mathcal{F}_{\text{OSet}}$ in a manner that is consistent with its conditions. Formally:

**Proposition 6.** *Assuming an environment that follows Conditions 5, 6, 7, 8 and 9 when accessing $\Pi_{OHTable}$, Conditions 1, 2, 3 and 4 will also be satisfied in calls to $\mathcal{F}_{OSet}$.*

The proof is in Appendix G.3

**Proposition 7.** *Assuming an environment that follows Conditions 5, 6, 7, 8 and 9 $\Pi_{OHTable}$ is a secure implementation of $\mathcal{F}_{OHTable}$ with abort in the $\mathcal{F}_{ABB}, \mathcal{F}_{OSet}$-hybrid model in the 3-party setting against one static malicious adversary, where $\mathcal{F}_{OSet}$ is subject to Conditions 1, 2, 3 and 4.*

The proof is in Appendix G.4. Finally, in Appendix H.3 we show that:

**Proposition 8.** *$\Pi_{OHTable}$.Build has complexity $O(n(\kappa + D))$ and $\Pi_{OHTable}$. Query has complexity $O(\kappa)$ and $\Pi_{OHTable}$.Extract has complexity $O(nD)$.*

## 9 Maliciously-Secure Oblivious Map Construction

As noted above, Oblivious Hash Tables (Sect. 8) have multiple limitations (formalized by Conditions 5–9). In particular, it does not allow real items to be queried multiple times and has very particular restrictions about how the stash is used by the environment. In this section, we present an Oblivious Map (OMap) construction that removes these limitations.

We will use the hierarchical solution, but with a twist. We will define our OMap recursively[2]. An OMap will consist of an Oblivious Hash Table and a smaller OMap of roughly half the capacity. This implicitly creates a hierarchy of OHTables, with the levels corresponding to levels of the recursion. Viewing the hierarchical solution in terms of recursion will make it much simpler to present our protocols and prove them secure. We will evidently need a base case: we use $\Pi_{\text{QuietCache}}$ for this as $\Pi_{\text{QuietCache}}$ already implements OMap (although it is only efficient for smaller table sizes). Our OMap will have a limitation that it can only be queried a certain number of times. Our final ORAM will be able to remove this limitation by taking advantage of the fact that its capacity is equal to the size of the index space. Our condition on the order that OMap should be accessed is formally stated below.

**Condition 10 (OMap Call Pattern).** *First Init$(\llbracket X \rrbracket, \llbracket Y \rrbracket, n)$ is called, where len$(\llbracket X \rrbracket)$ = len$(\llbracket Y \rrbracket) = w \leq \log(N) < \frac{\kappa}{4}$.*
*Then there are at most $n - w$ calls to Query$(\llbracket x \rrbracket)$ each followed immediately by a call to Add$(\llbracket x \rrbracket, \llbracket y \rrbracket)$ (for the same $x$ and some value of $y$ other than $\perp$).*
*Finally, there is a call to Extract.*

In more detail, an OMap of capacity $n$ will contain two data objects: an OHTable with capacity roughly $\frac{n}{2}$ and a smaller sub-OMap of capacity roughly $\frac{n}{2}$. We first store items in the sub-OMap, until it becomes full. When this happens, we extract the contents of the sub-OMap and build an OHTable from its contents. We then initialize a new sub-OMap, in which we store new items. To avoid querying an item to the OHTable more than once, we first query the sub-OMap. If the item has already been queried, it will have been re-added (see Condition 10) and therefore placed in the sub-OMap. If it is found in the

[2] Recall that we need to recurse on OMaps rather than ORAMs, since the smaller levels in the hierarchy need to be able to hold indices from the full space.

