DORAM revisited: Maliciously secure RAM-MPC with logarithmic overhead

Brett Falk[1], Daniel Noble[1], Rafail Ostrovsky[2], Matan Shtepel[2], and Jacob Zhang[2,3]

[1]University of Pennsylvania
[2]University of California, Los Angeles
[3]Jane Street Capital

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 and computation 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 and computation per query, for a database of size N with D-bit values, where κ is the security parameter. Our hidden constants in a 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 and computation complexity of the best semi-honest DORAM protocol, 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 construction.

Poly-logarithmic overhead Oblivious RAM (ORAM) [Ost90, Ost92, GO96] was developed to allow a client to access a database held by an untrusted server, 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 (Figure 1b), OptORAMa achieved amortized logarithmic overhead in the single-server setting [AKL$^+$20] (Figure 1a) with constant in front big-O notation $> 2^{228}$. The constant was later reduced to 9405 in [DO20]) and de-armotized in [AKLS21]. However, despite all these improvements, these works are of only theoretical interest, due to large constants. In Section 3.4 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 κ, we prove the following theorem:

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

The server-to-server communication overhead of $O((\kappa + D) \log(N))$ matches the best communication and computation complexity of a DORAM protocol in the semi-honest model [FNO22, LO13], thus we achieve security against malicious adversaries with *no* asymptotic increase in communication costs.

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.

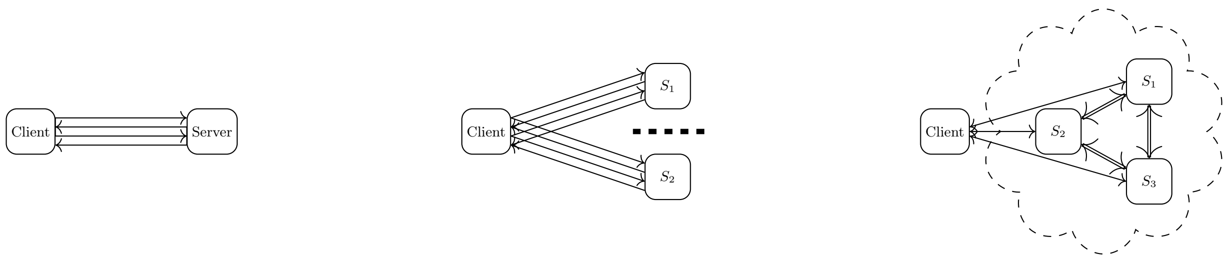
1.1 MPC in the RAM model

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 they cannot be computed efficiently under MPC. 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 ith element in a list or the function $W(i, Y, y_1, \ldots, y_N)$ which

[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.



(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 amongst themself for several rounds, and respond to the client. The client's work is the lowest in the DORAM setting.*

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, [SCSL11, GGH$^+$13, WCS15, SVDS$^+$13] developed *circuit ORAM*, an ORAM maintaining the stringent one-trusted-client one-untrusted-server security model of traditional ORAM while decreasing the circuit-complexity of the client. Another example of such efforts, are *multi-server ORAM* protocols where the trusted client's data is shared and accessed across multiple servers. Assuming some fraction of the servers are honest [OS97, GKK$^+$12, GKW18, KM19] these works shift some of the communication burden to servers. These multi-server ORAMs can also be adapted to the MPC context by simulating the client using (circuit-based) MPC, allowing the MPC participants to play the role of the additional ORAM servers. Some of these protocols have been implemented [GKK$^+$12, LO13, ZWR$^+$16, WHC$^+$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 at the cost of *asymptotic cost of the best known semi-honest construction.*

2 Notation, Conventions, and Definitions

Notation. We denote the 3 parties P_1, P_2, P_3 and $\mathbb{F}_{2^l}$ the finite field of 2^l elements. For the bulk of our symbolic definitions, see Table 1.

Symbol	Definition
N	The (maximum) number of elements storable by the ORAM data structure
D	The size (in bits) of each element
κ	The security parameter $\kappa = \omega(\log N)$
$[\![x]\!]$	A secret sharing of the value x
$[x]^{(i,j)}$	A two-party (additive) sharing of x between participants i and j
$[x]^{(i,j,k)}$	A three-party (additive) sharing of x between participants i, j and k.
$[\![x]\!]_{P_i}$	P_i's share of x.
$x^{(i)}$	An individual component of an (additive) share.
P_i	Participant i in the DORAM protocol
$x \overset{?}{=} y$	Boolean expression evaluating to 1 if x equals y, else 0
$(b?x : y)$	Expression which evaluates to x if b, else evaluates to y
$x[i : j]$	For $x = x_0 \dots x_{n-1} = x \in \mathbb{F}^n$ we let $x[i : j] = x_i \dots x_j$
C	An arithmetic or boolean circuit
$x \in_R S$	$x \overset{U}{\longleftarrow} (S)$ where $U(S)$ is the uniform distribution on S.
$\perp$	$\perp = 0$ reserved null value
$\mathfrak{e}$	Refers to the the "Alibi" bits (described in Section 5.)

Table 1: Notation

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

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

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

DORAM. We use N to denote the total number of elements in the DORAM. Each element stored in the DORAM is a pair (X, Y), where X is the "virtual address" of the D-bit payload, Y. Thus a query will be a secret-shared index, X, and the response will be a secret-sharing of the corresponding payload, Y. We assume that only X's in the range $[N]$ are queried. We let κ be our computational security parameter. Since we want to achieve failures with probability negligible in the input size, N, we must have $\kappa = \omega(\log N)$. We use σ denote our statistical security parameter.
DOMap. A DOMap is a DORAM protocol where the index set, X, need not be $1, \dots, N$, but can instead be arbitrary bit strings.

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

Definition 1 (DORAM). *A protocol, Π, is said to be a UC maliciously-secure (n, t)-Distributed ORAM protocol if for all $N, D, \kappa \in \mathbb{Z}^+$, Π UC-realizes the DORAM functionality (Figure 2).*

Secret Sharing. Our protocol makes heavy use of (3,1) *replicated secret sharing* (also known as CNF sharing [CDI05]).

Definition 2 (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 use $[\![x]\!]$ to denote a replicated secret sharing of the value x held jointly by by P_0, P_1, P_2. Explicitly, this means that for P_i, $[\![x]\!] = \{x^{(j)}\}_{j \neq i}$. Let $[z]^{(i,j)}$ denote an additive sharing of z held by parties P_i and P_j. Concretely, $[z]^{(i,j)}$ denotes a two-party XOR sharing of the value z between participants i and j, so participant i holds a share $z^{(i)}$ where $z^{(i)} + z^{(j)} = z$. Let $[z]^{(i,j,k)}$ denote an XOR secret sharing between parties i, j, k. For a list $X \in \mathbb{F}^n$, we let $X_i^{(j)}$ denote the share of the value X_i held by player j.

3 Prior Work

3.1 Client-Server ORAM

ORAM protocols was originally introduced to allow a single client with $O(1)$ memory slots each of size $O(D)$ for $D > \log N$ to store an array of $O(N)$ elements on a single, untrusted server [Gol87, Ost90, Ost92, GO96]. Follow-up works aimed at reducing the communication complexity between the client and the server. Single server ORAM must incur at least logarithmic overhead [GO96, LN18]. A long line of research has culminated in ORAM protocols that achieve these bounds. In particular, [AKL$^+$20] achieves *amortized* overhead of $O(\log N)$, but the big-O notation hides at least 2^{228} constnat, which was later reduced to 9348 [DO20]. The work of [AKL$^+$20] was later de-amortized to achieve (logarithmic) overhead in the worst-case [AKLS21], following the de-amortization technique of [OS97]. Although these protocols are asymptotically optimal, the hidden constants are above 9,000 making them unsuitable from practical applications [DO20].

In the original ORAM model, the single server was considered to be *passive*, meaning it could read and write to locations specified by the client, but could not perform computation of its own. In *Active ORAM* protocols, the server is allowed to perform computation of its own, and the ORAM lower bounds can be circumvented (for sufficiently large blocks) and it is possible to achieve *constant* query overhead using single-server PIR machinery [KO97, OS97, AKST14, DvDF$^+$16, FNR$^+$15, RFK$^+$14]. Indeed, if one only considered asymptotic communication complexity, the client could encrypt its query using single-server PIR [KO97], and the active server could run a linear-sized selection circuit to retrieve a single, encrypted response. This clearly has low *communication complexity* but unacceptable *computation complexity* for the server.

3.2 Multi-server ORAM

A relaxation of the ORAM model is the notion of *multi-server* ORAMs [LO13, GKW18, KM19]. Inspired by multi-server PIR protocol [OS97], in a multi-server ORAM protocol, there is a single client, and multiple (non-colluding) servers. As shown by [LO13], adding a second (non-colluding) server can increase efficiency – [LO13] achieved logarithmic overhead (in the two server setting) years before a protocol with comparable asymptotic efficiency was found in the single-server setting [AKL$^+$20]. The two-server scheme still maintains much better *concrete* efficiency than the best known single-server ORAM (despite having similar asymptotic costs).

Having multiple servers also allows other techniques where servers don't just store client's data but compute on these data. For example as Function Secret-Sharing (FSS) [GI14, BGI15] can be used to support PIR techniques in order to reduce the client-server *communication* at the cost of increasing server-side *computation.* Several (D)ORAM protocols have been built using FSS [Ds17, KM19, VHG22], but these all require $O(N)$ server side computation, limiting their scalability.

3.3 DORAM

Unlike the original ORAM model, and the variants discussed above, in the *Distributed Oblivious RAM* (DORAM) model there is no client at all[2]. Instead, all information is secret-shared between the DORAM

[2] the client may be conceptualized secret sharing the query between the servers, but she does not participate in the retrieval protocol

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

Table 2:
Complexity of DORAM protocols. N denotes the number of records, κ is a cryptographic security parameter, σ is a statistical security parameter, and D is the record size. Note that communication always lower bounds computation, as comunnicating is considered a "computational step."

servers. Thus a DORAM protocol allows a group of servers to access a secret-shared array at a secret-shared index.

This leaves the question of how to build DORAM. As oberved in [OS97], notice that any ORAM protocol that requires a trusted client can be compiled into a DORAM protocol by executing the client's code within a generic MPC framework. The problem with this approach is that the client may not be "MPC-friendly," meaning that simulating the client under MPC would result in efficiency loss. One possible avenue towards creating efficient DORAM is to build ORAM protocols whose clients have the low circuit complexity and compile them into DORAM protocols using generic MPC. Circuit ORAM [WCS15] and Path ORAM [SVDS+13] took this approach. As shown in Table 2, compiling Circuit ORAM into a DORAM yields a DORAM protocol with $O\left(\log^3 N + D \log N\right)$ bits of communication per query.

Instead of compiling a client-server ORAM into a DORAM, an alternative approach is to build a DORAM directly. To date, several DORAM protocols have been developed [KS14, WHC+14, FJKW15, Ds17, JW18, HV20, BKKO20, FNO22, VHG22, JZLR22]. Among these DORAM protocols, [LO13, FNO22] is the most efficient, with logarithmic overhead in both communication and computation. Nevertheless, asymptotic complexity is not everything and [KS14, FJKW15, Ds17, JW18, WHC+14, VHG22] have all been implemented, and are reasonably efficient in practice. We stress that all of these protocols are only secure against semi-honest adversaries.

3.4 Limitations of the generic ORAM to DORAM transformation

The majority of ORAM and DORAM constructions have focused on the semi-honest model, where the server (or servers) observe the client's queries but do not tamper with any of the data stored by the client. In the passive model, where the ORAM server is simply a memory array where the client can store and retrieve data, [GO96] developed methods to authenticate data and compile an ORAM protocol designed for semi-honest servers into one designed for malicious servers. The [GO96] transformation breaks down, however, when the ORAM server is *active*.

One possible technique for building DORAM protocols secure against malicious adversaries, is to execute a single-server ORAM protocol using a malicious secure MPC framework. We can generically transform any semi-honest client-server ORAM, into a malicious-DORAM by simulating *both* the ORAM client and the ORAM server using an MPC protocol that provides security against malicious adversaries. To measure the efficiency of this transformation, we need to measure the efficiency of *both* the ORAM client and the ORAM server.

Remark 1 (Converting OptORAMa to a DORAM). Consider trying to transform OptORAMa (which has an $O(\log N)$ communication overhead in the client-server setting) into a malicious-secure DORAM.

Since the hidden constants are *enormous* [DO20], it will obviously be inefficient in practice, but would it be asymptotically efficient?

As it turns out, OptORAMa loses its asymptotic optimality when compiled into a DORAM. The main reason is that the OptORAMa client is assumed to perform operations on w-bit "words" in $O(1)$ time (Where $w = O(\log N)$). This allows them to perform SIMD operations on N bits in $O(N/w)$ time by bit packing (e.g. [AKL$^+$20][Theorem 4.5]). Simulating the client under MPC, transmitting a w-bit word requires (at least) w-bits of communication. So the packing techniques crucially used in [AKL$^+$20] do not yield the necessary a w-fold speedup under MPC. Another difference in the model is how PRFs are evaluated. In the client-server setting, OptORAMa assumes that a PRF can be computed in constant time. This makes sense in the client-server setting where the PRF can be implemented with a hardware circuit (e.g. AES-NI). When simulating the client under MPC, however, the PRF calls need to be simulated under MPC, and this requires at least an additional $O(\kappa)$ communication.

Thus simulating OptORAMa (or follow-up works like [AKLS21, MV23]) under a malicious MPC protocol would *not* yield a DORAM with $O((\kappa + D) \log N)$ communication complexity.

Remark 2 (Converting [LO13] to a DORAM). The 2-server ORAM protocol of [LO13] achieves logarithmic overhead, and the [LO13] client is reasonably efficient. Essentially, the [LO13] client makes $O(\log N)$ PRF calls for each ORAM query. This means that the [LO13] *client* could be run under MPC with a communication complexity of $O((\kappa + D) \log N)$. The issue is that the [LO13] servers are *active* – they generate permutations and shuffle data. This means that running the [LO13] servers under an MPC would require verifying consistency of permutations will introduce additional overhead.

Remark 3 (Converting [FNO22] to a Maliciously-secure DORAM). The 3-server DORAM of [FNO22] achieves security against one semi-honest device with communication cost $O((\kappa+D) \log(N))$. However it is not easily convertible to a malicious protocol. In particular, [FNO22] heavily depends on Bloom Filters to implement a Distributed Oblivious Set. These Bloom Filters are secret-shared between 2 parties, so have no protection against a party providing incorrect shares.

Thus, for these generic conversions (running an efficient ORAM under malicious-secure MPC) it is not clear how to build a malicious-secure DORAM with $O((\kappa + D) \log N)$ communication per access.

Our protocol builds on the (3,1)-DORAM of [FNO22], which has logarithmic overhead in the semi-honest setting. Our protocol achieves the same computational and communication complexity but is secure against *malicious* adversaries.

4 Technical Overview

In this work, we construct a 3-player 1-malicious distributed ORAM (DORAM) with logarithmic amortized-per-query overhead. To our knowledge, this is the first DORAM protocol shown to be secure against malicious adversaries. Our construction has an amortized-per-query computational and communication complexity of $O((\kappa + D) \log N)$, where κ is our computational security parameter, D is the number of bits in each data block, and N is the total number of elements stored in the DORAM. This is identical to best DORAMs in the semi-honest setting ([LO13, FNO22]). We remark that we can also de-amortize our constructions using standard machinery and without substantialy increasing the overhead.

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 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 becomes 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 $\Theta(\log(N))$ OHTable 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 somewhat more inefficient "ORAM."

Remark 4 (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 Figure 3. Note that most existing ORAM protocols actually instantiate the slightly stronger OMap functionality.

The Hierarchical Solution for DORAMs: *Distributed* ORAMs can also be built using the Hierarchical solution using the same technique. Distributed Oblivious Hash Tables (OHTables) 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 OHTable 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 the OHTable; if it is found, the parties execute a protocol that is indistinguishable from a real, unique query to the OHTable. By recursively implementing the sub-DORAMs using this technique, a DORAM can be constructed using $\Theta(\log(N))$ OHTables.

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, that is 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. In the classical ORAM setting, shuffling n items requires $\Theta(n \log(n) D)$ communication with larrge 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 re-using randomness and using oblivious tight compaction instead of shuffles. This brings the cost down to $\Theta(nD)$ but the constants are still large even when using tight compaction of [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^{th} item for any secret-shared $k \in [n]$ with only $\Theta(n + D)$ communication. (See Section 7 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 iteslf, and secret-share this to the holders. This is then used to build a OHTable.)

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? 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 (section 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 mentioned 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 who 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 (Section 7), 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/computation. 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 of (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 and computation 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 (section 6), and prove it UC-secure against a $(3, 1)$ malicious adversary. This greatly simplifies our later protocol descriptions and proofs.

5 Cuckoo Hashing, Stashes and the Alibi Technique

Cuckoo Hashing is a core technique in our protocol, as well as many other Hierarchical ORAMs and DORAMs [PR10, GMOT12, LO13, KM19, PPRY18, AKL+20, FNO22] . Using Cuckoo Hashing in (D)ORAMs correctly requires understanding some subtle technicalities. In this section we explain Cuckoo Hashing and describe an efficient method for using Cuckoo Hashing in (D)ORAMs that avoids leaking information about the access pattern.

Functionality $\mathcal{F}_{\textbf{OMap}}$

$\mathcal{F}_{\textbf{OMap}}$**.Init(**$[\![X]\!]$**,** $[\![Y]\!]$**,** w**,** n**,** N**):** Store (X_i, Y_i) in a dictionary for all $1 \leq i \leq w$, where $\text{len}(X) = \text{len}(Y) = w$.
$\mathcal{F}_{\textbf{OMap}}$**.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 $[\![res]\!]$.
$\mathcal{F}_{\textbf{OMap}}$**.Add(**$[\![x]\!]$**,** $[\![y]\!]$**):** Store (x, y) in the dictionary (writing over an old value if need be).
$\mathcal{F}_{\textbf{OMap}}$**.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 [PR01] is a form of multiple choice hashing. It has multiple variants, we use the following common, simple variant.[3] Let n be the number of elements we wish to store, where each element has a unique identifier chosen from $\{0,1\}^S$ and a data value chosen from $\{0,1\}^D$. We will have 2 tables, T_0 and T_1, each of capacity c, where c is the smallest power of 2 larger than $(1+\epsilon)n$ for some fixed, and fairly small, constant ϵ. (Empirical evidence shows that $\epsilon = 0.2$ suffices for typical parameter ranges [PSSZ15].) Tables T_0 and T_1 are associated with hash functions h_0 and h_1 respectively, where $h_i : \{0,1\} \to [c]$. An item with indicator $(x, y) \in \{0,1\}^S \times \{0,1\}^D$ will either be stored in location $h_0(x)$ in T_0 or $h_1(x)$ in T_1. A builder, given the n items, can determine a satisfying allocation of items to locations (if one exists) in $\Theta(n)$ operations.

Cuckoo Hashing has the appealing property that each item can only be stored in a small constant number of locations (in our case 2). However, there is a non-negligible probability that no satisfying assignment exists. For a simple example, observe that 3 (out of n) items may be mapped to the same two locations with non-negligible probability. For a cuckoo hash table storing n elements in a table of size $\Theta(n)$, the probability of failure is $\Theta(\frac{1}{n})$ (see [DK12] for the explicit constant.) For many applications, a build failure is inconsequential, as you can simply try again with new hash functions. However, in security-oriented applications, this failure can leak information about the access pattern. This is not because the build failure event by itself leaks information: if the hash functions are secret and appear random the event is independent of the input sets. Rather, by guaranteeing that the table was built correctly, an adversary can learn information during queries. For instance, if 3 distinct queried items access the same 2 locations it is evident that one of the queried items was not stored in the table.

In order to reduce the probability of build failure, it is common to add an additional *stash*, of size s, which can the items that cannot be stored in the main table [KMW09]. This reduces the failure probability to $O(n^{-(s+1)})$ for constant s [KMW09] and $O\left(\left(\frac{O(s)}{n}\right)^s\right)$ [Nob21] for super-constant s. By picking $s = \Theta(\log(N))$, for any $n = \omega(\log(N))$ the failure probability then becomes negligible in N.[4]

The Stashing variant allowed Cuckoo Hashing to be applied to Hierarchical ORAMs [GMOT12, KLO12]. Recall that in Hierarchical ORAM, we implement an OMap using an OHTable of size N and a (recursively implemented) OMap of capacity $N/2$. It was suggested that, rather than having an OHTable that can store all N elements, it was possible to use Cuckoo Hashing with a Stash to implement a pseudo-OHTable that could efficiently store $N-s$ elements (the non-stash elements) and refuse to store s elements (the stash). These stash elements could then be stored in the sub-OMap. This reduced the number of queries to the sub-OMap before it became full by a small constant, but allowed Hierarchical ORAMs to finally take advantage of Cuckoo Hashing.[5]

[3] Some variants have more than 2 hash tables (though usually the number is still constant). Others allow each location to have capacity more than 1 (though again, this is usually constant.). Alternatively, some use multiple hash functions within a single table (with size at least $2(1+\epsilon)n$. For an excellent survey of variants of Cuckoo Hashing and open problems relating to these, see [Mit09].

[4] Setting $s = \Theta(\log(n))$ makes the failure probability negligible in n, but this is not suitable for Hierarchical (D)ORAMs, where table sizes may be polylogarithmic in the (D)ORAM size.

[5] The cache-the-stash solution [KLO12] was originally described in terms of levels rather than recursive calls to a sub-OMap, but the effect is the same.

One catch is that the choice of rejected elements is not independent of the structure of the pseudo-OHTable. In particular, recall that in Hierarchical ORAM, if the item is found in the sub-ORAM, the item will not be queried in the OHTable, but random locations will be accessed instead. Falk et al. [FNO21] observed that this causes the memory locations accessed for queries to stashed items to be *resampled* during a query, whereas non-stash items are accessed in their original hash locations. Furthermore, this causes the access pattern when querying all items in an OMap to be distinguishable from an access pattern to items exclusively not in the OMap. This can be solved by tagging items with an "Alibi" bit for that level of the recursion to indicate that, although the item has been found in the sub-OMap, it must still be queried to the OHTable.

Our DORAM uses Cuckoo Hashing with a Stash and the Alibi technique. This means that our OHTables are not true full dictionary implementations as they refuse to store s of the items given to them. Also we choose to always reinsert a full s items. This effectively means that we always store s items in the stash even if some of these items could have been stored in the main tables.

6 The Arithmetic Black Box (ABB) Model

Initially, MPC was used solely as a method of Secure Function Evaluation (SFE), that is it was used to perform once-off evaluations of stateless functionalities. However, today many use cases require MPC to implement a *reactive* functionality, that is, a functionality can retain a state between executions. In order to facilitate the development of protocols for reactive functionalities, Damgård and Nielsen introduced the Arithmetic Black Box (ABB) model for secure computation [DN03]. In their words, an "ABB can be thought of a secure general-purpose computer." The ABB is a reactive functionality that retains state between executions. Parties can input private values to the ABB, assigning it a public *handle*. The parties may instruct the ABB to perform basic operations on the values it holds. The parties specify the inputs to the operation using their public handles, and also specify a public handle for the result which will also be stored in the ABB. Lastly, the parties may instruct the ABB to output a result, either to a single party, or to all parties. In the case of a malicious adversary, the secret-sharing scheme must protect against an adversary inputting incorrect shares, either by successfully reconstructing secrets even if some shares are corrupted, or by aborting the protocol if corrupted shares are detected. The ABB model greatly simplifies descriptions and proofs of complex MPC-based protocols.

In the ABB-model, the inputs and outputs of functionalities are often only values stored in the ABB. In the UC model, the environment may call the functionality/protocol in arbitrary ways, and may later do arbitrary things with the ABB-stored inputs and outputs, including revealing them. The only security guarantee that the protocols (can) provide is that they are indistinguishable from the functionality (or more specifically, an adversary interacting with the protocol is indistinguishable from a simulator interacting with the functionality). This necessitates that any *local variables* stored in the ABB by the protocol (i.e. not inputs or outputs), cannot be accessed by the environment, as the functionality does not have any such local variables.

Some of our sub-protocols will have conditions on how the environment can call them. For instance, our OHTables only allow each item to be queried at most once.[6] We will explicitly state any such conditions on how the environment can call such sub-protocols and prove that they are indistinguishable from desired functionalities under these conditions. Observe that this means that the sub-protocol is not actually a UC-secure implementation of the functionality as the protocol cannot be called in an arbitrary way by the environment. Nevertheless, when such a sub-protocol is used to implement a functionality in a larger protocol, the larger protocol ensures that its calls to the functionality satisfy the required conditions. This means that the sub-protocol is "shielded" from the environment by the larger protocol: the sub-protocol is a local object[7] in the larger protocol, and so cannot be called in arbitrary ways by the environment. Therefore, the sub-protocol will be indistinguishable from its functionality in this setting. Furthermore, the larger protocol may be accessed by the environment in arbitrary ways and be a UC secure implementation of its functionality.

[6] We actually implement something weaker: a Distributed Oblivious Partial Hash Table, which rejects a few items, and has a further condition that rejected items must be queried with the same probability as un-rejected items.

[7] That is, it is a local variable with associated functions as conceptualized in object-oriented programming.

We present $\mathcal{F}_{\text{ABB}}$, which we UC-realize in the malicious model in Figure 4. We divide our ABB into four functionalities $\mathcal{F}_{\text{ABB1}}$, $\mathcal{F}_{\text{ABB2}}$, $\mathcal{F}_{\text{ABB3}}$ and $\mathcal{F}_{\text{ABB4}}$, where $\mathcal{F}_{ABBi}$ is implemented in the $\mathcal{F}_{ABB(i-1)}$-hybrid model. In future sections we refer to all these functionalities under the single functionality $\mathcal{F}_{\text{ABB}}$.

6.1 UC-realizing $\mathcal{F}_{\textbf{ABB1}}$ in the malicious model: Our base MPCs

We base our ABB on the (3,1) replicated-secret sharing, maliciously UC-secure protocol of [FLNW17].[8] This ensures that they maintain their security when composed generally with other protocols. MPCs of [FLNW17] and [CGH+18]. Using [FLNW17] and [CGH+18] we can preform bit-efficient-operations (e.g. equality tests) and algebraic-efficient-operations (e.g. multiset polynomial check) on secret-shared values, respectively. To transition between "bit secret sharing" and "field secret sharing" *for free*, our protocol uses (3,1) *replicated secret sharing* (sometimes called CNF sharing c.f. [CDI05]) over $\mathbb{F}_2$ and $\mathbb{F}_{2^l}$. As standard in BGW-style MPCs, additions and randomness generation gates are free, but unlike in most protocols input, output, and multiplication gates cost $O(\text{number of input bits} + \text{number of output bits})$, that is *constant overhead* over cleartext evaluation. [9] These functionalities implemented by [FLNW17] and [CGH+18] make up for $\mathcal{F}_{\text{ABB1}}$(Figure 4).

We briefly review the properties of $\mathbb{F}_{2^l}$, which play a central part in our protocol. Let $x, y \in \mathbb{F}_{2^l}$. Recall that, in bits, x can be represented as $x_0 \cdots x_{l-1}$ where x_{l-1} is the highest order bit, and similarly, y can be represented as $y_0 \cdots y_{l-1}$ where $y_i, x_i \in \mathbb{F}_2$. Field addition in $\mathbb{F}_{2^l}$ is thus equivalent to bit-wise XOR, i.e.,

$$x + y = (x_0 \oplus y_0) \cdots (x_{l-1} \oplus y_{l-1}) = x \oplus y$$

Thus, secret sharing over $\mathbb{F}_{2^l}$ allows us to transition between bit and field secret sharing "for free" using bit-slicing. Let $x, y \in \mathbb{F}_{2^l}$ be additive secret shares of $z \in \mathbb{F}_{2^l}$. That is, $x + y = x \oplus y = z$. This implies that $x_i \oplus y_i = z_i$. Thus, if x is held by P_0 and y by P_1, to get a secret sharing of $z[i : j] \in \mathbb{F}_{2^{j-i}}$, P_0 holds $x[i : j]$ and P_1 holds $y[i : j]$. This, of course, requires no communication. In the special case where $i = j - 1$, we have transitioned to bit secret-sharing. Furthermore, the same truncation tricks work over replicated secret sharing: to obtain $[\![x[i : j]]\!]$, P_k can simply hold $(x^{(k)}[i : j], x^{(k+1)}[i : j])$. Additionally, we can let $[\![z]\!] = [\![X||Y]\!]$ by bitwise appending the shares of Y to the corresponding shares of X. This allows us to swap back from a Boolean sharing to an Arithmetic sharing with only local operations. For notational simplicity, this "casting" of data-types is usually implicit. In particular, if operations occur on fields of different sizes, $\mathbb{F}_{2^l}$, $\mathbb{F}_{2^{l'}}$ where $l > l'$, the item from the smaller field is implicitly first cast to the large field by prefixing it with zeros.