**Protocol $\Pi_{\mathbf{OMap}}$**

**$\Pi_{\mathbf{OMap}}$.Init($\llbracket X \rrbracket$, $\llbracket Y \rrbracket$, $w$, $n$, $N$):**

1. Initialize counter for the number of stored items (including overwrites): $t = w$
2. Create a sub-OMap. Store the initial values in this sub-OMap. (Alibi bits are not needed here, as there is no OHTable yet.)
   (a) $s = \log(N)$
   (b) $\mathcal{F}_{\text{OMap}}.\text{Init}(\llbracket X \rrbracket, \llbracket Y \rrbracket, w, \frac{n}{2} + \frac{s}{2}, N)$

**$\Pi_{\mathbf{OMap}}$.Query($\llbracket x \rrbracket$):**

1. If $t < \frac{n}{2} + \frac{s}{2}$: This means the OHTable has not yet been built. Only query the sub-OMap and pass on the value:
   (a) Return $\mathcal{F}_{\text{OMap}}.\text{Query}(\llbracket x \rrbracket)$
2. Else: The OHTable has been built. First query the sub-OMap. If the item is not found, search for it in the OHTable and return the result. If the item was found but has an Alibi bit of 1 it was stashed from the OHTable, so must also be searched for in the OHTable. Otherwise, perform a no-op query:
   (a) $\llbracket y \rrbracket = \mathcal{F}_{\text{OMap}}.\text{Query}(\llbracket x \rrbracket)$
   (b) $\llbracket b_{Alibi} \rrbracket = \llbracket y[-1] \rrbracket$
   (c) $\llbracket b_{found} \rrbracket = \mathcal{F}_{\text{ABB}}.\text{Equal}(\llbracket y \rrbracket, \llbracket \perp \rrbracket)$
   (d) $\llbracket x_{query} \rrbracket = \mathcal{F}_{\text{ABB}}.\text{IfThenElse}(\llbracket b_{Alibi} \rrbracket \cup (\neg \llbracket b_{found} \rrbracket), \llbracket 0 \rrbracket || \llbracket x \rrbracket, \llbracket \perp \rrbracket)$
   (e) $\llbracket y_{table} \rrbracket = \mathcal{F}_{\text{OHTable}}.\text{Query}(\llbracket x_{query} \rrbracket)$
   (f) $\llbracket y_{map} \rrbracket = \llbracket y[1:-1] \rrbracket$ (Drop the Alibi bit for this level.)
   (g) $\llbracket y_{ret} \rrbracket = \mathcal{F}_{\text{ABB}}.\text{IfThenElse}(\llbracket b_{found} \rrbracket, \llbracket y_{map} \rrbracket, \llbracket y_{table} \rrbracket)$
   (h) Return $\llbracket y_{ret} \rrbracket$