6.2 UC-realizing $\mathcal{F}_{\textbf{ABB2}}$ in the malicious model: composing on $\mathcal{F}_{\textbf{ABB1}}$

Recall the the Universal Composition Theorem [Can01]:

Theorem 2 (The Universal Composition Theorem (as in [Tof07])). *Let $\mathcal{F}$ and $\mathcal{G}$ be two ideal functionalities, and let π UC-realise $\mathcal{F}$ in the $\mathcal{G}$-hybrid model. Moreover, let ρ UC-realize $\mathcal{G}$. Then the composed protocol, $\pi^{\rho/\mathbf{G}}$, UC-realises $\mathcal{F}$.*

Let π implement functionality $\mathcal{F}$ using only calls to $\mathcal{F}_{\text{BasicABB}}$, and with no calls to the $\mathcal{F}_{\text{BasicABB}}$.Output. Then the Universal Composition theorem implies that trivial simulator running the exact same code as the adversary exists. This simulator works because both the real and ideal executions are composed only of oracle calls, and hence are indistinguishable to any environment $\mathcal{Z}$. Therefore, the Universal Composition Theorem implies the following:

[8] The fact that [FLNW17] is UC-secure is stated in the ePrint version: `https://eprint.iacr.org/2016/944.pdf`, p35. The UC-security of [CGH+18] comes from instantiating their base protocol using [LN17], which is UC-secure, c.f. `https://eprint.iacr.org/2017/816.pdf`, p21.

[9] We only to evaluate MPC multiplication over $\mathbb{F}_{2^l}$ for $l > \kappa$ and $\mathbb{F}_2$. Assuming that $\kappa > 4\sigma$ (which is reasonable as in practice $\kappa \in \{128, 256\}, \sigma \in \{40, 80\}$) where σ is our computational security parameter, the constant overhead over $\mathbb{F}_{2^l}$ is only 2 [CGH+18][Page 3]. The overhead over $\mathbb{F}_2$ depends on the number of AND gates, G, we will evaluate throughout the protocol. For $G = 2^{20}, \sigma = 2^{-40}$ we have that the constant overhead is 3 [FLNW17][Corollary 5.4]. This is reasonable, as for $N > 2^{15}$ just instantiating the DORAM would involve evaluating over 2^{20} gates.

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. We denote communication/computation costs **in green.**

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

Input(x, pId, varName): Receive x from party pId and store it as $[\![varName]\!]$ $\boldsymbol{O(|x|)}$.
RandomElement(l, outName): Sample $x \in_R \mathbb{F}_{2^l}$ and store x in $[\![outName]\!]$ $\boldsymbol{O(l)}$.
OR($[\![x]\!]$, $[\![y]\!]$, outName): For $x, y \in \mathbb{F}_2$, compute $z = x \vee y$. Store z in $[\![outName]\!]$ $\boldsymbol{O(1)}$.
Add($[\![x]\!]$, $[\![y]\!]$, outName): Compute $z = x + y$ and store z in $[\![outName]\!]$ $\boldsymbol{O(|x|)}$.
Mult($[\![x]\!]$, $[\![y]\!]$, outName): Compute $z = x * y$ and store z in $[\![outName]\!]$ $\boldsymbol{O(|x|)}$.
Output($[\![z]\!]$, to, localVarName): Receive $[\![z]\!]$ from at least one P_i for $i \neq to$. Send z to to, who store it in the local variable $localVarName$ $\boldsymbol{O(|x|)}$.

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

Equal($[\![x]\!]$, $[\![y]\!]$, outName): If $x \overset{?}{=} y$ set z to 1, otherwise to 0. Store z in $[\![outName]\!]$ $\boldsymbol{O(|x|)}$.
IfThenElse($[\![b]\!]$, $[\![x]\!]$, $[\![y]\!]$, outName): If $b \overset{?}{=} 1$, set z to x, otherwise set z to y. Store z in $[\![outName]\!]$ $\boldsymbol{O(|x|)}$.
CreateMAC($[\![x]\!]$, outName): Create a "tag," τ, on the data X, and store t in $[\![outName]\!]$ $\boldsymbol{O(|x| + \kappa)}$.
CheckMAC($[\![x]\!]$, $[\![\tau]\!]$, outName): Check whether the tag, τ is a valid tag on the message, x, and store the (boolean) result in $[\![outName]\!]$ $\boldsymbol{O(|x| + \kappa)}$.
PRFEval($[\![x]\!]$, $[\![k]\!]$, outName): Compute $z = \mathrm{PRF}_k(x)$, a pseudorandom function on input x over key $k \in \mathbb{F}_{2^\kappa}$ and store z in $[\![outName]\!]$ $\boldsymbol{O(|x| + \kappa)}$.

$\mathcal{F}_{ABB3}$**: 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 $[varName]^{(i,j)}$ $\boldsymbol{O(|x|)}$.
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 $[\![varName]\!]$ $\boldsymbol{O(|x|)}$.

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

ObliviousShuffle($[\![X]\!]$, outName): Let $X = X_0, \ldots, X_{n-1}$. Sample a random (secret) permutation $\pi \in_R S_n$ and store $[\![\pi(X)]\!] = [\![\pi(X_0)]\!], \ldots, [\![\pi(X_l)]\!]$ in $[\![outName]\!]$ $\boldsymbol{O(\sum_{i=0}^{n-1}(|X_i| + \kappa))}$. 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($[\![X]\!]$, $[\![Y]\!]$, $[\![M]\!]$, outName): $X = X_0, \ldots, X_{n-1}$, $Y = Y_0, \ldots, Y_{n-1}$. $M_i = \alpha Y_i$ for all $0 \leq i \leq n-1$. Compute $z = \sum_{i=0}^{n-1} X_i Y_i$ and store z in $[\![outName]\!]$ $\boldsymbol{O(|X_i| + |Y_i| + \kappa)}$.

Fig. 4: Arithmetic Black Box functionality.

Corollary 1. *Let Π_{ABB1} UC-realize $\mathcal{F}_{ABB1}$. Let π implement functionality $\mathcal{F}$ using only calls to $\mathcal{F}_{BasicABB}$, and with no calls to the $\mathcal{F}_{ABB1}$.Output. Then $\pi^{\Pi_{\mathrm{ABB1}}/\mathcal{F}_{ABB1}}$ UC-realizes $\mathcal{F}$.*

Via the corollary above, we instantiate all the functionalities in $\mathcal{F}_{\mathrm{ABB2}}$ (as done by, for example [DFK$^+$06, Lau15]). Equal, IfThenElse, CreateMAC and CheckMAC are evaluated via standard (folklore) circuits, and inputs are first mapped to $\mathbb{F}_{2^\kappa}$ (or a larger extension field) if they are not already in this field. PRFEval is instantiated via the LowMC cipher[10] [ARS$^+$15].

Analysis in [FNO22][Appendix B] shows that the the LowMC circuit for evaluating κ-bit security LowMC on l bits has size $O(\kappa + l)$. Note that evaluating a secure PRF inside of a secure computation is a more challenging task than the well-studied primitive of an Oblivious PRF (OPRF), in which the inputs and outputs are private, rather than secret-shared [CHL22].

6.3 UC-realizing $\mathcal{F}_{\mathrm{ABB3}}$ in the malicious model: 2 party functionalities

Our (3,1) DORAM construction will crucially rely on the ability of 2 parties to store secret-shared data between them. This will allow them to access data using handles that are known to them, but unknown to the third party. For instance, an array of items may be secret-shared between 2 parties. A protocol can reveal to these 2 parties an index of this array to access. They can access this item and convert the secret to an item stored in the ABB (i.e., stored a replicated CNF sharing), without the third party ever learning which index in the array was accessed.

However, this creates a challenge. A malicious party may tamper with their shares. In the case of CNF sharing, the redundancy in the secret-sharing allows this tampering to be detected. For non-replicated secret sharing schemes, we detect such tampering by using MACs. That is, the 2 parties hold an XOR secret-sharing of both the secret and of the MAC. While an adversary can cause an additive error to both the reconstructed secret and the reconstructed MAC, since it does not know the MAC key, it will not be able to cause the protocol to reconstruct an incorrect value, except with negligible probability.

Since this technique is ubiquitous in the literature, we move the protocol Π_{ABB3} implementing $\mathcal{F}_{\mathrm{ABB3}}$ to Figure 11 in Appendix A. In Appendix A we also give a full proof to the following theorem:

Theorem 3. *Against a static malicious adversary controlling at most one party, Protocol Π_{ABB3} (Figure 11) statistically UC-realizes functionality $\mathcal{F}_{ABB3}$ with abort in the $\mathcal{F}_{ABB1}$,$\mathcal{F}_{ABB2}$-hybrid model.*

6.4 UC-realizing $\mathcal{F}_{\mathrm{ABB4}}$ in the malicious model: specialized functionalities

In this section, we show how to securely implement the ObliviousShuffle and the SilentDotProduct of $\mathcal{F}_{\mathrm{ABB4}}$.

Oblivious Shuffle. We present an adaptation of [LWZ11] to the malicious-adversary setting which is particularly convenient to prove in the $\mathcal{F}_{\mathrm{ABB1}}, \mathcal{F}_{\mathrm{ABB2}}, \mathcal{F}_{\mathrm{ABB3}}$-hybrid model. The protocol begins by resharing an array of items stored in the ABB to P_1 and P_2. That is the parties call $[Y_i^{1,2}]^{(1,2)} =$ ReplicatedTo2Sharing($[\![X_i]\!], 1, 2$) for all $i \in [n]$. P_1 then sends P_2 a random permutation, π. P_1 and P_2 then re-insert the items to the ABB, shuffled according to this permutation[11]. That is, they call $[\![Z_i]\!] =$ 2SharingToReplicated($[Y_{\pi(i)}^{1,2}]^{(1,2)}$) for $i \in [n]$. $[\![Z]\!]$ now contains the same items as $[\![X]\!]$, but permuted according to a permutation unknown to P_3. The above protocol is repeated with the roles of (P_1, P_2) being taken by (P_2, P_3) and then by (P_3, P_1), the final array will be permuted according to a random permutation unknown to any of the parties. Due the MACs implicitly added by ReplicatedTo2Sharing and checked by 2SharingToReplicated the functionality costs $O(n(\kappa + |X_i|))$ communication and computation.

Since the functionality is randomized, we need to show that the combined distributions of the adversary's view and the protocol's output are indistinguishable from the distribution of a simulator's view and the functionality's output. The protocol and the functionality both result in the array stored by the ABB being randomly permuted according to a permutation unknown to the adversary. (While this

[10] The security of LowMC has been adapted into the Picnic signature scheme [CDG$^+$17], a 3rd round candidate in the NIST post-quantum digital signature contest [NIS21]. Additionally, LowMC has received several thorough cryptanalysis attempts [LIM20, BBVY21, DLMW15] motivated by an ongoing Microsoft-funded challenge, `https://lowmcchallenge.github.io/`

[11] For improved efficiency, the permutation could instead be generated by a PRG, where the seed is known by P_1 and P_2, but not P_3. This would yield a computationally-secure shuffle.

permutation is known to the 2 honest parties, this does not assist a distinguisher $\mathcal{Z}$ who only has access to the adversary/simulator.) Furthermore, this distribution is independent of the adversary's/simulator's actions. Therefore it suffices to show that we can simulate the adversary. A simple simulator exists for this protocol: $\mathcal{S}$ generates a random permutation and sends it to $\mathcal{A}$ (as the honest party would). $\mathcal{S}$ then outputs the same result as $\mathcal{A}$.

SilentDotProduct: Using the silent-multiplication property of (3,1)-CNF shares (a.k.a. replicated secret shares) and MACs to build a secure "silent" dot product between $[\![X]\!] = [\![X_1]\!], ...[\![X_n]\!]$ and $[\![Y]\!] = [\![Y_1]\!], \ldots, [\![Y_n]\!]$ for $O((|X_i| + |Y_i|)\kappa)$ communication and computation. Since similar tricks have previously appeared in the literature (e.g. [CDI05, BIKO12]) we reserve the full explanation to the Appendix B.

6.5 Additional remarks

In this section we present three important remarks about the nature of our ABB which impact the rest of our presentation, future work, and the security of our protocol.

Remark 5 (Notational note). For notational clarity, we use standard assignment notation to show a new variable being stored in the ABB, dropping its name from the function call. For example, we will replace $\mathcal{F}_{\text{ABB}}$.PRFEval($[\![x]\!], [\![k]\!], Q$) with $[\![Q]\!] = \mathcal{F}_{\text{ABB}}$.PRFEval($[\![x]\!], [\![k]\!]$). Occasionally, we use infix notation to call a functionality, for instance, replacing $\mathcal{F}_{\text{ABB}}$.Mult($[\![x]\!], [\![y]\!], z$) with $[\![z]\!] = [\![x]\!] * [\![y]\!]$. Also, we occasionally place constants in the secret-sharing notation (e.g. $[\![true]\!]$), in which case, the constant is first implicitly input as a secret variable.

Remark 6 (Security up to an adversarial abort). Our ABB is secure up to *up to an adversarial abort*[12], and thus our DORAM function also achieves security with abort. We believe this relaxation is appropriate with respect to the DORAM functionality because the input and output of the DORAM functionality are *secret shared* and reveal no information. Thus, by selecting to not give output to a subset of the honest players, an adversary does not gain an information advantage over the honest players, who will abort the protocol immediately after not receiving output. Moreover, the adversary cannot abort in response to the access-pattern, input, etc, preventing output delivery upon some (meaningful) event. Additionally, it is sometimes the case (Figure 6) that the functionalities allow the adversary to choose its secret share of the output (while not allowing it to choose the output). This is of course not a violation to any meaningful notion of security.

7 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 $\mathcal{F}_{\text{OMap}}$, (Figure 3).

In similar works, the cache is often instantiated by executing a linear-scan under MPC [FNO22] with 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.

[12] See e.g. [IKK+11][Definition 2.1]

Protocol $\mathit{\Pi}_{\mathbf{QuietCache}}$

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

$\mathit{\Pi}_{\mathbf{QuietCache}}$**.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 $i \in [w]$: $\llbracket M_i \rrbracket = \mathcal{F}_{\text{ABB}}.\text{CreateMAC}(\llbracket Y_i \rrbracket)$

$\mathit{\Pi}_{\mathbf{QuietCache}}$**.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: $\llbracket M_t \rrbracket = \mathcal{F}_{\text{ABB}}.\text{CreateMAC}(\llbracket y \rrbracket)$

$\mathit{\Pi}_{\mathbf{QuietCache}}$**.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 $i = t, \ldots, 1$
      i. $\llbracket \mathit{isMatch}_i \rrbracket = \mathcal{F}_{\text{ABB}}.\text{Equal}(\llbracket X_i \rrbracket, \llbracket X \rrbracket)$
      ii. $\llbracket \mathit{isBeforeMatch}_i \rrbracket = \llbracket \mathit{isBeforeMatch}_{i+1} \rrbracket \vee \llbracket \mathit{isMatch}_{i+1} \rrbracket$ (Except that $\llbracket \mathit{isBeforeMatch}_t \rrbracket = 0$)
      iii. $\llbracket b_i \rrbracket = \llbracket \mathit{isMatch}_i \rrbracket \wedge (\neg \llbracket \mathit{isBeforeMatch}_i \rrbracket)$
2. Then efficiently select the item based on this indicator array using $\mathcal{F}_{\text{ABB}}$.SilentDotProduct:
   (a) $\llbracket y \rrbracket = \mathcal{F}_{\text{ABB}}.\text{SilentDotProduct}((\llbracket b_i \rrbracket^D)_{i=1}^t, \llbracket Y_i \rrbracket_{i=1}^t, \llbracket M_i \rrbracket_{i=1}^t)$
   (b) Return $\llbracket y \rrbracket$.

$\mathit{\Pi}_{\mathbf{QuietCache}}$**.Extract():**

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

Fig. 5: $\mathit{\Pi}_{\text{QuietCache}}$: Protocol for the cache (implementation of smallest $\mathcal{F}_{\text{OMap}}$).

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 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 (Figure 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 $\Pi_{QuietCache}$ (Figure 5) UC-realizes functionality $\mathcal{F}_{QuietCache}$ (Figure 5) with abort in the $\mathcal{F}_{ABB}$-hybrid model.*

Proof. For the correctness of $\Pi_{\text{QuietCache}}$.Query, note that while $\Pi_{\text{QuietCache}}$.Store does not delete old index-value pairs when a new copy of the index is stored, it places the new item at the end of the array. This effectively overwrites the old value, since $\Pi_{\text{QuietCache}}$.Query always uses the last-occurring version of the item.

For the correctness of $\Pi_{\text{QuietCache}}$.Extract, note that every item (and only items) inserted (either during Init or Add) will be stored somewhere in the array, and that the most recently inserted value will occur latest. If there is a later occurrence of an item, the protocol replaces the first occurrence with $(\llbracket\perp\rrbracket, \llbracket\perp\rrbracket)$. Therefore, the list $\llbracket\hat{X}\rrbracket, \llbracket\hat{Y}\rrbracket$ will contain all stored indices with their most-recently stored value. It will have a number of items equal to the total number of insertions (including overwrites). Finally, the output is shuffled before being returned as required.

Since $\Pi_{\text{QuietCache}}$ only makes calls to $\mathcal{F}_{\text{ABB}}$ and is correct, security follows directly from Corollary 1.

In Appendix E.1 we prove that:

Proposition 2. *The communication and computational 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.*

8 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 (Section 9) we use this as a sub-protocol to build (a variant of) an OHTable.

At a high level, Π_{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.

Functionality $\mathcal{F}_{\mathbf{OSet}}$

$\mathcal{F}_{\mathbf{OSet}}$**.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.

$\mathcal{F}_{\mathbf{OSet}}$**.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 $\Pi_{\mathbf{OSet}}$

Hybrids: $\mathcal{F}_{\text{ABB}}, \mathcal{F}_{\text{ZKPOfValidCHT}}$.
$\Pi_{\mathbf{OSet}}$**.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 $\mathsf{CHT} \cup \mathsf{Stash} = \{\mathsf{CHT}, (i_1, \ldots, i_s)\} = \mathsf{BuildCHTwS}(\hat{Q}, h_0, h_1)$ (protocol, of complexity $O(n)$, defined in Appendix C).
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) $[\![b]\!] = \mathcal{F}_{\text{ZKPOfValidCHT}}.\mathbf{Verify}([\![\tilde{Q}]\!], [\![\mathsf{CHT}]\!], c, h_0, h_1, \log(N))$
   (b) $b = \mathcal{F}_{\text{ABB}}.\text{Reveal}([\![b]\!])$. Abort if not b.
10. Secret-share the CHT to P_1 and P_2:
   $[\mathsf{CHT}]^{(1,2)} = \mathcal{F}_{\text{ABB}}.\text{ReplicatedTo2Sharing}([\![\mathsf{CHT}]\!], \{1,2\})$.

$\Pi_{\mathbf{OSet}}$**.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 Π_{OSet}: Functionality and Protocol for a Distributed Oblivious Partial Set

Functionality $\mathcal{F}_{\textbf{ZKPOfValidCHT}}$

$\mathcal{F}_{\textbf{ZKPOfValidCHT}}$**.Verify(**$[\![X]\!]$**,** $[\![\mathsf{CHT}]\!]$**,** c**,** h_0**,** h_1**,** ℓ**, 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 $\Pi_{\textbf{ZKPOfValidCHT}}$

$\Pi_{\text{ZKPOfValidCHT}}$.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^ℓ and add the prefix $1||O^{\ell-S-1}$ to all real elements.
   (c) Pick a random evaluation point for the polynomial:
   $[\![w]\!] = \mathcal{F}_{\text{ABB}}.\text{RandomElement}(\kappa)$
   (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:
   $[\![check_1]\!] = \mathcal{F}_{\text{ABB}}.\text{Equal}([\![u]\!], [\![v]\!])$

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.

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 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 PRFs of the items 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 Section 6.1). 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. As mentioned in section 5, 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 (section 5) 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 Figure 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 Figure 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 Figure 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 $\Pi_{\text{ZKPOfValidCHT}}$ 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 that Π_{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 Π_{OSet} (Figure 6) statistically (with failure probability negligible in N) realizes functionality $\mathcal{F}_{OSet}$ (Figure 6) with abort in the $\mathcal{F}_{ABB}, \mathcal{F}_{ZKPOfValidCHT}$-hybrid model.*

Proof. We will have one simulator S_0, when $\mathcal{A}$ corrupts P_0, and a second simulator S_1, when $\mathcal{A}$ corrupts P_1 or P_2.

S_0 will work as follows. During the build, S_0 will pick n random values from $GF(2^\kappa)$. S_0 will provide these as the Q values to a local copy of $\mathcal{A}$. If $\mathcal{A}$ generates an invalid Cuckoo Hash Table, or if $\mathcal{A}$ aborts, then S_0 aborts. Otherwise S_0 calls $\mathcal{F}_{\text{OSet}}$.Build($[\![X]\!], n, N, stash$), which will generate some S and store S in $[\![stash]\!]$ in the ABB. During queries, S_0 simply executes $\mathcal{F}_{\text{OSet}}$.Query($[\![x]\!], res$). Throughout, S_0 responds to any messages from the environment in the same way its local copy of $\mathcal{A}$ would.

The environment attempts to distinguish $\mathcal{A}$ acting as P_0 interacting with Π_{OSet} and S_0 interacting with $\mathcal{F}_{\text{OSet}}$. Firstly, we show that during an execution, the probability that the environment can distinguish them is negligible (in N). Since the PRF key is always kept inside the ABB, the sequence of evaluations of the PRF on the items in X is computationally indistinguishable from a sequence of n κ-bit values. As such, $\mathcal{A}$'s behavior in the real setting (when given PRF outputs) will be the same as $\mathcal{A}$'s behavior in the simulation, except with a probability corresponding to $\mathcal{A}$'s ability to distinguish random values from distinct PRF outputs, which is negligible. In particular, the difference in the probabilities of $\mathcal{A}$ aborting in the real setting and the simulation is negligible. Furthermore, the difference in the probabilities of $\mathcal{A}$ building an invalid Cuckoo Hash Table or sending an invalid S will also be negligible. Both of these events result in the protocol aborting and are the only actions $\mathcal{A}$ can take to cause an abort (apart from aborting directly itself). Therefore the differences in the probabilities of abort are negligible. Furthermore, since the PRF outputs are computationally indistinguishable from random, the copy of $\mathcal{A}$ run by S_0 (whose messages S_0 forwards to the environment) will have outputs indistinguishable from $\mathcal{A}$ in the real execution. Thus, the environment cannot distinguish $\mathcal{A}$ from S_0.

Note that an honest P_0 may also fail to build a valid Cuckoo Hash Table since there is an inherent probability of failure when building a Cuckoo Hash Table with a stash. Since the stash is of size $log(N)$

and $n = \omega(\log(N))$, this probability is negligible [Nob21]. We do not actually need this fact here as S_0 can simulate $\mathcal{A}$ even if $\mathcal{A}$ behaves honestly. However, it is necessary for the non-degeneracy requirement: that is if all parties are honest the protocol should not abort except with negligible probability.

Now we consider what may occur after the Build has finished executing. The value $[\![stash]\!]$ in both cases must be a bit array of length n with s ones. Note, however, that the stash generated by $\mathcal{F}_{\text{OSet}}$ in the simulation may not be the same (or from the same distribution) as the one generated by the copy of $\mathcal{A}$ run by S_0. (S_0 knows the stash $\mathcal{A}$ generated, but has no knowledge or control over $[\![stash]\!]$ as it is generated by the functionality.) However, this discrepancy will never be noticed. Condition 3 guarantees that any outputs from the ABB by the environment executing other protocols will not leak any information about $[\![stash]\!]$. As such, even if $\mathcal{A}$ were to provide its stashed items to the environment, there is nothing the environment can do to learn about whether these items were used as the real stash or not. Finally, $\mathcal{A}$ receives no information during a Query, so S_0 can trivially simulate $\mathcal{A}$ during Queries.

For S_1, simulating the Build is trivial. Since P_1 and P_2 receive no outputs during the Build, S_1 will simply do whatever $\mathcal{A}$ would do. During a Query, S_1 generates a fresh random κ-bit value and inputs this to $\mathcal{A}$ as q. It then responds as $\mathcal{A}$ would.

The PRF key is stored in the ABB and the PRF is evaluated inside of a computation, so the output of a single PRF evaluation is indistinguishable from a random value. Furthermore, since the same item is never queried more than once (Condition 1), the PRF outputs will all be distinct (pseudo)random values. In order to demonstrate their independence though, we need Condition 4. Condition 4 implies two things. Firstly, the environment cannot insert some bias either increasing, or reducing the probability that stash items are queried. Preventing biasing is not sufficient. For instance, if it is later revealed that a queried item was in the stash this would be essentially the same as the adversary having caused a stashed item to be queried. Therefore Condition 4 also implies that any revealed value will not leak information about whether queried items were stashed. Note that conditioning based on whether a stash item was (or wasn't) queried, either due to biasing or leakage causes the PRF outputs to not be fully independent, as stash items have PRFs that result in more collisions with other items in the table. Since the environment cannot influence, or learn, whether stashed items are queried, the outputs of Query on items that were in X will be computationally indistinguishable.

The environment can, however, choose to influence whether to query items in X or items that were not in X. This can create a problem. The hashes of the PRF outputs may correspond to a choice of locations which is incompatible with the items all being stored in the table (i.e. the stash size would be greater than s if these items were all stored). If the environment queries only items in the table, this can never happen in the real execution because it is known that the table was built correctly. It may, though, happen in the simulation as a new sequence of n random values are chosen. However, since there are at most n queries (Condition 2), the probability of this occurring is at most the probability of a build failure in a Cuckoo Hash Table of n items with a stash of size s, which is negligible in N [Nob21].

Therefore, the behavior of $\mathcal{A}$ in the simulated setting will be computationally indistinguishable from that of $\mathcal{A}$ in the real setting. Thus, the environment will be unable to distinguish S_1 interacting with $\mathcal{F}_{\text{OSet}}$ from $\mathcal{A}$ interacting with Π_{OSet}.

In Appendix E.2 we prove that:

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

9 Maliciously-secure Oblivious Hash Table Construction

In the previous section, we presented a Distributed Oblivious Set protocol. We now show how to use this to build a Distributed Oblivious Hash Table (OHTable): that is a protocol for securely mapping indices to values provided each item is only queried once. In short, this will be achieved by first using the OSet to check whether the item 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

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

Build(($[\![X]\!]$,$[\![Y]\!]$, 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^{stash} and Y^{stash} respectively.
Query($[\![x]\!]$, res):
$x \in [N] \cup \{\bot\}$. If $\exists j \in [n]; x = X_j$ and $x \notin X^{\text{stash}}$, set $z = [\![Y_j]\!]$. Else, set $z = [\![\bot]\!]$. 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 $(\bot, \bot)$. 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($[\![X]\!]$, $[\![Y]\!]$, n N, $X^{\mathbf{stash}}$, $Y^{\mathbf{stash}}$):

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

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

Extract(res):

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

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

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 $(\perp, \perp)$).

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 after all calls to Query have been executed.

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 (Π_{OHTable}) and functionality ($\mathcal{F}_{\text{OHTable}}$) in Figure 8. We now prove its security. Firstly, we need to demonstrate that if Π_{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 $\mathcal{F}_{OSet}$.*

Proof. Firstly, we observe that satisfying Condition 6 implies Condition 2 is satisfied. $\mathcal{F}_{\text{OSet}}$ can only be queried if Π_{OHTable} is queried. Since there are at most n queries to Π_{OHTable} (Condition 6) there will be at most n queries to $\mathcal{F}_{\text{OSet}}$, satisfying Condition 2.