**$\Pi_{\mathbf{OMap}}$.Add($\llbracket x \rrbracket, \llbracket y \rrbracket$):**

1. If $t \geq \frac{n}{2} + \frac{s}{2}$ (the OHTable has been built, so the Alibi bit must be appended to show this is not a stashed item):
   (a) $\llbracket y \rrbracket = \llbracket y \rrbracket || \llbracket 0 \rrbracket$
2. $\mathcal{F}_{\text{OMap}}.\text{Add}(\llbracket x \rrbracket, \llbracket y \rrbracket)$
3. $t = t + 1$
4. If $t = \frac{n}{2} + \frac{s}{2}$: The sub-OMap is full. It must be extracted and built into the OHTable. The sub-OMap may contain empty items due to overwrites, these are assigned an index from a disjoint space so they can be inputs to the build:
   (a) $(\llbracket X \rrbracket, \llbracket Y \rrbracket) = \mathcal{F}_{\text{OMap}}.\text{Extract}()$
   (b) For $i \in [\frac{n}{2} + \frac{s}{2}]$:
      i. $\llbracket \hat{X}_i \rrbracket = \mathcal{F}_{\text{ABB}}.\text{IfThenElse}(\llbracket X_i \rrbracket \overset{?}{=} \llbracket \perp \rrbracket, \llbracket 1 \rrbracket || \llbracket i \rrbracket, \llbracket 0 \rrbracket || \llbracket X_i \rrbracket)$
   (c) $(\llbracket stashX \rrbracket, \llbracket stashY \rrbracket) = \mathcal{F}_{\text{OHTable}}.\text{Build}(\llbracket \hat{X} \rrbracket, \llbracket Y \rrbracket, \frac{n}{2} + \frac{s}{2}, 2N)$
   (d) Set the Alibi bits to 1 for stashed items:
      i. For $i \in [s]$ $\llbracket stashY_i \rrbracket = \llbracket stashY_i \rrbracket || \llbracket 1 \rrbracket$
   (e) If an item was empty before it was put in the table, make it empty again: For $i \in [s]$
      i. $\llbracket stashX_i \rrbracket = \mathcal{F}_{\text{ABB}}.\text{IfThenElse}(\llbracket stashX_i[1] \rrbracket, \llbracket \perp \rrbracket, \llbracket stashX_i[2:] \rrbracket$
   (f) $\mathcal{F}_{\text{OMap}}.\text{Init}(\llbracket stashX \rrbracket, \llbracket stashY \rrbracket, s, \frac{n}{2} + \frac{s}{2}, N)$

**$\Pi_{\mathbf{OMap}}$.Extract():**

1. Extract contents from the OMap and the OHTable. Combine and shuffle them, then return the result (which may include empty items):
   (a) $\llbracket X^{map} \rrbracket, \llbracket Y^{map} \rrbracket = \mathcal{F}_{\text{OMap}}.\text{Extract}()$
   (b) $\llbracket X^{table} \rrbracket, \llbracket Y^{table} \rrbracket = \mathcal{F}_{\text{OHTable}}.\text{Extract}()$
   (c) If an item was empty before it was put in the table, make it empty again. For $i \in [\frac{n}{2} - \frac{s}{2}]$:
      i. $\llbracket X_i^{table} \rrbracket = \mathcal{F}_{\text{ABB}}.\text{IfThenElse}(\llbracket X_i^{table} \rrbracket[1], \llbracket \perp \rrbracket, \llbracket X_i^{table}[2:] \rrbracket$
   (d) $\llbracket X \rrbracket = \llbracket X^{map} \rrbracket || \llbracket X^{table} \rrbracket$, $\llbracket Y \rrbracket = \llbracket Y^{map} \rrbracket || \llbracket Y^{table} \rrbracket$
   (e) $\llbracket \hat{X} \rrbracket, \llbracket \hat{Y} \rrbracket = \mathcal{F}_{\text{ABB}}.\text{ObliviousShuffle}(\llbracket X \rrbracket, \llbracket Y \rrbracket)$
   (f) Return $\llbracket \hat{X} \rrbracket, \llbracket \hat{Y} \rrbracket$

**Fig. 9.** Recursive OMap protocol

sub-OMap we therefore do a no-op query to the OHTable. Extract will be called exactly when the sub-OMap becomes full again, and the contents of both the OHTable and sub-OMap will be extracted and returned.

Things are complicated slightly by the fact that because of the "cache-the-stash" technique, our OHTable for storing $n$ elements, actually stores only $n - s$ elements, and returns a stash of $s$ items which is intended to be "cached." To handle this, we increase the capacity of the both the sub-OMap and the OHTable by $\frac{s}{2}$, thus both have a size of $\frac{n}{2} + \frac{s}{2}$. Note that since the OHTable caches $s$ items, it will only hold $\frac{n}{2} - \frac{s}{2}$ real items. This means that each recursive call to the OMap causes the size to be reduced by slightly less than half; nevertheless as $s$ is very small relative to $n$ ($s = \Theta(\log(N)) = o(\kappa)$ and $\kappa$ is the size of the base level), the total recursive depth will still be $\Theta(\log(N))$. Additionally, since stashed items need to be queried in the OHTable with probability equal to stored items, the OMap will tag stashed items with an Alibi bit (c.f. Appendix B) before placing them in the sub-OMap. This will slightly increase the size of payloads at smaller levels of the recursion, but will not affect asymptotic performance.

Our protocol $\Pi_{\text{OMap}}$, as well as the functionality $\mathcal{F}_{\text{OMap}}$ that it implements, are presented in detail in Figs. 9 and 3 respectively. We next prove the security of $\Pi_{\text{OMap}}$ with respect to $\mathcal{F}_{\text{OMap}}$. Note that since $\Pi_{\text{OMap}}$ reveals no values from the ABB, this security proof is not particular to our 3-party honest-majority setting. Rather, it applies in any setting given a $\mathcal{F}_{\text{ABB}}, \mathcal{F}_{\text{OHTable}}, \mathcal{F}_{\text{OMap}}$-hybrid setting, where $\mathcal{F}_{\text{OHTable}}$ is subject to at most Conditions 5, 6, 7, 8 and 9, and $\mathcal{F}_{\text{OMap}}$ is of a smaller capacity and subject to at most Condition 10.

Since $\Pi_{\text{OMap}}$ does not reveal any values from the ABB, to prove its security we need only prove two things (see Corollary 1): that the outputs (to the ABB) are correct and that the conditions on the functionalities it uses are upheld. We prove these below.

**Proposition 9.** *Assuming an environment that follows Condition 10 and that $n \geq \kappa = \omega(\log(N))$, $\Pi_{OMap}[n, N]$ is a secure implementation of $\mathcal{F}_{OMap}[n, N]$ in the $\mathcal{F}_{ABB}, \mathcal{F}_{OHTable}, \mathcal{F}_{OMap}[\frac{n}{2} + \frac{\log(N)}{2}, N]$-hybrid setting, where $\mathcal{F}_{OHTable}$ is subject to Conditions 5, 6, 7, 8 and 9, and $\mathcal{F}_{OMap}$ occurs as a single instance of $\mathcal{F}_{OMap}[\frac{n}{2} + \frac{\log(N)}{2}, N]$ and is subject to Condition 10.*

The proof is in Appendix G.5.

**Proposition 10.** *If $\Pi_{OMap}$ is implemented with its functionalities instantiated in the following ways:*

- *$\mathcal{F}_{OMap}$ implemented recursively with $\Pi_{OMap}$ for all $n \geq \kappa$ and with $\Pi_{QuietCache}$ once $n < \kappa$.*
- *$\mathcal{F}_{OHTable}$ implemented using $\Pi_{OHTable}$, which in turn implements $\mathcal{F}_{OSet}$ using $\Pi_{OSet}$*
- *$\mathcal{F}_{ABB}$ is implemented as described in Sect. 5*

*the resulting protocol will have the following costs:*

- *Init* : $\Theta(\kappa w)$
- *Query:* $\Theta(\log(N)(\kappa + D)$
- *Add and Extract (combined, amortized over n accesses):* $\Theta(\log(N)(\kappa + D))$

The proof is in Appendix H.4

**Acknowledgments.** Supported in part by ONR under grant N00014-15-1-2750, Ripple Labs Inc., DARPA under Cooperative Agreement HR0011-20-2-0025, the Algorand Centers of Excellence programme managed by Algorand Foundation, NSF grants CNS-2001096 and CCF-2220450, US-Israel BSF grant 2015782, Amazon Faculty Award, Cisco Research Award and Sunday Group. Any views, opinions, findings, conclusions or recommendations contained herein are those of the author(s) and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ONR, Ripple Labs Inc., DARPA, the Department of Defense, the Algorand Foundation, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes not withstanding any copyright annotation therein.