Π_{OHTable} can only be queried on distinct real values, but may be queried multiple times on the input $\perp$ (Condition 5). In such a case, this is identified, and a distinct item from $\{N+1, \ldots, N+n\}$ is queried to $\mathcal{F}_{\text{OSet}}$. Note that this set is disjoint from the possible real values of x, which are in $\{1, \ldots, N\}$. Therefore, the queries to $\mathcal{F}_{\text{OSet}}$.Query will be distinct, satisfying Condition 1 of $\mathcal{F}_{\text{OSet}}$.

We now wish to show that if Conditions 7 and 8 are satisfied, Conditions 3 and 4 will also be satisfied.

First we need to show that no values output by Π_{OHTable} itself undermine Condition 3. During the build, the value $\tilde{S}$ is revealed. This is a random permutation of S, which from the correctness of Π_{OHTable} is a bit array consisting of s ones. Therefore, the revealed $\tilde{S}$ will be a random bit array with s ones. This leaks no information about S itself as it was already known that S has s ones. The build also reveals the values $\hat{Q}$. These are secure evaluations of a PRF under an unknown key, where the key has never been queried on these inputs before. Therefore these values will be computationally indistinguishable from random, and in particular will not leak any information about whether the inputs were in the stash. Thus, Condition 3 remains satisfied.

During Query, the value q is revealed. By the correctness of $\mathcal{F}_{\text{OSet}}$, this value is equal to $\text{PRFEval}(x, k)$ only if $x \in X$ and $x \notin X^{\text{stash}}$. Otherwise it is $\text{PRFEval}(N + t, k)$. Since the set $\hat{Q}$ contains exactly the PRF evaluations of the non-stash indices and the PRF evaluations of $N+1, \ldots, N+n$ we are guaranteed that q will be in the original set $\hat{Q}$. (Here we use Condition 6, so $1 \leq t \leq n$.) Furthermore, since t is incremented during each query, and non-null values are never re-queried (Condition 5), this will be a value of q that has never been revealed before, so $(\hat{X}_i, \hat{Y}_i, \hat{Q}_i)$ will not have been deleted. Lastly, due to the randomness of the shuffle of tagged items in the Build, the value i such that $q = \hat{Q}_i$ will be a random value from $[2n - s]$ that has not yet been deleted. Therefore, the Query reveals no information about whether inputs were in the stash of $\mathcal{F}_{\text{OSet}}$ and Condition 3 remains satisfied.

Before continuing, we need two basic correctness facts. First, observe that the input set, X, to $\mathcal{F}_{\text{OSet}}$.Build is exactly the same as the input indices, X, in $\mathcal{F}_{\text{OSet}}$.Build. Second, note that for all $X_i \in X$, $X_i \in X^{\text{stash}}$ in $\mathcal{F}_{\text{OHTable}}$.Build exactly when $S_i = 1$. By Condition 8, during Π_{OHTable}.Query, for every $x \in X$, the probability that x is the input to Query is (computationally indistinguishable from) independent of whether $x \in X^{\text{stash}}$. Therefore when $\mathcal{F}_{\text{OSet}}$.Query is called, the probability that $x = X_i$ is also (computationally indistinguishable from) independent of S_i. Therefore 4 remains satisfied.

During the extract, no values are revealed.

We now need to prove that outside Π_{OHTable}, the environment cannot do anything to break Conditions 3 or 4 without breaking Conditions 7 or 8. First, note that S is shuffled immediately after being output by $\mathcal{F}_{\text{OSet}}$.Build and is never used in its unshuffled form. The only information flow from S is into the choice of *stashX*. However, no information about *stashX* may be revealed by the environment (Condition 7). Therefore, no information may be leaked about *stash* outside of Π_{OHTable} showing that Condition 3 is always satisfied.

Π_{OHTable}.Query, given an input x, queries this exact same value to $\mathcal{F}_{\text{OSet}}$.Query, except when $x = \perp$. Clearly, when $x = \perp$, $x \notin X$, so for all $x \in X$, the query is passed on the $\mathcal{F}_{\text{OSet}}$. Our conditions are not concerned with the case $x \notin X$. If $x = X_i \in X$ we have from Condition 8 that the event of choosing x was (computationally indistinguishable from) independent of the choice of X^{stash} (conditioned on all values revealed by $\mathcal{F}_{\text{ABB}}$ outside of the protocol whether before or after). However, the only information flow from *stash* is to X^{stash}, so this event must also be independent of the choice of *stash*, and Condition 4 is also always satisfied.

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.*

Proof. Our simulator is simple. During a build it generates a random bit-string $\tilde{S}$ of length n consisting of s ones. It provides this to its local copy of $\mathcal{A}$ (and likewise for all other values it generates). It then generates $2n - 2s$ values chosen uniformly at random from $\{0, 1\}^\kappa$ and outputs these as $\hat{Q}$. During a query it selects a random value from $\hat{Q}$ that has not yet been selected and outputs it as q. The simulator calls the functionality on the same inputs and aborts if $\mathcal{A}$ aborts.

Since $\tilde{S}$ is known to have s ones and is shuffled before being revealed, it is from the same distribution as the value generated by the simulator. The values $\hat{Q}$ are the result of PRF evaluations on an unknown key, so to a computationally-bounded $\mathcal{A}$ will be indistinguishable from random κ-bit values except with negligible probability. Lastly, the values of q revealed in the query are guaranteed to be some new value from $\hat{Q}$. (See proof of Proposition 6 for more details.) As to which value it is, this is completely randomized by the Oblivious Shuffle of the tagged items during the build. Therefore, q will be chosen uniformly at random from the unaccessed values in $\hat{Q}$, exactly the same as in the simulation. Therefore our simulator will provide its copy of $\mathcal{A}$ with values that are from either identical, or computationally indistinguishable from, the distributions generated by a real execution. Therefore, $\mathcal{A}$ will behave identically in both executions except with negligible probability. Since the simulator responds as $\mathcal{A}$ would, the environment will not be able to distinguish the two executions.

Finally, in Appendix E.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 $O(nD)$.*

10 Maliciously-secure Oblivious Map Construction

As noted above, Oblivious Hash Tables (Section 9) 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[13]. 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 with an unscalable efficiency). 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) *Firstly Init*($[\![X]\!], [\![Y]\!], n$) *is called, where len*($[\![X]\!]$) $=$ *len*($[\![Y]\!]$) $= w \leq \log(N) < \frac{\kappa}{4}$
Then there are at most $n - w$ *calls to Query*($[\![x]\!]$) *each followed immediately by a call to Add*($[\![x]\!], [\![y]\!]$) *(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 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 κ 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 D) 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 Π_{OMap}, as well as the functionality $\mathcal{F}_{\text{OMap}}$ that it implements, are presented in detail in Figures 9 and 3 respectively. We next prove the security of Π_{OMap} with respect to $\mathcal{F}_{\text{OMap}}$ is formally stated and proven below. Note that since Π_{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 Π_{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 that* $\Pi_{OMap}[n, N]$ *is called according to Condition 10, it is a correct implementation of* $\mathcal{F}_{OMap}[n, N]$ *in the* $\mathcal{F}_{ABB}, \mathcal{F}_{OHTable}, \mathcal{F}_{OMap}[\frac{n}{2} + \frac{s}{2}, N]$*-hybrid model.*

Proof. As Init and Add do not return any values, we only need to show that the values returned by Query and Extract are correct. The protocol maintains an invariant that before and after every pair

[13] Recall that we need to recursve on OMaps rather than ORAMs. See Remark 4)

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 on the OHTable:
   (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

of Query, Add calls, each index which has ever been inserted (either through Init or Add) is stored in exactly one place in either $\mathcal{F}_{\text{OMap}}$ or $\mathcal{F}_{\text{OHTable}}$ and is stored with its most recent value. By "stored" we mean two things: if queried to this functionality it would be found and that if extract were called the item would be returned. Initially this is true as all items from Init are placed in the sub-OMap. By the correctness of $\mathcal{F}_{\text{OMap}}$, the associated values would be returned by a query and the full dictionary (with no old values) returned by an extract call. Initially, calls to query/add will simply be passed directly to $\mathcal{F}_{\text{OMap}}$. Since $\mathcal{F}_{\text{OMap}}$ implements a dictionary, it will always return the most recent version of a value that was written, and furthermore when extracted will contain the contents of the dictionary. When t reaches $\frac{n}{2} + \frac{s}{2}$, the contents of $\mathcal{F}_{\text{OMap}}$ will be extracted and built into $\mathcal{F}_{\text{OHTable}}$. This will contain all indices ever added to Π_{OMap} (including via initialization) with their most recent values. However, the stash items will not be stored in $\mathcal{F}_{\text{OHTable}}$, but will be used to initialize a new $\mathcal{F}_{\text{OMap}}$. Still the invariant holds: each item used for the build will either be located in $\mathcal{F}_{\text{OHTable}}$ or $\mathcal{F}_{\text{OMap}}$. Henceforth, when an item is queried it may be found either in $\mathcal{F}_{\text{OMap}}$ or $\mathcal{F}_{\text{OHTable}}$. If found in $\mathcal{F}_{\text{OMap}}$ it will not be removed from $\mathcal{F}_{\text{OMap}}$ by Query, but the subsequent call to Add will overwrite the old value in $\mathcal{F}_{\text{OMap}}$ with the updated value. If found in $\mathcal{F}_{\text{OHTable}}$, the query effectively deletes the item from the OHTable, as extract returns only unqueried items and the item is not allowed to be requeried to $\mathcal{F}_{\text{OHTable}}$. The item, with its new value, is then written to $\mathcal{F}_{\text{OMap}}$, so the invariant is maintained.

Since the invariant holds before every call to Query, if the item exists in the dictionary, it will either be found (with its most recent value) in $\mathcal{F}_{\text{OMap}}$ or will be found (with its most recent value) in $\mathcal{F}_{\text{OHTable}}$. The most recent value will then be returned. If the item does not exist in the dictionary, it was never added to $\mathcal{F}_{\text{OMap}}$, so will not be in $\mathcal{F}_{\text{OMap}}$ or $\mathcal{F}_{\text{OHTable}}$ and $\perp$ will be returned as required. Therefore Query returns correct values.

Since the invariant holds after every call to Add, it will hold before the call to Extract. This means every item that was stored that was every added (including through initialization) will either be in the result of $\mathcal{F}_{\text{OMap}}$.Extract or the result of $\mathcal{F}_{\text{OHTable}}$.Extract and will appear with its most recent value. Therefore $[\![X]\!], [\![Y]\!]$ will contain the full up-to-date dictionary.

As for empty items, the extract of $\mathcal{F}_{\text{OMap}}$ will, apart from the real values, only contain copies of $(\perp, \perp)$. $\mathcal{F}_{\text{OHTable}}$.Extract may contain some additional non-real items. This is because when the first $\mathcal{F}_{\text{OMap}}$ is extracted, it may contain empty items, and these are given 1-prefixed indices before being passed to $\mathcal{F}_{\text{OHTable}}$.Build. $\mathcal{F}_{\text{OHTable}}$.Extract will return all of these items (they will not have been queried because their indices are 1-prefixed). Π_{OMap}.Extract handles this by setting the indexes of these will be set back to $\perp$ during the extract (the values will still be $\perp$ as they were $\perp$ during the build). Therefore, apart from the contents of the dictionary, all other items will be copies of $(\perp, \perp)$. The total number of items returned by $\mathcal{F}_{\text{OMap}}$.Extract will be $\frac{n}{2} + \frac{s}{2}$. The total number of items returned by $\mathcal{F}_{\text{OHTable}}$.Extract will be $\frac{n}{2} + \frac{s}{2} - s = \frac{n}{2} - \frac{s}{2}$. Therefore the total number of items returned by Extract will be n as required. Finally, the items are shuffled before being returned, so will be in an arbitrary order. Therefore Π_{OMap}.Extract will return a correct value (chosen from a correct distribution).

Now we prove that the necessary conditions hold on the sub-functionalities.

Proposition 10. *Assuming that Π_{OMap} is called according to Condition 10, it calls sub-functionality $\mathcal{F}_{OHTable}$ according to Conditions 5, 2, 7, 8 and 9.*

Proof. Firstly, we observe the pattern of calls to $\mathcal{F}_{\text{OHTable}}$. It is first built during the call to Π_{OMap}.Add when $t = \frac{n}{2} + \frac{s}{2}$. It is built from the extracted contents of $\mathcal{F}_{\text{OMap}}$ which contains exactly $\frac{n}{2} + \frac{s}{2}$ items. (Some of these may have been empty, but if so they are given unique indices.) Therefore $\mathcal{F}_{\text{OHTable}}$ may be queried at most $\frac{n}{2} + \frac{s}{2}$ times. Since Extract is called when $t = n$, there will in fact be only $\frac{n}{2} - \frac{s}{2}$ calls to $\mathcal{F}_{\text{OHTable}}$, so Condition 6 is satisfied. Following these queries $\mathcal{F}_{\text{OHTable}}$ is extracted and never used again, satisfying Condition 9.

Next we show there are no repeated queries to $\mathcal{F}_{\text{OHTable}}$, except for no-op queries. If a real item, x, is queried once to $\mathcal{F}_{\text{OHTable}}$, this query will necessarily be followed by a call of Π_{OMap}.Add($[\![x]\!], [\![y]\!]$) for some value of y. Therefore, x shall be stored in the sub-OMap. Furthermore, the Alibi bit will be set to 0. If x is queried again, it will be found in the sub-OMap, so b_{found} will be 1. The Alibi bit will still be 0, so b_{Alibi} will be 0. Therefore, $\perp$ will be queried instead of $[\![x]\!]$. Thus, there are no repeated queries of real items, and Condition 5 is satisfied.

Finally, we show the stash conditions are upheld. No values are revealed from the ABB in this protocol. $\mathcal{F}_{\text{OMap}}$ is a local object, so the environment cannot access it directly to learn information about the stashed items. Therefore the only way information about the stash could leak is by values that are returned by Π_{OMap}, either in Query or Extract. But the value returned by Query is pre-determined: it is the value that was last written to the queried index. (This will be proven later when we show correctness.) The contents of Extract are also pre-determined: they are the contents of the dictionary padded with $(\perp, \perp)$. Their order is re-randomized by the final shuffle, so leaks no information about which items were stashed. Therefore, it is impossible for any ABB-revealed value to reveal information about the contents of the stash, satisfying Condition 7.

Lastly, we need to show that the query pattern does not depend on whether an item was stashed. This is simple, if Query(x) is called after $\mathcal{F}_{\text{OHTable}}$ has been built, x is always queried to $\mathcal{F}_{\text{OHTable}}$ the first time (and, as already shown $\perp$ queried any subsequent times). If x was in the stash, this is ensured by the fact that x was placed in $\mathcal{F}_{\text{OMap}}$ with an Alibi bit set to 1. It will be found in the sub-OMap and, due to its Alibi bit, still queried in $\mathcal{F}_{\text{OHTable}}$. If x was not in the stash, and has not been queried since $\mathcal{F}_{\text{OHTable}}$ was built, it cannot be stored in the sub-OMap. Therefore it will not be found, b_{found} will be 0 and x will be queried in $\mathcal{F}_{\text{OHTable}}$. Therefore Condition 8 will be satisfied.

Proposition 11. *Assuming that $\Pi_{OMap}[n, N]$ is called according to Condition 10, it calls sub-functionality $\mathcal{F}_{OMap}[\frac{n}{2} + \frac{s}{2}, N]$ according to Conditions 10.*

Proof. Π_{OMap} calls the full cycle of calls of $\mathcal{F}_{\text{OMap}}$ twice, so we check that Condition 10 holds in both cycles.

In the first cycle, Π_{OMap} initializes $\mathcal{F}_{\text{OMap}}$ with exactly the same inputs that Π_{OMap} was initialized with. Therefore, if $w < \frac{\kappa}{4}$ holds in Π_{OMap}, it will also hold in $\mathcal{F}_{\text{OMap}}$. Following this, there are $\frac{n}{2} + \frac{s}{2} - w$ repetitions of Π_{OSet}.Query followed by Π_{OSet}.Add, using the same index. Thus there will be $\frac{n}{2} + \frac{s}{2} - w$ alternating calls to $\mathcal{F}_{\text{OMap}}$.Query and $\mathcal{F}_{\text{OMap}}$.Add. Recall that the capacity of $\mathcal{F}_{\text{OMap}}$ is $\frac{n}{2} + \frac{s}{2}$, so at this point, $\mathcal{F}_{\text{OMap}}$ has to be extracted. This is exactly what occurs: Π_{OHTable}'s counter t has reached $\frac{n}{2} + \frac{s}{2}$ so the sub-OMap is extracted and its contents built into the OHTable. Thus, in the first cycle, Condition 10 is satisfied on $\mathcal{F}_{\text{OMap}}$.

In the second cycle, $\mathcal{F}_{\text{OMap}}$ is initialized with the contents of the stash. This is of size $s = \log(N)$ which can be presumed to be smaller than $\kappa/4$ since $\kappa = \omega(\log(N))$. Again, since Condition 10 holds on Π_{OHTable} calls to Query are alternated with calls to Add using the same index. Extract will occur in Π_{OHTable} after a total of at most $n - w$ calls to Query and Add in Π_{OHTable}. Since there were $\frac{n}{2} + \frac{s}{2} - w$ of these alternating calls in the first cycle, the total number of these alternating calls to $\mathcal{F}_{\text{OMap}}$ in the second cycle will be at most $\frac{n}{2} - \frac{s}{2} \leq \frac{n}{2} + \frac{s}{2} - w$. These are followed by Π_{OMap}, and therefore $\mathcal{F}_{\text{OMap}}$, being extracted. Therefore, Condition 10 is satisfied in the sub-OMap $\mathcal{F}_{\text{OMap}}$.

Combining these shows that Π_{OMap} is a secure implementation of $\mathcal{F}_{\text{OMap}}$ subject to the necessary conditions.

Proposition 12. *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.*

Proof. The protocol satisfies all conditions on $\mathcal{F}_{\text{OHTable}}$ (Proposition 10) and $\mathcal{F}_{\text{OMap}}$ (Proposition 11). There are no conditions on $\mathcal{F}_{\text{ABB}}$. The protocol is also correct with regard to the functionality (Proposition 9. Therefore, since Π_{OMap} doesn't reveal any values, by Corollary 1, given the above (conditioned) hybrids, as long as Condition 10 is satisfied, Π_{OMap} is a secure implementation of $\mathcal{F}_{\text{OMap}}$.

Proposition 13. *If Π_{OMap} is implemented with its functionalities instantiated in the following ways:*

- *$\mathcal{F}_{OMap}$ implemented recursively with Π_{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 Π_{OSet}*
- *$\mathcal{F}_{ABB}$ is implemented as described in Section 6*

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):* $\Theta(\log(N)(\kappa + D))$

Proof. The initialization cost will be passed on all the way to the base case, where it will have cost $\Theta(\kappa w)$ (Proposition 2).

For Query, let us first look at the cost ignoring the recursion. The cost of the two IfThenElse statements are $\log(N)$ and D respectively and the cost of $\mathcal{F}_{\text{OHTable}}$.Query is $\Theta(\kappa)$ (Proposition 17). These dominate all other costs.

Now, there are $O(\log(N))$ levels of the recursion, since $\frac{n}{2} + \frac{s}{2} < \epsilon n$ for some constant $\epsilon < 1$ when $n \geq \kappa = \omega(\log(N))$. The recursive calls may have slightly larger values due to the appended Alibi bits, but since there is only one bit added per layer of recursion, the size of the values will be at most $D + \Theta(\log(N))$. Finally, the cost of the base level is $\Theta(\kappa + D)$ (Proposition 2). Therefore, the total cost of all recursive levels is $\Theta(\log(N)(D + \log(N) + \kappa) + (\kappa + D)) = \Theta(\log(N)(\kappa + D))$.

For Add and Extract, we compute the total amortized combined cost of these throughout the data structure.

At the base level, the cost of Add is cheap, only $\Theta(\kappa)$. However the cost of extracting is expensive as this is when deleting old copies occurs. Given that the base level is of capacity $\Theta(\kappa)$, the cost of extracting will be $\Theta(\kappa^2 \log(N) + \kappa D)$ (Proposition 2). Amortized over $\Theta(\kappa)$ accesses, this is $\Theta(\kappa \log(N) + D)$ per access.

At the recursive OMap levels, the cost of Add in most cases is basically free: an Alibi bit is added (which requires no communication) and the call is passed onto the sub-OMap. However, Add will result in a rebuild when $t = \frac{n}{2} + \frac{s}{2}$. We skip over the cost of $\mathcal{F}_{\text{OMap}}$.Extract here, it will be included in the amortized cost of Extracts in the level below. The other costs are dominated by $\mathcal{F}_{\text{OHTable}}$.Build, which costs $\Theta(n(\kappa + D))$. (There is also the cost of $\mathcal{F}_{\text{OMap}}$.Init, but this is only $\Theta(w\kappa) = o(n\kappa)$.) Amortized over n accesses, this cost is $\Theta(\kappa + D)$.

Now we compute the amortized cost of Extract on the recursive OMap levels. Note that there will be exactly one instance of $\mathcal{F}_{\text{OMap}}$ at each level of the recursion at the start and end of each call to Add in the top level. If some OMap becomes full during an Add, it will be extracted during the Add, but will be re-initialized by the end of the Add. Therefore, if we compute the amortized cost of Extract in a recursive OMap during its lifetime, ignoring recursive calls, and multiply this by the depth of recursive calls, we obtain the amortized cost of extract over all levels. The cost of extracting (ignoring the recursive call) is one call to Π_{OHTable}.Extract at a cost of $\Theta(nD)$, resetting of empty elements add a cost of $\Theta(n \log(N))$ and an oblivious shuffle at a cost of $\Theta(nD)$ leading to a total cost of $\Theta(nD)$. Amortized per access this is $\Theta(D)$. Since there are $\Theta(\log(N))$ levels of the recursion, the amortized access over all recursive OMaps is $\Theta(\log(N)D)$.

Combining this to the cost of the base-level, the combined amortized cost of Add and Extract over all levels is $\Theta(\kappa \log(N) + D + \kappa + D + \log(N)D) = \Theta((\kappa + D)\log N)$.

11 Maliciously-secure DORAM

Our recursive OMap protocols has handled most of the limitations of our other protocols. However, it still has one limitation: it can only be called a certain number of times before the contents must be extracted.

Note that this is an inherent challenge to defining OMaps. In OMaps the index space may be much larger than the capacity of the OMap. This can be useful, our recursive solution is only possible with OMaps because smaller levels must use the same index space but have smaller capacities. However, this also means that if new items are constantly added, eventually the capacity will become depleted. This leads to a dilemma. One option is that the OMap can only handle a certain number of calls to Add. This is the choice used by Π_{OMap} and is reflected in the limit to the calls to Query/Add in Condition 10. The other option is to have a bound on the number of distinct *items* that may be stored in the underlying dictionary.

In an ORAM, however, the capacity is equal to the index space. That is every possible index stores some value.

We implement such a DORAM. It uses $\mathcal{F}_{\text{OMap}}$, implemented by Π_{OMap}, to store items once they are queried. It is essentially a wrapper around $\mathcal{F}_{\text{OMap}}$. It stores all N items using Add. The OMap is of capacity $2N$ to allow for N additional queries. When the OMap is extracted it will necessarily contain N

items with indices $1, \dots, N$ and N empty indices, in a shuffled order. The indices can safely be revealed, empty values deleted, and the non-empty values re-built into a new $\mathcal{F}_{\text{OMap}}$. The protocol is formally presented in Figure 10 below.

Protocol $\Pi_{\mathbf{DORAM}}$

$\Pi_{\mathbf{DORAM}}$**.Init($[\![Y]\!]$, N):**

1. Initialize an OMap with capacity twice that of the DORAM. This will allow it to store N items and support a further N queries before being rebuilt. It is not initialized with any items:
   (a) $\mathcal{F}_{\text{OMap}}.\text{Init}([\![]\!], [\![]\!], 0, 2N, N)$
2. Store all of the items. (For technical reasons we need to query them each first.) For $i \in [N]$:
   (a) $\mathcal{F}_{\text{OMap}}.\text{Query}([\![i]\!])$
   (b) $\mathcal{F}_{\text{OMap}}.\text{Add}([\![i]\!], [\![Y_i]\!])$
3. Set counter $t = 0$

$\Pi_{\mathbf{DORAM}}$**.ReadAndWrite($[\![x]\!]$, $[\![y_{new}]\!]$):**

1. $[\![y_{new}]\!] = \mathcal{F}_{\text{OMap}}.\text{Query}([\![x]\!])$
2. $\mathcal{F}_{\text{OMap}}.\text{Add}([\![x]\!], [\![y_{new}]\!])$
3. $t = t + 1$
4. If $t = N$, a rebuild must occur:
   (a) $[\![X]\!], [\![Y]\!] = \mathcal{F}_{\text{OMap}}.\text{Extract}()$
   (b) For $i \in [2N]$: $X_i = \mathcal{F}_{\text{ABB}}.\text{Output}([\![X_i]\!])$.
   (c) The list X will contain the items in $[N]$ exactly once and N copies of $\perp$.
   (d) Delete the $\perp$ items.
   (e) Sort remaining items according to the indices.
   (f) Let $[\![\hat{Y}]\!]$ be the values so sorted.
   (g) Call $\Pi_{\text{DORAM}}.\text{Init}([\![\hat{Y}]\!])$

Fig. 10: Π_{DORAM}, the DORAM protocol

We prove the security below.

Theorem 4. *Π_{DORAM} is a UC-secure implementation of $\mathcal{F}_{DORAM}$ when $\mathcal{F}_{OMap}$ is instantiated with Π_{OMap}, $\Pi_{OHTable}$, Π_{OSet}, $\Pi_{QuietCache}$ (for the smallest recursion of $\mathcal{F}_{OMap}$ of size $(\kappa/2, \kappa]$) and the ABB of section 6; specifically it is secure with abort against one static malicious adversary.*

Proof. The fact that $\mathcal{F}_{\text{OMap}}$ is called consistently with Condition 10 is visible by inspection.

The only values revealed by the protocol are the opened indexes. Since $\mathcal{F}_{\text{OMap}}$ was given all N values during initialization, and since $\mathcal{F}_{\text{OMap}}$ is of capacity $2N$, by the definition of $\mathcal{F}_{\text{OMap}}$.Extract, X will contain all N indices (paired with their most recent values) and N copies of $\perp$, in a random order. Revealing N is therefore easily simulatable: the simulator generates N copies of $\perp$ and the indices $1, \dots, N$ in a random order.

Correctness follows from the correctness of $\mathcal{F}_{\text{OMap}}$.

Since $\mathcal{F}_{\text{OMap}}$ is secure when implemented with the mentioned protocols, Condition 10 is satisfied, the protocol is correct and all outputs of Π_{DORAM} are simulatable, Π_{DORAM} is a UC-secure implementation of $\mathcal{F}_{\text{DORAM}}$ given these instantiations.

Adding the wrapper only creates a minor overhead relative to Π_{OMap}. This is proven formally below.

Theorem 5. *If Π_{DORAM} is instantiated with Π_{OMap}, $\Pi_{OHTable}$, Π_{OSet}, $\Pi_{QuietCache}$ (for the smallest recursion of $\mathcal{F}_{OMap}$ of size $(\kappa/2, \kappa]$) and the ABB of section 6; specifically it has amortized cost per read/write of $\Theta(\log(N)(\kappa + D))$*

Proof. When $\mathcal{F}_{\text{OMap}}$ is initialized, there are N calls to Query and Add at total cost $\Theta(\log(N)(\kappa + D)N)$. Initialization happens every N accesses, so the amortized cost of it is $\Theta(\log(N)(\kappa+D)$ per ReadAndWrite.

Ignoring the cost of re-initialization, the costs of a ReadAndWrite involve one Query (cost $\Theta(\log(N)(\kappa + D)))$ and one Add. There is also an Extract every N accesses. The combined amortized cost of Add and Extract is $\Theta(\log(N)(\kappa + D))$. These costs dominate other steps of the protocol.