## References

[AKL+20] Asharov, G., Komargodski, I., Lin, W.-K., Nayak, K., Peserico, E., Shi, E.: OptORAMa: optimal oblivious RAM. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12106, pp. 403–432. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45724-2_14

[AKLS21] Asharov, G., Komargodski, I., Lin, W.-K., Shi, E.: Oblivious RAM with *Worst-Case* logarithmic overhead. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021. LNCS, vol. 12828, pp. 610–640. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84259-8_21

[AKST14] Apon, D., Katz, J., Shi, E., Thiruvengadam, A.: Verifiable oblivious storage. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 131–148. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54631-0_8

[ARS+15] Albrecht, M.R., Rechberger, C., Schneider, T., Tiessen, T., Zohner, M.: Ciphers for MPC and FHE. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 430–454. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_17

[BBVY21] Banik, S., Barooti, K., Vaudenay, S., Yan, H.: New attacks on LowMC instances with a single plaintext/ciphertext pair. IACR ePrint 2021/1345 (2021)

[BGI15] Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 337–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_12

[BIKO12] Beimel, A., Ishai, Y., Kushilevitz, E., Orlov, I.: Share conversion and private information retrieval. In: 2012 IEEE 27th Conference on Computational Complexity, pp. 258–268. IEEE (2012)

[BKKO20] Bunn, P., Katz, J., Kushilevitz, E., Ostrovsky, R.: Efficient 3-party distributed ORAM. In: Galdi, C., Kolesnikov, V. (eds.) SCN 2020. LNCS, vol. 12238, pp. 215–232. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57990-6_11

[BOGW88] Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: STOC, New York, NY, USA. ACM (1988)

[Can01] Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: FOCS, pp. 136–145. IEEE (2001)

[CCD88] Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols. In: STOC (1988)

[CDG+17] Chase, M., et al.: Post-quantum zero-knowledge and signatures from symmetric-key primitives. IACR ePrint 2017/279 (2017)

[CDI05] Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In: Kilian, J. (ed.) TCC 2005. LNCS, vol. 3378, pp. 342–362. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30576-7_19

[CFP13] Cramer, R., Fehr, S., Padró, C.: Algebraic manipulation detection codes. Sci. China Math. **56**, 1349–1358 (2013)

[CGH+18] Chida, K., et al.: Fast large-scale honest-majority MPC for malicious adversaries. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10993, pp. 34–64. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96878-0_2

[CHL22] Casacuberta, S., Hesse, J., Lehmann, A.: SoK: oblivious pseudorandom functions. In: EuroS&P, pp. 625–646. IEEE (2022)

[DFK+06] Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 285–304. Springer, Heidelberg (2006). https://doi.org/10.1007/11681878_15

[DK12] Drmota, M., Kutzelnigg, R.: A precise analysis of Cuckoo hashing. ACM Trans. Algorithms (TALG) **8**(2), 1–36 (2012)

[DLMW15] Dinur, I., Liu, Y., Meier, W., Wang, Q.: Optimized interpolation attacks on LowMC. IACR ePrint 2015/418 (2015)

[DN03] Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 247–264. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_15

[DO20] Dittmer, S., Ostrovsky, R.: Oblivious tight compaction in $O(n)$ time with smaller constant. In: Galdi, C., Kolesnikov, V. (eds.) SCN 2020. LNCS, vol. 12238, pp. 253–274. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57990-6_13

[DPSZ12] Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_38

[Ds17] Doerner, J., Shelat, A.: Scaling ORAM for secure computation. In: CCS (2017)

[DvDF+16] Devadas, S., van Dijk, M., Fletcher, C.W., Ren, L., Shi, E., Wichs, D.: Onion ORAM: a constant bandwidth blowup oblivious RAM. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9563, pp. 145–174. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49099-0_6

[FJKW15] Faber, S., Jarecki, S., Kentros, S., Wei, B.: Three-party ORAM for secure computation. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 360–385. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48797-6_16

[FLNW17] Furukawa, J., Lindell, Y., Nof, A., Weinstein, O.: High-throughput secure three-party computation for malicious adversaries and an honest majority. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10211, pp. 225–255. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56614-6_8

[FNO21] Hemenway Falk, B., Noble, D., Ostrovsky, R.: Alibi: a flaw in Cuckoo-hashing based hierarchical ORAM schemes and a solution. In: Canteaut, A., Standaert, F.-X. (eds.) EUROCRYPT 2021. LNCS, vol. 12698, pp. 338–369. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77883-5_12

[FNO22] Falk, B.H., Noble, D., Ostrovsky, R.: 3-party distributed ORAM from oblivious set membership. In: Galdi, C., Jarecki, S. (eds.) Security and Cryptography for Networks. SCN 2022. LNCS, vol. 13409, pp. 437–461. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-14791-3_19

[FNR+15] Fletcher, C.W., Naveed, M., Ren, L., Shi, E., Stefanov, E.: Bucket ORAM: single online roundtrip, constant bandwidth oblivious RAM. IACR ePrint 2015/1065 (2015)

[GHL+14] Gentry, C., Halevi, S., Lu, S., Ostrovsky, R., Raykova, M., Wichs, D.: Garbled RAM revisited. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 405–422. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_23

[GI14] Gilboa, N., Ishai, Y.: Distributed point functions and their applications. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 640–658. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_35

[GKK+12] Gordon, S.D., et al.: Secure two-party computation in sublinear (amortized) time. In: CCS (2012)

[GKW18] Gordon, S.D., Katz, J., Wang, X.: Simple and efficient two-server ORAM. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018. LNCS, vol. 11274, pp. 141–157. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03332-3_6

[GMOT12] Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Privacy-preserving group data access via stateless oblivious RAM simulation. In: SODA (2012)

[GMW87] Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: STOC (1987)

[GO96] Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. JACM **43**(3), 431–473 (1996)

[Gol87] Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: STOC 1987, pp. 182–194. ACM (1987)

[HV20] Hamlin, A., Varia, M.: Two-server distributed ORAM with sublinear computation and constant rounds. IACR ePrint 2020/1547 (2020)

[IKH+23] Ichikawa, A., Komargodski, I., Hamada, K., Kikuchi, R., Ikarashi, D.: 3-party secure computation for RAMs: optimal and concretely efficient. IACR ePrint 2023/516 (2023)

[IKK+11] Ishai, Y., Katz, J., Kushilevitz, E., Lindell, Y., Petrank, E.: On achieving the "best of both worlds" in secure multiparty computation. SIAM J. Comput. **40**(1), 122–141 (2011)

[JW18] Jarecki, S., Wei, B.: 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In: Preneel, B., Vercauteren, F. (eds.) ACNS 2018. LNCS, vol. 10892, pp. 360–378. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93387-0_19

[JZLR22] Ji, K., Zhang, B., Lu, T., Ren, K.: Multi-party private function evaluation for RAM. IACR ePrint 2022/939 (2022)

[KLO12] Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in) security of hash-based oblivious RAM and a new balancing scheme. In: SODA (2012)

[KM19] Kushilevitz, E., Mour, T.: Sub-logarithmic distributed oblivious RAM with small block size. In: Lin, D., Sako, K. (eds.) PKC 2019. LNCS, vol. 11442, pp. 3–33. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17253-4_1

[KMW09] Kirsch, A., Mitzenmacher, M., Wieder, U.: More robust hashing: Cuckoo hashing with a stash. SIAM J. Comput. **39**, 1543–1561 (2009)

[KO97] Kushilevitz, E., Ostrovsky, R.: Replication is NOT needed: SINGLE database, computationally-private information retrieval. In: FOCS (1997)

[KS14] Keller, M., Scholl, P.: Efficient, oblivious data structures for MPC. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8874, pp. 506–525. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45608-8_27

[Lau15] Laud, P.: Parallel oblivious array access for secure multiparty computation and privacy-preserving minimum spanning trees. In: PoPETs (2015)