Therefore, the amortized cost per ReadAndWrite is $\Theta(\log(N)(\kappa + D)$

Remark 7 (Deamortization). As described, our DORAM protocol has *amortized* query complexity of $O((\kappa + D) \log N)$. Our protocol (like most hierarchical (D)ORAMs) can be deamortized using Ostrovsky and Shoup's pipelining technique [OS97], yielding a protocol with $O((\kappa + D) \log N)$ communication in *the worst case.*

Acknowledgments

Supported in part by DARPA under Cooperative Agreement HR0011-20-2-0025, the Algorand Centers of Excellence programme managed by Algorand Foundation, NSF grants CNS-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 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] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In *EUROCRYPT*, 2020.

[AKLS21] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. Oblivious RAM with worst-case logarithmic overhead. In *CRYPTO*, pages 610–640. Springer, 2021.

[AKST14] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In *PKC*, 2014.

[ARS$^+$15] Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *EUROCRYPT*, pages 430–454. Springer, 2015.

[BBVY21] Subhadeep Banik, Khashayar Barooti, Serge Vaudenay, and Hailun Yan. New attacks on LowMC instances with a single plaintext/ciphertext pair. Cryptology ePrint Archive, Paper 2021/1345, 2021. `https://eprint.iacr.org/2021/1345`.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT*, 2015.

[BIKO12] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and Ilan Orlov. Share conversion and private information retrieval. In *2012 IEEE 27th Conference on Computational Complexity*, pages 258–268. IEEE, 2012.

[BKKO20] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed ORAM. In *SCN*, 2020.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty Unconditionally Secure Protocols. In *STOC*, 1988.

[CDG$^+$17] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. Post-quantum zero-knowledge and signatures from symmetric-key primitives. Cryptology ePrint Archive, Paper 2017/279, 2017. `https://eprint.iacr.org/2017/279`.

[CDI05] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005. Proceedings 2*, pages 342–362. Springer, 2005.

[CGH$^+$18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *Annual International Cryptology Conference*, pages 34–64. Springer, 2018.

[CHL22] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. SoK: Oblivious pseudorandom functions. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 625–646. IEEE, 2022.

[DFK$^+$06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography: Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006. Proceedings 3*, pages 285–304. Springer, 2006.

[DK12] Michael Drmota and Reinhard Kutzelnigg. A precise analysis of cuckoo hashing. *ACM Transactions on Algorithms (TALG)*, 8(2):1–36, 2012.

[DLMW15] Itai Dinur, Yunwen Liu, Willi Meier, and Qingju Wang. Optimized interpolation attacks on LowMC. Cryptology ePrint Archive, Paper 2015/418, 2015. `https://eprint.iacr.org/2015/418`.

[DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *Advances in Cryptology-CRYPTO 2003: 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003. Proceedings 23*, pages 247–264. Springer, 2003.

[DO20] Sam Dittmer and Rafail Ostrovsky. Oblivious tight compaction in $o(n)$ time with smaller constant. In *SCN*, pages 253–274. Springer, 2020.

[Ds17] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In *CCS*, 2017.

[DvDF$^+$16] Srinivas Devadas, Marten van Dijk, Christopher Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *TCC*, 2016.

[FJKW15] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-party ORAM for secure computation. In *ASIACRYPT*, 2015.

[FLNW17] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT*, pages 225–255. Springer, 2017.

[FNO21] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. Alibi: A flaw in cuckoo-hashing based hierarchical ORAM schemes and a solution. In *EUROCRYPT*, pages 338–369. Springer, 2021.

[FNO22] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. 3-party distributed ORAM from oblivious set membership. In *International Conference on Security and Cryptography for Networks*, pages 437–461. Springer, 2022.

[FNR$^+$15] Christopher W Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket ORAM: Single online roundtrip, constant bandwidth oblivious RAM. IACR ePrint 2015/1065, 2015.

[GGH$^+$13] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. Optimizing oram and using it efficiently for secure computation. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 1–18. Springer, 2013.

[GHL$^+$14] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. Garbled RAM revisited. In *EUROCRYPT*, 2014.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT*, 2014.

[GKK$^+$12] S Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

[GKW18] S Dov Gordon, Jonathan Katz, and Xiao Wang. Simple and efficient two-server ORAM. In *ASIACRYPT*, 2018.

[GMOT12] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *STOC*, 1987.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *JACM*, 43(3), 1996.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 182–194. ACM, 1987.

[HV20] Ariel Hamlin and Mayank Varia. Two-server distributed ORAM with sublinear computation and constant rounds. IACR ePrint 2020/1547, 2020.

[IKK$^+$11] Yuval Ishai, Jonathan Katz, Eyal Kushilevitz, Yehuda Lindell, and Erez Petrank. On achieving the "best of both worlds" in secure multiparty computation. *SIAM journal on computing*, 40(1):122–141, 2011.

[JW18] Stanislaw Jarecki and Boyang Wei. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In *ACNS*, 2018.

[JZLR22] Keyu Ji, Bingsheng Zhang, Tianpei Lu, and Kui Ren. Multi-party private function evaluation for RAM. Cryptology ePrint Archive, Paper 2022/939, 2022. `https://eprint.iacr.org/2022/939`.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[KM19] Eyal Kushilevitz and Tamer Mour. Sub-logarithmic distributed oblivious RAM with small block size. In *PKC*, 2019.

[KMW09] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 2009.

[KO97] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 364–373. IEEE Computer Society, 1997.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 506–525. Springer, 2014.

[Lau15] Peeter Laud. Parallel oblivious array access for secure multiparty computation and privacy-preserving minimum spanning trees. *Proc. Priv. Enhancing Technol.*, 2015(2):188–205, 2015.

[LIM20] Fukang Liu, Takanori Isobe, and Willi Meier. Cryptanalysis of full LowMC and LowMC-M with algebraic techniques. Cryptology ePrint Archive, Paper 2020/1034, 2020. `https://eprint.iacr.org/2020/1034`.

[LN17] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 259–276, 2017.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In *CRYPTO*, 2018.

[LO13] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, 2013.

[LWZ11] Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-efficient oblivious database manipulation. In Xuejia Lai, Jianying Zhou, and Hui Li, editors, *Information Security*, pages 262–277, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Mit09] Michael Mitzenmacher. Some open questions related to cuckoo hashing. In *ESA*, 2009.

[MV23] Surya Mathialagan and Neekon Vafa. MacORAMa: Optimal oblivious RAM with integrity. Cryptology ePrint Archive, Paper 2023/083, 2023. `https://eprint.iacr.org/2023/083`.

[MZ14] John C Mitchell and Joe Zimmerman. Data-oblivious data structures. In *STACS*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

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

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

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage. In *STOC*, volume 97, 1997.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, 1990.

[Ost92] Rafail Ostrovsky. *Software Protection and Simulation On Oblivious RAMs.* PhD thesis, Massachusetts Institute of Technology, 1992.

[PPRY18] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In *FOCS*, 2018.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *ESA*, 2001.

[PR10] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 515–530, 2015.

[RFK$^+$14] Ling Ren, Christopher W Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Ring ORAM: Closing the gap between small and large client storage oblivious RAM. IACR ePrint 2014/997, 2014.

[SCSL11] Elaine Shi, T-H Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

[SVDS$^+$13] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*, 2013.

[Tof07] Tomas Toft. Primitives and applications for multi-party computation. *Unpublished doctoral dissertation, University of Aarhus, Denmark*, 2007.

[VHG22] Adithya Vadapalli, Ryan Henry, and Ian Goldberg. Duoram: A bandwidth-efficient distributed ORAM for 2- and 3-party computation. Cryptology ePrint Archive, Paper 2022/1747, 2022. `https://eprint.iacr.org/2022/1747`.

[Vol99] Heribert Vollmer. *Introduction to circuit complexity: a uniform approach.* Springer Science & Business Media, 1999.

[WCS15] Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *CCS*, 2015.

[WHC$^+$14] Xiao Shaun Wang, Yan Huang, T-H Hubert Chan, abhi shelat, and Elaine Shi. SCORAM: oblivious RAM for secure computation. In *CCS*, 2014.

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

[Yao86] Andrew Yao. How to generate and exchange secrets. In *FOCS*, 1986.

[Yeo22] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. *Cryptology ePrint Archive*, 2022.

[ZWR$^+$16] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: efficient random access in multi-party computation. In *S & P*, 2016.

A Proofs and protocols for UC-realizing $\mathcal{F}_{\mathbf{ABB3}}$ in the malicious model

Theorem 6. *Against a static malicious adversary controlling at most one party, Protocol Π_{ABB3} (Figure 11) statistically UC-realizes functionality $\mathcal{F}_{ABB3}$ with abort in the $\mathcal{F}_{ABB1}$,$\mathcal{F}_{ABB2}$-hybrid model.*

Proof. We show that a simulator interacting with the functionality will be indistinguishable from an adversary interacting with the protocol. Our simulator, $\mathcal{S}$, will work as follows. $\mathcal{S}$ will run a copy of $\mathcal{A}$, which it will access in a black-box, non-rewinding manner. Let P_i be the corrupted party. If P_i is one of the recipients of ReplicatedTo2Sharing, the simulator generates random x_i, M_i from $\mathbb{F}_{2^\sigma}$, and gives these to $\mathcal{A}$. Upon a call to 2SharingToReplicated, the simulator calls $\mathcal{A}$ and checks whether it returns

Protocol $\Pi_{\mathbf{ABB3}}$

Hybrids: Operates in the $\mathcal{F}_{\text{ABB1}}$, $\mathcal{F}_{\text{ABB2}}$-hybrid model.
ReplicatedTo2Sharing($\llbracket x \rrbracket$, i, j, varName^{i,j}):

1. Compute a MAC of the secret: $\llbracket M \rrbracket = \text{CreateMAC}(\llbracket x \rrbracket)$.
2. Generate two masks: $\llbracket r_x \rrbracket = \text{RandomElement}(\sigma)$, $\llbracket r_M \rrbracket = \text{RandomElement}(\sigma)$.
3. Reveal the masks to P_i as its shares: $x_i = \text{Output}(r_x, i)$, $M_i = \text{Output}(r_M, i)$.
4. Generate masked values: $\llbracket m_x \rrbracket = \text{Add}(\llbracket x \rrbracket, \llbracket r_x \rrbracket)$, $\llbracket m_M \rrbracket = \text{Add}(\llbracket M \rrbracket, \llbracket r_M \rrbracket)$.
5. Reveal the masked items to P_j as its shares: $x_j = \text{Output}(m_x, j)$, $M_j = \text{Output}(m_M, j)$.
6. P_i and P_j associate handle varName^{i,j} with the shares (x_i, M_i) and (x_j, M_j) respectively.

2SharingToReplicated($[x^{i,j}]^{(i,j)}$, varName):

1. Let (x_i, M_i) be P_i's shares associated with $x^{i,j}$ and let (x_j, M_j) be P_j's shares.
2. Input all variables: $\llbracket x_a \rrbracket = \text{Input}(x_i, i)$, $\llbracket M_a \rrbracket = \text{Input}(M_i, i)$, $\llbracket x_b \rrbracket = \text{Input}(x_j, j)$, $\llbracket M_b \rrbracket = \text{Input}(M_j, j)$.
3. Reconstruct the secret and MAC inside the ABB: $\llbracket varName \rrbracket = \text{Add}(x_a, x_b)$, $\llbracket M \rrbracket = \text{Add}(M_a, M_b)$.
4. Verify the MAC: $\llbracket z \rrbracket = \text{CheckMAC}(x, M)$, $y = \text{Output}(\llbracket z \rrbracket, \{1, 2, 3\})$.
   If $y == 0$ (i.e. the MAC is incorrect), all players abort.

Fig. 11: Two-Sharing functionality.

the secret-share and MAC-share that it was given for this handle. If it does, then $\mathcal{S}$ lets $\mathcal{F}_{\text{ABB3}}$ continue and store the secret using the new handle in the ABB. If $\mathcal{A}$ has modified either the secret-share or the MAC-share, then $\mathcal{S}$ commands $\mathcal{F}_{\text{ABB3}}$ to abort.

The view of $\mathcal{A}$ during ReplicatedTo2Sharing is completely identical in both executions. In both cases it receives two values chosen uniformly at random from $\mathbb{F}_{2^\sigma}$. If $\mathcal{A}$ provides correct secret shares to 2SharingToReplicated, then the correct value will be reconstructed, the MAC will be correct, and the real and ideal executions will be identical.

If $\mathcal{A}$ provides incorrect secret shares to ReplicatedTo2Sharing, then the ideal execution will always abort. If $\mathcal{A}$ provides incorrect secret shares, the real execution will abort if the secret-MAC pair is inconsistent. Let us compute the probability that the modified secret-MAC pair is consistent. Let δ_x be the additive error $\mathcal{A}$ introduces to the secret, and δ_M be the additive error $\mathcal{A}$ introduces to the MAC. The MAC will be viewed as correct provided:

$$(x + \delta_x)\alpha = M + \delta_M$$

But $x\alpha = M$, so this simplifies to:

$$\delta_x \alpha = \delta_M$$

If $\mathbb{F}_{2^l}$ is a field, for α chosen uniformly at random from $\mathbb{F}_{2^l}$, as long as $y \neq 0$, $y\alpha$ is uniformly distributed over $\mathbb{F}_{2^l}$. This comes from the "0-divisor property" of fields. Since $\mathcal{A}$ has introduced some error, $\delta_x \neq 0$, so $\delta_x\alpha$ is uniformly distributed over $\mathbb{F}_{2^\sigma}$. The probability that $\delta_x\alpha = \delta_M$ is therefore $\frac{1}{|\mathbb{F}_{2^\sigma}|} = 2^{-\sigma}$, which is negligible in σ, our statistical security parameter. Therefore, except with negligible probability, the real execution will also abort if $\mathcal{A}$ provides incorrect sharings.

Therefore, the behavior of the $\mathcal{S}$ interacting with $\mathcal{F}_{\text{ABB3}}$ is indistinguishable from that of $\mathcal{A}$ interacting with Π_{ABB3}, except with negligible probability.

B Full implementation of $\mathcal{F}_{\text{ABB4}}$.SilentDotProduct

SilentDotProduct: Another useful property of replicated secret-shares (RSS) is 2-ary *silent* multiplication. Let x and y be RSS-shared, such that P_i holds $(x^{(i)}, x^{(i+1)})$ and $(y^{(i)}, y^{(i+1)})$, where $x^{(1)} + x^{(2)} + x^{(3)} = x$ and $y^{(1)} + y^{(2)} + y^{(3)} = y$. Note that, by the distributive property $xy = (x^{(1)}y^{(1)} + x^{(2)}y^{(1)} + x^{(1)}y^{(2)}) + (x^{(2)}y^{(2)} + x^{(3)}y^{(2)} + x^{(2)}y^{(3)}) + (x^{(3)}y^{(3)} + x^{(1)}y^{(3)} + x^{(3)}y^{(1)})$. Furthermore, observe that P_i can compute $x^{(i)}y^{(i)} + x^{(i+1)}y^{(i)} + x^{(i)}y^{(i+1)}$ using only local operations. Thus, using only local operations, the players can obtain an additive sharing of the product, that is P_i holds some $z^{(i)}$ such that $z^{(1)} + z^{(2)} + z^{(3)} = z = xy$. Moreover, by computing the an additive sharing of $Z_i = X_i Y_i$ and summing the shares, the players can *silently* compute the dot product (with additive error) of two replicated secret shared vectors $[\![X]\!] = [\![X_1]\!], \dots, [\![X_n]\!]$ and $[\![Y]\!] = [\![Y_1]\!], \dots, [\![Y_n]\!]$. To go back to replicated secret sharing, each party inputs their shares into the ABB. Summing the player's shares inside the ABB, the parties obtain $[\![Z]\!]$. This protocol preserves privacy (the parties receive no new information), but a malicious party can hold an incorrect additive share, allowing it to introduce arbitary additive error to the result. Thus, we call the above protocol DotProductWithError($[\![X]\!], [\![Y]\!]$).

Assuming that each $[\![Y_i]\!]$ is MACed by $[\![M_i]\!]$ (which can be reused across many executions) under a secret key $[\![\alpha]\!]$ it is not too difficult to make the above protocol maliciously secure. A protocol to implement $\mathcal{F}_{\text{ABB}}$.SilentDotProduct($[\![X]\!], [\![Y]\!], [\![M]\!]$) works as follows:

1. Set $[\![z]\!] = \text{DotProductWithError}([\![X]\!], [\![Y]\!])$.
2. Set $[\![m]\!] = \text{DotProductWithError}([\![X]\!], [\![M]\!])$.
3. Compute $[\![b]\!] = \text{CheckMAC}([\![z]\!], [\![m]\!])$.
4. Output $[\![b]\!]$ to all parties. If it is 0, abort, else, output $[\![z]\!]$.

The proof of security is similar to that of the 2-sharing protocols (Appendix A), so we just provide a proof sketch. $\mathcal{S}$ runs $\mathcal{A}$ (who receives no outputs on this protocol) to determine the errors δ_z and δ_m that $\mathcal{A}$ would insert into the first and second calls to DotProductWithError respectively. If either of these is non-zero, $\mathcal{S}$ aborts. If $(z + \delta_z)\alpha = m + \delta_m$, then $\mathcal{S}$ will cause an abort, but the protocol will not abort. However, this only occurs if $\delta_z \alpha = \delta_m$, which since $\delta_z \neq 0$ and α is chosen at random from $\mathbb{F}_{2^\sigma}$ occurs with negligible probability. Therefore, the real and ideal executions are identical except with negligible probability.

C Instantiating BuildCHTwS

Cuckoo Hash tables are well-studied, and there are several different methods for efficiently building Cuckoo Hash Tables [Yeo22][Appendix C] in $O(n)$ time. Our protocol could work with any of them, so in this section, we lay out the semantics of the BuildCHTwS algorithm, but do not prescribe a specific procedure for building the cuckoo hash table. We do, however, require the cuckoo build algorithm will always output a stash of size s,

Since all of our cuckoo hash builds occur locally (i.e., not under MPC), there are no constraints on which build procedure is used.

The BuildCHTwS algorithm takes a set of indices as well as two hash functions h_0, h_1.

$$\mathsf{CHT} \cup \mathsf{Stash} = \{\mathsf{CHT}, (i_1, \dots, i_s)\} = \mathsf{BuildCHTwS}(\hat{Q}, h_0, h_1) \tag{3}$$

BuildCHTwS

1. Parse Q as n key-value pairs, $((X_1, Y_1), \dots, (X_n, Y_n))$.
2. Parse h_b as a hash function $h_b : \{0,1\}^* \to [c]$.
3. Build the Cuckoo Hash Table as in [Yeo22][Appendix C]
4. Return the table(s) CHT, as well as the set of "stashed" indices $i_1, \dots, i_s$ where $i_j \in [n]$, and CHT is a cuckoo hash table containing all elements $\{X_i\}$ for $i \in [n] \setminus \{i_1, \dots, i_s\}$.

Fig. 12: Building a Cuckoo Hash Table with a Stash

D The Alibi method for "caching the stash"

At a high level "Alibi" [FNO21] shows that a naive "cache the stash" technique popular in the literature leaks information about the access pattern. In their paper, [FNO21] offer an alternative "cache the stash" technique which effectively "simulates" the no-caching (D)ORAM access-pattern. [FNO21]'s technique which appends $\mathsf{numLevels}$ bits of information, $\mathfrak{e}_i$, in each payload Y_i. Note this does not mean we need to make the assumption that $D \geq \log N$. In fact, we do not rely on that assumption anywhere in the paper to obtain our asymptotics.

Leveraging this information, [FNO21] shows a "just-in-time" protocol for simulating the no-caching access pattern.

The Alibi technique works as follows: For all $j \in [N]$, the alibi bit vector, $\mathfrak{e}_j$, is initially set to $0^{\mathsf{numLevels}} = 0^{\Theta(\log N)}$ First, if $(\llbracket x \rrbracket, \llbracket y \rrbracket)$ could not fit in the OHTable at level i and was reinserted into the linear level, we let their corresponding $\mathfrak{e}[i] = 1$. Then, if $(\llbracket x \rrbracket, \llbracket y \rrbracket)$ is found at L_j, and $k > j$ where $k = \max(\{i : \mathfrak{e}[i] = 1\} \cup \{0\})$, continue querying $L_{j+1}, \ldots L_k$ as if $(\llbracket x \rrbracket, \llbracket y \rrbracket)$ has not been found, and query $L_{k+1}, \ldots, L_{\mathsf{numLevels}}$ if $(\llbracket x \rrbracket, \llbracket y \rrbracket)$ was not found in L_k. Intuitively, this simulates the query pattern we would make if our (D)ORAM did not reinsert elements. Resetting the mark, if $(\llbracket x \rrbracket, \llbracket y \rrbracket)$ was queried by the DORAM, when we put $(\llbracket x \rrbracket, \llbracket y \rrbracket)$ back in the linear level, we, let $\mathfrak{e} = 0^{\mathsf{numLevels}}$ since now $(\llbracket x \rrbracket, \llbracket y \rrbracket)$ are where they "should" be. Of course, we do each of the above boolean operation using our $\mathcal{F}_{\mathrm{ABB}}$(Section 6), which is efficient due to our $GF(2^l)$ representation. These ideas are presented and proved formally in [FNO21].

Concretely, for a given element $(\llbracket x \rrbracket, \llbracket y \rrbracket)$ we reserve the last $\mathsf{numLevels}$ bits of y for $\mathfrak{e}$, letting $\llbracket \mathfrak{e} \rrbracket = \llbracket y[D - \mathsf{numLevels} + 1 : D] \rrbracket$. Below we present our instantiation of Alibi, and the rest of the DORAM protocol.

E Proofs about the costs of protocols

We give straightforward proofs about the cost of our protocols in this section.

E.1 Cost of $\mathit{\Pi}_{\mathbf{QuietCache}}$

Proof (Proposition 2). We tally the communication for each subprotocol:

1. $\mathit{\Pi}_{\mathrm{QuietCache}}$.Init: no communication.
2. $\mathit{\Pi}_{\mathrm{QuietCache}}$.Store: $\mathcal{F}_{\mathrm{ABB}}$.CreateMAC for an element of size D costs $O(\max\{D, \kappa\})$.
3. $\mathit{\Pi}_{\mathrm{QuietCache}}$.Query: The FindOne($t$) circuit has computes t equality tests (on $\log(N)$-bit elements), t AND gates, and t OR gates. Thus, for $t = O(n)$ evaluating the FindOne(t) circuit requires $O(n \log N)$ communication. Additionally, we pay $O(\max\{D, \kappa\})$ for $\mathcal{F}_{\mathrm{ABB}}$.CheckMAC and $O(D)$ for $\mathcal{F}_{\mathrm{ABB}}$.Input. Thus the total communication is $O(D + n \log N)$.
4. $\mathit{\Pi}_{\mathrm{QuietCache}}$.Extract: We call $\mathcal{F}_{\mathrm{ABB}}$.Equal $O(n^2)$ times, each time on $\Theta(\log N)$ bits, which requires a total of $O(n^2 \log N)$ communication. The $O(n^2)$ $\mathcal{F}_{\mathrm{ABB}}$.ORcalls we make cost a constant number of bits each. The $O(n)$ calls to $\mathcal{F}_{\mathrm{ABB}}$.IfThenElse, we make each cost $O(n \log N)$. Thus the calls to $\mathcal{F}_{\mathrm{ABB}}$.Equal dominate the and the complexity of $\mathit{\Pi}_{\mathrm{QuietCache}}$.Extract is $O(n^2 \log N)$.

Since all operations involve storing and retrieving elements or doing computations under MPC, the asymptotic computational complexity of each sub-protocol is the same as its communication complexity.

E.2 Propositions and proofs about the complexity of $\mathit{\Pi}_{\mathbf{OSet}}$

We present propositions and proofs concerning the communication and computation complexity of $\mathit{\Pi}_{\mathrm{OSet}}$.

Proposition 14. $\mathit{\Pi}_{OSet}$.Build *runs at communication and computation cost of* $O(n(\kappa + D))$.

Proof. Let us tally the cost of every step. Step 1 is free, step 2 costs $O(n(\kappa + \log N))$, step 3 costs $O(n(\kappa{+}D{+}\log N))$, step 4 costs $O(\kappa n)$. Step 5 is silent, but costs the builder $O(n)$ (See Appendix C). Step 6 costs $O(\log N \log n)$, steps 7,8,9,10 cost $O(n\kappa)$, step 11 costs 1, and step 12 is free. Since $\kappa = \omega(\log N)$ Our tally is $O(n(\kappa + D))$ communication and computation.

Note that since we rebuild a level of n elements every $O(n)$ queries, the amortized build complexity per query is $O(\kappa + D) \leq O((\kappa + D) \log N)$, as deisred.

Proposition 15. $\mathit{\Pi}_{OSet}$.Query *runs at communication and computation cost of* $O(\kappa)$.

Proof. Step 1 costs $O(\kappa + \log N)$, step 2,3,4 cost $O(\kappa)$ step 5 costs 1 and step 6 costs $O(\kappa)$.

Note that since we query $O(\log N)$ levels per query we have that the amortized cost per query of querying Osets across all levels is $O(\kappa \log N)$

E.3 Propositions and proofs about the costs of Π_{OHTable}

We analyze the communicaitonal and computational complexity of Π_{OHTable} with the following propositions:

Proposition 16. *$\Pi_{OHTable}$.Build has complexity $O(n(\kappa + D))$.*

Proof. Step 1 has complexity $O(n(\kappa + D))$, step 2 has complexity 0, step 3,4 have complexity $O(n\kappa)$ since $\mathsf{numDummies} = O(n)$, step 5 has complexity $O(n(\kappa + D))$, step 6 has complexity $O(n\kappa)$, and step 7 is free.

Proposition 17. *$\Pi_{OHTable}$.Query has complexity $O(\kappa)$.*

Proof. Step 1 is free, step 2 costs κ, step costs $O(\log N)$, step 4,5 costs $O(\kappa)$, step 6,7 are free.

Proposition 18. *$\Pi_{OHTable}$.Extract has complexity $O(n)$*

Proof. Communication is free and computation is at most $O(n)$, depending on the underlying data structure representation.

E.4 Propositions and proofs about the costs of Π_{DORAM}

Before we can prove our main theorem, we must determine that Π_{DORAM} realizes $\mathcal{F}_{\text{DORAM}}$ *efficiently.* Thus let us analyze the complexity of Π_{DORAM}:

Proposition 19. *The complexity of Π_{DORAM}.Init is $O(N\kappa)$*

Proof. Step 1 is silent has computational complexity $O(N)$, step 2 costs $O(N\kappa)$ by Proposition 16, step 3 is free.

Proposition 20. *The complexity of Π_{DORAM}.ReadAndWrite not including calls to Π_{DORAM}.Rebuild (step 8) is $O((\kappa + D)\log N)$*

Proof. Steps 1 and 2 costs 0. By Proposition 2 step 3 costs $O(\kappa \log N + D)$. By Proposition 17 and our instantiation of $\mathcal{F}_{\text{ABB}}$.Equal and $\mathcal{F}_{\text{ABB}}$.IfThenElse, step 4 costs $O(\kappa + D)$ per iteration for an total of $((\kappa + D)\log N)$ over all iterations. Step 5 costs D while steps 6 and 7 free. As stated in the proposition, we ignore the cost of step 8.

We can consider the complexity of calling Π_{DORAM}.Rebuild on the t'th query, but is much cleaner to leverage the well-known hierarchical solution fact that a level of m elements is built every $O(m)$ queries to prove the following proposition:

Proposition 21. *The amortized cost per query of calling Π_{DORAM}.Rebuild in step 8 of Π_{DORAM}.ReadAndWrite is $O(\kappa \log N)$.*

Proof. Step 1,2,3 are free. Assume that we are rebuilding L_ℓ for $\ell < \mathsf{numLevels}$ where $|L_\ell| = m$. Then, step $4a$ costs $O(m \log N)$ and step $4b$ costs $O(m\kappa)$. Now assume that $\ell = \mathsf{numLevels}$ where $|L_\ell| = m = O(N)$. Then step $5a$ costs $O(m(\log N + D))$, step $5b$ costs $O(m \log N)$ and step $5c$ costs $O(m\kappa)$. Steps 6,7,8, and 9 are free regardless of ℓ. Thus by the standard fact "level of m elements is built every $O(m)$ queries," we have our result.