[LIM20] Liu, F., Isobe, T., Meier, W.: Cryptanalysis of full LowMC and LowMC-M with algebraic techniques. IACR ePrint 2020/1034 (2020)

[LN17] Lindell, Y., Nof, A.: A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In: CCS, pp. 259–276 (2017)

[LN18] Larsen, K.G., Nielsen, J.B.: Yes, there is an oblivious RAM lower bound! In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10992, pp. 523–542. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96881-0_18

[LO13] Lu, S., Ostrovsky, R.: Distributed oblivious RAM for secure two-party computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 377–396. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36594-2_22

[LWZ11] Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In: Lai, X., Zhou, J., Li, H. (eds.) ISC 2011. LNCS, vol. 7001, pp. 262–277. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24861-0_18

[Mit09] Mitzenmacher, M.: Some open questions related to Cuckoo hashing. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 1–10. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04128-0_1

[MV23] Mathialagan, S., Vafa, N.: MacORAMa: optimal oblivious RAM with integrity. IACR ePrint 2023/083 (2023)

[MZ14] Mitchell, J.C., Zimmerman, J.: Data-oblivious data structures. In: STACS. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2014)

[NIS21] NIST. Post-quantum cryptography PQC: Round 3 submissions (2021). https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions

[Nob21] Noble, D.: Explicit, closed-form, general bounds for cuckoo hashing with a stash. IACR ePrint 2021/447 (2021)

[OS97] Ostrovsky, R., Shoup, V.: Private information storage. In: STOC, vol. 97 (1997)

[Ost90] Ostrovsky, R.: Efficient computation on oblivious RAMs. In: STOC (1990)

[Ost92] Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Ph.D. thesis, Massachusetts Institute of Technology (1992)

[PPRY18] Patel, S., Persiano, G., Raykova, M., Yeo, K.: PanORAMa: oblivious RAM with logarithmic overhead. In: FOCS (2018)

[PR01] Pagh, R., Rodler, F.F.: Cuckoo hashing. In: ESA (2001)

[PR10] Pinkas, B., Reinman, T.: Oblivious RAM revisited. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 502–519. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_27

[PSSZ15] Pinkas, B., Schneider, T., Segev, G., Zohner, M.: Phasing: private set intersection using permutation-based hashing. In: USENIX, pp. 515–530 (2015)

[RFK+14] Ren, L., et al.: Ring ORAM: closing the gap between small and large client storage oblivious RAM. IACR ePrint 2014/997 (2014)

[SVDS+13] Stefanov, E., et al.: Path ORAM: an extremely simple oblivious RAM protocol. In: CCS (2013)

[Tof07] Toft, T.: Primitives and applications for multi-party computation. Unpublished doctoral dissertation, University of Aarhus, Denmark (2007)

[VHG22] Vadapalli, A., Henry, R., Goldberg, I.: Duoram: a bandwidth-efficient distributed ORAM for 2- and 3-party computation. IACR ePrint 2022/1747 (2022)

[Vol99] Vollmer, H.: Introduction to Circuit Complexity: A Uniform Approach. Springer, Cham (1999). https://doi.org/10.1007/978-3-662-03927-4

[WCS15] Wang, X., Chan, H., Shi, E.: Circuit ORAM: on tightness of the Goldreich-Ostrovsky lower bound. In: CCS (2015)

[WHC+14] Wang, X.S., Huang, Y., Chan, T.-H.H., Shelat, A., Shi, E.: SCORAM: oblivious RAM for secure computation. In: CCS (2014)

[Yao82] Yao, A.: Protocols for secure computations (extended abstract). In: FOCS (1982)

[Yao86] Yao, A.: How to generate and exchange secrets. In: FOCS (1986)

[ZWR+16] Zahur, S., et al.: Revisiting square-root ORAM: efficient random access in multi-party computation. In: S & P (2016)